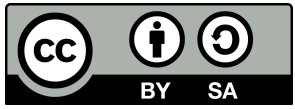
Evolución y Gestión de la Configuración

Material de estudio del curso 2026/2027

David Benavides **Jesús Moreno León**
David Romero Organvídez **José Manuel Sánchez Ruiz**

por orden alfabético de apellidos

Ingeniería Informática - Ingeniería del Software
Universidad de Sevilla



Dedicado a todo el alumnado y profesorado que ha cursado o ha participado en la asignatura Evolución y Gestión de la Configuración, del grado en Ingeniería Informática - Ingeniería del Software de la Universidad de Sevilla, desde el curso 2013-2014. Especial mención merecen José Ángel Galindo, Belén Ramos, Pablo Neira, Antonio M. Gutiérrez y Andrés Jiménez. Sin vuestras múltiples aportaciones a lo largo de los últimos años, ni la asignatura ni esta propia obra serían lo que son.

Prefacio

Este documento ha sido escrito con la intención de servir como material de apoyo para el alumnado de la asignatura Evolución y Gestión de la Configuración del grado Ingeniería Informática - Ingeniería del Software de la Universidad de Sevilla.

No pretende, por tanto, ser una documentación exhaustiva de todos los contenidos del curso. Ni tampoco un *salvoconducto* para ya no tener que asistir a clase :)

De hecho, lo ideal sería que, como estudiante, leyeras el capítulo del tema correspondiente antes de acudir a clase, de manera que en el aula puedas plantear dudas que te hayan surgido con la lectura, compartir anécdotas personales o curiosidades relacionadas con el tema, sugerir posibles matices, correcciones o ampliaciones, participar en los debates y discusiones que tendremos en el aula, así como colaborar con tus compañeros en la resolución de los retos prácticos que se proponen.

Si no eres estudiante de la asignatura y estás leyendo este documento, esperamos igualmente que haya cuestiones que puedan servirte de interés y utilidad, quizá para preparar tus propias clases si eres docente, o incluso para mejorar tu práctica diaria como profesional en la industria.

En cualquiera de los casos, ¡esperamos que disfrutes de la lectura!

Historial de versiones

Este material se revisa y amplía cada curso. La tabla recoge las versiones publicadas, de la más reciente a la más antigua.

Versión	Fecha	Cambios
1.0	6 de septiembre de 2026	Primera versión del material de estudio para el curso 2026/2027.

Contenido

Capítulo 1

Introducción

La máquina solo puede hacer aquello que sabemos ordenarle que haga. Ada Lovelace.

1.1. Del código fuente al producto en uso

Es muy probable que en asignaturas de los primeros cursos del grado tu objetivo al realizar un proyecto haya sido que el software compilara y que produjera el resultado esperado. Sin embargo, en los proyectos reales ese objetivo es solo un paso del camino que hay que recorrer hasta que el producto es utilizado por los usuarios finales.

Por ejemplo, es probable que te haya pasado que un proyecto en el que has estado trabajando funcionaba correctamente en tu ordenador, pero cuando lo has compartido con otros compañeros resulta que no han podido ejecutarlo. O quizá te resulte familiar que al repartir tareas de un proyecto entre distintas personas, cuando habéis tratado de integrar el trabajo han surgido conflictos, errores inesperados o comportamientos difíciles de explicar. No digamos ya que un robot conversacional te haya propuesto una solución para algo que le has pedido y al trasladarlo a tu código lo haya estropeado o empiece a hacer algo que no esperabas.

Incluso puede que hayas vivido situaciones como la siguiente al incorporarte a un equipo. Te pasan el repositorio, desde el que, en teoría, deberías poder construir, probar y ejecutar el sistema. Pero en la práctica alguien te tiene que explicar detalles de palabra, descubres que hay ficheros que se necesitan pero no están en el repo, o dependes de una versión concreta de una biblioteca que alguien instaló en el servidor hace meses pero nadie había documentado.

Para evitar situaciones así, y para poder afirmar con cierta confianza que una funcionalidad está lista realmente para ser utilizada por los usuarios finales, en esta asignatura vamos a estudiar técnicas para construir, integrar, probar, empaquetar, entregar y desplegar software de manera fiable y reproducible. El objetivo final es que podamos crear software mejor y más rápido.

En realidad esto es algo que vemos muy claro cuando miramos a otras industrias. Por ejemplo, piensa en una fábrica de montaje de bicicletas. Es evidente que no es suficiente con fabricar una rueda perfecta, un manillar perfecto o un pedal perfecto. Además hay que ensamblar las piezas, comprobar que todo encaja, y probar que el funcionamiento de pedales, dirección, cambios y frenos es correcto. Pero ni siquiera ahí habría terminado el trabajo: todavía tenemos que empaquetar la bicicleta y enviarla al punto de venta. Y además hacer todo esto de manera previsible, para que podamos planificar el trabajo y evitar sorpresas.

Pues con el software ocurre algo equivalente. Escribir el código es necesario, pero no es suficiente. En este capítulo introductorio vamos a ver, a alto nivel, cómo se desarrolla este proceso en la industria del software. Lo que nos va a servir para estructurar el resto del documento.

Fases desde que se escribe el software hasta que está en uso en producción

El software genera valor cuando puede ser utilizado de forma fiable por sus usuarios o por otros sistemas. Para llegar a ese punto desde el código fuente es necesario pasar por diferentes fases[1].

La primera fase es la construcción (*build*), que básicamente consiste en un proceso que transforma el código fuente y los recursos del proyecto para que el software pueda instalarse o ejecutarse. El resultado de este proceso se suele llamar artefacto o paquete.

En algún momento (luego de pasar pruebas, revisiones, refactorizaciones, modificaciones, etcétera), este paquete se libera o entrega, es decir, se crea una *release*, para lo que se suele publicar el paquete en algún tipo de repositorio de artefactos.

La siguiente fase consiste en desplegar (*deploy*) el paquete o artefacto publicado en el entorno de producción. El software desplegado simplemente está colocado, instalado o configurado en ese entorno, pero no necesariamente se encuentra todavía disponible para su uso.

En muchos casos existe una última fase (*rollout*), que consiste en activar progresivamente el software, poniéndolo a disposición para su uso. En servicios web, esto puede significar mover tráfico hacia la nueva versión. En una app móvil o de

[1] Lamentablemente los términos que se utilizan en la industria no son del todo estándar. Quizá hayas hecho prácticas en empresa ya, y hayas visto que en tu equipo se llamaba de manera diferente a algunos pasos. O que incluso usaba alguno de estos términos para referirse a cosas completamente diferentes. Y, sí, solo podemos decirte que la vida profesional es así de imperfecta. No obstante, hemos tratado de usar los términos más generalizados en la academia y en la industria.

escritorio, puede significar activar una actualización progresiva o hacerla visible para un porcentaje de usuarios o usuarias finales.

En posteriores capítulos profundizamos en diferentes aspectos de las fases de construcción, entrega y despliegue. La fase de activación queda fuera del alcance de esta asignatura, pero en las lecturas recomendadas de cada capítulo puedes encontrar información al respecto, y también hemos incluido un anexo con algunas ideas fundamentales de esta fase. La figura 1.1 resume este recorrido.

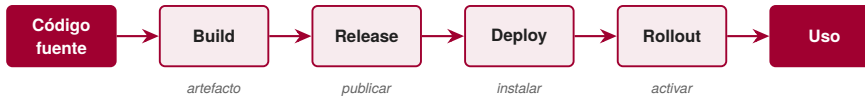


Fig. 1.1 Del código fuente al producto en uso: construcción (*build*), entrega (*release*), despliegue (*deploy*) y activación progresiva (*rollout*).

1.2. ¿Cómo está estructurado el libro?

El Capítulo 2 presenta las ideas fundamentales de la integración continua, la entrega continua y el despliegue continuo. Estas prácticas, conocidas de forma habitual como CI/CD, permiten construir, validar, entregar y desplegar software de manera más frecuente, fiable y reproducible. Frente a enfoques más manuales o basados en grandes entregas, CI/CD ayuda a reducir trabajo repetitivo, incertidumbre y riesgo, al tiempo que mejora la calidad del software y la capacidad de los equipos para reaccionar ante los cambios.

Estas prácticas forman parte de lo que se conoce, de forma amplia, como gestión de la configuración, que es el conjunto de prácticas que nos permite identificar, organizar, controlar y conocer el estado de los elementos que forman parte de un sistema software.

El Capítulo 3 se centra en cómo organizar y gestionar el código fuente y otros recursos de un proyecto software. Veremos por qué los sistemas de control de versiones son una pieza central de la gestión de la configuración, ya que permiten conservar la historia del proyecto, coordinar el trabajo del equipo y relacionar los cambios del código con pruebas, incidencias, versiones y artefactos.

El Capítulo 4 está dedicado a las pruebas del software, otro de los pilares fundamentales de las prácticas CI/CD. Estudiaremos por qué las pruebas automatizadas son necesarias para obtener retroalimentación rápida y ganar confianza en cada cambio. E insistiremos en que, aunque la automatización es clave, probar bien exige

criterio para diseñar buenos casos de prueba, entender los riesgos del sistema y combinar distintos niveles de prueba de forma razonable.

El Capítulo 5 presenta técnicas y estrategias para gestionar tareas, incidencias y cambios durante el ciclo de vida del proyecto. Veremos cómo organizar el trabajo pendiente, documentar decisiones, priorizar problemas, coordinar al equipo y mantener trazabilidad entre lo que se planifica, lo que se modifica en el código y lo que finalmente se entrega.

Por último, el Capítulo 6 profundiza en las ideas de aislamiento y reproducibilidad de los entornos y artefactos software. Estudiaremos cómo distintas metodologías y tecnologías nos ayudan a reducir los problemas derivados de la dependencia del entorno, facilitando que el software pueda construirse, probarse y ejecutarse de forma más previsible en distintas máquinas.

Como ves, el recorrido del libro sigue el camino que va desde el código fuente hasta el software en uso: gestionar los cambios, validarlos, empaquetarlos, entregarlos y desplegarlos con la mayor fiabilidad posible. Vamos, por tanto, a comenzar este viaje acercándonos a las prácticas de CI/CD.

Capítulo 2

Integración continua, entrega continua y despliegue continuo

La frase más peligrosa es: "siempre lo hemos hecho así".
Grace Hopper.

Resumen En este capítulo veremos que, más allá de compilar, crear software implica preparar entornos reproducibles, gestionar dependencias, ejecutar pruebas, revisar código, empaquetar artefactos, mantener trazabilidad y realizar despliegues repetibles. Esto nos permite entender qué aportan la integración continua, la entrega continua y el despliegue continuo, y por qué la automatización es esencial para reducir errores, acortar ciclos de retroalimentación y hacer más fiable y eficiente el proceso.

2.1. Integración continua para construir y validar cada cambio de forma automática

Como hemos mencionado en el Capítulo 1, construir software es el proceso técnico de convertir código fuente, recursos, configuraciones y otros elementos del proyecto en un producto funcional que puede ejecutarse. Sin embargo, gestionar la construcción es algo más amplio: consiste en organizar, automatizar y supervisar ese proceso para que ocurra de forma previsible y con criterios compartidos. Básicamente, construir es ejecutar pasos, y gestionar la construcción es decidir cuáles son esos pasos, cómo se validan, cuándo se ejecutan y qué hacemos si fallan.

La integración continua propone integrar cambios pequeños de manera muy frecuente en una línea principal compartida del repositorio de software, y validar automáticamente que el sistema sigue construyéndose correctamente y superando las pruebas. Pero para hacer esto posible necesitamos entender antes los pasos principales de la gestión de la construcción: entornos, dependencias, análisis estático, pruebas y empaquetado.

2.1.1. Entornos locales y reproducibilidad

El primer obstáculo para poder construir software que nos solemos encontrar es que necesitamos preparar el *entorno*. Un *entorno* en el desarrollo de software es un espacio de trabajo físico o virtual configurado con herramientas, sistemas operativos y en general cualquier configuración de paquetes software específicos. Se suele hablar de *entorno local* (o de desarrollo) al del equipo en el que se desarrolla el software, *entorno de pruebas* en donde se realizan las pruebas y *entorno de producción* en donde se ejecuta el software para su uso. Por ejemplo, tu proyecto puede necesitar una versión concreta de Java, Node o Python, o una extensión específica del editor. Pero si cada estudiante o desarrollador configura todo manualmente, el equipo termina teniendo tantos entornos como personas. Y entonces aparece la frase clásica: "pues en mi máquina funciona".

Una configuración reproducible intenta que el entorno pueda reconstruirse a partir de unas instrucciones, que mantendremos también en nuestro repositorio. El entorno local suele incluir ajustes de editor, extensiones, perfiles de compilación, variables de entorno y configuraciones específicas del proyecto. Algunas de estas decisiones pueden ser preferencias personales, pero otras forman parte del contrato del proyecto. Por ello, si el equipo necesita un perfil de compilación, una configuración concreta de formato, o una extensión para el IDE, tendremos que documentarlo o automatizarlo para facilitar que cualquier persona pueda incorporarse al proyecto sin incidencias.

A lo largo del libro, y también a lo largo del curso en clases de prácticas, veremos diferentes opciones para conseguirlo: usando scripts, ficheros de configuración o contenedores, por ejemplo. Lo importante es que el conocimiento no esté solo en la memoria de una persona. Tiene que estar documentado y versionado.

> **Para saber más**
>
> Revisa la documentación de Black[a], un formateador de código Python. Haz una prueba con uno de tus ficheros para entender el tipo de cambios que realiza en tu código.
>
> [a] `https://github.com/psf/black`

Para saber más

Investiga cómo puedes añadir un archivo EditorConfig[a] a tu proyecto, para indicar a tu IDE determinadas configuraciones (como el tamaño de las indentaciones) que quieres que se mantengan a nivel de proyecto.

[a] https://editorconfig.org/

2.1.2. Dependencias

Una *dependencia* es un elemento en el que tu código se apoya y necesita para funcionar. Son piezas externas que tu proyecto utiliza, como bibliotecas, frameworks, plugins o herramientas. Gestionarlas bien implica decidir qué se instala, en qué versión y de dónde procede.

Por ejemplo, el siguiente programa, que comprueba si la web de nuestra escuela está accesible, depende de la biblioteca *requests*. Es decir, que necesitamos instalar *requests* para que el programa pueda ejecutarse.

Ejemplo de dependencia en Python

```
import requests

response = requests.get("https://www.informatica.us.es/")
if response.status_code == 200:
    print("¡Funciona! :)")
else:
    print("Hay algún problema :(")
```

También debes tener en cuenta las dependencias transitivas, es decir, las dependencias que llegan indirectamente porque unas bibliotecas que usas dependen de otras a su vez. Por ejemplo, *requests* necesita a su vez *urllib3* y *certifi*, y un framework web como *flask* arrastra *click*, entre otras (figura 2.1).

> **Para saber más**
>
> *pipdeptree* es una herramienta que permite mostrar el árbol de dependencias de los paquetes instalados en tu entorno Python. Utiliza *pipdeptree* para ver las dependencias concretas de *requests*.

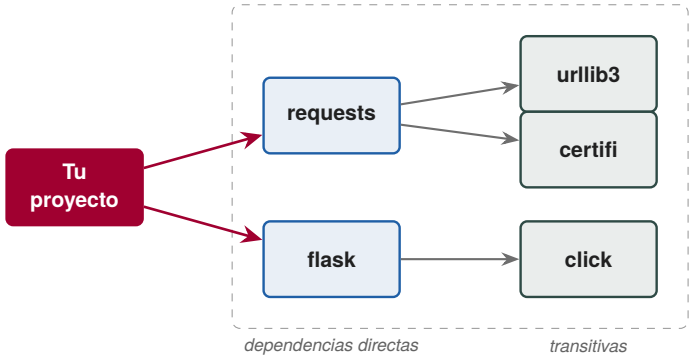


Fig. 2.1 Una dependencia directa puede arrastrar dependencias *transitivas*.

Debido a las dependencias transitivas es fácil que el número de dependencias crezca rápidamente. También pueden aparecer árboles de dependencias difíciles de entender, paquetes retirados, versiones vulnerables o combinaciones que el gestor de paquetes no puede resolver automáticamente. Así que es raro encontrarse con profesionales de la industria que no hayan vivido en primera persona un infierno de dependencias[1] (*dependency hell*).

> **Para saber más**
>
> Investiga sobre el incidente *left-pad* que ocurrió en el año 2016. ¿Cómo se relaciona con las dependencias transitivas que discutimos en esta sección?

Existen muchas soluciones y procesos que nos ayudan a gestionar dependencias[2]. Un fichero como `package.json`, `requirements.txt`, `pom.xml` o `build.gradle`

[1] https://en.wikipedia.org/wiki/Dependency_hell

[2] Por ejemplo, es posible que hayas visto contribuciones a repositorios creadas automáticamente por *Dependabot* para actualizar dependencias vulnerables u obsoletas.

describe dependencias. Un fichero de bloqueo, como `package-lock.json` o `poetry.lock`, ayuda a fijar versiones concretas para que distintas máquinas resuelvan las mismas versiones.

Pero no todas las decisiones se pueden automatizar, ni mucho menos. Un gestor de paquetes puede instalar versiones, pero no puede decidir por ti si conviene introducir una librería pesada, si la licencia es aceptable o si el mantenimiento del paquete es saludable, por ejemplo. Estas son decisiones que debemos tomar las personas, por lo que el equipo debe seguir decidiendo si actualiza, sustituye, congela o elimina dependencias.

> **Para saber más**
>
> En la Universidad de Sevilla hemos hecho investigaciones[a] sobre dependencias transitivas y el impacto que pueden tener en la vulnerabilidad de los proyectos que pueden ser de interés para conocer más sobre esta cuestión.
>
> [a] https://doi.org/10.1016/j.cose.2023.103669

2.1.3. Análisis estático de código con *linters*

Es muy habitual utilizar un tipo de herramientas de análisis estático de código, llamadas *linters*, que analizan el código sin ejecutarlo para detectar posibles problemas, inconsistencias de estilo y antipatrones.

Estas herramientas suelen venir configuradas con una serie de reglas predefinidas. La documentación justifica cada regla, explicando lo que se busca en el código, por qué puede ser un potencial problema, y cómo podría modificarse para evitarlo.

Por ejemplo, esta es la regla SIM-114 de *Ruff*[3], que detecta dos ramas de un if con el mismo cuerpo:

> **Regla SIM-114 de Ruff**
>
> Ejemplo:
> ```
> if x == 1:
>     print("Hola")
> elif x == 2:
>     print("Hola")
> ```

[3] https://docs.astral.sh/ruff/rules/if-with-same-arms/

Utilizar en su lugar:

```
if x == 1 or x == 2:
    print("Hola")
```

Para saber más

Instala *Ruff* y comprueba su funcionamiento. Quizá puedes probar a introducir en tu código algún antipatrón intencionadamente para que salte una regla específica.

El objetivo de los *linters* es detectar problemas de forma temprana, aplicar criterios de consistencia a todo el proyecto y escalar comprobaciones que serían tediosas de revisar manualmente en cada cambio.

2.1.4. Revisión de código

En la sección anterior hemos visto una forma automatizada de revisar ciertos aspectos del código. Sin embargo, hay problemas que una herramienta no puede interpretar bien: intención de diseño, responsabilidades mal repartidas, decisiones de arquitectura o claridad de una solución. Por eso, está muy extendida en la industria la revisión humana, que implica que los cambios en el código sean revisados por otra persona del equipo antes de ser aceptados.

Estas revisiones tienen beneficios obvios, ya que añaden una capa más de seguridad para detectar errores y mantener el código limpio. Pero si se tiene una cultura que fomenta revisiones de código de calidad, pueden ofrecer otros beneficios muy interesantes.

Por un lado, son una gran herramienta de aprendizaje, ya que cuando alguien revisa tu código, te puede dar ideas sobre bibliotecas que no conocías, por ejemplo. Además, si el código te lo revisa alguien con más experiencia, puedes aprender buenas prácticas y familiarizarte con el estilo de código del equipo.

Por otro lado, el hecho de que otra persona revise el código hace que ya haya al menos dos personas que conocen esas líneas. Este conocimiento compartido permite a los equipos evolucionar el código de manera más consistente. Y además te permite irte de vacaciones con más tranquilidad, ya que no tendrás que conectarte por videoconferencia si se produce algún problema con tu código.

Por último, hay que considerar que las revisiones también son una fuente de documentación muy interesante, ya que los comentarios de las revisiones explican por qué las cosas se han hecho de un modo determinado.

> **Para saber más**
>
> En algunos sectores las revisiones de código pueden estar relacionadas con obligaciones contractuales, normativas o de certificación. Investiga brevemente la Ley de Ciberresiliencia europea (*Cyber Resilience Act*) para conocer qué se indica a los fabricantes de productos con elementos digitales en relación a la revisión de código.

2.1.5. Pruebas

Para poder tener confianza en que nuestro código es correcto es imprescindible que contemos con una batería de pruebas bien diseñada. Es decir, que además de escribir el código, tenemos que escribir también código de pruebas que usa el código principal y lanza un error si no se comporta como se espera.

Aunque tenemos un capítulo completo dedicado a las pruebas, el Capítulo 4, desde el comienzo queríamos recalcar la idea de que las pruebas tienen una parte automatizable y una parte humana. En las prácticas aprenderemos a automatizar la ejecución de las pruebas, pero también veremos que el diseño de buenas pruebas exige entender el dominio, los riesgos y los criterios de aceptación.

Es posible escribir y ejecutar cientos de pruebas, pero si estas están mal diseñadas, simplemente tendremos una falsa sensación de seguridad. Por el contrario, unas pocas pruebas bien elegidas pueden aportar más valor que muchas pruebas redundantes o mal diseñadas.

Como veremos en detalle en el Capítulo 4, hay que distinguir entre niveles de prueba. Las pruebas unitarias verifican piezas pequeñas y suelen ser rápidas. Las pruebas de integración validan colaboraciones entre componentes, servicios o bases de datos. Y las pruebas extremo a extremo recorren flujos completos de usuario y suelen ser más costosas.

Un buen diseño de pruebas combina estos niveles de forma estratégica: con cada cambio se ejecutan las rápidas y baratas; mientras que se es más selectivo con la ejecución de las pruebas más lentas y costosas.

2.1.6. Artefactos y empaquetado

Un *artefacto software* es una salida empaquetada y potencialmente distribuible que se produce a partir de un código fuente. Un artefacto puede ser algo tan simple como un archivo de código que ejecutamos o un archivo zip, hasta algo tan complejo como un contenedor o una máquina virtual que contiene todo lo necesario para una aplicación, como veremos en el Capítulo 6.

En muchos proyectos, antes de empaquetar hay que compilar o transformar el código. Esto puede significar traducir Java a bytecode, transpilar TypeScript a JavaScript, minificar (proceso de borrar caracteres sobrantes del código de una página, como espacios, saltos de línea y comentarios, sin alterar cómo funciona) recursos front-end o generar código intermedio. Lo importante para este capítulo es que ese paso también debe poder ejecutarse de forma automática y reproducible, normalmente desde la línea de comandos [4] (la consola o la terminal).

Por tanto, el empaquetado aparece al final de la construcción software, tras la resolución de dependencias, la ejecución de los *linters*, el compilado y la ejecución de las pruebas (figura 2.2).

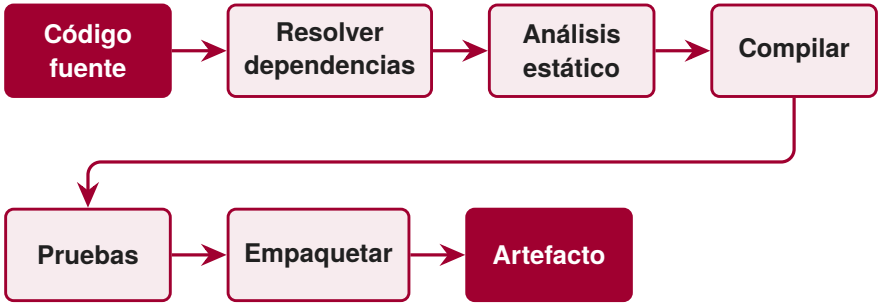


Fig. 2.2 La construcción (*build*): del código fuente al artefacto empaquetado, pasando por la resolución de dependencias, el análisis estático, la compilación y las pruebas.

En función del software que estemos construyendo, el contenido y la estructura del empaquetado será diferente, aunque en muchos casos se trata simplemente de un directorio comprimido que se organiza siguiendo una especificación determinada. Por ejemplo, en Python, empaquetar una biblioteca consiste en producir un artefacto, que se llama *wheel*, siguiendo unas reglas que permitan luego a los instaladores de

[4] Línea de comandos es la traducción mal hecha de *command line*. Sin embargo, la traducción correcta, línea de órdenes, es muy poco usada en español por lo que hemos decidido usar línea de comandos.

paquetes como *pip* o *uv* instalarla en el sistema. Las *wheels* tienen que contener, por tanto, todo lo necesario para su instalación: el código, los metadatos del paquete (como su nombre y versión), información sobre sus dependencias, e incluso instrucciones que indiquen dónde hay que colocar los archivos en el entorno.

> **Para saber más**
>
> Aunque pueda sonar complicado, en realidad crear una *wheel* en Python es muy simple. Investiga sobre *pyproject.toml* para crear una *wheel* de un programa tuyo sencillo.

El objetivo de construir paquetes es reducir o evitar que haya que reconstruir el software desde el código fuente en cada máquina que vaya a ejecutarlo. Además, los paquetes preconstruidos ofrecen más estabilidad, ya que pueden aparecer inconsistencias si cada máquina construye el software en su propio entorno. En consecuencia, si nuestro software va dirigido a más de un entorno, tendremos que producir diferentes paquetes, adaptados a distintos sistemas operativos o arquitecturas de CPU, por ejemplo.

Como en todos los pasos discutidos, la automatización juega un papel importante en el empaquetado. Pero el equipo debe gestionar la estrategia: qué formato conviene, cuándo se empaqueta, qué recursos deben incluirse o excluirse, qué optimizaciones son necesarias para una plataforma o cómo se resuelven conflictos entre recursos o dependencias.

2.1.7. Integración continua

Hasta ahora hemos visto varias piezas del proceso de construcción: preparar el entorno, gestionar dependencias, compilar, ejecutar análisis estático, lanzar pruebas y empaquetar artefactos. Pero ¿cuándo hacemos todo eso? ¿Y quién se encarga de ello?

Una posibilidad sería esperar al final del proyecto, juntar el trabajo de todas las personas del equipo y comprobar entonces si todo encaja. Ya puedes imaginar el problema: cuanto más tiempo pasa un cambio aislado, más probable es que choque con otros cambios. Puede fallar la compilación, pueden romperse pruebas, puede haber dependencias incompatibles y es muy probable que dos partes del sistema hayan evolucionado en direcciones distintas generando conflictos.

La integración continua, o *Continuous Integration* (CI), aparece como respuesta a ese problema en el contexto de prácticas como Extreme Programming, durante la década de 1990, y se populariza después dentro del movimiento ágil. CI propone integrar cambios frecuentemente en una línea principal compartida del repositorio y comprobar, mediante un proceso automático de construcción y pruebas, si esos cambios conviven correctamente con el resto del sistema.

Esto implica trabajar con cambios pequeños, integrar con frecuencia, mantener la línea principal del repositorio en buen estado y hacer que los fallos aparezcan pronto, cuando son más sencillos de entender y corregir.

El dolor de integrar tarde

Imagina que estás trabajando en una práctica en grupo. Cada persona se encarga de una parte y durante dos semanas nadie integra nada. Al final, juntáis el trabajo. Una persona ha cambiado el modelo de datos, otra ha actualizado una dependencia, otra ha reorganizado los nombres de varias funciones y otra ha escrito pruebas contra una versión antigua de la API. Cada cambio parecía razonable por separado, pero al unirlos aparecen errores difíciles de diagnosticar.

Esta situación es tan común que tiene nombre propio: *integration hell* o *merge hell*. Y por eso, una de las ideas principales de CI es: *Si algo duele, hazlo más a menudo y adelanta el dolor.* Es decir, que si sabemos que integrar al final del proyecto es doloroso, la solución es integrar antes y con más frecuencia, de forma que los problemas aparezcan cuando todavía son pequeños. Cuanto más frecuentemente integramos, en menos sitios tenemos que buscar si se produce un error, y arreglamos los problemas mucho más rápido. Por eso en este capítulo hablamos de *integración continua*. Todavía puedes tener dudas sobre qué significa en este contexto "continua" (¿una vez al día?, ¿una vez cada hora?, ¿una vez a la semana?) y la respuesta es que depende. Vamos poco a poco.

El ciclo básico de CI

Un ciclo básico de CI tiene las siguientes ideas principales (figura 2.3):

1. Una persona realiza un cambio en el código.
2. Ese cambio se integra en la línea principal compartida del repositorio, normalmente tras una revisión.
3. Un servidor o servicio de CI detecta el cambio.
4. El servidor obtiene una copia limpia del repositorio.

5. Ejecuta automáticamente el proceso de construcción: resuelve dependencias, compila, analiza, pasa las pruebas y, si procede, empaqueta.
6. Publica el resultado del proceso: éxito, fallo, logs, informes de pruebas o artefactos generados.
7. El equipo recibe feedback y actúa en consecuencia.

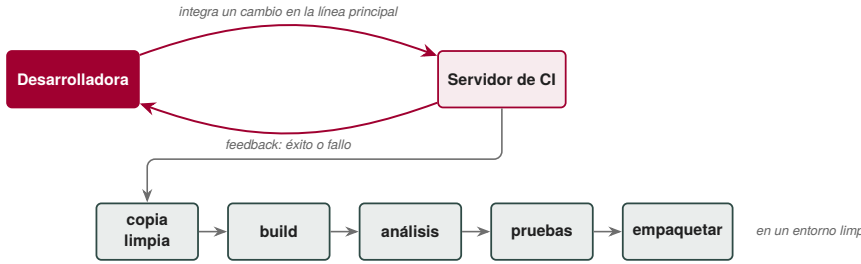


Fig. 2.3 El ciclo de integración continua: cada cambio integrado dispara una construcción automática en un entorno limpio, que devuelve feedback rápido al equipo.

Servicios de CI pueden ser Jenkins[5], Travis CI[6], Circle CI[7], GitHub Actions[8] o GitLab CI/CD[9] entre otras muchas opciones.

Un detalle importante es que el servidor de CI parte de un entorno limpio que va configurando con las instrucciones que le demos (lo que se conoce como *pipeline*). Eso permite detectar errores que en tu ordenador pueden quedar ocultos, como ficheros que olvidaste añadir al repositorio, dependencias instaladas manualmente, variables de entorno no documentadas o configuraciones que solo existen en tu máquina.

Principios prácticos de CI

Una formulación clásica de CI, asociada al artículo de referencia de Martin Fowler[10] (escrito en 2001, actualizado en 2006 y reescrito más recientemente en 2023), puede resumirse en los siguientes principios:

[5] https://www.jenkins.io/
[6] https://www.travis-ci.com/
[7] https://circleci.com/es/
[8] https://docs.github.com/es/actions
[9] https://docs.gitlab.com/ci/
[10] https://martinfowler.com/articles/continuousIntegration.html

1. Poner todo en la línea principal de un sistema de control de versiones.
2. Automatizar el build.
3. Hacer que el build sea auto-testeable.
4. Integrar cambios en la línea principal al menos una vez al día, idealmente con más frecuencia.
5. Lanzar una integración automática por cada cambio integrado.
6. Mantener el build rápido.
7. Arreglar un build roto inmediatamente.
8. Esconder el trabajo que no está terminado.
9. Probar en un entorno lo más parecido posible al entorno real.
10. Hacer visible para todo el equipo qué está ocurriendo.

Destacamos a continuación algunos aspectos interesantes sobre cada uno de esos principios. En primer lugar, aunque discutiremos en profundidad esta cuestión en el Capítulo 3, es importante entender que CI se apoya en una línea principal compartida en el repositorio. En Git, esa línea suele ser *main*; en otros contextos puede llamarse *trunk*, *mainline* o *master*. El objetivo es que el equipo tenga una referencia común que se mantiene siempre en un estado sano.

Por otra parte, el build debe poder ejecutarse sin pasos manuales. Si para construir tu proyecto necesitas abrir un IDE, pulsar varios botones y recordar instrucciones que no están documentadas, todavía no tienes una base sólida para CI. Idealmente debería ser posible que cualquier persona del equipo pueda abrir una máquina limpia, traerse las fuentes del repositorio, ejecutar una única orden, y tener listo el sistema en su propio entorno. Y para que eso sea posible el repositorio debe poder devolver el código fuente, las pruebas, el esquema de la base de datos, los datos de prueba, los archivos de configuración, las configuraciones del IDE, los scripts de instalación, dependencias y cualquier herramienta requerida para construir el software[11].

Tal como se presenta en el Capítulo 4, en CI cada vez que se publica un cambio en la línea principal se lanza un build que debe pasar una batería de pruebas. Esta suite de pruebas debe ser suficientemente detallada como para que el equipo pueda estar bastante seguro de que el sistema está funcionando correctamente y el cambio no ha roto nada, es decir, que el cambio se ha integrado correctamente con el resto del sistema sin producir regresiones. Y para ello trataremos de pasar las pruebas en un entorno que se parezca lo máximo posible al entorno real donde se ejecuta nuestro software en producción, para lo que los entornos virtuales, las máquinas virtuales y los contenedores pueden ser de gran ayuda, como veremos en el Capítulo 6.

[11] Esto no significa que el repositorio deba almacenar todos estos elementos, pero sí debe poder permitir al equipo conseguir todos ellos automáticamente en sus versiones adecuadas, como veremos en detalle en el Capítulo 6.

Una regla práctica clásica es que cada persona integre al menos una vez al día. Kent Beck incluso afirma en su libro sobre Extreme Programming que *ningún código debe estar sin integrarse durante más de un par de horas*. Pero, entonces, ¿qué ocurre con las funcionalidades algo más grandes que quizá no están listas ni siquiera al final del día? Como se presenta en el Apéndice A, existen estrategias para poder trabajar en ellas e ir integrando cambios pequeños y verificables, y hacer que la nueva funcionalidad no esté visible para los usuarios. Sería parecido a una obra de ampliación en un gran edificio en la que el equipo trabaja detrás de un panel para no molestar a quienes lo usen, pero va conectando poco a poco electricidad, climatización, seguridad o fontanería al edificio existente para comprobar que todo encaja antes de abrir la nueva zona al público.

Además es importante que el feedback llegue pronto al equipo tras realizar un cambio. Una regla básica de referencia es que tarde alrededor de diez minutos o menos. No es una ley universal, por supuesto, pero si el proceso tarda demasiado, existe el riesgo de que el equipo deje de prestarle atención.

Para que este enfoque funcione, el equipo debe mantener una cultura de arreglar rápidamente la línea principal si algo se rompe. En palabras de Kent Beck, *nadie tiene una tarea con prioridad mayor que arreglar el build*. Y Jez Humble, uno de los autores más influyentes en este campo, llega a plantear que, para poder hablar realmente de CI, un build roto debería repararse en cuestión de 10 minutos.

Por último, es fundamental que el resultado del proceso sea claro. Un buen sistema de CI te dice rápida y claramente si el cambio se integra. Si el proceso falla, el sistema debe ofrecer información suficiente para actuar, detallando qué etapa concreta es la que falló, si es una prueba específica que no pasó, por ejemplo, y qué cambio introdujo probablemente el problema.

Si el proceso se completa sin fallos, ¡genial!, porque sabemos que el cambio reciente se integra correctamente con el sistema, lo que aumenta la confianza del equipo en que la línea principal sigue siendo construible y comprobable. Pero ¿qué ocurre después de construir y validar un cambio? ¿Cómo se convierte en una *release*? ¿Cómo se despliega en entornos reales? ¡Vamos a verlo en las siguientes secciones!

2.2. Entrega continua para que el software siempre esté en una versión liberable

En la sección anterior hemos visto cómo la integración continua nos ayuda a construir y validar cambios de manera frecuente. Si todo va bien, al final de ese proceso tendremos un artefacto: por ejemplo, una imagen Docker, un paquete Python,

un archivo `.jar`, un instalador o cualquier otro resultado generado por el proceso de construcción.

En ese momento el equipo tiene que decidir qué artefacto concreto va a considerar válido, qué identidad le asigna, dónde lo deja disponible, documentar qué pruebas o aprobaciones ha superado y registrar de qué código concreto salió.

A este paso lo llamamos *release*, o entrega, y consiste en decidir si una versión concreta del producto puede avanzar hacia su distribución o despliegue.

2.2.1. Qué significa hacer una release

Hacer una *release* consiste en tomar un artefacto construido, validarlo y convertirlo en una versión identificada, trazable y potencialmente entregable. Por ejemplo, imagina que se ha construido una imagen de una máquina virtual a partir del último código del repositorio. Esa imagen puede funcionar correctamente, pero todavía necesitamos responder algunas preguntas, como: ¿qué versión del producto representa?, ¿a partir de qué cambio se construyó?, ¿qué pruebas ha superado?, ¿dónde queda almacenada?, ¿puede usarse para desplegar en preproducción o producción? o ¿podemos volver a encontrarla dentro de tres meses si aparece un problema?

Estas son precisamente las preguntas que tratamos de responder a la hora de hacer una *release*. En consecuencia, suele incluir actividades como asignar un número de versión, etiquetar el código, generar notas de versión, almacenar el artefacto en un repositorio o canal de distribución, y dejar registrada la relación entre código fuente, artefacto, versión y entorno. Y para estas tareas es habitual apoyarse en un repositorio de artefactos, como *Artifactory*[12], *Nexus Repository*[13] o *GitLab Package Registry*[14].

Es importante ser consciente de que no todos los builds generados son tratados de igual forma. Algunos builds sirven solo para comprobar que un cambio compila. Otros builds sirven para ejecutar pruebas internas. Otros builds generan artefactos temporales. Pero no todos esos resultados tienen por qué convertirse en una release.

Una release, en cambio, debe poder reconocerse y recuperarse (figura 2.4). Si dentro de un tiempo alguien pregunta qué versión se desplegó, qué cambios incluía o qué artefacto exacto se utilizó, el equipo debería poder responder sin depender de la memoria de una persona.

[12] https://jfrog.com/artifact-management/

[13] https://github.com/sonatype/nexus-public

[14] https://docs.gitlab.com/user/packages/generic_packages/

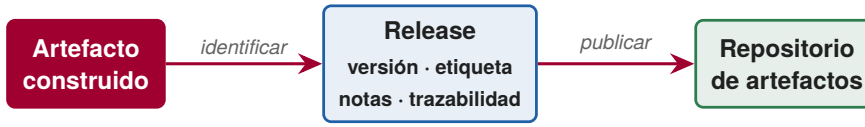


Fig. 2.4 Una *release* toma un artefacto, lo convierte en una versión identificada y trazable (número de versión, etiqueta, notas) y lo publica en un repositorio para poder recuperarlo después.

2.2.2. Cómo se hacían tradicionalmente las releases

En esta asignatura estamos conociendo técnicas y metodologías que permiten que un cambio pueda avanzar automáticamente por varias etapas hasta obtener un artefacto validado y listo para convertirse en una versión entregable. Pero durante mucho tiempo, y todavía hoy en algunos contextos, hacer una release era un evento muy importante, planificado y bastante manual.

En algún momento, el equipo decidía que había suficientes cambios para preparar una nueva versión. A partir de ahí se estabilizaba el código, se elegían los cambios que iban a entrar, se generaba un candidato de versión, se ejecutaban pruebas, se corregían errores, se volvía a generar otro candidato y se preparaba un plan para llevar esa versión a producción o distribuirla a los usuarios.

En ese modelo, liberar o entregar software era algo que ocurría cada cierto tiempo: una vez al mes, una vez cada trimestre, una o dos veces al año, o cuando el equipo consideraba que había suficientes cambios acumulados. Cuanto más tiempo pasaba entre releases, más grande era cada una de ellas. Y cuanto más grande era la release, más difícil era entender qué podía fallar.

Esto genera un problema muy parecido al que vimos en la sección anterior. Si acumulas muchos cambios durante semanas o meses, el día de la release se convierte en un momento delicado. Hay más cosas que probar, más personas que coordinar, más documentación que preparar y más riesgo de que algo salga mal.

Y como el proceso era así de complicado, en las empresas era muy habitual que existiera una persona responsable de su coordinación. Dependiendo de la organización, esta figura podía llamarse *release manager*, *release captain* o recibir otro nombre parecido.

Clare Liguori describe cómo se desarrollaba este proceso en Amazon alrededor del año 2010[15] y cuáles eran las tareas que debía coordinar como parte de su trabajo, como decidir cuándo se preparaba la release, determinar qué cambios entraban y cuáles quedaban fuera, asignar un número de versión, comprobar que se habían

[15] https://d1.awsstatic.com/builderslibrary/pdfs/CICD_pipeline_as_release_captain.pdf

ejecutado las pruebas necesarias, preparar instrucciones de despliegue o distribución, coordinar el orden de despliegue en distintos entornos, obtener aprobaciones, y preparar un plan por si algo salía mal.

Era una función necesaria porque el proceso tenía muchas partes manuales. Y, por tanto, era también una tarea muy costosa, lenta y propensa a errores. La persona que asumía ese papel tenía que dedicar tiempo a coordinar la release en lugar de desarrollar nuevas funcionalidades, mejorar el producto o resolver otros problemas técnicos. Además, si el proceso dependía demasiado de esa persona, el equipo podía volverse frágil: entregar software requería conocimiento manual, listas de pasos, experiencia acumulada y mucha atención.

Por todo ello, muchas organizaciones han intentado reducir el tamaño de las releases y aumentar su frecuencia. La idea es parecida a la de CI: si liberar software duele, conviene hacerlo más a menudo, con cambios más pequeños y con más automatización. El objetivo es que el software esté siempre en estado entregable, reduciendo la necesidad de grandes eventos manuales de release (figura 2.5).

Fig. 2.5 Del modelo tradicional (pocas releases grandes y arriesgadas) a la entrega continua (muchas releases pequeñas y frecuentes).

Para saber más

Cada cierto tiempo y desde hace algunos años, se publican los informes y métricas de DORA (*DevOps Research and Assessment*[a]). Estos informes son algo así como un estándar de la industria para evaluar la eficacia, la velocidad y la estabilidad de los procesos de desarrollo y entrega de software. En el entorno de CI/CD, su importancia radica en que ofrecen un marco de medición objetivo basado en cinco métricas clave: frecuencia de *deploy* y tiempo de ciclo (*lead time*) para medir la velocidad, junto con el porcentaje de fallos en cambios y el tiempo medio de recuperación (*MTTR*) para medir la estabilidad,

además del porcentaje de despliegues de retrabajo (una nueva métrica, incluida en 2024, que mide la proporción de despliegues no planificados realizados para corregir errores visibles para los usuarios).

[a] https://dora.dev/

2.2.3. Versionado

Versionar significa asignar una identidad única y reconocible a un estado concreto del software y a los artefactos que lo representan. Una estrategia habitual, especialmente cuando el software expone una API pública, es el versionado semántico[16], que usa la forma `X.Y.Z`, también escrita como `MAJOR.MINOR.PATCH`. El primer número suele representar cambios mayores o incompatibles con versiones anteriores; el segundo, cambios menores de funcionalidad compatibles; y el tercero, correcciones o cambios pequeños que no alteran la funcionalidad pública. Por ejemplo, pasar de `1.4.2` a `1.5.0` sugiere una funcionalidad nueva compatible, mientras que pasar a `2.0.0` avisa de cambios más profundos (figura 2.6).

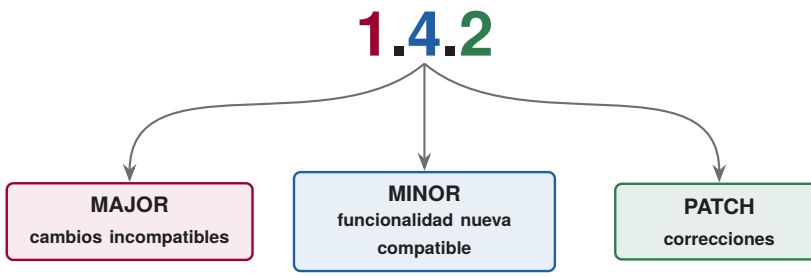


Fig. 2.6 Versionado semántico `X.Y.Z`: cada número comunica el tipo de cambio respecto a la versión anterior.

También es frecuente etiquetar el estado de madurez de una versión. Una versión *alpha* suele ser temprana e inestable; una *beta* ya permite validación más amplia, pero puede contener fallos; una *release candidate* o `rc` es candidata a convertirse en la versión final si no aparecen problemas relevantes; y la *final release* es la entrega definitiva.

[16] https://semver.org/lang/es/

Algunos proyectos usan versionado por fecha, como `26.04` o `20260414`, cuando la fecha de entrega comunica mejor la evolución del producto. En cualquier caso, el esquema de versionado que utilices debería garantizar que las versiones son únicas, comparables e informativas. Por ejemplo, hay fabricantes que a veces utilizan esquemas de versionado con nombres comerciales, como *Android Ice Cream Sandwich* o *Android Jelly Bean*, o también *Claude Sonnet* y *Claude Opus*, que si bien son nombres únicos, por sí solos no son suficientes para ordenar versiones, comparar compatibilidad o automatizar decisiones técnicas.

Para saber más

Es interesante leer las notas de entrega, o *release notes*, de diferentes versiones de proyectos impulsados por la comunidad. Por ejemplo, puedes consultar las notas de versión de Blender [a], comprobando la entidad de los cambios, correcciones y nuevas funcionalidades aportadas por los colaboradores en cada iteración.

[a] `https://www.blender.org/download/releases/`

2.2.4. Artefactos inmutables y trazabilidad

Una buena release necesita trazabilidad. Esto significa que debemos poder reconstruir la relación entre el código fuente, el proceso de construcción, el artefacto generado y los entornos donde ese artefacto se ha usado.

Si aparece un error en producción necesitamos saber qué versión exacta está desplegada, a partir de qué commit se construyó, las pruebas que pasó antes de ser entregado o cuál era la versión anterior, por si hay que volver atrás.

El versionado nos ayuda, por supuesto, pero para que funcione los artefactos de release deberían ser inmutables. Es decir, una vez construido y publicado un artefacto, no deberíamos modificarlo. Si necesitamos cambiar algo, generamos un artefacto nuevo con una identidad nueva.

Esto evita una fuente muy peligrosa de confusión. Imagina que publicas una imagen Docker con la etiqueta `app:1.4.2`. Luego descubres un pequeño problema, reconstruyes la imagen y vuelves a publicar otra imagen distinta con la misma etiqueta. Ahora tienes dos artefactos diferentes con el mismo nombre. Puede que una máquina tenga una versión, otra máquina tenga otra, y dentro de unas semanas nadie sepa exactamente qué se probó o qué se desplegó.

Por eso es buena práctica usar identificadores únicos, checksums, etiquetas asociadas al commit o mecanismos equivalentes que permitan saber qué artefacto exacto se está utilizando. En sistemas modernos de entrega continua, el propio sistema puede registrar automáticamente la relación entre commit, build, artefacto, release y entorno.

2.2.5. Entrega continua

Llegados a este punto, podemos entender mejor qué aporta *Continuous Delivery* (CD), o entrega continua[17].

Como vimos en la sección anterior, CI se centra en comprobar si cada cambio se integra correctamente con el resto del sistema, y para ello se usan pruebas automatizadas que deben devolver feedback de manera rápida al equipo de desarrollo. CD va un paso más allá: después de construir y validar técnicamente un cambio, se pregunta si el software queda en un estado entregable.

En la práctica, la entrega continua se materializa mediante una *delivery pipeline*: una secuencia automatizada de etapas que toma cada cambio integrado, construye un artefacto desplegable, lo almacena como *release candidate*, lo valida progresivamente y determina si puede considerarse liberable. Por tanto, la pipeline puede entenderse como la ruta definida por el equipo para decidir, de forma repetible y trazable, si una versión estaría lista para avanzar hacia producción.

En muchos casos esta fase de entrega también incluye pasar otro conjunto de pruebas automatizadas, que tienen un enfoque más centrado en el usuario, se ejecutan en entornos más parecidos a producción todavía, y suelen ser más lentas y costosas que las pruebas de CI. Por ejemplo, Dave Farley explica en su libro *Continuous Delivery Pipelines* que para construir el London Multi-Asset Exchange en CI usaban unas 40.000 pruebas pequeñas que se pasaban en unos 4 minutos. Si las pruebas se superaban, la persona que había realizado el cambio podía seguir trabajando con confianza en una nueva funcionalidad, mientras el *release candidate* seguía pasando las pruebas de CD, que tardaban unos 40 minutos.

CD significa que cada cambio que supera el proceso de validación deja el software en condiciones de ser entregado. Eso no significa necesariamente que se despliegue automáticamente a producción o se distribuya a los usuarios. La decisión final puede seguir dependiendo de una persona, de una política de negocio, de una ventana de mantenimiento, de una tienda de aplicaciones o de una validación externa. Pero el

[17] En esta sección usamos CD para referirnos a *Continuous Delivery*. Más adelante distinguiremos esta práctica de *Continuous Deployment*, que automatiza también el despliegue a producción.

producto debería estar preparado para liberarse sin tener que improvisar un proceso manual cada vez.

Dicho de otra forma, cuando se usa CD, la release deja de ser un gran evento excepcional y se convierte en una capacidad habitual del equipo.

En algunas organizaciones llaman a esta idea *el pipeline como release captain*. En lugar de que una persona tenga que coordinar manualmente cada release, la pipeline se encarga automáticamente de detectar los cambios integrados, construir el artefacto, ejecutar pruebas y comprobaciones, asignar identidad a la release, almacenar el artefacto, registrar trazabilidad entre commit, artefacto y release, impedir que una release avance si falla una comprobación, y mostrar al equipo información de lo que está ocurriendo en cada momento.

Es importante que tengamos claro que esto no elimina el criterio humano. El equipo sigue teniendo que decidir qué comprobaciones son necesarias, qué riesgos son aceptables, qué política de versionado se usa, qué significa que una release está lista y cuándo conviene publicarla a usuarios reales. La diferencia es que esas decisiones se incorporan al proceso, en lugar de depender de una persona que recuerda pasos manuales.

De este modo, si el proceso de CD se implementa correctamente, nuestro software está siempre en un estado entregable, ya que el proceso de construcción, validación, empaquetado y almacenamiento del artefacto en el repositorio de releases está automatizado y es repetible.

Este enfoque reduce el tamaño de las releases, acorta el tiempo entre escribir un cambio y tenerlo disponible, y disminuye el riesgo asociado a grandes lotes de cambios. También permite que el equipo dedique menos tiempo a coordinar releases manuales y más tiempo a mejorar el producto.

En la Tabla 2.1 puedes ver un resumen de las diferencias entre integración continua y entrega continua.

Pero todavía falta una pregunta importante. Una vez que tenemos una release construida, validada, identificada y disponible, ¿cómo la llevamos a un entorno real? Esa será la siguiente fase: el despliegue.

2.3. Despliegue continuo para llevar de manera fiable las entregas al entorno de producción

En la sección anterior hemos visto que una *release* es una versión identificada, trazable y potencialmente entregable del software. Pero que una release exista no significa todavía que esté ejecutándose en ningún sitio.

Característica	Integración continua	Entrega continua
Objetivo principal	Integrar y validar frecuentemente los cambios de código en una línea principal compartida.	Mantener el software en un estado liberable y preparado para avanzar hacia producción cuando se decida.
Fase del ciclo	Integración, construcción y validación (*build*).	Entrega y validación de la versión (*release*).
Automatización	Resolución de dependencias, construcción, análisis y ejecución de pruebas tras cada cambio integrado en la línea principal.	Generación, almacenamiento y validación progresiva de una *release candidate* en entornos cada vez más parecidos a producción.
Frecuencia	Tras cada cambio integrado en la línea principal, normalmente varias veces al día.	Cada cambio que supera CI avanza por la *delivery pipeline*, aunque la liberación final se produzca cuando la organización lo decida.
Resultado	Cambio integrado y validado, normalmente acompañado de un artefacto construido y probado.	*Release candidate* identificada, trazable y lista para liberarse en cualquier momento.
Beneficios esperados	Detección temprana de errores y reducción de los problemas derivados de integrar tarde (*merge hell*).	Reducción del tiempo de entrega (*time-to-market*), del tamaño y riesgo de las releases, y de la coordinación manual necesaria.

Tabla 2.1 Comparativa entre integración continua y entrega continua.

La siguiente fase es el despliegue, o *deploy*. Desplegar consiste en llevar una release a un entorno concreto para que pueda ejecutarse allí. Por ejemplo, desplegar puede significar instalar un paquete en un servidor, arrancar una nueva imagen Docker en un clúster, actualizar una función serverless, copiar una web estática a un sistema de publicación o instalar una nueva versión de un servicio interno.

La idea importante es que el despliegue trata de colocar una versión concreta del software en un entorno determinado y hacer que pueda ejecutarse allí.

2.3.1. Desplegar no es lo mismo que entregar

Insistimos en la diferencia entre el concepto de entregar (*release*) y desplegar (*deploy*) porque es habitual encontrarse con usos de estos términos como si fueran sinónimos. Pero no lo son. Si los confundes, puede que no te entiendas bien con tu equipo de trabajo.

Hacer una *release* (entrega) significa que tenemos una versión identificada y disponible. Desplegar (*deploy*) significa instalar o ejecutar esa versión en un entorno concreto. Por tanto, una release puede existir aunque no se haya desplegado todavía. Y una misma release puede desplegarse en varios entornos distintos. Por ejemplo, imagina que has terminado la práctica de una asignatura y ya le has asignado una versión y has creado, por ejemplo, un archivo comprimido con todo lo que se necesita. Eso sería una entrega. Tomar ese artefacto, descomprimirlo, instalar sus dependencias, instalar el propio sistema y ponerlo en funcionamiento, sería desplegarlo. Así, un equipo podría generar la release `1.4.2`, almacenarla en un repositorio de artefactos y desplegarla primero en un entorno de pruebas. Si todo va bien, esa misma release puede desplegarse luego en preproducción y, más tarde, en producción En este caso la release es la misma, pero está desplegada en varios entornos diferentes.

2.3.2. Entornos de despliegue: desarrollo, preproducción y producción

En un proyecto real no suele existir un único entorno. Lo habitual es que haya varios entornos con objetivos distintos.

Un entorno de desarrollo permite probar cambios de forma rápida, normalmente con menos restricciones. Un entorno de pruebas o integración sirve para validar que distintas partes del sistema funcionan juntas. Un entorno de preproducción intenta parecerse lo máximo posible a producción para detectar problemas antes de afectar a usuarios reales. Y producción es el entorno donde el software presta servicio a usuarios o sistemas reales (figura 2.7).

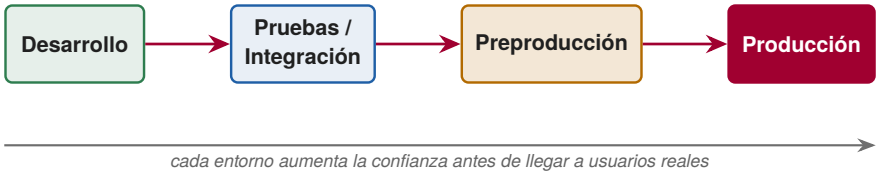


Fig. 2.7 Cadena de entornos de despliegue. Más allá de los nombres concretos, cada entorno busca aumentar la confianza antes de exponer el cambio a usuarios reales.

Puede que leyendo artículos, o incluso si ya has hecho prácticas en empresa, hayas visto que hay proyectos que no usan exactamente estos nombres. Algunas organizaciones hablan de *dev*, *sandbox*, *staging* o *testing*. También hay empresas que usan entornos llamados *alpha* y *beta*, que se utilizan para validar funcionalidad

e integración, y *gamma*, que intenta parecerse lo máximo posible a producción. Más allá del nombre, en cualquier caso, lo importante es la función que cumplen. Y la idea es que cada entorno debería aumentar nuestra confianza antes de exponer el cambio a usuarios reales.

2.3.3. Despliegues seguros y automatizados

Un despliegue manual puede parecer sencillo cuando el sistema es pequeño: copiar unos ficheros, reiniciar un servicio y comprobar que todo sigue funcionando. Pero a medida que el sistema crece, ese proceso se vuelve frágil. Alguien puede olvidar un paso, ejecutar una orden en el servidor equivocado o usar una versión incorrecta del artefacto, por ejemplo.

Automatizar el despliegue ayuda a reducir esos errores. Un proceso automatizado puede tomar una release concreta, desplegarla siempre de la misma forma, registrar qué ha ocurrido y detenerse si alguna comprobación falla, como veremos en detalle en las prácticas y en el Capítulo 6. Además, existen herramientas específicas que nos ayudan con estos despliegues automatizados, como *Spinnaker*[18] o *Harness*[19].

Pero para que esta automatización funcione es importante usar artefactos identificados e inmutables, ejecutar pasos repetibles y dejar registro de qué release se ha desplegado y dónde. Y si el proceso es automático, es fundamental comprobar que el entorno queda en un estado saludable y permitir detectar rápidamente si algo ha ido mal, para evitar que una versión avance si no supera las comprobaciones necesarias.

> **Para saber más**
>
> Investiga el concepto de infraestructura como código. Reflexiona sobre cómo se relaciona esta idea con los procesos de CI/CD que hemos estudiado. Si además te sigue picando la curiosidad, intenta buscar sobre *DevOps* y piensa qué relación tiene con todo esto que estamos viendo.

[18] https://spinnaker.io/

[19] https://www.harness.io/

2.3.4. Despliegue continuo

Como hemos visto, la integración continua nos ayuda a comprobar si un cambio se integra correctamente. Por su parte, la entrega continua nos ayuda a mantener el producto en estado liberable. El despliegue continuo va un paso más allá: si una release supera todas las validaciones necesarias, el sistema la despliega automáticamente en producción (figura 2.8).

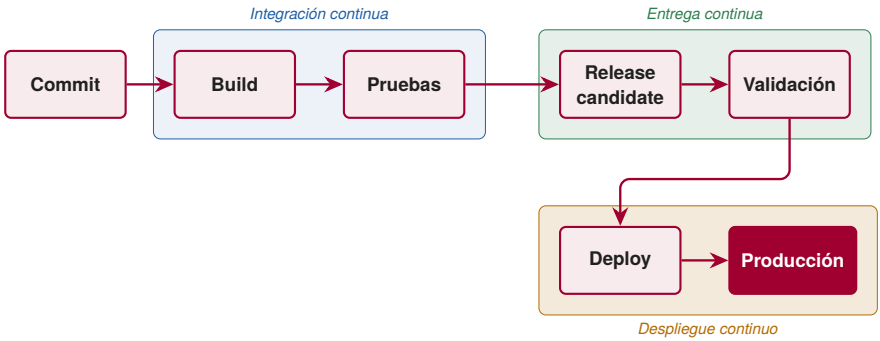


Fig. 2.8 El *pipeline* de CI/CD por fases: integración continua (construir y validar cada cambio), entrega continua (dejar el producto siempre liberable) y despliegue continuo (llevarlo a producción de forma automática).

Este modelo encaja especialmente bien en servicios web, APIs y sistemas en la nube donde el equipo controla el entorno de ejecución. En esos casos, si el pipeline tiene buenas pruebas, buenos controles y buena observabilidad, puede tener todo el sentido que el despliegue a producción no requiera una decisión manual en cada cambio.

En empresas de internet con sistemas muy grandes, la automatización del despliegue ha llegado a escalas que impresionan. Por ejemplo, Jon Jenkins, que era el director de análisis de la plataforma en Amazon.com en 2014, afirmaba en una charla[20] que hacían cambios a producción cada 11,6 segundos de media. En 2021, Meta describía Conveyor, su plataforma de despliegue continuo para servicios, como capaz de realizar más de 100.000 despliegues por semana[21].

[20] https://www.youtube.com/watch?v=dxk8b9rSKOo

[21] https://atscaleconference.com/conveyor-continuous-deployment-at-facebook/

Independientemente de la escala, lo que estas cifras muestran es que, cuando el proceso está muy automatizado, los despliegues dejan de ser eventos excepcionales y se convierten en una actividad normal del equipo.

En cualquier caso, es importante recalcar que no todos los productos encajan igual de bien con despliegue continuo. Una app móvil puede depender de una tienda. Una aplicación de escritorio puede requerir instaladores, firma y actualizaciones graduales. Un firmware puede necesitar validaciones muy estrictas. En esos casos, puede ser más razonable practicar entrega continua: el producto está siempre preparado para liberarse, aunque la publicación o distribución final siga una política más controlada.

Por tanto, no debe entenderse el despliegue continuo como el objetivo obligatorio para cualquier proyecto. Es una posibilidad que tiene sentido cuando el equipo controla suficientemente el entorno, puede automatizar las validaciones necesarias y tiene mecanismos para detectar problemas rápidamente y reaccionar ante ellos.

> **Para saber más**
>
> Escucha el episodio de podcast *CI/CD nativo en la nube: llevando DevOps al siguiente nivel*, que está dividido en dos partes [a] [b]. En la entrevista participan dos responsables de tecnología de una empresa de comercio al por menor con más de 18.000 trabajadores en España, en la que explican cómo son sus procesos de integración, entrega y despliegue continuos. Aunque hay conceptos que todavía no hemos tratado, seguro que muchas de las cosas que discuten ya te resultan familiares.
>
> [a] https://www.ivoox.com/cicd-cloud-native-llevando-devops-al-siguiente-audios-mp3_rf_69773817_1.html
>
> [b] https://www.ivoox.com/cicd-cloud-native-llevando-devops-al-siguiente-audios-mp3_rf_70434837_1.html

En la Tabla 2.2 puedes ver un resumen de las diferencias entre entrega continua y despliegue continuo.

2.4. ¿Y cómo afecta la IA a todo esto?

Las herramientas de inteligencia artificial para programar pueden ayudarnos a escribir código, generar pruebas, explicar errores o proponer cambios en menos

Característica	Entrega continua	Despliegue continuo
Objetivo principal	Mantener el software en un estado liberable y preparado para avanzar hacia producción.	Llevar automáticamente a producción cada release que supera las validaciones establecidas.
Paso a producción	No es automático. Requiere de una acción expresa del equipo.	Es automático cuando la release supera todas las comprobaciones definidas por el equipo.
Decisión de liberación	La organización decide cuándo conviene publicar o desplegar la release.	La decisión se expresa previamente mediante políticas y comprobaciones automatizadas en el *pipeline*.
Automatización	Automatiza la construcción, identificación, almacenamiento y validación progresiva de la *release candidate*.	Automatiza el despliegue de la release en el entorno de producción.
Pruebas y controles	Valida la release en entornos progresivamente más parecidos a producción para determinar si está lista para liberarse.	Comprueba que el despliegue se completa correctamente y que el entorno queda en un estado saludable.
Resultado	Una release identificada, trazable y disponible para ser desplegada cuando se decida.	Una release desplegada automáticamente y en ejecución en producción.
Contextos habituales	Productos cuya publicación está sujeta a decisiones externas, tiendas de aplicaciones, ventanas de mantenimiento o procesos de validación controlados.	Servicios web, API y sistemas en la nube en los que el equipo controla el entorno de ejecución y puede automatizar las validaciones necesarias.

Tabla 2.2 Comparativa entre entrega continua y despliegue continuo.

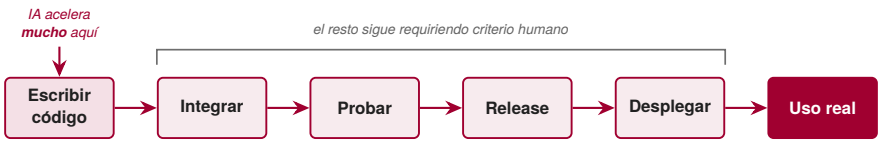


Fig. 2.9 La IA puede acelerar mucho la escritura de código, pero el resto del camino hasta el uso real (integrar, probar, entregar, desplegar) sigue dependiendo de criterio humano.

tiempo. Esto puede aumentar mucho la velocidad con la que una persona produce modificaciones en un repositorio.

Sin embargo, como hemos visto, escribir código no es lo mismo que entregar y desplegar software. Para que un cambio genere valor tiene que integrarse con el resto del sistema, construirse de forma reproducible, superar pruebas, empaquetarse como artefacto, convertirse en una release trazable, desplegarse en un entorno adecuado y, finalmente, llegar a usuarios o sistemas reales (figura 2.9).

Esta distinción empieza a aparecer también en estudios recientes sobre el impacto de la IA en el desarrollo de software. Por ejemplo, en el artículo *Writing code vs shipping code*[22], Demirer, Musolff y Yang analizan datos de más de 100.000 desarrolladores de GitHub y observan que distintas generaciones de herramientas de IA incrementan de forma notable la actividad de codificación, medida en commits, con aumentos de hasta un 180 %. Pero esos efectos se reducen drásticamente cuando se observan niveles más altos de la cadena de producción: proyectos, releases y uso real del software. Analizando los cuatro mercados principales de apps, detectan un incremento moderado en el número de apps, pero no detectan incremento alguno en el uso total.

El resultado es intuitivo si pensamos en el proceso que hemos estudiado en este capítulo. La IA puede acelerar una tarea concreta, como escribir una función o preparar una propuesta de cambio, pero el flujo completo sigue dependiendo de pruebas, revisiones, decisiones de producto, integración con otros componentes, validaciones externas, coordinación del equipo y despliegues seguros. Si esos pasos siguen requiriendo criterio humano, una mejora muy grande en la escritura de código puede traducirse solo parcialmente en más software entregado y utilizado.

Además, al aumentar el número de cambios que un equipo puede producir, también aumenta la importancia de contar con mecanismos fiables para validarlos. Una organización que genera código más rápido, pero no mejora sus procesos de construcción, pruebas, empaquetado, trazabilidad y despliegue, puede acabar acumulando más cambios pendientes, más ruido y más riesgo.

Por tanto, si bien la IA puede ayudarnos a escribir código más rápido, es evidente que seguimos necesitando prácticas de ingeniería que nos permitan convertir ese código en software entregable, desplegable y útil. En ese contexto, CI/CD sigue siendo una pieza fundamental para transformar productividad local en valor real.

2.5. Lecturas y recursos recomendados

- Capítulo 10, *DevOps*, del libro *Software Engineering: A Modern Approach* (Marco Tulio Valente).
- Capítulos 5, *Managing Dependencies*, y 8, *Delivering Software*, del libro *The Missing README: A Guide for the New Software Engineer* (Chris Riccomini y Dmitriy Ryaboy).
- Sesiones *Packaging and Shipping Code* y *Code Quality* del curso *The Missing Semester of Your CS Education* (MIT).

[22] https://papers.ssrn.com/sol3/papers.cfm?abstract_id=6859839

- Artículo *Continuous Integration* (Martin Fowler).
- Artículo *My CI/CD pipeline is my release captain* (Clare Liguori, Amazon Builders' Library).
- Libro *Continuous Integration: Improving Software Quality and Reducing Risk* (Paul Duvall, Steve Matyas y Andrew Glover).
- Libro *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* (Jez Humble y David Farley).
- Libro *Continuous Delivery Pipelines: How to Build Better Software Faster* (David Farley).
- Libro *Configuration Management Best Practices: Practical Methods that Work in the Real World* (Bob Aiello y Leslie Sachs).

Capítulo 3

Gestión del código fuente

Si Git te resulta confuso, no te preocupes: no estás solo.
Julia Evans.

Resumen En este capítulo estudiamos cómo organizar y gestionar el código fuente de un proyecto software, y veremos por qué los sistemas de control de versiones son una pieza central de la gestión de la configuración. Nos centraremos especialmente en Git, que es la referencia en la industria. Primero veremos cómo trabaja una persona con un repositorio propio; después, cómo cambia el escenario cuando trabajamos en equipo usando servicios como GitHub, GitLab o Bitbucket. Finalmente discutiremos distintas estrategias de uso del repositorio, reflexionando sobre su relación con la integración, entrega y despliegue continuos que introdujimos en el capítulo anterior.

3.1. Introducción: por qué el código necesita historia

Es probable que durante los primeros cursos del grado hayas realizado copias de seguridad de tu propio proyecto de una forma bastante artesanal. Quizá has tenido una carpeta llamada `practica-final`, otra llamada `practica-final-buena` y tal vez otra llamada `practica-final-ahora-sí-v2`. Si además estabas trabajando en grupo, puede que cada persona tuviera su propia copia, y hayáis compartido código por mensajería o mediante alguna carpeta en red (figura 3.1).

Este tipo de situaciones pueden parecer inofensivas cuando el proyecto es pequeño. Pero incluso en trabajos de clase aparecen pronto algunos problemas. ¿Cuál era la versión que funcionaba ayer? ¿Quién cambió este fichero? ¿Por qué se modificó esta función? ¿Qué parte del código tengo que conservar si dos personas han tocado la misma línea? ¿Qué versión exacta se entregó finalmente?

En el Capítulo 2 vimos que, para construir, probar, empaquetar, entregar y desplegar software de manera fiable, necesitamos procesos reproducibles y trazables.

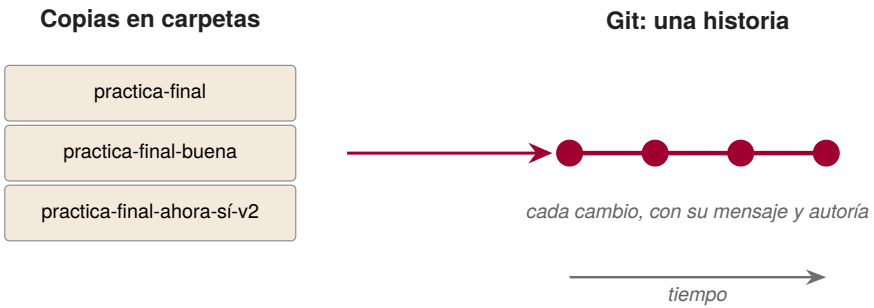


Fig. 3.1 Del caos de carpetas con copias (¿cuál era la buena?) a una única historia ordenada en Git, donde cada cambio queda registrado con su mensaje y su autoría.

Y esa trazabilidad comienza en el código fuente. Por eso los repositorios de código son una pieza central de la gestión de la configuración, ya que, además de almacenar el código, conservan la historia del proyecto: los cambios realizados, las versiones relevantes, las líneas de trabajo paralelas, las decisiones que se tomaron y la relación entre el código, las incidencias, las pruebas y los artefactos que se generan más adelante.

En este capítulo vamos a estudiar cómo funcionan estos sistemas centrándonos fundamentalmente en Git, porque es la herramienta más extendida en la industria. Pero en lugar de explicar órdenes y parámetros, que muchas veces es lo que se encuentra en tutoriales, vamos a intentar entender realmente los conceptos a alto nivel. Así, durante las clases de prácticas, te resultará más fácil coger soltura con la terminal, porque entenderás qué estás haciendo en cada momento.

Una breve progresión histórica

Antes de centrarnos en Git, merece la pena mirar un momento hacia atrás. La forma más simple de conservar versiones de un proyecto es hacer copias manuales. Cada cierto tiempo duplicas una carpeta y le pones un nombre diferente. Es una estrategia muy fácil de entender, pero también muy propensa a errores, ya que puedes editar la carpeta equivocada, borrar la copia buena o no recordar qué diferencia había entre dos versiones.

Para mejorar esta situación aparecieron los sistemas de control de versiones locales. La idea era mantener una base de datos en la propia máquina con los cambios realizados a los ficheros. Esto permitía recuperar versiones anteriores sin depender

de una colección desordenada de carpetas. El problema es que seguía siendo una solución pensada principalmente para una persona.

Pero como los equipos necesitan colaborar de forma más organizada, con el tiempo se popularizaron los sistemas centralizados, como *CVS* o *Subversion* (aún presentes en entornos profesionales). En este modelo hay un servidor central que contiene la historia del proyecto, y las personas del equipo obtienen desde ahí una copia de trabajo. El modelo es sencillo de administrar y permite controlar quién puede hacer qué, pero tiene una limitación importante, y es que el servidor central se convierte en un punto crítico. Si no puedes conectarte al servidor, muchas operaciones dejan de estar disponibles. Y si el servidor pierde la historia y no hay copias adecuadas, el proyecto queda en una situación delicada. Este modelo es el que siguen lo que se conocen como *sistemas de control de versiones centralizados*.

Los *sistemas de control de versiones distribuidos*, como Git o Mercurial, cambian esta idea. En lugar de tener únicamente una copia completa de la historia en un servidor central, cada persona puede tener en su máquina una copia completa del repositorio, con toda su historia. Esto permite trabajar localmente, crear *commits* sin conexión, consultar la historia del proyecto de forma rápida y sincronizar cambios con otros repositorios cuando sea necesario. A diferencia de los sistemas centralizados, en un sistema distribuido no es necesario tener un repositorio central. Sin embargo, es muy común usar aproximaciones que usando sistemas de control de versiones distribuidos usan un repositorio central por convención y de esta forma sacan el mayor partido de ambos mundos.

> **Para saber más**
>
> Ya que estamos hablando de historia, es interesante conocer cómo y por qué nace Git, por lo que te animamos a echar un ojo a este breve artículo de la web oficial del proyecto[a]. Y así quizá entiendas mejor cómo es posible que su versión inicial fuera creada en tan solo unos pocos días.
>
> [a] https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Una-breve-historia-de-Git

3.2. Git como máquina del tiempo del proyecto

Una forma útil de empezar a entender Git es imaginarlo como una máquina del tiempo. En distintos momentos del proyecto podemos guardar una fotografía del estado de los ficheros. Más adelante podremos consultar esa fotografía, compararla

con otra, saber quién la creó o recuperar parte de su contenido. Podríamos decir que el sistema de control de versiones es como el famoso Delorean de la película *Regreso al futuro*. Con una máquina del tiempo podemos ir adelante y atrás en el tiempo e incluso crear líneas temporales paralelas que evolucionen de manera independiente. Podremos incluso mezclar luego historias o seleccionar lo que nos gusta más de una parte de una historia para combinarla a nuestro gusto.

En Git, para construir esta máquina del tiempo, cada una de esas fotografías o instantáneas se llama *commit*. Un commit representa un estado concreto del proyecto, acompañado de información adicional, como quién lo creó, cuándo, qué mensaje dejó, a qué ficheros afectaron los cambios, o desde qué commit o commits anteriores procede.

Por tanto, Git almacena la historia de un proyecto como una secuencia de instantáneas o estados del proyecto, con las relaciones entre ellos. Cuando guardas un nuevo estado, Git registra una nueva instantánea y la conecta con la anterior. Incluso, como veremos más adelante, es posible que en algún momento dos líneas de trabajo se separen y luego se vuelvan a unir, de forma que la historia deja de ser una línea simple y se convierte en un grafo de commits.

Para conocer la historia de un proyecto necesitamos almacenar información sobre el significado de cada cambio. Es decir, saber por qué hemos pasado de una instantánea a otra. Por eso los mensajes que acompañan cada instantánea, que se llaman mensajes de commit, son importantes. Supongamos que quieres entender por qué alguien ha realizado un cambio y el mensaje es algo como "cambios varios". Algo así apenas ayuda, ¿verdad? En cambio, un mensaje que explica que se ha corregido la validación de fechas en el formulario de registro permite entender mejor la evolución del sistema (figura 3.2). Y aunque pueda parecer algo evidente, es habitual encontrar en la industria equipos que no siguen buenas prácticas al respecto. De hecho, en un estudio sobre 1600 mensajes de commit de cinco proyectos libres muy activos, alrededor del 44 % de los mensajes analizados necesitaban mejorar[1].

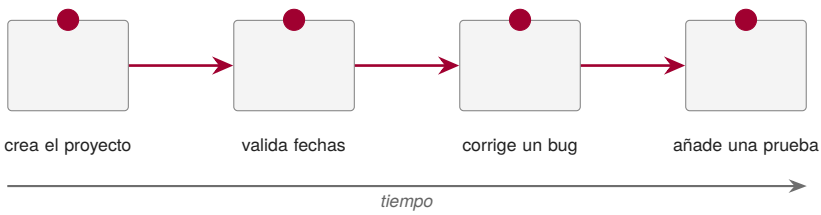


Fig. 3.2 Git guarda la historia como una secuencia de instantáneas (*commits*). Cada commit es una fotografía del proyecto acompañada de un mensaje que explica el porqué del cambio.

[1] https://dl.acm.org/doi/10.1145/3510003.3510205

Para saber más

Por ejemplo, el proyecto Django, uno de los frameworks web libres más conocidos del ecosistema Python, mantiene una guía para las personas que contribuyen al proyecto en la que explica cómo deben prepararse los commits[a]. Lee el apartado en el que se detalla cómo deberían escribirse los mensajes de commit y reflexiona sobre las recomendaciones.

[a] https://docs.djangoproject.com/en/dev/internals/contributing/committing-code/#committing-guidelines

Para saber más

Pero estamos hablando aquí de commits y de historia de un proyecto como algo muy abstracto, ¿no? Existe un videojuego muy interesante, *Oh My Git!*[a], que permite visualizar la estructura interna de los repositorios Git, y ver claramente el resultado de nuestras acciones en tiempo real. El juego permite realizar las acciones usando cartas, que representan órdenes git, o escribiendo las propias órdenes en un terminal, como prefieras. Instala *Oh my git* en tu equipo y trata de resolver los siguientes niveles: *Living dangerously, Making backups, Enter the time machine, The command line* y *Your first commit* de la sección *Intro*.

[a] https://ohmygit.org/

Además, cada commit queda asociado a una identidad. Por eso es importante configurar correctamente el nombre y el correo de la persona que contribuye al repositorio, y usar una identidad consistente. Si una misma persona aparece con tres nombres distintos, reconstruir la autoría del proyecto se vuelve más difícil[2].

Así, con los datos y metadatos que se almacenan en el repositorio, los equipos pueden revisar contribuciones, auditar cambios, relacionar commits con incidencias o saber qué código se incluyó en una release concreta, por ejemplo.

Pero antes de ver todo lo que Git nos ofrece al colaborar en equipo, quizá resulte de ayuda comenzar por las cosas que puede hacer una persona cuando trabaja individualmente con un repositorio para un proyecto. Vamos a verlo con un ejemplo sencillo.

[2] Cada curso esta situación le ocurre a más de un estudiante de la asignatura, que realiza contribuciones al proyecto con distintas identidades. Esto dificulta valorar correctamente su trabajo. ¡Esperemos que este año no seas tú a quien le pase!

3.3. Trabajar en solitario: conceptos y prácticas fundamentales de Git

Imagina que vas a comenzar a trabajar en una práctica nueva. Y lo vas a hacer usando Git para ayudarte a mantener la historia del proyecto.

Como primer paso inicias el repositorio en el directorio del proyecto[3]. Luego creas un archivo. A continuación escribes el código con algo de funcionalidad. Escribes una prueba y pasa correctamente. ¡Esto marcha! Quizá ese podría ser un buen momento para tomar una instantánea, ¿no?.

Hasta ese momento has estado trabajando en lo que Git llama tu espacio de trabajo, o *Working directory*, que es la carpeta donde editas los ficheros, ejecutas pruebas, creas archivos o borras cosas. En ocasiones podrás leer o escuchar este concepto como *Sandbox*, *Work space*, *Playground* o *Working tree*.

Para poder guardar en el repositorio el estado del nuevo archivo en el que has estado trabajando, primero hay que añadirlo (con la orden `git add`) a lo que se conoce como *staging area*, zona de preparación, índice o algunas veces *cache*. Esta zona sirve para decidir qué cambios concretos quieres incluir en el próximo commit. Puede parecer una complicación al principio, pero es una de las ideas que ayudan a construir una historia limpia, como veremos un poco más adelante. Y hay que tener en cuenta que, la primera vez que se añade un archivo al índice, Git empieza a considerarlo como parte del proyecto, es decir, pasa a estar bajo seguimiento o *tracked*. A partir de ahí, la herramienta monitorizará el archivo y podrá detectar modificaciones futuras.

Una vez que has añadido el archivo al índice, entonces ya puedes crear un commit (con la orden `git commit`) para hacer una instantánea y guardar el estado del proyecto en ese momento, lo que se acompañará del mensaje de commit correspondiente y otros metadatos, como hemos comentado previamente. Básicamente, al confirmar un commit, Git toma lo que habías preparado en el índice y lo incorpora a la historia del proyecto como una nueva instantánea (figura 3.3).

[3] Este es un paso que a veces la gente pierde de vista: la materialización de todo lo que se puede y se hace con Git requiere de dos elementos fundamentales: *i*) tener instalada la herramienta Git en el entorno y *ii*) crear una máquina del tiempo, es decir, un repositorio de git. Un repositorio de git se representa en un conjunto de archivos ocultos que están dentro de la carpeta de la que quieres hacer un seguimiento. Con esas dos cosas hechas, ya puedes empezar a trabajar con Git

Fig. 3.3 Las tres zonas de Git al trabajar en local: editas en el espacio de trabajo, preparas cambios concretos en el *staging area* con `git add` y los confirmas en el repositorio con `git commit`.

> **Para saber más**
>
> Pon en práctica esta idea con *Oh My Git!* en los niveles *Add new files to the index* y *Update files in the index* de la sección *Index*.

Sencillo, ¿verdad? Venga, pues vamos a continuar avanzando con esa práctica que tienes que terminar.

3.3.1. Commits atómicos

Como la cosa ha ido bien, resulta que te has venido un poco arriba y has estado trabajando ya con varios ficheros y varias funcionalidades de la práctica. Todo parece funcionar bien. ¡Pero no has tomado ninguna otra instantánea en todo este rato! ¿Hacemos ahora un commit con todos los cambios?

Podría ser una opción, pero lo ideal es trabajar con lo que se conoce como commits atómicos. Un *commit atómico* es aquel que incluye un solo cambio lógico y completo. No significa que un commit tenga que modificar un único fichero, ni que tenga que ser pequeño a toda costa. Significa que agrupa un cambio coherente. Si mañana ese commit hubiera que revisarlo, revertirlo o explicarlo, debería poder entenderse como una unidad.

Un mensaje que capture la esencia de lo que es un commit atómico podría ser "añadir validación de correo electrónico en el registro". Ese cambio puede afectar al formulario, la lógica de validación y una prueba. Aunque afecte a tres ficheros, conceptualmente es una sola unidad. En cambio, un commit que mezclara esa validación con una reorganización visual de la página de inicio y una actualización de dependencias sería más difícil de revisar y de deshacer.

Así que, en una situación así, lo que tendrías que hacer es añadir selectivamente al índice los cambios que quieres que formen el próximo commit. Y una vez listo, confirmar el commit para añadir la instantánea al repositorio, junto a su mensaje y

metadatos. Y así para cada uno de los commits atómicos que quieres que formen parte de la historia del proyecto.

> **Para saber más**
>
> Pon en práctica esta idea con *Oh My Git!* en el nivel *Adding changes step by step* de la sección *Index*.

3.3.2. Qué debe vigilarse en la máquina del tiempo

Otra decisión importante es qué ficheros queremos versionar. Aunque hablemos de repositorios de código fuente, en realidad un repositorio puede contener más cosas que código: las propias pruebas, documentación, ficheros de configuración del proyecto, scripts de construcción, plantillas, ejemplos de datos o instrucciones para preparar el entorno, entre otras.

En general, todo lo que sea necesario para construir, probar, mantener o entender el proyecto debería estar versionado. Pero, ojo, que no todo debe entrar en el repositorio. Los binarios generados, los archivos temporales, los logs, las carpetas de dependencias descargadas automáticamente o los resultados de compilación suelen quedar fuera. También es importante dejar fuera los secretos (contraseñas, tokens, claves privadas o credenciales de acceso). No sería la primera vez que alguien se deja un secreto del acceso a algún recurso en un repositorio, provocando efectos no deseados y disgustos.

Para indicar a Git qué ficheros o patrones queremos ignorar se usa normalmente un fichero `.gitignore`. Este fichero también debe estar versionado si las reglas afectan al proyecto. Así, todo el equipo comparte el mismo criterio sobre qué cosas no deben entrar en la historia.

> **Para saber más**
>
> Consulta esta plantilla de `.gitignore` para un proyecto Java[a] y esta otra para un proyecto Python[b]. Reflexiona sobre los patrones propuestos (aunque quizá haya cuestiones que no entiendas todavía).
>
> [a] https://github.com/github/gitignore/blob/main/Java.gitignore
> [b] https://github.com/github/gitignore/blob/main/Python.gitignore

3.3.3. Volver atrás no significa siempre lo mismo

Sigamos con esa práctica que tienes que entregar. Si recuerdas, habías creado tu primer archivo, escrito algo de código, preparado los cambios en el índice y realizado un commit. Después habías seguido trabajando y añadido varias nuevas funcionalidades, creando commits atómicos. Todo parece ir bien.

Ahora pensemos que, justo después de modificar varios ficheros en tu espacio de trabajo, ejecutas las pruebas y empiezan a fallar. Miras el código y te das cuenta de que el enfoque seguido no tiene mucho sentido.

En esta situación todavía no habías añadido los cambios al índice ni habías hecho commit. En un caso así, volver atrás significa simplemente descartar o deshacer cambios de tu espacio de trabajo. Es decir, tan solo tenemos que volver al estado que tenía el proyecto en el último commit.

> **Para saber más**
>
> Pon en práctica esta idea con *Oh My Git!*. Para ello puedes abrir el nivel *Sandbox with three commits* de la sección *Sandbox*. Luego puedes editar el archivo *you*, por ejemplo añadiendo la línea *Abro la puerta* al final del archivo. Y para eliminar ese cambio y devolver el archivo al estado en el que estaba en el último commit, tendrás que investigar qué carta (o qué orden) utilizar.

Ahora pensemos en otra situación. Has modificado dos ficheros y has preparado ambos en el índice, porque pensabas que formarían parte del próximo commit. Pero al revisarlo con calma te das cuenta de que uno de esos ficheros no debería entrar todavía. Quizá pertenece a otro cambio distinto, o quizá contiene una modificación temporal que no quieres confirmar. En este caso, volver atrás no significa borrar el cambio, sino sacarlo de la zona de preparación. El fichero puede seguir modificado en tu espacio de trabajo, pero ya no formará parte del próximo commit.

Este matiz es importante. A veces no queremos deshacer el trabajo, sino reorganizarlo mejor. Como decíamos antes, la zona de preparación nos ayuda a decidir con cuidado qué va a formar parte de la siguiente instantánea. Por eso, cuando trabajamos con commits atómicos, es normal mover cambios dentro y fuera del índice antes de confirmar.

> **Para saber más**
>
> Pon en práctica esta idea con *Oh My Git!* en el nivel *Resetting files in the index* de la sección *Index*.

Incluso si ya hemos registrado un commit y nos damos cuenta de un problema justo después, Git ofrece opciones para subsanarlo, ya que es una situación muy habitual. Mediante la enmienda de commits (*amend*), podemos modificar el mensaje de la última confirmación o incorporar cambios que debieron ir en ella pero olvidamos preparar en el índice. Esto evita tener que crear un nuevo commit con mensajes poco profesionales como "se me olvidó añadir la prueba" que ensucian la historia. En realidad, como los commits en Git son inmutables, lo que ocurre entre bambalinas es que Git descarta el commit defectuoso y lo sustituye por uno nuevo que integra la corrección. Pero a efectos de la historia del proyecto, el resultado es una historia limpia y lineal.

Por tanto, a la hora de deshacer un cambio, tenemos que pensar si está solo en nuestro espacio de trabajo, si está ya en el índice, o si incluso se ha guardado en un commit, para decidir la estrategia adecuada.

3.3.4. Ramas para explorar y mezclar selectivamente

Hasta ahora hemos trabajado de forma que la historia de tu práctica avanza como una línea recta de commits: haces un cambio, lo preparas, lo confirmas; haces otro cambio, lo preparas, lo confirmas; y así sucesivamente. Para muchas situaciones, esta forma de trabajar es suficiente.

Pero pensemos ahora en otro escenario. La práctica funciona, las pruebas pasan y tienes varios commits que representan avances razonables. Sin embargo, se te ocurre una idea para mejorar una parte importante del diseño. Quizá quieres cambiar la forma en la que organizas los módulos, probar una biblioteca nueva o rehacer una funcionalidad que ya tienes implementada. No sabes si esa idea va a salir bien. Puede que mejore el proyecto, pero también puede que te lleve a un callejón sin salida.

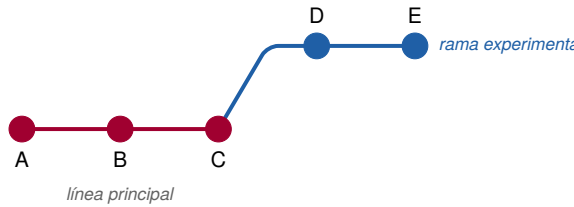
Podrías empezar a modificar directamente tu línea principal de trabajo y descartar los cambios si no funcionan bien, como hemos visto antes. Pero, para situaciones así, Git también nos ofrece una opción en la que, en lugar de deshacer después, podemos directamente separar antes de empezar a trabajar.

Para eso sirven las ramas. Una rama es una línea de trabajo independiente que parte de un punto concreto de la historia. Puedes pensar en una rama como un desvío

temporal en la máquina del tiempo del proyecto. Desde el commit en el que estás, creas una nueva línea y empiezas a avanzar por ella sin modificar la línea principal.

Por ejemplo, supongamos que tu práctica tiene una historia como esta, con tres commits:

El commit `C` representa un estado estable, en el que el proyecto funciona y las pruebas pasan. Ahora quieres probar una nueva forma de organizar la validación del registro. En lugar de hacerlo directamente sobre la línea principal, creas una rama para esa idea. Conceptualmente, la historia podría quedar así:



Los commits `D` y `E` pertenecen a la rama experimental. Ahí puedes probar, equivocarte, borrar, reorganizar y volver a probar sin ensuciar la línea principal. Si al final descubres que la idea no merece la pena, puedes abandonar esa rama y volver al estado estable que tenías en `C`. En ese caso, la rama habrá servido como un espacio seguro para experimentar.

Pero también puede pasar lo contrario. La idea funciona, las pruebas siguen pasando y el nuevo diseño mejora el código. Entonces tiene sentido incorporar esos cambios a la línea principal. A ese proceso lo llamamos integrar o mezclar una rama, o hacer *merge*. Y una vez integrados los cambios, ya podemos eliminar la rama experimental. En el caso más simple, si no has tocado la línea principal mientras tanto, la historia podría quedar así.

Si te fijas, es como si Git hubiese movido el puntero de la línea principal hacia delante, por eso a este tipo de merge se le llama *fast forward*.

Para saber más

Pon en práctica esta idea con *Oh My Git!*. Para ello puedes abrir el nivel *Sandbox with three commits* de la sección *Sandbox*. Investiga cómo crear una nueva rama y crea la rama *me_la_juego*. Luego puedes editar el archivo *you*, por ejemplo añadiendo la línea *Abro la puerta* al final del archivo. Realiza el commit. Ahora añade la línea *Uff, solo era un gato* al final del archivo, y realiza otro commit. Y ahora tendrás que investigar cómo hacer el *merge* de la rama *me_la_juego* en la principal. Finalmente puedes eliminar la rama *me_la_juego*.

No obstante, es importante no idealizar las ramas, ni pensar que no hay que gestionarlas. Si creas una rama para cada pequeño cambio y luego esas ramas quedan olvidadas, el repositorio puede volverse confuso. Una rama debería tener un propósito: probar una idea, desarrollar una funcionalidad, corregir un problema o preparar una reorganización concreta. Y cuando ese propósito termina, hay que gestionarla, ya sea integrándola si aporta valor, o eliminándola cuando ya no hace falta.

Básicamente, una rama te permite avanzar por una línea de trabajo separada sin comprometer todavía la historia principal del proyecto. En proyectos personales, las ramas pueden ayudarte a trabajar con más tranquilidad. En proyectos en equipo, además, se convierten en una herramienta de coordinación.

De hecho, en los proyectos profesionales reales pocas veces trabajaremos de manera individual. Así que ha llegado el momento de explorar las cosas que nos ofrece Git cuando varias personas necesitan trabajar sobre el mismo proyecto. Para eso necesitamos introducir dos nuevas ideas: los repositorios remotos y los servicios de alojamiento.

3.4. Trabajar con otras personas: repositorios remotos y servicios de alojamiento

Hasta ahora hemos trabajado de forma individual. Tienes tu propio repositorio en tu equipo, vas creando commits, decides qué ficheros entran en la historia, corriges errores y usas ramas para probar ideas sin mezclarlo todo.

Pero lo normal en un proyecto software es que no trabajes en soledad. Supongamos ahora que tienes que hacer una práctica con otras personas. Una compañera implementa la autenticación, tú estás trabajando en la validación de formularios y

otra persona está preparando la configuración del entorno. Todas necesitáis ver el trabajo de las demás, incorporar cambios y evitar que cada ordenador acabe teniendo una versión distinta del proyecto.

Para resolver esta situación Git permite trabajar con repositorios remotos. Un repositorio remoto es una copia del repositorio que está disponible en otro lugar, normalmente en un servidor. Así, cada persona puede trabajar en su copia del repositorio en local, crear commits y consultar la historia, y después sincronizar cambios con el repositorio remoto cuando sea necesario. Y, además, como cada persona del equipo tiene una copia completa de la historia del proyecto, el servidor remoto no es el único lugar donde existe la historia.

3.4.1. Git no es GitHub (ni viceversa)

Aquí conviene aclarar una confusión muy habitual. Git y GitHub no son lo mismo. Ni las operaciones que se hacen en uno y en otro son iguales. Habrá operaciones que son de Git y otras que son de Github, como veremos a continuación, y es importante no confundirse.

Git es la herramienta de control de versiones. Es decir, el sistema que permite crear commits, consultar la historia, trabajar con ramas, comparar cambios o sincronizar repositorios. Mientras que, por ejemplo, GitHub[4], GitLab[5], Bitbucket[6], Codeberg[7] o Gitea[8] son servicios de alojamiento de repositorios Git. Permiten guardar repositorios en la web y añaden muchas funciones alrededor, como gestión de usuarios, permisos, incidencias, revisión de código, integración con sistemas de CI/CD, wikis, tableros de proyecto y otras herramientas de colaboración.

Dicho de otra forma, puedes usar Git sin usar GitHub. Y también puedes usar Git con GitLab, Codeberg, un servidor propio o cualquier otro servicio compatible. Git es la tecnología de control de versiones. GitHub y soluciones similares son plataformas que facilitan la colaboración alrededor de repositorios Git.

Esta distinción es importante porque, cuando trabajamos en equipo, muchas acciones ocurren en Git, pero otras ocurren en la plataforma. Por ejemplo, crear un commit es una operación de Git, mientras que abrir una propuesta de cambio para que otra persona la revise es una funcionalidad que ofrecen servicios como

[4] https://github.com/
[5] https://gitlab.com/
[6] https://bitbucket.org/
[7] https://codeberg.org/
[8] https://about.gitea.com/

GitHub o GitLab. Las dos cosas se complementan, pero es importante tener clara la distinción.

3.4.2. Sincronizar trabajo

Volvamos a la práctica en grupo. Supongamos que el equipo decide crear un repositorio remoto en GitLab. Una persona crea el repositorio inicial y todas obtienen una copia en su máquina. A partir de ese momento, cada integrante tiene un repositorio local con la historia del proyecto, y el equipo tiene además un repositorio remoto que actúa como referencia común.

Insistimos en que al traer el repositorio por primera vez a tu máquina, no estás descargando simplemente una carpeta con ficheros. Lo que estás obteniendo es una copia del repositorio completo, con su historia. Por eso puedes consultar commits anteriores, crear nuevas ramas o trabajar sin conexión.

Pero aunque puedas trabajar en local, en en algún momento querrás enviar tus cambios al repositorio remoto (hacer *push*) para que el resto del equipo pueda verlos e incorporarlos. En ese momento tus commits dejan de estar solamente en tu máquina y pasan a estar disponibles para las demás personas.

Y, del mismo modo, tú también tendrás que incorporar cambios que otras personas han publicado. Si una compañera ha enviado nuevos commits al remoto, tu copia local puede quedarse desactualizada. Para seguir trabajando con una versión reciente del proyecto, tendrás que traer esos cambios (hacer *pull* o *fetch*) e incorporarlos a tu repositorio local.

La idea principal es que en Git trabajas localmente y sincronizas con otros repositorios cuando lo necesitas. Por tanto, hacer un commit no significa automáticamente que el resto del equipo lo vea. Un commit creado en tu repositorio local seguirá estando en tu máquina hasta que lo envías a un remoto. Y, del mismo modo, que alguien haya publicado cambios en el remoto no significa que estén ya en tu copia local. Para tenerlos en tu copia local, tienes que traerlos expresamente (figura 3.4).

> **Para saber más**
>
> Pon en práctica estas ideas con *Oh My Git!* en el nivel *Friend* de la sección *Remotes*. Este nivel parte de una situación en la que un amigo y tú estáis trabajando conjuntamente en un proyecto, y cada cual tenéis una copia local de un repositorio remoto.

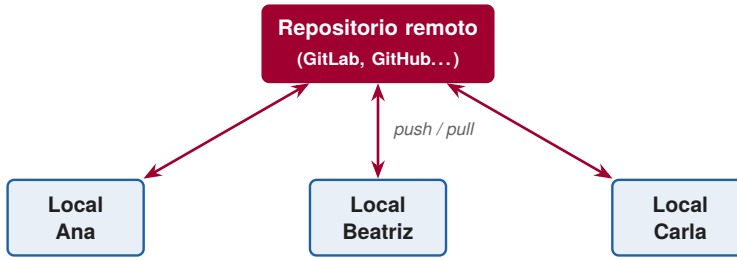


Fig. 3.4 Cada persona trabaja en su repositorio local (con su copia completa de la historia) y sincroniza con un repositorio remoto común mediante *push* (enviar) y *pull*/*fetch* (traer).

3.4.3. Cuando los cambios se pisan y aparecen conflictos

Trabajar con un remoto común no elimina todos los posibles problemas de programar colaborativamente, por supuesto, pero los hace más visibles y gestionables.

Imagina que tú modificas el mensaje de error que aparece cuando una fecha no es válida. Al mismo tiempo, otra persona modifica ese mismo mensaje para mejorar la traducción. Las dos personas trabajáis localmente, las dos creáis commits, y en algún momento queréis incorporar los cambios al repositorio común. En ese momento Git puede detectar que hay dos modificaciones sobre la misma zona del fichero y pedir ayuda para decidir qué versión debe quedar.

A esto lo llamamos conflicto. Un conflicto no significa necesariamente que alguien haya hecho algo mal. Simplemente indica que dos líneas de trabajo han tocado una misma parte del proyecto y Git no puede decidir automáticamente cuál es la solución correcta.

Este tipo de conflictos sintácticos son mecánicos y relativamente fáciles de solucionar. Git lanza el aviso, marca la zona problemática y pide que una persona la revise.

> **Para saber más**
>
> Pon en práctica esta idea con *Oh My Git!* en el nivel *Problems* de la sección *Remotes*.

Pero también existen conflictos más difíciles de detectar y solucionar. Por ejemplo, tú podrías cambiar el nombre de una función en tu repositorio local, mientras otra persona escribe código nuevo en su repositorio usando todavía el nombre antiguo de la función. Cuando intentéis combinar ambos cambios, quizá Git puede mezclar los ficheros sin que aparezca un conflicto textual, pero el programa deja de funcionar.

Este es un ejemplo de conflicto semántico, en el que los cambios se pueden combinar como texto sin problema, pero generan comportamientos incorrectos del sistema.

Por tanto, resolver conflictos exige entender qué estaba intentando hacer cada cambio. La automatización puede ayudar mucho aquí, evidentemente, ya que un sistema de integración continua puede ejecutar pruebas cada vez que se propone incorporar un cambio. Pero, como vimos en el Capítulo 2, la automatización no sustituye el criterio humano. Si dos cambios son incompatibles desde el punto de vista del diseño, alguien tendrá que entender el problema y tomar una decisión. Y por eso las revisiones de código resultan de gran ayuda en estos casos, como también se discutió en el capítulo anterior.

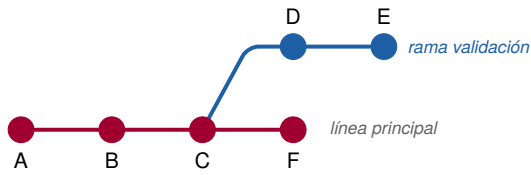
Por todo ello, los sistemas como Github o Gitlab incorporan o se conectan con herramientas de CI/CD que permiten integrar y gestionar tanto pruebas automáticas como revisiones de código de manera sencilla, generando informes que informan al equipo y le ayudan a tomar decisiones. Y también permiten incorporar otro tipo de comprobaciones que contribuyen a mantener sano el repositorio. Por ejemplo, permiten establecer reglas para que los commits tengan mensajes con un formato determinado o que cada cambio esté asociado a una incidencia concreta, como veremos en las clase de prácticas y detallaremos en el Capítulo 5.

3.4.4. Integrar código desde ramas

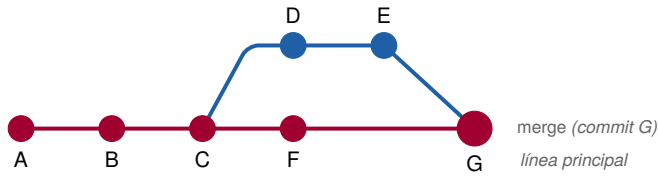
Ya hemos visto que cuando una rama contiene cambios que queremos incorporar a otra, necesitamos integrarla. Y hemos visto cómo lograrlo haciendo *merge*.

Cuando trabajamos individualmente puede ser que la línea principal no haya avanzado desde que se creó la rama. Y en ese caso Git puede incorporar los commits de la rama como si hubieran ocurrido directamente después del último commit de la línea principal, de manera que la historia queda lineal.

Pero en proyectos reales, trabajando en equipo, es muy habitual que la línea principal sí haya avanzado mientras tú trabajabas en tu rama. Por ejemplo, tú estabas implementando la validación de fechas, mientras otra persona integraba una mejora en la página de inicio. En ese caso, al integrar tu rama, Git tiene que combinar dos líneas de historia que han evolucionado desde un punto común. Así, podríamos tener una situación como la siguiente:



Si ahora hicieras un merge de la rama de validación en la línea principal, la historia quedaría así (aunque si aparecen conflictos, primero habría que resolverlos y después completar el merge):



En este caso vemos que Git ha creado automáticamente un nuevo commit, G, que se crea comparando tres instantáneas: C (que es el ancestro común), F (que es el último commit de la línea principal) y E (que es el último commit de la rama). Por esto este tipo de merge es conocido como *three-way merge*. Al commit G se le llama commit de *merge* y es un commit especial, ya que tiene más de un commit padre. Y eso hace que se refleje claramente de dónde proviene ese commit en la historia del proyecto.

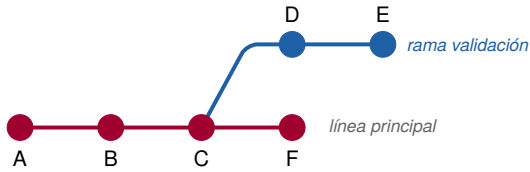
> **Para saber más**
>
> Pon en práctica esta idea con *Oh My Git!*. Para ello puedes abrir el nivel *Sandbox with three commits* de la sección *Sandbox*. Realiza los pasos igual que en el reto de la Sección 3.3.4, pero en esta ocasión crea también un commit nuevo en la línea principal que modifica el tipo de bebida de la línea 2 del archivo *you*, por ejemplo cambiando café por té. Luego haz *merge* de la rama y comprueba la diferencia en el resultado.

El commit de merge creado no introduce una funcionalidad nueva por sí mismo, pero existe para dejar constancia de que se han combinado dos historias. Este enfoque, por tanto, conserva muy bien lo que ocurrió. Se ve que hubo una rama, que la línea principal avanzó por su cuenta y que después ambas líneas se integraron. Pero ten en cuenta que cuando tenemos muchas ramas pequeñas, la historia puede volverse más difícil de leer.

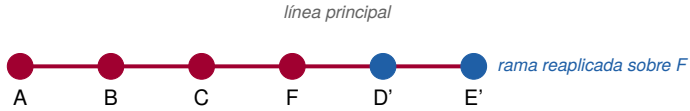
3.4.5. Recolocar cambios: rebase

Existe otra forma de preparar una rama antes de integrarla: hacer *rebase*, que muchas veces se define como *reescribir la historia*.

Volvamos de nuevo al escenario en el que habías creado una rama para trabajar en la validación de fechas. Mientras tanto, la línea principal ha avanzado porque otra persona ha integrado otros cambios. La historia estaba así:



Tu rama empezó en C, pero ahora la línea principal ya va por F. Hacer rebase consiste en tomar tus commits D y E y volver a aplicarlos como si hubieran empezado desde F. Conceptualmente, la historia pasaría a verse así:



Usamos `D'` y `E'` porque no son exactamente los mismos commits. Representan los mismos cambios, pero aplicados sobre una nueva base de la historia. Por eso se llama *rebase*, porque se cambia la base desde la que parte la rama.

Ahora simplemente habría que hacer merge (fast-forward) de la línea principal y la historia quedaría así (tras eliminar la rama de validación):

> **Para saber más**
>
> Pon en práctica esta idea con *Oh My Git!*. Para ello puedes prepararte un escenario similar al discutido usando el nivel *Sandbox with three commits* de la sección *Sandbox*. O también puedes practicar con el nivel *Rebasing* de la sección *Changing the past*.

La ventaja es que la historia queda más lineal. Parece que primero se incorporaron los cambios de la línea principal y después se hicieron los cambios de validación.

Esto puede facilitar la lectura de la historia y evitar algunos commits de merge. Pero reescribir historia exige tener cuidado.

En un trabajo individual, o en una rama que todavía no depende de nadie más, usar rebase puede ser una forma razonable de ordenar tus commits antes de integrarlos. En cambio, si otras personas ya están trabajando sobre esa misma rama, reescribirla puede causar confusión. Es como cambiar las páginas de un cuaderno que otra persona ya había fotocopiado para seguir trabajando.

Por eso una regla práctica habitual es que puedes reordenar y limpiar historia que todavía es privada, pero deberías evitar reescribir historia que ya sirve de base para otras personas. Como afirman Scott Chacon y Ben Straub en el libro *Pro Git*[9], *si sigues esa pauta, todo irá bien. Si no, la gente te odiará y tus amigos y familiares te despreciarán*. Así que ya ves que no es algo que tomar a la ligera.

La diferencia conceptual entre merge y rebase puede resumirse así. Merge une dos líneas de historia y deja constancia de esa unión. Rebase recoloca una línea de trabajo sobre una base más reciente para producir una historia más lineal. Merge conserva mejor la forma en que ocurrió el trabajo. Rebase puede hacer que la historia final sea más limpia, pero modifica la forma en que se presenta.

3.4.6. Otras opciones útiles de Git al trabajar en equipo

Git tiene muchas operaciones que aparecen con frecuencia en el trabajo diario. Las siguientes son especialmente comunes al trabajar en equipo.

Por un lado, las etiquetas, o *tags*, sirven para marcar un punto concreto de la historia con un nombre reconocible. Por ejemplo, cuando un equipo publica la versión `1.0.0` de una aplicación, puede crear una etiqueta sobre el commit exacto desde el que se generó esa release. Como vimos en el capítulo anterior, esta idea es muy importante para la trazabilidad, porque si sabemos qué etiqueta corresponde a una release, podemos relacionar código fuente, artefacto, pruebas y despliegue.

Otra operación que puede aparecer es *cherry-pick*, que consiste en tomar un commit concreto de una parte de la historia y aplicarlo en otra. Por ejemplo, podrías corregir un error como parte del trabajo de una rama, y quieres aplicar la corrección en otra rama distinta. Pero en lugar de mezclar toda la rama, puedes querer traer solo el commit concreto que arregla el error. Es una operación muy interesante, pero si se abusa de ella la historia puede volverse más difícil de seguir.

Por último, *stashing* permite guardar temporalmente cambios que tienes en tu espacio de trabajo sin convertirlos todavía en un commit. Por ejemplo, imagina que

[9] https://git-scm.com/book/es/v2

estás a mitad de una modificación de una nueva funcionalidad. Todavía no está en un estado que quieras guardar en la historia, y justo te avisan que tienes que ponerte urgentemente a arreglar un error grave que se acaba de detectar, por lo que tienes que cambiar de rama. Con *stash* puedes apartar esos cambios, dejar el espacio de trabajo limpio y recuperarlos más tarde.

3.5. Modelos de uso del repositorio

Como vimos en el Capítulo 2, los paquetes de las entregas se construyen a partir de un estado concreto y trazable del código fuente, normalmente identificado mediante una rama, una etiqueta o un commit concreto del sistema de control de versiones. Y como hemos visto en este capítulo, es habitual usar ramas para permitir que los equipos de desarrollo puedan trabajar en paralelo, mezclando sus cambios en la línea principal cuando están listos para ser integrados.

Pero existen múltiples estrategias de gestión de ramas que establecen cuánto tiempo deberían durar las ramas, cuál es la línea principal, cuándo se crea una release, cómo se relacionan las ramas con las versiones liberadas del software o cómo se propagan los cambios a otras ramas, entre otras cuestiones.

Y no existe un modelo perfecto para todos los proyectos. Un equipo que desarrolla una biblioteca con varias versiones mantenidas en paralelo puede necesitar una estrategia distinta a la de un equipo que despliega una aplicación web varias veces al día. Lo importante es que el modelo sea consciente, esté documentado y encaje con la forma en que el equipo construye, prueba, entrega y despliega el software.

3.5.1. Trunk-based development

Una estrategia de ramas que está muy relacionada con la integración y la entrega continua es el desarrollo basado en la línea principal, o *trunk-based development*[10].

En este modelo existe una línea principal, normalmente llamada *trunk* o `main`, y el equipo integra cambios en ella con mucha frecuencia. Puede haber ramas, pero suelen ser ramas muy cortas, de vida muy breve (en general, unas horas o unos pocos días), para desarrollar una actualización muy concreta. La idea es evitar que el trabajo permanezca aislado durante mucho tiempo.

[10] https://trunkbaseddevelopment.com/

Como decíamos, este enfoque encaja especialmente bien con CI/CD. Si el equipo integra cambios pequeños y frecuentes en la línea principal, los cambios se propagan rápidamente a todo el equipo, haciendo poco probable que se separen demasiado, y el pipeline puede construir, probar y dar feedback rápidamente. Si algo se rompe, el cambio responsable suele ser reciente y fácil de localizar.

Para que este enfoque funcione hacen falta buenas pruebas automáticas, commits pequeños, capacidad para ocultar funcionalidades incompletas cuando sea necesario y una cultura de arreglar rápido la línea principal si se rompe. Y esto último tiene implicaciones, porque cuando se detecta un problema en la línea principal se produce un freno en el trabajo de todos los equipos.

3.5.2. Ramas por funcionalidad: *feature branch-based development*

Otra estrategia muy habitual consiste en crear ramas por funcionalidades, de manera que los equipos trabajan simultáneamente en estas ramas, que pueden tener una vida larga (días o incluso semanas). Este enfoque permite aislar el trabajo entre equipos, ya que mientras una funcionalidad no está lista, permanece en su rama. Cuando se considera terminada, se revisa y se integra en la línea principal (figura 3.5).

El problema es que, si una rama se separa de la línea principal mucho tiempo, cada vez será más probable que aparezcan problemas cuando tratemos de integrarla. Para evitar estos problemas los equipos de cada rama pueden traerse cambios de la línea principal en momentos determinados.

Cuando se prepara una entrega se suele utilizar una rama para la release, en la que se integran los cambios de las ramas de funcionalidades. Los paquetes se construyen a partir de estas ramas de release, una vez se han probado y estabilizado.

Una cuestión interesante es que, mientras se está probando la rama de release, el desarrollo en las ramas de funcionalidades puede seguir avanzando en paralelo con trabajo para la próxima entrega.

Durante mucho tiempo, uno de los modelos más populares dentro de esta familia fue GitFlow, que propone una estructura muy clara con ramas por funcionalidades, correcciones urgentes, desarrollo, línea principal y entregas. En proyectos donde se publican versiones grandes cada cierto tiempo y hay que mantener varias líneas de release, un modelo como este puede tener sentido.

Pero en proyectos que buscan integración continua, entregas frecuentes y despliegues automatizados, estas estrategias encajan peor. Como afirma el propio Vincent Driessen, creador de GitFlow, *recuerda siempre que las soluciones milagrosas no existen. Considera tu propio contexto. No odies. Y sé tú quien decide.*

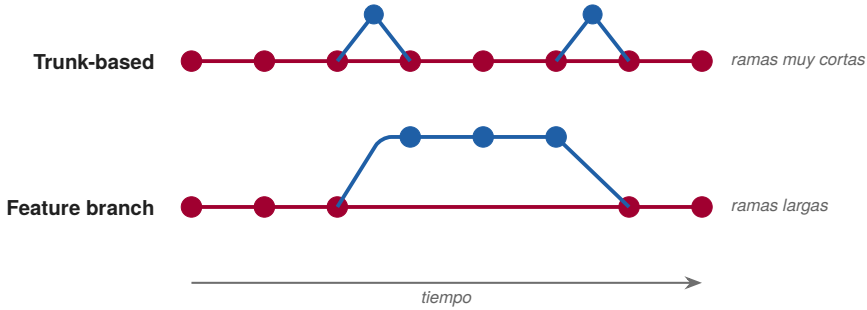


Fig. 3.5 Dos modelos de ramas: *trunk-based* (integración muy frecuente en la línea principal, con ramas de vida muy corta) frente a *feature branch* (ramas de funcionalidad de vida larga que se integran al terminar).

Para saber más

Lee el artículo de presentación de Git Flow del año 2010[a]. Presta especial atención a la nota que añadió en 2020, y reflexiona sobre sus implicaciones. A continuación lee el artículo sobre GitHub Flow[b]. ¿En qué familia de modelos de uso del repositorio encajarías a Git Flow y GitHub Flow respectivamente?

[a] https://nvie.com/posts/a-successful-git-branching-model/

[b] https://docs.github.com/en/get-started/using-github/github-flow

3.5.3. EGCFlow: el modelo que usaremos en la asignatura

A partir de las ideas anteriores, para el proyecto de la asignatura vamos a usar un modelo de trabajo que llamamos *EGCFlow*.

Este modelo encaja bien cuando el equipo de desarrollo no trabaja todos los días de forma continua. Este el caso habitual en una asignatura, con el alumnado avanzando más bien a ráfagas: una tarde se trabaja bastante, después el proyecto queda parado unos días porque hay otras entregas o exámenes, y más adelante se retoma[11].

[11] Algo parecido ocurre muchas veces en el Trabajo Final de Grado, y por eso bastantes estudiantes están usando EGGFlow también como modelo de uso del repositorio para su TFG en la Universidad de Sevilla.

Por eso no queremos aplicar de forma estricta un modelo de integración continua en el que todo el mundo esté integrando cambios constantemente todos los días. Pero tampoco queremos que existan ramas que se mantienen durante muchas semanas y que el equipo descubra problemas de integración justo antes de la entrega. EGCFlow intenta situarse en un punto intermedio.

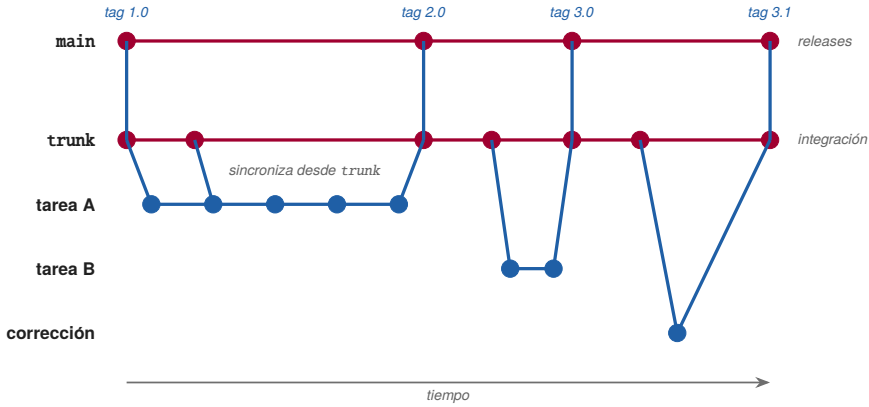


Fig. 3.6 Modelo EGCFlow. El trabajo se desarrolla en ramas de tarea y se integra con frecuencia en `trunk`. Los estados estables que se entregan se llevan a `main` y se identifican mediante etiquetas.

En este modelo hay dos ramas que no se destruyen. La rama `trunk` es la línea de integración del equipo. Ahí se mezclan con frecuencia los cambios que se van terminando. No significa que `trunk` tenga que contener únicamente versiones listas para entregar, pero sí debería mantenerse en un estado razonable: el proyecto debería construir, las pruebas básicas deberían pasar y el resto del equipo debería poder seguir trabajando a partir de ella.

La rama `main`, en cambio, representa la línea de entregas. Cuando un estado de `trunk` se considera suficientemente estable para una entrega, se incorpora a `main` y se marca con una etiqueta, por ejemplo `1.0`, `2.0` o `3.1`. Así podemos saber qué estado exacto del código corresponde a cada entrega. Esto es importante porque, como hemos insistido varias veces, una release debe tener una referencia trazable dentro del repositorio.

El trabajo diario se realiza en ramas de tarea. Una rama debería representar una funcionalidad, una corrección, una mejora de documentación, una refactorización concreta o cualquier otro cambio con un propósito claro. Por eso evitaremos las ramas por persona. Por el contrario, una rama llamada `login`, `fix-fechas` o `docs-api` ayuda a entender qué está ocurriendo en el repositorio. Una rama llamada

con el nombre de una persona suele acabar mezclando cambios distintos y hace más difícil interpretar la historia del proyecto.

Las ramas de tarea pueden durar algo más que en un modelo *trunk-based* estricto porque, como decíamos, asumimos que el trabajo en la asignatura será intermitente. Pero eso no significa que puedan quedar abandonadas durante semanas. Mientras una rama siga viva, habrá que mantenerla razonablemente cerca de `trunk`. Y cuando la tarea termine, habrá que integrarla en `trunk` y eliminarla (figura 3.6).

Insistimos en esto último: las ramas que dejan de usarse se destruyen. Mantener ramas antiguas sin propósito suele añadir ruido. Por tanto, si una rama sigue existiendo, el equipo debería saber por qué existe, qué contiene y qué tendría que ocurrir para cerrarla. Es decir, debe existir una justificación para ello.

Por último, otra cuestión importante es que en EGCFlow no usaremos *pull requests* como mecanismo para integrar trabajo dentro del mismo equipo. Y la razón es fundamentalmente didáctica. Queremos que aprendas a hacer integraciones con Git directamente para traer cambios, mezclar ramas, resolver conflictos, comprobar el resultado y dejar el repositorio en un estado coherente. Las plataformas como GitHub ofrecen interfaces muy cómodas para hacer parte de este trabajo, pero en esta asignatura no nos interesa que el proceso quede oculto detrás de un botón.

Los *pull requests* los reservaremos para las situaciones en que un equipo quiera proponer cambios al proyecto de otro equipo. Ahí sí tienen sentido, porque hay una frontera clara entre proyectos. El equipo que recibe la propuesta puede revisarla, discutirla y decidir si la acepta o no.

3.6. ¿Y cómo afecta la IA a todo esto?

Hace un par de cursos académicos los mensajes de commit de los proyectos de los estudiantes de la asignatura comenzaron a llenarse de emojis, algo poco habitual hasta la fecha. Esto parecía indicar que se estaban utilizando sistemas con IA como asistentes para algunas tareas.

Pero la situación ha cambiado bastante desde entonces. Los agentes de programación actuales pueden recibir una tarea, explorar el repositorio, modificar ficheros, ejecutar pruebas, crear commits, trabajar en una rama y terminar abriendo una *pull request* para que sus cambios sean revisados. Y una misma persona puede poner a trabajar varios agentes al mismo tiempo.

Esto empieza a tener consecuencias incluso para las propias plataformas que alojan los repositorios, ya que están creciendo a toda velocidad la creación de repositorios, el número de commits, peticiones de merge o el uso de APIs. En

consecuencia, GitHub está trabajando para que su infraestructura pueda soportar hasta treinta veces su escala anterior[12].

Y esto conecta también con lo que hemos estudiado en este capítulo. Si producir cambios se vuelve mucho más rápido, revisar e integrar esos cambios puede convertirse en el nuevo cuello de botella[13]. Podemos automatizar parte de la revisión, ejecutar pruebas o pedir a otros agentes que busquen problemas, pero decidir si un cambio tiene sentido desde el punto de vista del diseño, si encaja con el resto del proyecto o si queremos incorporarlo a la historia sigue requiriendo criterio.

Un estudio reciente sobre *pull requests* creadas por agentes observa que los cambios de mayor tamaño tienen más dificultades para integrarse[14]. Por eso están ganando interés estrategias que intentan mantener los cambios pequeños y revisables. Por ejemplo, recientemente GitHub ha incorporado soporte para *stacked pull requests*, que permiten dividir un cambio grande en una secuencia de propuestas más pequeñas que pueden revisarse de forma independiente e incluso en paralelo[15].

Esto conecta claramente con muchas de las ideas que hemos visto en el capítulo: commits con un propósito claro, cambios pequeños, ramas de vida corta, integración frecuente y una línea principal que se mantiene sana. Si un agente puede producir mucho código muy rápido, quizá sea todavía más importante evitar que acumule durante horas una rama enorme que después resulte difícil de entender y revisar.

También está ocurriendo algo interesante en la dirección contraria: algunas funcionalidades de Git que durante años han pasado bastante desapercibidas están encontrando nuevos casos de uso. Un ejemplo son los *worktrees*[16]. Como los worktrees permiten asociar varios directorios de trabajo al mismo repositorio, es posible tener varias ramas abiertas simultáneamente sin estar cambiando continuamente de una a otra. Y esto encaja especialmente bien con un escenario en el que varios agentes trabajan en paralelo, ya que cada agente puede trabajar en su propio directorio y sobre su propia rama, compartiendo la misma historia del repositorio con los demás[17].

Pero, ojo, que separar los ficheros y las ramas no significa aislar completamente el entorno: dos agentes todavía podrían intentar usar el mismo puerto o los mismos

[12] https://github.blog/news-insights/company-news/an-update-on-github-availability/

[13] https://github.blog/ai-and-ml/generative-ai/agent-pull-requests-are-everywhere-heres-how-to-review-them/

[14] https://arxiv.org/abs/2602.19441

[15] https://github.blog/changelog/2026-07-30-stacked-pull-requests-are-now-in-public-preview/

[16] https://git-scm.com/docs/git-worktree

[17] https://openai.com/index/introducing-the-codex-app/

recursos, como veremos en mayor detalle cuando estudiemos el aislamiento de los entornos en el Capítulo 6.

3.7. Reflexión final: gestionar código fuente es gestionar configuración

En este capítulo hemos recorrido el camino desde una situación muy sencilla, como guardar copias de una práctica en carpetas distintas, hasta modelos de uso de repositorios pensados para equipos que integran y entregan software con distintas estrategias.

La idea principal es que los repositorios de código van mucho más allá de almacenar ficheros, ya que permiten mantener la historia del proyecto, coordinar trabajo, recuperar estados, revisar decisiones y conectar el código fuente con pruebas, incidencias, artefactos y entregas.

Git nos da mecanismos muy potentes, como hemos visto. Pero la herramienta no debería marcar cómo se organiza el trabajo del equipo. Esa decisión forma parte de la gestión de la configuración, y requiere criterio humano. Como veremos en las clases de prácticas, podemos ejecutar pruebas automáticamente, generar registros de cambios o preparar pipelines con múltiples pasos para realizar diferentes comprobaciones. Pero el equipo sigue teniendo que decidir qué entra en el repositorio, cómo se escriben los commits, cuándo se integran ramas, qué modelo de uso se adopta y qué significa mantener una línea principal sana.

Si el capítulo anterior nos ayudaba a entender el recorrido del código hasta convertirse en producto en uso, este capítulo nos ha mostrado que todo ese recorrido empieza en la historia del código fuente. Una historia cuidada no garantiza por sí sola que el software sea bueno, por supuesto, pero es evidente que sin ella se hace mucho más difícil construir, probar, revisar, entregar y desplegar software de forma fiable y reproducible.

3.8. Lecturas y recursos recomendados

- Libro *Pro Git* (Scott Chacon y Ben Straub).
- Artículo *How to explain git in simple words?* (Muhammad Usama).
- Sitio `https://ohshitgit.com/es`
- Sesión *Version Control and Git* del curso *The Missing Semester of Your CS Education* (MIT).

- Capítulo 8, *Delivering Software*, del libro *The Missing README: A Guide for the New Software Engineer* (Chris Riccomini y Dmitriy Ryaboy).
- Capítulo 10, *DevOps* y Apéndice, *Git*, del libro *Software Engineering: A Modern Approach* (Marco Tulio Valente).

Capítulo 4

Pruebas de software

Me alegré más de que el software funcionara que del hecho de que aterrizáramos. Margaret Hamilton.

Resumen En este capítulo estudiamos por qué las pruebas de software son una pieza fundamental para construir, integrar, entregar y desplegar software con confianza. Veremos que probar requiere diseñar con criterio situaciones que puedan revelar fallos relevantes. Para ello distinguiremos entre errores, defectos y fallos, analizaremos el papel de los oráculos de prueba, presentaremos diferentes tipos y niveles de pruebas, y estudiaremos técnicas para seleccionar buenos casos sin intentar probarlo todo. También introduciremos el desarrollo guiado por pruebas como una práctica que usa las pruebas para guiar el diseño y la construcción del código. Finalmente, reflexionaremos sobre cómo la IA empieza a transformar el diseño, la generación y la evaluación de pruebas. Y cerraremos el capítulo con una idea central: aunque una buena suite de pruebas no elimina el riesgo completamente, sí nos ayuda a gestionarlo con evidencia y confianza.

4.1. Probar software para trabajar con confianza

Como se describe en el Capítulo 2, un pilar fundamental de la integración continua es que la construcción sea auto-testable, es decir, que cada vez que se publica un cambio en la línea principal se lanza una construcción que debe pasar una batería de pruebas. Y este conjunto de pruebas debe ser suficientemente detallado como para que el equipo pueda estar bastante seguro de que el sistema está funcionando correctamente y el cambio se ha integrado correctamente con el resto del sistema sin producir regresiones.

La idea básica es que, como la presenta Martin Fowler[1], además de construir el sistema, construimos también un detector de posibles problemas. Así, si alguien introduce un cambio que rompe un comportamiento importante, ese detector debería activarse. Esto no nos garantiza que todos los fallos posibles estén cubiertos, pero sí nos permite trabajar con más seguridad.

Este enfoque, que parece tan simple, en realidad ha cambiado la forma en la que desarrollamos software. Porque si no tenemos este mecanismo de retroalimentación continuo, modificar cualquier parte de un sistema existente puede darnos miedo. Incluso cuando vemos algo en el código que podríamos mejorar de forma sencilla, podemos estar tentados a pensar "mejor no lo toco, no vaya a romper algo". Por ello, como defiende Robert C. Martin en su libro *Clean Craftsmanship*, las pruebas automatizadas reducen el miedo del equipo de desarrollo a incluir nuevas funcionalidades o a refactorizar el código. Si la suite pasa (está *en verde*), se puede seguir adelante con confianza; si falla (está *en rojo*), avisa antes de que el problema llegue lejos (figura 4.1).

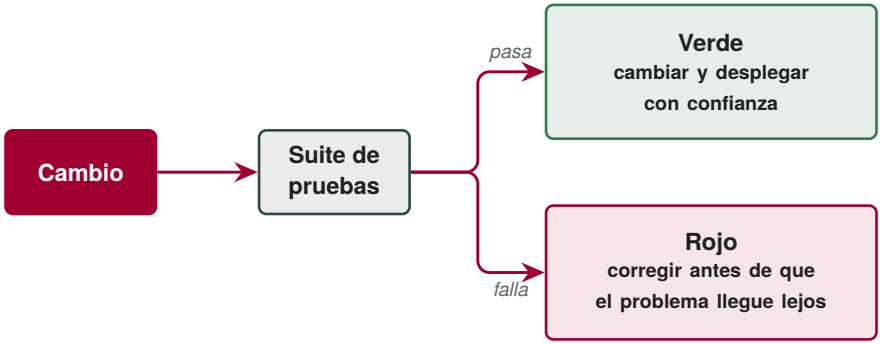


Fig. 4.1 Las pruebas automatizadas actúan como un detector de problemas: si pasan (verde), el equipo puede cambiar y desplegar con confianza; si fallan (rojo), avisan antes de que el problema llegue lejos.

Pero, además, las pruebas son una documentación magnífica, puesto que muestran cómo se debe interactuar con los componentes de un sistema. Y, por tanto, cuando tienes que comenzar a trabajar con un proyecto existente, las pruebas son el primer sitio al que sueles dirigirte[2].

[1] https://martinfowler.com/bliki/SelfTestingCode.html

[2] Por ello, cuando comiences a trabajar en el proyecto de la asignatura para realizar modificaciones a UVLHub, quizá pueda ser buena idea que eches un vistazo a sus pruebas para familiarizarte con el código.

Y, por otra parte, escribir las pruebas obliga al equipo de desarrollo a pensar sobre la interfaz y el diseño del sistema. Y le ayuda a darse cuenta de que si es complicado escribir pruebas para un programa, probablemente es que no está bien diseñado o implementado. Por lo que las pruebas contribuyen a que los equipos escriban código limpio, mejorando la separación de responsabilidades y reduciendo el acoplamiento.

¡Qué maravillosas son las pruebas! Todo son ventajas, ¿no? Bueno, sí, pero para lograr todos estos beneficios es fundamental el criterio humano a la hora de diseñar un buen conjunto de pruebas.

Esto lo vemos claramente en otras industrias, como la automovilística. Cuando visualizamos un vídeo de las pruebas de choque de un coche, entendemos claramente que antes de ejecutar la prueba un equipo ha tenido que tomar muchas decisiones: a qué velocidad se va a producir el impacto, contra qué tipo de obstáculo, con qué ángulo, qué sensores se van a colocar, qué medidas se van a observar, qué condiciones se consideran aceptables y qué conclusiones podremos sacar después.

Es decir, que hay una parte de diseño de la prueba, otra parte de ejecución de la prueba y otra de evaluación de los resultados de la prueba. Y para que podamos sacar conclusiones útiles, el diseño tiene un papel fundamental. Es más, automatizar pruebas mal diseñadas puede darnos una sensación de seguridad muy peligrosa. Por eso, a lo largo de este capítulo vamos a insistir mucho en esta idea: "Automatizar pruebas sin diseño es como hacer una pizza sin base"[3].

4.2. Qué es una prueba de software

Las pruebas de software nos permiten evaluar de forma sistemática si un programa cumple sus requisitos y funciona como esperamos. Por tanto, una prueba de software ejecuta una parte del sistema bajo unas condiciones concretas, observa el resultado y lo compara con lo que esperábamos que ocurriera. Si el resultado observado coincide con el esperado, la prueba pasa. Si no coincide, la prueba falla y nos está indicando que hay algo que debemos investigar.

Por ejemplo, imagina que tenemos una función `tiene_matricula_reducida` que decide si una persona tiene derecho a matrícula reducida por edad. Supongamos que el requisito dice que tienen matrícula reducida las personas menores de 18 años y las mayores o iguales que 65.

Una prueba sencilla podría comprobar que la función devuelve `True` para una persona de 17 años, indicando que tiene matrícula reducida. En este caso, la

[3] Frase atribuida a David Benavides, profesor de la asignatura

entrada de la prueba es el valor `17`. La parte del sistema bajo prueba es la función `tiene_matricula_reducida`. Y el resultado esperado es que la función devuelva `True`. Si la condición se cumple, la prueba pasa; si no se cumple, la prueba falla.

4.2.1. Entradas, salidas y oráculo

En un caso de prueba solemos distinguir tres elementos fundamentales. En primer lugar, las entradas, que son los datos, eventos o condiciones bajo las que se ejecuta el sistema. En segundo lugar, las salidas observadas, que son los resultados que el sistema produce al ejecutarse. Y, en tercer lugar, el oráculo de prueba, que es el criterio que nos permite decidir si esas salidas son correctas.

El término *oráculo* puede sonar un poco exagerado y solemne, la verdad, pero básicamente es el mecanismo que nos permite saber si la prueba ha pasado o ha fallado.

En una prueba automatizada, el oráculo suele expresarse mediante aserciones. Por ejemplo:

Entrada, salida y oráculo

```
def calcular_total(precios):
    return sum(precios)

def test_calcula_total_de_varios_productos():
    precios = [10, 15, 5]
    total = calcular_total(precios)
    assert total == 30
```

La lista `[10, 15, 5]` es la entrada. El valor `total` es la salida observada. Y la aserción `assert total == 30` es el oráculo de prueba (figura 4.2).

Muchas pruebas automatizadas siguen una estructura bastante parecida: primero se prepara el contexto, después se ejecuta la acción que queremos comprobar, y finalmente se verifica el resultado. En inglés se suele hablar de *arrange, act, assert*, es decir, preparar, actuar y comprobar.

Esta estructura ayuda a escribir pruebas más legibles, de forma que cuando alguien lee una prueba pueda entender rápidamente qué situación se está preparando, qué comportamiento se está ejercitando y qué resultado se considera correcto.

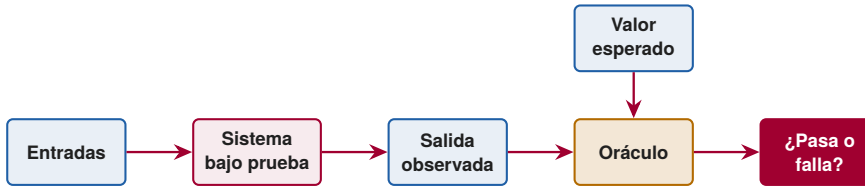


Fig. 4.2 Los tres elementos de un caso de prueba: las *entradas*, la *salida observada* y el *oráculo*, que compara la salida con lo esperado para decidir si la prueba pasa o falla.

4.2.2. Qué puede y qué no puede demostrar una prueba

Una prueba que falla puede demostrar que existe una discrepancia que debemos investigar. Si esperábamos que una función devolviera `True` y devuelve `False`, tenemos una evidencia clara de que algo no se comporta como esperábamos.

Sin embargo, una prueba que pasa no demuestra que el programa sea correcto en todos los casos. Solo nos dice que, para ese caso concreto, no hemos observado un fallo. Esta diferencia es fundamental. Edsger Dijkstra lo expresó con una frase muy conocida: las pruebas muestran la presencia de fallos, pero no su ausencia.

Veámoslo con una pequeña variación del ejemplo anterior:

Función con un defecto

```
def tiene_matricula_reducida(edad):
    return edad < 18 or edad > 65
```

La función contiene un defecto, porque el requisito decía que las personas de 65 años también tienen matrícula reducida. Sin embargo, las siguientes pruebas pasarían:

Pruebas que no revelan el defecto

```
def test_menores_de_18_tienen_matricula_reducida():
    assert tiene_matricula_reducida(17)

def test_mayores_de_65_tienen_matricula_reducida():
    assert tiene_matricula_reducida(70)
```

¿Significa eso que la función es correcta? No. Significa únicamente que esas dos pruebas no han detectado el problema. Para revelar el defecto necesitamos probar justo el límite:

Prueba que revela el defecto

```
def test_personas_de_65_tienen_matricula_reducida():
    assert tiene_matricula_reducida(65)
```

Esta prueba fallaría y nos indicaría que el comportamiento en el límite de 65 años no coincide con el requisito.

Por tanto, insistimos en que el diseño de casos de prueba es la base fundamental. Podemos tener pruebas automatizadas, ejecutarlas en cada cambio y ver todo en verde, pero aun así dejar sin cubrir comportamientos relevantes. Las pruebas solo pueden aumentar nuestra confianza en el software en la medida en que los casos elegidos sean significativos.

4.2.3. Error, defecto y fallo

En el lenguaje cotidiano usamos palabras como error, bug, defecto o fallo de forma bastante intercambiable. Incluso en documentación técnica es habitual encontrarse con ello. Pero para razonar sobre pruebas conviene distinguir algunos conceptos.

Un *error* es una equivocación humana[4]. Por ejemplo, una persona puede interpretar mal un requisito, olvidar un caso particular o escribir una condición incorrecta.

Un *defecto*, *fault* o *bug* es la causa que queda introducida en el software como consecuencia de ese error. En el ejemplo anterior, el defecto estaba en escribir `edad >65` cuando el requisito exigía `edad >= 65`.

Un *fallo*, o *failure*, es el comportamiento incorrecto que observamos cuando el sistema se ejecuta. Si una persona de 65 años intenta matricularse y el sistema no le aplica la matrícula reducida, entonces estamos observando un fallo.

La cadena sería, por tanto, algo así: una persona comete un error, ese error introduce un defecto en el código, y ese defecto puede provocar un fallo cuando el sistema se ejecuta bajo ciertas condiciones (figura 4.3).

Decimos "puede provocar" porque un defecto no siempre se manifiesta. Es posible que el código defectuoso no se ejecute en una prueba concreta. O que se ejecute,

[4] O quizá del agente IA que escribe el software.

Fig. 4.3 La cadena error → defecto → fallo. Un error humano introduce un defecto en el código, que *puede* (no siempre) provocar un fallo observable al ejecutar.

pero con datos que no hacen visible el problema. O que produzca un estado interno incorrecto que nunca llega a observarse en la salida final.

Esto explica por qué una prueba puede pasar aunque exista un defecto en el programa. Para observar un fallo se suelen necesitar tres condiciones, conocidas como modelo RIP: *reachability*, *infection* y *propagation*.

La primera condición es la accesibilidad. La prueba debe llegar hasta la parte del código donde está el defecto. Si el defecto está en una rama de un `if` que nunca se ejecuta, la prueba no podrá revelarlo.

La segunda condición es la infección. Una vez ejecutado el código defectuoso, el estado interno del programa debe volverse incorrecto. Hay casos en los que se ejecuta una línea defectuosa, pero para ciertos datos el resultado coincide casualmente con el que habría producido el código correcto.

La tercera condición es la propagación. El estado incorrecto debe propagarse hasta una salida observable. Si el programa calcula un valor interno incorrecto, pero luego ese valor no se usa o no afecta al resultado final, no veremos el fallo.

Considera este ejemplo:

Defecto que no siempre se observa

```
def aplicar_descuento(precio, porcentaje):
    descuento = precio * porcentaje / 100
    return precio + descuento
```

La función tiene un defecto, porque está sumando el descuento en lugar de restarlo. Sin embargo, esta prueba pasa:

Prueba que no revela el defecto

```
def test_descuento_cero():
    assert aplicar_descuento(100, 0) == 100
```

La prueba ejecuta el código defectuoso, pero no revela el problema porque sumar cero y restar cero producen el mismo resultado. En cambio, esta otra prueba sí lo revelaría:

Prueba que revela el defecto

```
def test_descuento_del_diez_por_ciento():
    assert aplicar_descuento(100, 10) == 90
```

Por eso, además de que una prueba ejecute una línea de código, es necesario que los casos hagan visible el comportamiento incorrecto.

4.2.4. Testing no es lo mismo que debugging

La distinción entre defecto y fallo también nos ayuda a separar dos actividades que a veces se mezclan: *testing* y *debugging*.

Hacer *testing* consiste en evaluar el software observando su ejecución. Diseñamos casos de prueba, ejecutamos el sistema y comprobamos si el resultado coincide con lo esperado. Si una prueba falla, sabemos que hay un comportamiento que debemos investigar.

Hacer *debugging*, en cambio, consiste en localizar la causa del fallo. Es decir, partimos de un fallo observado e intentamos encontrar el defecto que lo provoca. Para ello podemos leer el código, reproducir el problema, inspeccionar variables, revisar cambios recientes, usar un depurador o añadir registros temporales. como veremos en detalle en el Capítulo 5.

Una prueba, por tanto, puede decirnos que algo va mal, pero no siempre nos dice exactamente dónde está el problema. Si falla una prueba pequeña, que prueba una única función concreta, seguramente tendremos el defecto bastante acotado. Si falla una prueba que recorre una funcionalidad completa, quizá sepamos que el registro de usuarios no funciona, pero todavía tendremos que investigar si el problema está en la validación, la base de datos, el envío de correo, la configuración del entorno o la interfaz.

Esto no implica que las pruebas amplias sean malas. Simplemente cada tipo de prueba aporta información distinta. Algunas pruebas están pensadas para localizar rápido un problema concreto. Otras están pensadas para ganar confianza en que un flujo completo funciona. Como veremos en la siguiente sección, una buena estrategia de pruebas combina distintos niveles para equilibrar rapidez, coste, precisión y confianza.

4.3. Tipos y niveles de prueba

Hasta ahora hemos hablado de pruebas, así, en general. Pero no todas las pruebas tienen el mismo objetivo ni el mismo alcance. Algunas comprueban una función muy pequeña. Otras ejercitan la interacción entre varios módulos. Otras recorren el sistema completo desde el punto de vista de un usuario final. Y otras no se centran tanto en qué hace el sistema, sino en cómo lo hace.

Distinguir estos tipos y niveles es importante para diseñar una estrategia razonable, en la que cada nivel observa el sistema desde una distancia distinta y nos ofrece una confianza diferente. Además, veremos que no todas las pruebas tienen el mismo coste, por lo que no todas deben ejecutarse en el mismo momento ni con la misma frecuencia.

4.3.1. Pruebas funcionales y no funcionales

Una primera distinción útil es la que separa pruebas funcionales y pruebas no funcionales.

Las pruebas funcionales evalúan si el sistema hace lo que se supone que debe hacer. Por ejemplo, si una persona puede crear una cuenta con datos válidos, si el sistema rechaza una contraseña incorrecta, si una función calcula correctamente el precio de una matrícula o si un archivo UVL se transforma al formato esperado.

En cambio, las pruebas no funcionales evalúan características de calidad del sistema. Se centran en aspectos como rendimiento, seguridad, usabilidad, accesibilidad, escalabilidad, compatibilidad o disponibilidad.

Pensemos en un sistema de automatrícula de la universidad. Desde el punto de vista funcional, quizá permite seleccionar asignaturas, comprobar requisitos, calcular tasas y confirmar la matrícula. Pero si el primer día se conectan miles de estudiantes y el sistema deja de responder, tenemos un problema muy serio aunque muchas reglas funcionales estén implementadas correctamente.

Algo parecido ocurre con la seguridad. Un formulario de inicio de sesión puede aceptar credenciales válidas y rechazar credenciales incorrectas, y aun así tener problemas graves si permite ataques de fuerza bruta, no protege adecuadamente las sesiones o almacena contraseñas de forma insegura.

4.3.2. Pruebas unitarias

Las pruebas unitarias ejercitan partes pequeñas y concretas del software. Normalmente se centran en funciones, métodos o clases, y tratan de comprobar un comportamiento específico de forma bastante aislada.

Por ejemplo, podríamos probar la función de matrícula reducida que hemos usado en la sección anterior:

Pruebas unitarias

```
def tiene_matricula_reducida(edad):
    return edad < 18 or edad >= 65

def test_menores_de_18_tienen_matricula_reducida():
    assert tiene_matricula_reducida(17)

def test_adultos_no_tienen_matricula_reducida():
    assert not tiene_matricula_reducida(30)

def test_personas_de_65_tienen_matricula_reducida():
    assert tiene_matricula_reducida(65)
```

Estas pruebas tienen varias ventajas. Suelen ser muy rápidas, porque no necesitan arrancar una aplicación completa ni conectarse a servicios externos. Son precisas, porque si fallan el problema suele estar bastante acotado. Y son fáciles de ejecutar con mucha frecuencia, incluso cada vez que guardamos un cambio o antes de hacer un commit.

Por eso encajan muy bien con la integración continua que vimos en el Capítulo 2, que busca que el pipeline nos dé retroalimentación en pocos minutos y de forma fiable.

Pero las pruebas unitarias también tienen límites. Que muchas funciones pequeñas funcionen por separado no garantiza que el sistema completo funcione cuando esas funciones se combinan. Puede que cada pieza haga lo correcto de forma aislada, pero que dos componentes no se entiendan bien, que haya un problema de configuración o que el formato de datos que produce un módulo no sea el que espera otro.

4.3.3. Pruebas de integración

Las pruebas de integración observan si varias partes del sistema funcionan correctamente cuando colaboran. Pueden comprobar la interacción entre módulos internos, entre una aplicación y una base de datos, entre un servicio y una API externa, o entre distintos componentes de una arquitectura.

Por ejemplo, en un sistema como UVLHub, podríamos tener una prueba unitaria que comprueba que una función transforma correctamente una cadena UVL válida. Pero una prueba de integración podría comprobar que el servicio recibe un archivo, invoca el transformador adecuado, guarda el resultado y permite recuperarlo después.

Este tipo de prueba detecta problemas que las unitarias no suelen ver: errores en interfaces entre componentes, formatos incompatibles, fallos de serialización, problemas con consultas a la base de datos, dependencias mal configuradas o suposiciones distintas entre dos partes del sistema.

El precio es que suelen ser más lentas y más difíciles de preparar. Por ejemplo, si una prueba necesita una base de datos, tendremos que decidir cómo se crea, con qué datos iniciales se carga, cómo se limpia después de cada ejecución y cómo evitamos que una prueba dependa de lo que haya dejado otra.

Este tipo de decisiones conectan directamente con el Capítulo 6 en el que hablaremos de entornos y construcción aislada. Para que una prueba de integración sea fiable necesitamos que el entorno pueda construirse de forma previsible y automática. De lo contrario, la prueba puede pasar en una máquina, fallar en otra y convertirse en una fuente de incertidumbre.

4.3.4. Pruebas de sistema

Las pruebas de sistema evalúan el comportamiento del sistema completo, o de una parte suficientemente grande como para considerarla una versión integrada del producto. En lugar de centrarse en una función o en la colaboración entre dos componentes concretos, observan si el sistema, como conjunto, se comporta como esperamos.

Por ejemplo, una prueba de sistema podría levantar la aplicación, preparar una base de datos de prueba y comprobar que una persona puede registrarse, iniciar sesión, subir un archivo UVL, transformarlo y consultar el resultado. Cuando estas pruebas recorren un flujo completo desde una entrada cercana a la persona usuaria hasta las capas internas del sistema, también se suelen llamar pruebas extremo a extremo.

Estas pruebas aportan mucha confianza porque se parecen más al uso real del sistema. También pueden revelar problemas de arquitectura, configuración, permisos, rutas, dependencias o interacción entre capas que no aparecerían en pruebas más pequeñas.

Pero, de nuevo, hay un coste. Las pruebas de sistema suelen ser más lentas, más frágiles y más difíciles de diagnosticar. Si falla una prueba unitaria de la función `tiene_matricula_reducida`, sabemos bastante bien dónde mirar. Si falla una prueba de sistema que recorre un flujo completo, no es tan sencillo saber si el problema está en la interfaz, en la validación, en la base de datos, en la configuración del entorno, en un servicio externo o en los datos de prueba.

Por eso conviene usarlas con criterio. No tiene sentido comprobar todos los detalles del sistema únicamente mediante pruebas de sistema. Sería lento, costoso y difícil de mantener. Pero sí tiene sentido automatizar algunos flujos importantes que nos den confianza en que las piezas principales encajan.

4.3.5. Pruebas de aceptación

Las pruebas de aceptación son un poco diferentes, porque se centran en comprobar si el software satisface los criterios acordados con la persona usuaria, el cliente o el equipo que define el producto.

Por ejemplo, un criterio de aceptación para una funcionalidad de registro podría ser:

> Una persona puede crear una cuenta proporcionando un correo válido, una contraseña que cumple las reglas de seguridad y aceptando las condiciones de uso. Tras registrarse, puede iniciar sesión en el sistema.

Una prueba de aceptación trataría de comprobar ese comportamiento desde una perspectiva completa. En proyectos profesionales suelen estar vinculadas a acuerdos con clientes, validaciones de producto o normas de negocio. Y básicamente permiten saber cuándo el trabajo está finalizado. La figura 4.4 resume los cuatro niveles según su alcance.

4.3.6. Dobles de prueba y dependencias externas

Al hablar de las pruebas unitarias al comienzo de la sección hemos hecho un poco de trampa. Las pruebas eran muy sencillas porque la función `tiene_matricula_reducida` no dependía de nada externo. Recibía un valor,

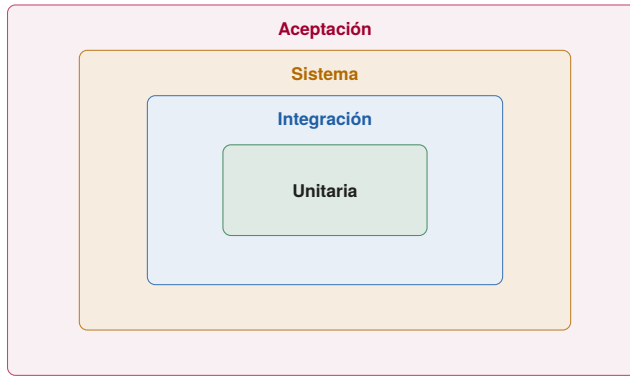


Fig. 4.4 Los niveles de prueba según su alcance: una prueba unitaria observa una pieza pequeña; la de integración, varias piezas juntas; la de sistema, la aplicación completa; y la de aceptación, el sistema desde el punto de vista del usuario.

calculaba un resultado y lo devolvía. Pero en los sistemas reales muchas piezas de código no trabajan solas. Consultan una base de datos, leen un fichero, llaman a una API, envían un correo, miran la hora actual o interactúan con otros componentes del sistema.

Esto es problemático, porque si queremos probar una parte pequeña del sistema, pero esa parte depende de elementos externos, la prueba puede volverse lenta, frágil o difícil de reproducir. Por ejemplo, si una prueba depende de una API real, puede fallar porque no tenemos conexión, porque el servicio está caído, porque ha cambiado su respuesta o porque nos ha bloqueado por hacer demasiadas peticiones. En ese caso, la prueba ya no depende solo de nuestro código, ya que lo hace también del estado de un sistema externo.

Para evitarlo, a veces usamos dobles de prueba. Un doble de prueba es un sustituto controlado de un componente real. En lugar de usar la base de datos real, la API real o el servicio de correo real, usamos un objeto preparado para comportarse de una forma concreta durante la prueba.

Por ejemplo, planteamos a continuación una función que comprueba si la web de la escuela está disponible. Para no acoplarla directamente a una biblioteca HTTP concreta, hacemos que reciba un cliente HTTP como parámetro:

Función con una dependencia externa

```
def web_disponible(cliente_http):
    respuesta = cliente_http.get("https://www.informatica.us.es/")
    return respuesta.status_code == 200
```

Si en una prueba llamamos a la web real, estaremos comprobando varias cosas a la vez: nuestra función, la biblioteca HTTP, la red, el servidor y quizá alguna configuración del entorno. Por tanto, deja de ser una prueba unitaria pequeña y controlada (figura 4.5).

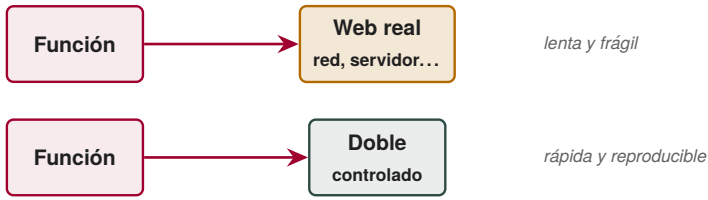


Fig. 4.5 Un doble de prueba sustituye una dependencia externa real (lenta y frágil) por un objeto controlado, de modo que la prueba ejercita solo la lógica de nuestra función.

Para probar únicamente la lógica de nuestra función podemos sustituir el cliente HTTP por un doble sencillo:

Prueba usando un doble sencillo

```
class RespuestaFalsa:
    status_code = 200

class ClienteHttpFalso:
    def get(self, url):
        return RespuestaFalsa()

def test_web_disponible_si_responde_200():
    cliente = ClienteHttpFalso()
    assert web_disponible(cliente)
```

Esta prueba es rápida, no necesita conexión a internet y siempre recibe la misma respuesta. Por tanto, nos permite comprobar de forma controlada que nuestra función devuelve `True` cuando el cliente HTTP responde con un código 200.

También podríamos probar el caso contrario:

Prueba con una respuesta de error

```
class RespuestaErrorFalsa:
    status_code = 500

class ClienteHttpConErrorFalso:
    def get(self, url):
        return RespuestaErrorFalsa()

def test_web_no_disponible_si_responde_error():
    cliente = ClienteHttpConErrorFalso()
    assert not web_disponible(cliente)
```

En la práctica encontrarás distintos nombres para estos sustitutos: *stubs*, *mocks*, o *fakes*. Aunque en la literatura se pueden encontrar matices entre ellos, básicamente todos ellos permiten sustituir una dependencia real por algo que controlamos durante la prueba, para que la prueba se mantenga rápida, determinista y centrada en una pieza pequeña del sistema.

Los lenguajes suelen contar con herramientas para crear este tipo de sustitutos sin tener que escribir clases falsas manualmente. Por ejemplo, Python incluye una herramienta estándar, `unittest.mock`, que permite crear dobles de prueba de forma más cómoda y flexible.

Para saber más

Investiga cómo funciona `unittest.mock.Mock` y úsalo para reescribir las pruebas de la función `web_disponible`, simulando dos respuestas distintas de la web de la escuela: una respuesta correcta con código `200` y una respuesta de error con código `500`.

Es importante entender que la prueba anterior no comprueba que la web esté realmente disponible. Tampoco comprueba que la biblioteca HTTP funcione correctamente, ni que la URL sea accesible desde nuestro entorno, ni que el sistema gestione bien una respuesta real con cabeceras, redirecciones o tiempos de espera. Solo comprueba qué hace nuestra función cuando su colaborador devuelve un objeto con un atributo `status_code`. Por eso los dobles de prueba no sustituyen a las pruebas de integración.

4.3.7. La pirámide de pruebas

Una vez que entendemos los distintos tipos de pruebas, se comprende mejor la necesidad de contar con una estrategia que equilibre confianza, coste y velocidad.

Una forma muy conocida de pensar en ese equilibrio es la pirámide de pruebas, popularizada por Mike Cohn (figura 4.6). La idea básica es que una suite de pruebas automatizadas debería tener muchas pruebas unitarias pequeñas y rápidas en la base, un número menor de pruebas de integración en la zona intermedia, y todavía menos pruebas amplias, lentas y cercanas al uso real en la parte superior.

Fig. 4.6 La pirámide de pruebas: muchas pruebas unitarias en la base; menos pruebas de integración y de sistema; y muy pocas pruebas de aceptación en la cima.

No conviene interpretar la pirámide como una ley exacta. Lo importante es la intuición que transmite: cuanto más amplia es una prueba, más partes del sistema cubre, pero normalmente también es más lenta, más costosa y menos precisa para localizar la causa de un fallo.

Ham Vocke resume esta idea de forma muy simple: escribe pruebas con distinta granularidad y, cuanto más alto nivel tenga la prueba, menos pruebas deberías tener[5].

En un pipeline de CI/CD este enfoque encaja de manera muy natural. Primero ejecutamos las pruebas más rápidas. Si esas pruebas pasan, podemos ejecutar pruebas más lentas y amplias. Y, en etapas posteriores, quizá pruebas de sistema,

[5] https://martinfowler.com/articles/practical-test-pyramid.html

aceptación, rendimiento o seguridad, usando los recursos más costosos solo cuando tiene sentido.

Como mencionábamos en el Capítulo 2, Dave Farley explica en su libro *Continuous Delivery Pipelines* que, para construir el London Multi-Asset Exchange, en la fase de CI usaban unas 40.000 pruebas unitarias y unos cientos de pruebas de integración que se pasaban en unos 4 minutos.

Si las pruebas no pasaban, el problema se detectaba muy rápido y podían arreglarlo o revertir el cambio. Si las pruebas pasaban, se generaba un *release candidate* que comenzaba a evaluarse en la fase de CD. Y, mientras se ejecutaban estas pruebas, el equipo podía comenzar a trabajar en una nueva funcionalidad con una confianza alta en que el código hacía lo que se esperaba.

En la fase de CD la suite estaba formada por varios miles de pruebas de sistema que tardaban en ejecutarse unos 40 minutos. Si cualquier prueba no pasaba, el candidato era descartado. Si se superaban todas, se pasaban pruebas de rendimiento, escalabilidad y calidad en el entorno de preproducción, y se generaba una entrega. En total, el pipeline tomaba 57 minutos en ejecutarse.

En este proyecto no seguían despliegue continuo. Los despliegues se hacían durante el fin de semana, cuando el mercado financiero estaba cerrado. Se tomaba la última entrega y se desplegaba en el entorno de producción, para lo cual utilizaban las mismas herramientas de gestión de la configuración que utilizaban en el pipeline de CI/CD. Por tanto, como las configuraciones, herramientas y procesos se habían probado en múltiples ocasiones en el pipeline, tenían la confianza en que existía poco riesgo de fallo al mover a producción. Este proceso tomaba menos de 10 minutos, incluyendo la migración de datos.

¿Y merecía la pena toda esta infraestructura de pruebas? ¿Funcionaba el enfoque? Dave Farley afirma con orgullo que pasaron 13 meses y 5 días hasta que un usuario notificó el primer bug. Pero, claro, esto no se debía solo a la automatización. Antes de poder ejecutar miles de pruebas en un pipeline, el equipo tuvo que diseñar buenos casos de prueba, seleccionando qué comportamientos merecía la pena comprobar y qué riesgos querían cubrir. Esa será precisamente la cuestión que abordaremos en la siguiente sección.

4.4. Diseño de casos de prueba

Ya hemos insistido a lo largo del capítulo en que automatizar muchas pruebas solo aporta valor si esas pruebas han sido diseñadas con criterio. Y ese criterio es el que nos ayuda a decidir qué casos concretos debemos probar en nuestro sistema.

Un enfoque ingenuo sería tratar de probarlo todo. Pero en software esto casi nunca es posible.

Imagina una función que recibe tres opciones booleanas: `notificaciones`, `modo_oscuro` y `recordatorios`. Cada opción puede estar activada o desactivada. En total tenemos $2^3 = 8$ combinaciones posibles. Eso parece manejable.

Pero si en lugar de tres opciones tenemos doce, ya tenemos $2^{12} = 4096$ combinaciones. Si además añadimos un desplegable con tres valores posibles, llegamos a 12.288 combinaciones. Probar exhaustivamente todas las combinaciones deja de ser razonable muy pronto. Por tanto, el objetivo de una buena suite de pruebas es elegir casos que tengan capacidad de revelar fallos relevantes.

En pruebas se habla a menudo de criterios de cobertura. Un criterio de cobertura es una regla o conjunto de reglas que nos ayuda a decidir qué debe cubrir nuestro conjunto de pruebas. Algunos criterios se basan en cubrir partes del código; otros, en cubrir decisiones lógicas; otros, en recorrer caminos de un grafo; y otros, en cubrir clases de datos de entrada.

Como veremos, estos criterios nos dan herramientas para no elegir casos al azar, aunque no sustituyen al juicio humano.

4.4.1. Particiones equivalentes

Una técnica clásica de diseño de casos de prueba consiste en dividir el espacio de entrada en particiones equivalentes. La idea es agrupar valores que, desde el punto de vista del comportamiento esperado, deberían tratarse de forma similar. En lugar de probar todos los valores de una partición, seleccionamos uno o varios representantes.

Volvamos al ejemplo de la matrícula reducida por edad. Supongamos que el requisito dice:

> Las personas menores de 18 años y las mayores o iguales que 65 tienen matrícula reducida. El resto paga matrícula ordinaria.

Podemos dividir el espacio de entrada en tres particiones:

- edades menores de 18;
- edades entre 18 y 64;
- edades mayores o iguales que 65.

Una primera suite de pruebas podría elegir un valor representativo de cada partición, de manera que comprueba un representante de cada comportamiento

esperado. Eso ya es mejor que elegir tres edades al azar que quizá caigan todas en la misma partición.

Por supuesto, las particiones no siempre son tan evidentes. En sistemas reales debemos entender el dominio, los requisitos y los riesgos. Si estamos validando archivos UVL, por ejemplo, podríamos distinguir archivos válidos, archivos con sintaxis incorrecta, archivos vacíos, archivos demasiado grandes o archivos que usan características no soportadas por nuestro transformador.

4.4.2. Valores límite

En muchas ocasiones los fallos aparecen en los límites entre particiones. Esto ocurre porque los límites suelen traducirse en comparaciones como <, <=, > o >=, y es muy fácil equivocarse justo ahí.

En el ejemplo de la edad, los límites importantes son 18 y 65. Por eso no deberíamos conformarnos con probar 12, 30 y 70. También deberíamos probar los valores que están justo en los bordes: 17, 18, 64, 65.

Si la implementación hubiera usado `edad >65` en lugar de `edad >= 65`, el caso de 65 revelaría el fallo. Si hubiera usado `edad <= 18` en lugar de `edad <18`, el caso de 18 lo revelaría.

Los valores límite son una técnica sencilla, pero muy poderosa. Siempre que veas rangos, tamaños máximos, fechas de corte, límites de longitud o umbrales de validación, sería buena idea pensar en ellos.

4.4.3. Combinaciones *pair-wise* y *n-wise*

Cuando tenemos una combinación de parámetros, unas técnicas muy utilizadas son las combinatorias, que intentan seleccionar un conjunto reducido de casos que cubra interacciones relevantes entre parámetros. Quizá la más conocida es *pair-wise testing* (o *all-pairs testing*), que busca cubrir todas las combinaciones posibles de pares de valores.

Por ejemplo, si tenemos tres variables booleanas `x`, `y` y `z`, probar todas las combinaciones requiere 8 casos:

$$(F,F,F), \quad (F,F,T), \quad (F,T,F), \quad (F,T,T),$$

$$(T,F,F), \quad (T,F,T), \quad (T,T,F), \quad (T,T,T)$$

Pero una selección *pair-wise* puede cubrir todas las combinaciones de pares con menos casos. Por ejemplo:

$$(F,F,F), \quad (F,T,T), \quad (T,F,T), \quad (T,T,F)$$

La clave está en mirar las variables de dos en dos. Para la pareja `x` y `y`, esos cuatro casos cubren `(False, False)`, `(False, True)`, `(True, False)` y `(True, True)`. Lo mismo ocurre con las parejas `x` y `z`, y `y` y `z`.

El objetivo es garantizar que cualquier interacción entre dos variables aparece al menos una vez en algún caso de prueba. Y se hace así porque muchos fallos se producen por la interacción entre dos parámetros. En ocasiones no bastará con cubrir pares, y tendrá sentido cubrir combinaciones de tres o más parámetros (*n-wise*).

4.4.4. Cobertura CRUD

En sistemas con datos persistentes es habitual comprobar las operaciones CRUD de cada entidad relevante: crear, leer, actualizar y borrar.

La operación de lectura actúa muchas veces como oráculo. Tras crear, actualizar o borrar, leemos el estado del sistema para comprobar si la operación tuvo el efecto esperado.

Además de esta secuencia básica, muchas veces conviene probar operaciones no permitidas: actualizar un elemento que no existe, borrar dos veces el mismo recurso, crear un recurso con datos incompletos o leer datos sin permisos suficientes.

Este tipo de cobertura es especialmente útil en pruebas de integración o de sistema, porque comprueba el comportamiento de la aplicación respecto a sus datos persistentes.

4.4.5. Errores conocidos y conocimiento del dominio

No todas las técnicas de prueba tienen que ser puramente sistemáticas. La experiencia también importa. Si un equipo sabe que en su sistema suelen aparecer fallos al procesar fechas con zonas horarias, al importar archivos con caracteres especiales o al gestionar usuarios no logueados, tiene sentido diseñar pruebas específicas para esos casos.

Esta técnica se basa en el conocimiento de errores habituales. Y esto es algo que se consigue con la experiencia profesional, con la que vamos desarrollando esa

intuición para saber sobre dónde suelen esconderse los problemas. Es una técnica que exige pararse a reflexionar, utilizar el criterio humano y también entender bien el dominio del problema.

Además, como veremos al estudiar la gestión de tareas e incidencias en el Capítulo 5, cada fallo encontrado en un proyecto puede convertirse en una nueva prueba de regresión. Si una incidencia describe un comportamiento incorrecto, una buena forma de cerrarla es añadir primero una prueba que reproduzca el fallo. Después corregimos el defecto y dejamos la prueba en la suite para evitar que el problema reaparezca en el futuro.

4.4.6. Cobertura de código y falsa sensación de seguridad

Hasta ahora hemos usado la idea de cobertura en un sentido amplio: cubrir particiones, límites, combinaciones u operaciones CRUD. Pero también existe una cobertura más ligada a la estructura del código.

Las herramientas de cobertura de código nos indican qué partes del código han sido ejecutadas durante las pruebas. En Python, por ejemplo, es habitual usar `coverage.py` o complementos como `pytest-cov`.

La cobertura puede ser muy útil para detectar zonas del código que nunca se ejercitan. Si una función crítica tiene 0 % de cobertura, probablemente deberíamos escribir alguna prueba. Si una rama de una condición nunca se ejecuta, quizá nos falta un caso importante.

Pero, ojo, porque la cobertura también puede ser engañosa. Ejecutar una línea no significa comprobar bien su comportamiento. Por eso conviene ver la cobertura como una herramienta de apoyo, no como un objetivo ciego. Un porcentaje alto de cobertura no demuestra que el sistema sea correcto. Aunque un porcentaje bajo sí puede indicar que hay zonas sin ejercitar.

En proyectos grandes, la cobertura resulta especialmente útil cuando se integra en el flujo de trabajo. En algunas empresas es habitual mostrar durante la revisión de código qué líneas añadidas o modificadas por un cambio han sido ejecutadas por las pruebas. Esta idea, conocida como *delta coverage*, ayuda a visualizar si el código que acabamos de tocar está respaldado por pruebas.

> **Para saber más**
>
> El artículo *Code coverage at Google*[a] analiza cómo utilizan la cobertura de código en esta empresa, donde calculan la cobertura diariamente sobre mil

> millones de líneas de código. Lee la sección 4.4, que incluye los resultados de una encuesta a más de 500 personas de los equipos de desarrollo. ¿Cómo perciben, en general, la utilidad de esta métrica?
>
> [a] https://research.google/pubs/code-coverage-at-google/

4.5. Desarrollo guiado por pruebas

Hasta ahora hemos tratado las pruebas como una forma de comprobar código que ya existe. Diseñamos casos de prueba, los automatizamos y los ejecutamos para ganar confianza en el sistema. Pero existe una práctica que cambia el orden habitual, y que es un poco llamativa la primera vez que se escucha. Esta práctica se conoce como desarrollo guiado por pruebas, o *Test Driven Development* (TDD), y propone escribir primero una prueba y después el código necesario para superarla.

TDD se asocia especialmente con Kent Beck, uno de los impulsores de *Extreme Programming* (XP) y autor del libro *Test-Driven Development: By Example*. El enfoque que plantea TDD es usar las pruebas para avanzar en pasos pequeños, concretando poco a poco el comportamiento esperado del sistema. Martin Fowler resume el ciclo básico como escribir una prueba para la siguiente funcionalidad, escribir el código necesario para que pase y refactorizar[6].

4.5.1. El ciclo rojo, verde y refactor

El ciclo clásico de TDD suele describirse con tres colores (figura 4.7):

1. **Rojo**: escribimos una prueba que falla, claro, porque la funcionalidad todavía no existe o no está completa.
2. **Verde**: escribimos el código mínimo necesario para que la prueba pase.
3. **Refactor**: mejoramos el diseño del código manteniendo las pruebas en verde.

Supongamos que queremos implementar usando TDD la función de la matrícula reducida con la que hemos estado trabajando en ejemplos anteriores. Podríamos empezar por un caso muy sencillo: una persona de 17 años tiene matrícula reducida. Primero escribiríamos una prueba que expresase ese comportamiento. Al principio, la prueba fallará, quizá porque la función ni siquiera existe. Después escribiremos el

[6] https://martinfowler.com/bliki/TestDrivenDevelopment.html

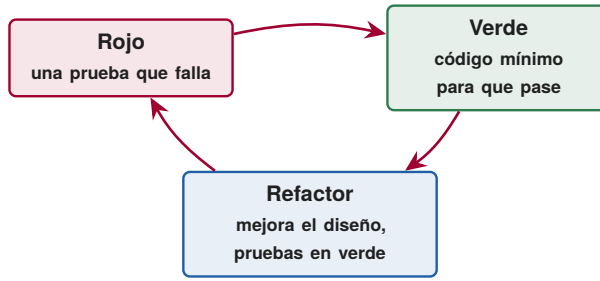


Fig. 4.7 El ciclo de TDD: *rojo* (una prueba que falla), *verde* (el código mínimo para que pase) y *refactor* (mejorar el diseño sin romper las pruebas), repetido en pasos pequeños.

código mínimo necesario para que pase. En este punto, incluso una implementación que devolviese siempre `True` sería suficiente para satisfacer el único caso que hemos especificado hasta ahora.

Una vez que la prueba está en verde, podemos revisar el código y preguntarnos si hay algo que convenga mejorar sin cambiar su comportamiento. En un ejemplo tan pequeño puede que todavía no haya nada que refactorizar, y eso también es perfectamente válido.

A continuación podemos añadir un segundo ejemplo: una persona de 30 años no tiene matrícula reducida. La implementación anterior ya no será suficiente, así que tendremos que generalizarla. Una posible solución mínima sería comprobar si la edad es menor de 18 años. Volvemos así a dejar todas las pruebas en verde y revisamos de nuevo si merece la pena refactorizar.

Después añadimos otro comportamiento relevante: una persona de 65 años sí tiene matrícula reducida. Este nuevo ejemplo obliga a ampliar la regla anterior para contemplar también a las personas de 65 años o más. La implementación va evolucionando así a medida que incorporamos nuevos ejemplos, en lugar de intentar resolver desde el principio todos los casos posibles.

Fíjate en que el proceso obliga a concretar el comportamiento esperado paso a paso. Cada prueba introduce un ejemplo concreto, cada nuevo ejemplo puede forzar una pequeña generalización de la implementación y, cuando el código lo necesita, la fase de refactorización permite mejorar su diseño sin modificar su comportamiento. Las pruebas proporcionan la confianza necesaria para realizar esos cambios de forma segura.

4.5.2. TDD también requiere diseño

A veces se presenta TDD como si eliminara la necesidad de diseñar. Pero eso sería una mala interpretación. En realidad, TDD exige mucho diseño, solo que lo reparte en pasos pequeños.

Antes de escribir pruebas conviene pensar en una lista de comportamientos que queremos cubrir. También tenemos que decidir en qué orden abordarlos. En ese sentido, TDD puede verse como una conversación continua entre requisitos, ejemplos, código y diseño.

Además, escribir la prueba primero nos obliga a pensar en cómo queremos usar el código. Antes de implementar una función tenemos que decidir cómo se llamará, qué parámetros recibirá, qué devolverá y qué comportamiento esperamos en un caso concreto. La prueba actúa como un primer cliente de nuestro diseño.

Esto puede ayudarnos a escribir código más testeable. Si una función es difícil de probar, quizá está mezclando demasiadas responsabilidades. Por ejemplo, una función que lee datos de teclado, consulta una base de datos, calcula un resultado y lo imprime por pantalla será más difícil de probar que una función que recibe datos y devuelve un valor. TDD nos empuja a separar mejor esas responsabilidades, porque necesitamos poder ejercitar el comportamiento desde una prueba automatizada.

Esta idea será especialmente importante cuando trabajemos con aplicaciones más complejas, como puede ser UVLHub. Si mezclamos lógica de negocio, acceso a base de datos, llamadas HTTP y presentación de resultados en la misma función, las pruebas serán más difíciles de escribir y mantener. Si separamos responsabilidades, el sistema será más testeable.

4.5.3. Qué sabemos realmente sobre TDD

TDD se ha defendido muchas veces como una práctica capaz de mejorar la calidad del software, reducir errores y favorecer diseños más simples. También ha recibido críticas. Hay equipos que lo consideran una disciplina fundamental y otros que lo ven como una práctica costosa, rígida o difícil de aplicar en determinados contextos[7].

Parte del debate se debe a que no siempre se entiende TDD de la misma manera. A veces se reduce a *test-first*, es decir, a escribir la prueba antes que el código.

[7] De hecho, TDD levanta debates bastante polarizados entre la comunidad. Podría incluso considerarse como un *flame war*. Así que si un día te invade el aburrimiento y estás con más colegas informáticos, puedes preguntar inocentemente qué piensan de TDD. Te auguramos un ratito muy entretenido.

Pero TDD defiende una visión más amplia: implica ciclos cortos, tareas pequeñas, ejecución frecuente de pruebas, código mínimo para avanzar y refactorización continua.

En el artículo *What Do We (Really) Know about Test-Driven Development?*[8], Itir Karac y Burak Turhan revisan la evidencia empírica disponible sobre TDD y señalan dos cosas muy interesantes. Por un lado, afirman que no hay evidencias sólidas de que TDD mejore de forma consistente la productividad. Y por otro, concluyen que los efectos positivos que TDD puede tener sobre la calidad del software, quizá no se deban solo a escribir las pruebas antes, sino a otros elementos del proceso: trabajar en tareas pequeñas y bien definidas, mantener ciclos de desarrollo cortos y estables, escribir solo el código necesario para superar la siguiente prueba y refactorizar con frecuencia.

Esta conclusión encaja bien con el enfoque de este capítulo. Escribir primero una prueba no garantiza que el diseño sea bueno, que el caso elegido sea relevante o que el sistema esté correctamente validado. Pero TDD puede ayudarnos porque nos obliga a trabajar con ejemplos concretos, recibir retroalimentación rápida y mantener una suite de pruebas ejecutable.

También hay que tener en cuenta que existen contextos donde aplicar TDD puede ser más difícil: interfaces gráficas, sistemas con mucho hardware, código legado muy acoplado, prototipos exploratorios o funcionalidades cuya solución todavía no entendemos bien. En esos casos, quizá necesitemos explorar primero, construir una pequeña prueba de concepto o diseñar mejor las fronteras del sistema antes de escribir pruebas útiles.

Por tanto, aunque TDD se utiliza en muchos contextos industriales y puede ser una práctica muy valiosa cuando se aplica con criterio, no es la única forma de llegar a una buena suite de pruebas. Martin Fowler recoge parte de este debate en la serie de conversaciones *Is TDD Dead?*, con Kent Beck y David Heinemeier Hansson[9]. Aunque el título sea un poco provocador, una conclusión interesante que presentan es que el objetivo siempre debería ser tener un código funcional, bien diseñado y acompañado de pruebas que permitan mantenerlo con confianza, independientemente de la técnica utilizada para lograrlo.

> **Para saber más**
>
> En la PyConEs de 2018 se realizó un taller de introducción a TDD con Python muy interesante. En este repositorio[a] puedes acceder a las diapositivas que

[8] https://www.cse.unr.edu/~dascalus/Paper_JAMES.pdf

[9] https://martinfowler.com/articles/is-tdd-dead/

se usaron para guiar el taller, a los enunciados de los retos prácticos y a las soluciones propuestas, además de otros recursos sobre TDD. Si te animas, puedes elegir uno de los retos y pensar cómo lo resolverías siguiendo el ciclo rojo-verde-refactor. Después puedes comparar tu solución con la propuesta.

[a] https://github.com/aleasoluciones/pycones2018

4.6. ¿Y cómo afecta la IA a todo esto?

Los agentes de IA están empezando a cambiar de forma significativa algunas tareas relacionadas con el diseño, generación y evaluación de pruebas. Pueden ayudarnos a explorar más casos, detectar huecos en nuestras suites, generar pruebas a partir de cambios concretos y revisar código dentro del propio pipeline, entre otros casos (figura 4.8).

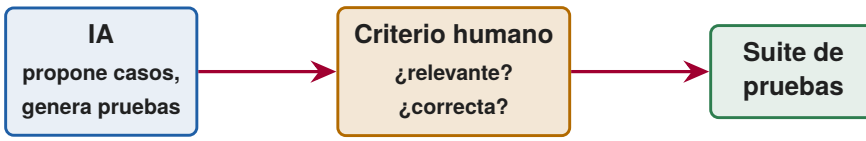


Fig. 4.8 La IA puede proponer muchos casos de prueba, pero sigue siendo el criterio humano quien decide cuáles son relevantes y correctos antes de incorporarlos a la suite.

Un primer uso relativamente natural consiste en usar modelos de lenguaje para ayudar en la planificación de pruebas. Mozilla, por ejemplo, ha estudiado el uso de LLMs para generar planes de prueba de funcionalidades de Firefox[10]. Los equipos diseñan planes de prueba manualmente, pero en sistemas grandes y con muchos escenarios posibles siempre existe el riesgo de olvidar casos relevantes. Así que utilizan LLMs para proponer escenarios adicionales. Algunas recomendaciones son redundantes, otras inválidas, pero alrededor del 27 % son recomendaciones que el equipo no había considerado y que resultan valiosas.

Un segundo uso más avanzado aparece cuando combinamos IA con técnicas como el *mutation testing*. Meta describe una herramienta llamada ACH, *Automated Compliance Hardening*, que usa LLMs para generar mutantes relevantes y después

[10] https://www.mozillafoundation.org/en/research/library/using-llms-to-bridge-the-gaps-in-qa-test-plans-at-firefox/

crear pruebas capaces de detectarlos[11]. La idea básica es utilizar la herramienta para construir fallos simulados más realistas y pruebas que endurezcan el sistema frente a regresiones futuras.

Un tercer caso es el de Cloudflare[12]. En su sistema de revisión de código con IA, cuando una persona abre una *merge request*, varios agentes especializados revisan aspectos como seguridad, rendimiento, calidad del código, documentación o cumplimiento de normas internas. La revisión se integra en el proceso de CI/CD, clasifica riesgos, coordina agentes y puede llegar a bloquear cambios cuando detecta problemas serios.

Estos ejemplos muestran una tendencia clara en la industria, en la que los sistemas de IA pueden participar en varias fases del proceso de pruebas: diseño de casos, generación de pruebas, análisis de cobertura, revisión de cambios, detección de riesgos y endurecimiento frente a regresiones.

Pero conviene mantener la misma prudencia que hemos aplicado durante todo el capítulo. Un agente de IA puede sugerir pruebas incorrectas, inventar problemas, pasar por alto riesgos importantes o producir una falsa sensación de seguridad. Por eso el papel del equipo sigue siendo fundamental a la hora de revisar las propuestas, decidir qué riesgos importan, validar los oráculos y mantener una suite de pruebas útil. Es decir, que aunque los agentes IA puedan ampliar nuestra capacidad para probar, no eliminan ni mucho menos la responsabilidad de pensar.

4.7. Reflexión final: probar es gestionar riesgo

En este capítulo hemos visto que las pruebas no son una actividad secundaria que aparece cuando el software ya está escrito, como lo fue durante una época en la historia de la informática. En la actualidad, las pruebas forman parte del proceso completo de construir, integrar, entregar y mantener software.

Las pruebas software no permiten demostrar que el sistema sea perfecto, por supuesto. Pero sí nos ayudan a diseñar y ejecutar comprobaciones que aumenten nuestra confianza, revelen fallos importantes lo antes posible y nos permitan cambiar el código sin miedo (figura 4.9).

También hemos visto que las pruebas tienen una parte automatizable y una parte humana. Podemos automatizar la ejecución, la comparación de resultados y la integración con el pipeline. Pero decidir qué casos probar, qué riesgos son más

[11] https://engineering.fb.com/2025/09/30/security/llms-are-the-key-to-mutation-testing-and-better-compliance/

[12] https://blog.cloudflare.com/ai-code-review/

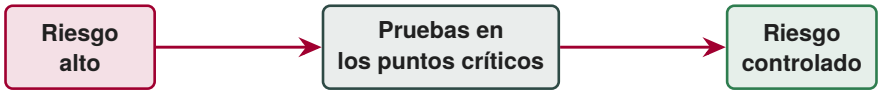


Fig. 4.9 Probar es gestionar riesgo: no elimina toda la incertidumbre, pero concentrar las pruebas en los puntos críticos lo reduce a un nivel que el equipo puede asumir con confianza.

importantes, qué nivel de prueba conviene usar o qué comportamiento se considera aceptable sigue requiriendo criterio.

Esta idea conecta directamente con los capítulos anteriores. En el Capítulo 2 vimos que la integración continua necesita una batería de pruebas que dé feedback rápido. En el Capítulo 3 vimos que la historia del código debe permitir entender y coordinar cambios; las pruebas forman parte de esa historia, porque documentan comportamientos esperados y ayudan a detectar regresiones. Y en el Capítulo 6 veremos que muchas pruebas solo son fiables si el entorno donde se ejecutan puede reproducirse correctamente.

También conecta con lo que estudiaremos sobre gestión de tareas e incidencias en el Capítulo 5. Cuando aparece un fallo, una buena práctica consiste en escribir primero una prueba que lo reproduzca. Después corregimos el defecto y dejamos la prueba en la suite para evitar que el problema reaparezca en el futuro. Así, cada incidencia importante puede convertirse en una mejora permanente del detector de errores del proyecto.

Por tanto, una suite de pruebas es una herramienta de gestión del riesgo. Nos ayuda a decidir si un cambio puede integrarse, si una release puede considerarse liberable o si una corrección ha resuelto realmente un problema.

Pero no olvides la advertencia principal del capítulo: una suite grande y mal diseñada puede dar una falsa sensación de seguridad. En cambio, una suite razonable, rápida, mantenible y bien pensada puede cambiar completamente la forma de trabajar del equipo.

Para saber más

Mercadona Tech ha publicado tres artículos divulgativos [a] [b] [c] en los que presentan cuáles son las nueve *verdades universales* que guían sus procesos de desarrollo de software. Lee los artículos prestando especial atención a cómo las prácticas de pruebas y CI/CD tienen un impacto muy importante en muchas de sus verdades universales.

[a] https://www.linkedin.com/pulse/las-verdades-universales-de-mercadona-en-el-desarrollo-sotfware-xaxof/

[b] https://www.linkedin.com/pulse/las-verdades-universales-de-mercadona-en-el-desarrollo-sotfware-aooff/
[c] https://www.linkedin.com/pulse/las-verdades-universales-de-mercadona-en-el-desarrollo-software-gatpe

4.8. Lecturas y recursos recomendados

- Capítulo 8, *Testing*, del libro *Software Engineering: A Modern Approach* (Marco Tulio Valente).
- Capítulo 6, *Testing*, del libro *The Missing README: A Guide for the New Software Engineer* (Chris Riccomini y Dmitriy Ryaboy).
- Libro *Introduction to Software Testing* (Paul Ammann y Jeff Offutt).
- Libro *The Art of Software Testing* (Glenford J. Myers, Corey Sandler y Tom Badgett).
- Libro *The Craft of Software Testing* (Brian Marick).
- Artículo *The Practical Test Pyramid* (Ham Vocke, en la web de Martin Fowler).

Capítulo 5

Gestión de tareas e incidencias

La informática trata de personas y redes en todos los niveles.
Wendy Hall.

Resumen En este capítulo estudiaremos cómo organizar tareas, peticiones de cambio e incidencias dentro de un proyecto software. Veremos que el cambio es inevitable, pero que, si no se gestiona de forma explícita, puede generar confusión, duplicidades, prioridades poco claras y pérdida de trazabilidad. Para evitarlo, los equipos usan gestores de tareas e incidencias, tableros de trabajo, estados, prioridades, tipos de cambio y roles. Después nos centraremos en las incidencias, analizando cómo informar bien de un problema y cómo abordarlo mediante un ciclo de depuración: reproducir, diagnosticar, reparar y aprender de lo ocurrido. Finalmente reflexionaremos sobre cómo los sistemas de IA pueden ayudar a describir, clasificar, reproducir y diagnosticar incidencias, aunque sin sustituir el criterio técnico del equipo.

5.1. Gestionar el cambio para no perder el control del proyecto

En los capítulos anteriores hemos hablado mucho de cambios. En el Capítulo 2 vimos que cada cambio debería poder construirse, probarse, empaquetarse, entregarse y, llegado el caso, desplegarse de forma fiable. En el Capítulo 3 vimos que Git permite conservar la historia del código y coordinar el trabajo del equipo. Y en el Capítulo 4 vimos que las pruebas nos ayudan a saber si un cambio ha roto algo que antes funcionaba.

Pero, ¿de dónde salen esos cambios? ¿Quién decide que hay que hacerlos? ¿Cómo sabemos qué problema intentan resolver? ¿Cómo se priorizan? ¿Cómo evitamos olvidar una incidencia que alguien detectó hace dos semanas? ¿Cómo relacionamos una conversación con el código que finalmente se modifica?

Estas preguntas son el punto de partida de este capítulo. Porque una parte importante del desarrollo de software consiste en decidir qué cambios merecen atención, organizarlos, comunicarlos, trazarlos y cerrarlos con criterio.

5.1.1. El cambio es inevitable

En un proyecto software el cambio es la situación normal y habitual. Puede aparecer porque una persona usuaria pide una nueva funcionalidad. Quizá porque el equipo descubre que una pantalla resulta confusa. O porque alguien se da cuenta de que cierta parte del código se ha vuelto difícil de modificar, por ejemplo.

Pero muchos cambios que llegan al equipo también están relacionados con la configuración. Puede que un mecanismo de monitorización detecte en producción un fallo que no apareció en otros entornos. Puede que una biblioteca necesaria deje de mantenerse. Una actualización de una dependencia puede romper una prueba. O quizá una combinación concreta de opciones activadas y desactivadas produzca un comportamiento inesperado. De hecho, un trabajo reciente de un equipo de la Universidad de Sevilla[1] estudió más de 300.000 informes de errores de proyectos de Eclipse y estimó que alrededor del 36 % estaba relacionado con cuestiones de configuración, en línea con otros estudios anteriores.

Incluso en una práctica de clase, donde el alcance parece mucho más controlado, seguro que has vivido esa sensación de que siempre hay algo que ajustar, corregir o volver a comprobar[2].

En consecuencia, el software evoluciona mediante cambios. Algunos son pequeños y rutinarios, otros son urgentes, otros pueden ser exploratorios, otros son correcciones, e incluso hay otros que son cambios técnicos que casi nadie ve desde fuera, pero que facilitan que el sistema pueda seguir evolucionando.

Y aunque, como decíamos, esta es la situación natural de cualquier proyecto software, si no se gestionan bien los cambios, pueden aparecer problemas (figura 5.1).

[1] https://dl.acm.org/doi/10.1145/3744915.3748477

[2] Y es que, como afirma John Allspaw, el modo en el que las cosas salen bien es solo un caso especial de los modos en los que salen mal: https://sre.google/sre-book/effective-troubleshooting/.

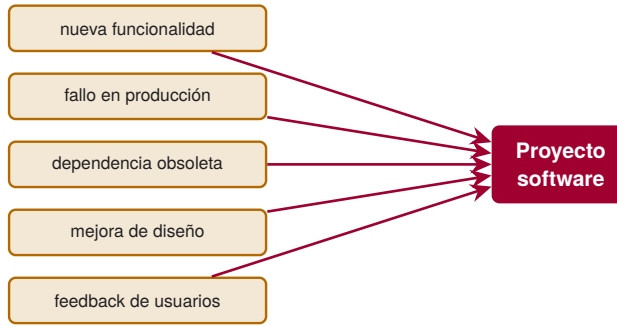


Fig. 5.1 El cambio es la situación normal de un proyecto: llega de muchas fuentes a la vez. Gestionarlo con método es lo que evita perder el control.

5.1.2. Cuando los cambios se gestionan de manera informal

Imagina un equipo que organiza su trabajo únicamente mediante mensajes sueltos en un grupo de mensajería. Una persona escribe: "habría que revisar lo de la descarga". Otra responde: "sí, creo que falla con algunos datasets". Alguien más dice: "lo miro luego". Dos días después otra persona pregunta: "¿esto estaba arreglado?". Y quizá en ese punto nadie está completamente seguro.

En proyectos pequeños, esta forma de trabajar puede parecer suficiente durante un tiempo. Pero pronto empiezan los problemas. Algunas tareas se olvidan. Otras se duplican. Las prioridades cambian sin que quede constancia. Una incidencia se comenta en una reunión, pero no se registra en ningún sitio. Un commit modifica una parte importante del código, pero nadie sabe qué necesidad lo motivó. Y cuando una persona deja el equipo, es probable que con ella se pierda información que nunca se documentó.

También puede ocurrir que la información esté repartida en demasiados lugares: un mensaje en chat, una nota en una libreta, una conversación en un pasillo, un correo, un comentario en una revisión de código y un commit con un mensaje poco descriptivo. En ese caso, aunque la información exista, reconstruir la historia del cambio se vuelve muy costoso.

Y esto afecta directamente a la integración continua. Si el equipo no sabe qué cambios están en marcha, cuáles están bloqueados, qué incidencias son críticas o qué tareas están listas para revisar, entonces será más difícil mantener sana la línea principal. Y por mucho que tengamos automatizado un pipeline que nos indique si una construcción falla, no podremos saber qué trabajo tenía prioridad o por qué se decidió modificar cierto comportamiento.

Por eso los proyectos software deben apoyarse en sistemas de gestión de tareas e incidencias, como un instrumento que nos permite contar con un lugar compartido donde hacer visible y coordinar el trabajo.

5.1.3. El gestor de tareas e incidencias dentro del ecosistema de desarrollo

Un gestor de tareas e incidencias es una herramienta que permite registrar, organizar y seguir unidades de trabajo. Dependiendo del contexto, estas unidades pueden llamarse *issues*, *tickets*, tareas, historias, incidencias, peticiones de cambio o *work items*. La terminología cambia mucho entre organizaciones y herramientas, pero la idea general es parecida: cada tarjeta o elemento describe algo que el equipo debe entender, decidir, hacer o verificar.

Herramientas como GitHub Issues, GitLab Issues, Jira, Azure DevOps, YouTrack o Redmine permiten crear estos elementos, asignar responsables, etiquetarlos, priorizarlos, moverlos entre estados, comentarlos y relacionarlos con commits, ramas, revisiones de código o releases.

Esto último es especialmente importante para esta asignatura. Un gestor de incidencias debe formar parte del ecosistema de desarrollo junto al repositorio de código, el sistema de integración continua, las pruebas, los artefactos, los requisitos y las releases. Si una incidencia se corrige mediante una rama, una petición de merge y una prueba nueva, debería ser posible reconstruir esa relación más adelante.

Por ejemplo, ante una pregunta como "¿por qué se cambió la validación de este formulario?", una buena trazabilidad permitiría responder algo así: se detectó la incidencia #238, afectaba a usuarios que introducían fechas límite, se reprodujo con estos pasos, se corrigió en esta rama, se revisó en esta pull request, se añadieron estas pruebas y la corrección entró en la release 1.4.3.

Ese tipo de trazabilidad es la memoria del proyecto, como ya adelantábamos en el Capítulo 3 (figura 5.2).

> **Para saber más**
>
> Explora la incidencia #14533 del repositorio de *pytest*[a] y trata de reconstruir su historia: quién la abrió, qué problema describía, qué discusión se produjo, qué pull request la cerró y qué cambios de código se realizaron. No hace falta que entiendas el código; céntrate en observar cómo se conecta la conversación con la historia del repositorio.

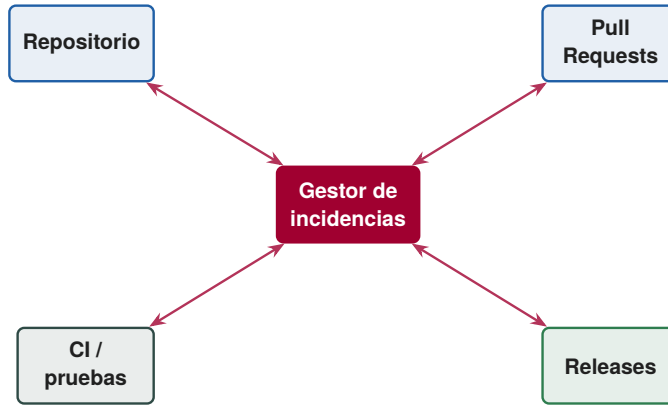


Fig. 5.2 El gestor de incidencias no vive aislado: se conecta con el repositorio, las *pull requests*, la CI y las releases. Esa red de relaciones es la que da trazabilidad al proyecto.

[a] https://github.com/pytest-dev/pytest/issues/14533

5.2. De una petición de cambio a una unidad de trabajo

Una de las ideas más importantes de este capítulo es que no todo lo que alguien pide debería convertirse inmediatamente en código.

Una persona puede pedir una funcionalidad, informar de un problema, sugerir una mejora o plantear una duda. Pero entre esa señal inicial y el trabajo que finalmente realiza el equipo hay un proceso de interpretación. Tenemos que entender qué se está pidiendo, por qué, a quién afecta, qué coste puede tener, si encaja con los objetivos del producto y cómo sabremos que está terminado.

En esta sección vamos a ver precisamente cómo llegamos a una tarea para el equipo desde que se recibe una petición (figura 5.3).

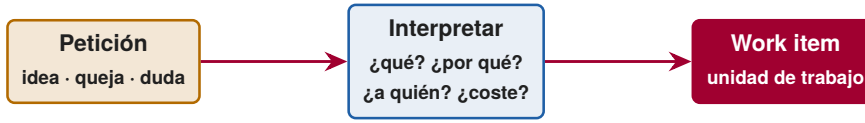


Fig. 5.3 No toda petición se convierte directamente en código. Entre la señal inicial y el trabajo del equipo hay un paso de interpretación que la transforma en una unidad de trabajo bien definida.

5.2.1. Work items, tareas e incidencias

En este capítulo usaremos las expresiones *work item*, tarea o elemento de trabajo de forma bastante cercana. Nos referiremos con ellas a cualquier unidad de trabajo que el equipo decide hacer visible y gestionar: una nueva funcionalidad, una mejora técnica, una tarea de documentación, una refactorización pendiente o una investigación. Por ejemplo una tarea podría ser: "añadir validación del campo de teléfono en el perfil de usuario".

Las incidencias son un caso particular de esos elementos de trabajo que describen un problema observado. Por ejemplo: "al guardar el perfil de usuario con el campo de teléfono vacío, el sistema devuelve un error 500". La incidencia, por tanto, es la descripción de una discrepancia entre lo que ocurre y lo que debería ocurrir.

Una petición de funcionalidad, o *feature request*, es una solicitud para añadir una nueva capacidad al producto o mejorar una existente. Este tipo de peticiones puede llegar por muchos canales: comentarios de usuarios, soporte, ventas, encuestas, reseñas de aplicaciones, conversaciones con clientes o formularios dentro del propio producto. Pero que una petición llegue no significa que haya que implementarla tal cual, evidentemente.

Por ejemplo, supongamos que un usuario de UVLHub pide: "quiero descargar todos los datasets en varios formatos". Esto suena razonable, pero deja varias preguntas abiertas. ¿Quién necesita esa descarga? ¿Para qué la quiere usar? ¿Qué formatos importan realmente? ¿Hay datasets muy grandes? ¿Debe descargarse todo en un único archivo? ¿Qué ocurre si un formato no se puede generar para algún modelo? ¿Cómo sabremos que la funcionalidad resuelve el problema?

Si saltamos directamente a programar, podemos invertir mucho esfuerzo en construir algo que quizá no sea lo que las personas necesitaban. Por eso muchas veces las peticiones de funcionalidad exigen un trabajo de exploración previo.

5.2.2. Peticiones de funcionalidad y exploración previa

Algunas peticiones de funcionalidad llegan suficientemente claras como para convertirse pronto en una tarea. Otras contienen demasiada incertidumbre. En esos casos puede ser más sensato realizar antes una actividad breve de exploración.

Un ejemplo conocido de este tipo de actividad es el *design sprint*, propuesto por Jake Knapp en el libro *Sprint*. En este contexto, la palabra *sprint* no se refiere a una iteración Scrum, sino a una actividad de unos cinco días para entender un problema, generar soluciones, decidir una dirección, construir un prototipo y hacer una evaluación. La idea básica es que si una funcionalidad es arriesgada, está poco

definida o puede tener mucho impacto, puede ser más inteligente aprender sobre ella antes de implementarla.

Durante el proceso, como apuntábamos, se define un objetivo, se construye un mapa del recorrido del usuario, se generan soluciones, se decide una propuesta, se crea un prototipo realista y se valida con personas usuarias. El prototipo no tiene que ser el producto real. De hecho, muchas veces lo importante es que sea una fachada suficientemente creíble para obtener información útil.

Esto conecta muy bien con la gestión de tareas. El resultado de una exploración así puede ser una lista de work items más concretos, criterios de aceptación más claros, decisiones de diseño o incluso la decisión de no construir la funcionalidad. Y esto último también es valioso. No implementar una idea después de aprender que no resuelve el problema adecuado puede ahorrar mucho trabajo al equipo.

En proyectos reales, por tanto, la gestión de cambios no se limita a meter todas las peticiones que se reciben en una cola. Debemos filtrar, agrupar, refinar y, a veces, experimentar antes de comprometer desarrollo.

5.2.3. Estados, prioridades, tipos y roles

Para que un proceso de gestión de cambios sea útil, cada work item debería tener al menos cierta información básica.

En primer lugar, necesitamos conocer su estado. No es lo mismo una incidencia recién creada que una tarea aceptada, una funcionalidad en desarrollo, una corrección pendiente de revisión o un cambio ya verificado. Un flujo sencillo podría usar estados como *nuevo*, *aceptado*, *en curso*, *corregido*, *verificado* y *cerrado*, con una salida lateral para lo que se rechaza o resulta estar duplicado (figura 5.4). Otros equipos usan columnas como *pendiente*, *en progreso*, *en revisión*, *bloqueado* y *hecho*. Los nombres dependen de la metodología que se siga, pero lo importante es que todo el equipo comparta qué significa cada estado.

Además necesitamos algún criterio de prioridad. Para ello hay que considerar cuestiones como el impacto del cambio, el número de personas afectadas, el riesgo, la urgencia, el coste estimado, la alineación con los objetivos del producto o las dependencias con otros trabajos.

También conviene identificar el tipo de cambio. Porque el equipo no va a gestionar igual una nueva funcionalidad, una corrección de error, una refactorización, una tarea de documentación, una actualización de dependencias o una mejora de rendimiento. El tipo de tarea nos ayuda a filtrar, organizar y analizar el trabajo.

Y, por último, necesitamos roles. Alguien informa o propone el cambio. Alguien lo analiza. Alguien decide si se acepta, se rechaza o se difiere. Alguien lo imple-

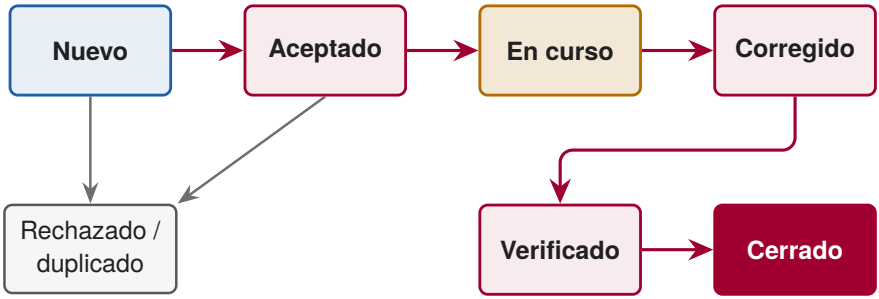


Fig. 5.4 Un ciclo de vida típico de una incidencia. Los nombres concretos varían según la metodología, pero el equipo debe compartir qué significa cada estado.

menta. Alguien revisa el código. Alguien verifica que el resultado es correcto. En equipos pequeños una misma persona puede asumir varios de estos roles, pero las responsabilidades deben estar claras.

Algunas incidencias pueden acabar cerrándose sin cambios de código porque son duplicadas, porque no se pueden reproducir con la información disponible, porque describen el comportamiento esperado o porque el equipo decide explícitamente no corregirlas. Lo importante es que esa decisión quede registrada y justificada.

5.2.4. Tableros para hacer visible el trabajo

Un tablero de trabajo, o tablero *kanban*, representa visualmente el estado de los work items. Normalmente cada tarjeta o elemento del tablero representa una unidad de trabajo y cada columna representa un estado. Al mover una tarjeta entre columnas, el equipo hace visible cómo avanza el trabajo (figura 5.5).

Y aunque la idea pueda parecer muy simple, en realidad ofrece bastante valor. Un tablero bien organizado permite detectar de un vistazo acumulación de trabajo en una fase, tareas bloqueadas o demasiadas cosas empezadas a la vez, por ejemplo. También ayuda a coordinar reuniones breves de seguimiento, porque el equipo puede revisar el tablero y preguntarse qué necesita atención.

Por supuesto, el tablero no hace el trabajo por nosotros. De poco sirve contar con un tablero si las tarjetas están mal descritas, no se actualizan los estados, o las prioridades no se revisan. La herramienta puede ayudarnos, pero el criterio y la responsabilidad deben seguir siendo del equipo.

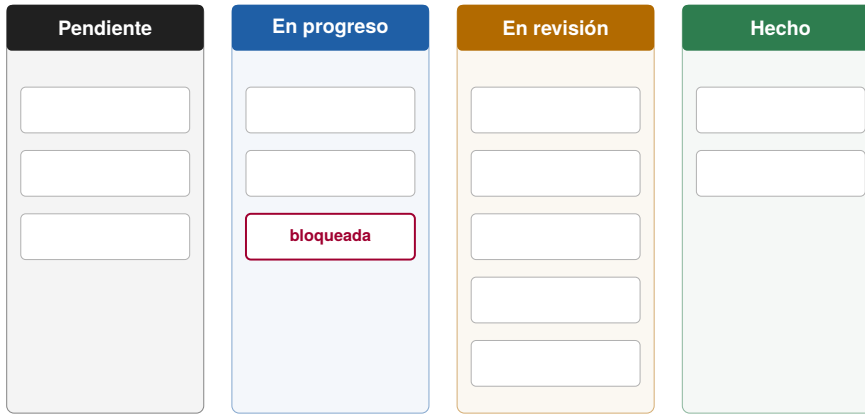


Fig. 5.5 Un tablero kanban hace visible el flujo de trabajo: cada tarjeta es una unidad de trabajo y cada columna, un estado. De un vistazo se detecta acumulación o tareas bloqueadas.

5.2.5. Trazabilidad entre tarea, código y validación

Como mencionábamos en la introducción del capítulo, la gestión de tareas se vuelve especialmente útil cuando se conecta con el resto del proceso de desarrollo.

Dependiendo de la estrategia de gestión del repositorio del equipo, una tarea podría dar lugar a una rama, por ejemplo. Esa rama puede contener varios commits. Esos commits pueden abrir una petición de merge. La petición puede lanzar el pipeline de CI. El pipeline puede ejecutar linters, pruebas y empaquetado. La revisión de código puede discutir decisiones de diseño. Y, si todo va bien, el cambio puede integrarse en la línea principal y formar parte de una release (figura 5.6).

Fig. 5.6 Trazabilidad de extremo a extremo: conectar tarea, rama, commits, petición de merge, CI y release permite responder qué incidencia arregla un cambio o qué tareas incluye una versión.

Cuando todo esto queda conectado, el equipo puede responder preguntas importantes como qué tareas incluye una release, qué incidencia arregla un commit, qué pruebas se añadieron para evitar una regresión, qué cambios están bloqueados porque falla CI, o qué funcionalidad se entregó finalmente, por ejemplo.

Es por ello que la gestión de tareas e incidencias forma parte de la gestión de la configuración. Nos ayuda a relacionar necesidades, decisiones, código, pruebas y

artefactos. Sin esa relación, aunque el proyecto pueda seguir avanzando, resultará muy complicado y costoso explicar su historia.

5.3. Informar bien de una incidencia

Como se ha definido anteriormente, una incidencia describe un comportamiento problemático observado en el sistema. Puede ser un fallo funcional, un problema de rendimiento, una inconsistencia visual, una cuestión de seguridad, una excepción inesperada o un comportamiento que contradice lo que el usuario esperaba.

Pero para poder ponernos a trabajar en solucionar una incidencia, es necesario que la persona que la reporta lo haga con un informe de calidad. Y cuando esto no ocurre se genera frustración y se producen retrasos.

5.3.1. Una incidencia mal descrita también cuesta tiempo

Imagina que alguien abre una incidencia con este texto:

> No funciona la descarga.

¿No funciona nunca? ¿Solo con algunos datasets? ¿En qué navegador? ¿Aparece algún mensaje de error? ¿La descarga empieza y se corta? ¿El botón no responde? ¿El archivo se genera vacío? ¿O quizá el problema es que el nombre del archivo no es el esperado?

Con una descripción así, la persona que intenta corregir el problema tiene que dedicar tiempo a adivinar qué significa exactamente "no funciona". Puede que pregunte por más información y tenga que esperar respuesta. Puede que intente reproducir el problema de varias formas y no lo consiga. O puede que investigue un comportamiento distinto al que la persona informante había observado.

En el artículo *What Makes a Good Bug Report?*[3], sus autores encuestaron a desarrolladores y personas que informaban errores en proyectos como Apache, Eclipse y Mozilla. Una de sus conclusiones principales es que existe un desajuste entre la información que los desarrolladores consideran más útil y la información que quienes reportan errores suelen proporcionar. En particular, los pasos para reproducir, las trazas de pila y los casos de prueba son muy valiosos para quienes corrigen, pero también son de los elementos más difíciles de aportar para muchos usuarios.

[3] https://doi.org/10.1145/1453101.1453146

Esto no significa que las personas usuarias sean descuidadas o perezosas. En realidad muchas veces simplemente no saben qué información necesita el equipo técnico. Y por eso el sistema de gestión de incidencias debería ayudar a recoger esa información.

5.3.2. Qué debe incluir un buen informe de incidencia

Un buen informe de incidencia debería reducir el esfuerzo necesario para que otra persona pueda observar el fallo, entenderlo y empezar a investigarlo.

En la práctica, suele ser útil incluir al menos los siguientes elementos:

- Un resumen breve y específico.
- El contexto en el que aparece el problema.
- Los pasos necesarios para reproducirlo.
- El comportamiento esperado.
- El comportamiento observado.
- Información del entorno: versión, navegador, sistema operativo, configuración relevante o datos utilizados.
- Mensajes de error, trazas, capturas de pantalla o archivos de ejemplo, cuando aporten información útil.

Para ayudar a que las incidencias incluyan la información que necesitamos, es habitual utilizar plantillas. Una plantilla sencilla podría tener esta forma (con datos de ejemplo ya rellenos):

Plantilla básica de incidencia

```
Resumen:
    Al guardar el perfil sin teléfono, aparece un error 500.

Contexto:
    Usuario autenticado en la versión 1.8.2 de UVLHub.

Pasos para reproducir:
    1. Hacer login.
    2. Ir a Profile.
    3. Borrar el campo Phone.
    4. Pulsar Save profile.

Resultado esperado:
```

```
    El perfil se guarda correctamente o se muestra un mensaje de
    ↪ validación.

Resultado observado:
    El sistema muestra un error 500.

Entorno:
    Firefox 128, Ubuntu 24.04, entorno local con Docker.

Información adicional:
    Se adjunta captura y fragmento del log.
```

Para saber más

Revisita la incidencia #14533 del repositorio de *pytest*[a]. ¿Incluye toda esta información aproximadamente? A continuación puedes echar un ojo a las tres plantillas del proyecto para reportar distintos tipos de incidencias[b]. ¿Qué información se solicita en cada caso?

[a] https://github.com/pytest-dev/pytest/issues/14533

[b] https://github.com/pytest-dev/pytest/tree/main/.github/ISSUE_TEMPLATE

En ocasiones un sistema de gestión de incidencias puede ayudar a recoger parte de esta información automáticamente. Por ejemplo, puede incluir en un informe el navegador, la versión del cliente, el identificador de la release, el usuario afectado, la ruta visitada o un identificador de correlación para buscar logs. Esto reduce la carga sobre quien reporta y disminuye el riesgo de información incorrecta.

En cualquier caso, debemos recordar que al otro lado de una incidencia suele haber una persona. Puede estar frustrada, puede no tener conocimientos técnicos o puede no saber explicar lo ocurrido. Y además hay que tener en cuenta que, por cada error que reporta un usuario, se estima que puede haber entre 10 y 100 usuarios que lo han sufrido y no lo han reportado. Así que tratar bien a quien lo hace también forma parte del proceso. Una incidencia escrita con poca precisión no debería interpretarse automáticamente como falta de interés. A veces es una señal de que nuestras plantillas, mensajes de error o mecanismos de observabilidad no están ayudando lo suficiente.

5.3.3. Ser específico, único y reproducible

Como venimos insistiendo, una incidencia debería ser específica. No es lo mismo escribir "sale una excepción al descargar un dataset" que indicar que "al pulsar el botón de descarga en un dataset con más de un modelo, usando Firefox, aparece la excepción `OverflowException`". La segunda descripción permite empezar a investigar en una dirección concreta.

También debería ser lo menos ambigua posible. Si hablamos de "el modelo grande", "la pantalla rara" o "el archivo que descargué ayer", estamos obligando a otras personas a interpretar demasiado. Conviene usar identificadores, nombres concretos, versiones y datos reproducibles.

Además, una incidencia debería ser única. Si el mismo problema ya existe en el gestor, puede ser mejor añadir información al informe existente que abrir otro nuevo. Dicho esto, los duplicados no siempre son inútiles. A veces una segunda persona aporta una captura, un entorno o unos pasos que ayudan a entender mejor el problema. Por eso, si se detecta un duplicado, lo razonable es cerrar o fusionar dejando constancia de la relación.

Y, por encima de todo, una buena incidencia debería ayudar a reproducir. Si no podemos observar el fallo, es mucho más difícil diagnosticarlo y demostrar que lo hemos corregido. Como veremos en la siguiente sección, reproducir es el primer paso de una depuración con método.

5.4. Depurar una incidencia con método

La depuración suele imaginarse como una actividad casi detectivesca: tras recibir una incidencia, alguien mira el código, ejecuta el programa, pone puntos de ruptura, revisa logs y, después de un rato, exclama algo así como "¡ah, ya lo tengo!". Y un poco de eso hay, la verdad. Pero podemos seguir métodos que nos ayuden a no depender solo de nuestra inspiración.

En el libro *Debug It!*, Paul Butcher insiste en que depurar no es simplemente "hacer que el bug desaparezca". Primero hay que entender por qué el software se comporta de forma inesperada. Después hay que corregir el problema sin romper otras cosas, mantener o mejorar la calidad del código y evitar que el mismo problema vuelva a aparecer.

Butcher propone un proceso central con cuatro pasos: reproducir, diagnosticar, reparar y reflexionar. Vamos a adaptarlo aquí al contexto de este capítulo (figura 5.7).

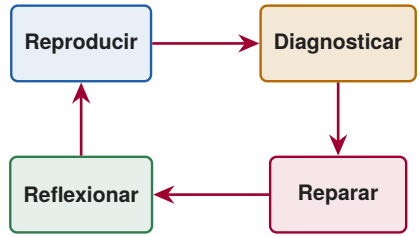


Fig. 5.7 El método de depuración en cuatro pasos: reproducir el problema, diagnosticar la causa, reparar sin romper nada y reflexionar para evitar que vuelva a ocurrir.

5.4.1. Reproducir

Antes de tocar el código, necesitamos intentar reproducir el problema.

Esto puede parecer una pérdida de tiempo cuando creemos tener clara la causa. Pero modificar el código antes de observar el fallo puede ocultarlo, cambiar sus condiciones o introducir otro problema. Además, si no hemos reproducido la incidencia, ¿cómo sabremos después que la hemos corregido?

Reproducir significa encontrar una forma fiable de hacer que el sistema muestre el comportamiento problemático. A veces es sencillo: seguimos los pasos del informe y el fallo aparece. Otras veces no ocurre en nuestro equipo, pero sí en producción. O solo aparece con un navegador concreto. O con una versión antigua. O con datos reales que no tenemos en local. O cuando varias personas usan el sistema a la vez.

Para reproducir, normalmente intentamos controlar tres cosas: el software, el entorno y las entradas.

Controlar el software significa usar la misma versión que tenía la persona afectada. En un proyecto con releases y artefactos inmutables, esto debería ser posible. Si la incidencia se reportó contra la versión 1.4.2, deberíamos saber qué commit y qué artefacto correspondían a esa versión.

Controlar el entorno significa acercarnos a las condiciones donde apareció el problema: sistema operativo, navegador, base de datos, variables de entorno, servicios externos, permisos o configuración. Aquí vuelven a ser importantes las ideas de reproducibilidad y aislamiento que veremos también en el Capítulo 6.

Controlar las entradas significa identificar qué datos, acciones o eventos provocan el fallo. Puede ser un archivo concreto, una secuencia de clics, una petición HTTP, una configuración determinada o una respuesta de un servicio externo.

Una buena reproducción debería ser, además, cómoda. Si para observar el fallo tenemos que seguir cincuenta pasos manuales durante media hora, cada experimento será lento y propenso a errores. Por eso conviene reducir la reproducción hasta

que sea lo más pequeña posible. Si el problema aparece con un archivo enorme, quizá podamos encontrar unas pocas líneas que lo provocan. Si aparece tras muchos clics, quizá podamos automatizar la secuencia. Si depende de un servicio externo, quizá podamos sustituirlo temporalmente por un doble de prueba, como vimos en el Capítulo 4.

También debemos intentar hacer determinista lo que parece no determinista. Los errores intermitentes son especialmente frustrantes, porque no sabemos si un experimento ha funcionado o simplemente hemos tenido suerte. A veces la fuente de no determinismo está en la concurrencia, en llamadas a servicios externos, en datos iniciales imprevisibles o en aleatoriedad. En esos casos, parte del trabajo consiste en controlar esas condiciones para que el fallo aparezca de forma más fiable.

¿Y si no conseguimos reproducir? Podemos sentir la tentación de cerrar la incidencia sin más con un "pues a mí me funciona". Pero lo ideal es documentar qué hemos intentado, qué información falta y qué datos adicionales podrían ayudar. También podemos pedir ayuda a soporte, a quien informó del problema o a otra persona del equipo. Que no podamos reproducir ahora no significa necesariamente que el fallo no exista.

5.4.2. Diagnosticar

Una vez que tenemos una reproducción, empieza el diagnóstico.

Diagnosticar consiste en construir hipótesis y comprobarlas mediante experimentos. Es decir, aplicamos una especie de método científico a nuestro software (figura 5.8).

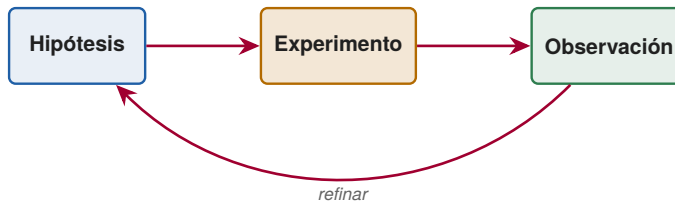


Fig. 5.8 Diagnosticar como método científico: se formula una hipótesis sobre la causa, se diseña un experimento que la confirme o descarte, se observa el resultado y se refina hasta localizar el defecto.

Una hipótesis es una explicación provisional. Por ejemplo: "el fallo ocurre porque esta variable puede llegar a `None`", "el sistema está usando una versión incompatible

de la biblioteca", "la consulta devuelve resultados duplicados" o "dos peticiones concurrentes están modificando el mismo recurso".

Un experimento es una acción diseñada para confirmar o descartar una hipótesis. Puede consistir en ejecutar el programa con otros datos, añadir una aserción temporal, inspeccionar una variable con el depurador, cambiar una configuración, desactivar un servicio externo o escribir una prueba específica.

Lo importante es que un experimento debería decirnos algo. Antes de hacerlo, conviene preguntarse: ¿qué aprenderé si el resultado es A? ¿Y si el resultado es B? Si ninguna respuesta cambia mi comprensión del problema, probablemente deberíamos replantear el experimento.

También es importante cambiar una sola cosa cada vez. Si modificamos tres variables, actualizamos una dependencia y cambiamos un archivo de configuración al mismo tiempo, incluso si el fallo desaparece, no vamos a saber por qué específicamente.

Para saber más

Una estrategia útil cuando estamos ante una regresión es comparar versiones del sistema. Si sabemos que en una versión anterior el fallo no ocurría y que en la versión actual sí ocurre, podemos buscar qué cambio introdujo el problema. Una idea a la que se llama *diff debugging* [a].
La orden `git bisect` puede ser de mucha ayuda para hacer *diff debugging*. Investiga cómo podrías usarla para localizar el commit que introdujo una regresión.

[a] https://www.martinfowler.com/bliki/DiffDebugging.html

Cuando una investigación se alarga, es muy útil anotar la traza de experimentos. Es suficiente con dejar constancia de qué se ha probado, con qué resultado y qué hipótesis quedan descartadas. Esto evita repetir caminos ya explorados y facilita pedir ayuda a otra persona sin tener que reconstruirlo todo de memoria.

Y, sí, pedir ayuda también forma parte del método. Explicar el problema a una compañera, revisar juntos los pasos o hacer *pair debugging* puede revelar supuestos que no habíamos visto. De hecho la Ley de Linus afirma que dado un número suficientemente elevado de ojos, todos los errores se convierten en obvios[4]. Así que siempre es bueno apoyarse en compañeros cuando nos atascamos.

[4] Así la formuló Eric S. Raymond en su ensayo y libro *La catedral y el bazar*.

Y a veces basta con contar el problema en voz alta para darse cuenta de que nuestra hipótesis no encaja. Incluso puede ser suficiente con que se la contemos a un patito de goma[5].

5.4.3. Reparar

Cuando creemos haber entendido la causa, llega el momento de reparar.

Pero para reparar no basta con quedarse con el último estado del último experimento. Durante el diagnóstico quizá hemos añadido trazas temporales, cambiado datos, probado configuraciones extrañas o hecho modificaciones rápidas para ver qué pasaba. La corrección final debería construirse limpiamente desde un estado conocido.

Aquí Git vuelve a ser fundamental. Podemos revisar el `diff`, descartar cambios exploratorios, crear un commit atómico con la solución real o, si procede, usar una rama específica para la corrección. Lo que llegue a la historia del proyecto debería ser el cambio que queremos mantener, no todos los rastros de la investigación.

Una buena secuencia para reparar una incidencia es la siguiente:

1. Reproducir el fallo.
2. Escribir una prueba que falle por ese fallo.
3. Comprobar que la prueba falla por la razón esperada.
4. Implementar la corrección.
5. Comprobar que la prueba nueva pasa.
6. Ejecutar las pruebas de regresión relevantes.
7. Revisar el cambio antes de integrarlo.

Esta secuencia conecta directamente con el Capítulo 4. Cada incidencia importante puede convertirse en una nueva prueba que evita que el problema reaparezca. Así, la suite de pruebas va creciendo con los errores del proyecto.

También es importante arreglar la causa y no solo el síntoma. Si una variable llega con un valor incorrecto, quizá sea tentador forzar un valor válido justo antes de que falle. Pero en realidad lo que nos interesa es por qué llegó ese valor incorrecto. Y si no entendemos eso, podemos estar ocultando el problema real.

Una última recomendación práctica es no mezclar corrección y refactorización en el mismo cambio. Si mientras corriges una incidencia reorganizas medio módulo, cambias nombres, extraes funciones y modificas comportamiento, la revisión será

[5] Sí, hasta existe un nombre para esto: el método de depuración del patito de goma: https://es.wikipedia.org/wiki/M%C3%A9todo_de_depuraci%C3%B3n_del_patito_de_goma.

más difícil. Además, si algo se rompe, no sabremos si fue por la corrección o por la refactorización. Primero arregla el fallo de forma clara. Después, si hace falta refactorizar, hazlo en otro cambio.

5.4.4. Analizar lo aprendido

Llegados a este punto, aunque la incidencia ya esté resuelta, es importante dedicar un momento a preguntarnos qué hemos aprendido (figura 5.9).

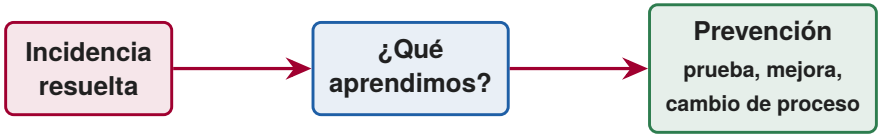


Fig. 5.9 Resolver la incidencia no es el final: analizar qué la permitió y convertirlo en una prevención (una prueba nueva, una mejora de diseño o de proceso) es lo que evita que vuelva a ocurrir.

¿Por qué no detectamos antes este problema? ¿Faltaba una prueba? ¿El diseño permitía estados inválidos? ¿La interfaz inducía a error? ¿El mensaje de excepción era poco útil? ¿La plantilla de incidencias no pedía información suficiente? ¿El pipeline no ejecutaba una comprobación relevante?

Esta fase puede parecer secundaria, especialmente cuando hay prisa, pero es la que ayuda al equipo a continuar creciendo. Si solo cerramos el ticket y seguimos adelante, puede que un tipo de problema similar vuelva a aparecer. Pero si aprendemos de la incidencia y tomamos medidas, es probable que el sistema y el proceso queden un poco mejor.

Hay una especie de recorrido emocional muy común al depurar. Primero pensamos: "no puede ser". Después: "en mi máquina no pasa". Luego: "no debería pasar". Más tarde: "¿por qué está pasando?". Finalmente: "ah, ya". Y, en los casos más inquietantes: "¿cómo podía estar funcionando antes?".

Esta última pregunta, aunque algo cómica, es muy valiosa. A veces un fallo revela que el sistema funcionaba de casualidad, que una prueba no comprobaba lo importante o que una parte del diseño era más frágil de lo que pensábamos. En ese momento tenemos una oportunidad de mejorar.

Para saber más

Cuando una incidencia es crítica y afecta de manera masiva a usuarios en producción, suele hablarse de *respuesta a incidentes*. En ese contexto las fases ante una incidencia varían un poco, y suelen incluir triaje, coordinación, mitigación, resolución y seguimiento, como puedes leer en el artículo *What happens when the pager goes off?*[a]. La mitigación busca reducir el impacto cuanto antes, aunque todavía no se conozca la causa raíz. Y la fase de seguimiento suele incluir la elaboración de un documento de *postmortem*. Uno de los postmortems más famosos y jugosos para leer es el del incidente de CrowdStrike de 2024 [b] que provocó que millones de sistemas Microsoft Windows experimentaran un *pantallazo azul de la muerte*, provocando interrupciones muy severas y costosas en múltiples industrias de todo el planeta. Otro postmortem reciente muy recomendable es el del incidente de intrusión sufrido por HuggingFace en julio de 2026 impulsado por un agente IA de OpenAI[c].

[a] https://increment.com/on-call/when-the-pager-goes-off/

[b] https://www.crowdstrike.com/wp-content/uploads/2024/08/Channel-File-291-Incident-Root-Cause-Analysis-08.06.2024.pdf

[c] https://huggingface.co/blog/agent-intrusion-technical-timeline

5.5. ¿Y cómo afecta la IA a todo esto?

Las herramientas de IA empiezan a aparecer en varias partes de la gestión de tareas e incidencias. Pueden ayudarnos a resumir conversaciones, clasificar solicitudes, detectar duplicados, mejorar informes de error, proponer preguntas aclaratorias, analizar logs o incluso sugerir parches.

Pero, como en capítulos anteriores, conviene distinguir entre acelerar una tarea y asumir la responsabilidad del proceso completo. Un sistema IA puede ayudarnos con ciertas tareas. Pero decidir qué problema merece resolverse, qué prioridad tiene, qué cambio es aceptable y qué evidencia necesitamos debe seguir siendo responsabilidad del equipo.

5.5.1. De comentarios dispersos a señales de cambio

En muchas ocasiones las solicitudes no llegan como issues bien escritas. Pueden llegar como reseñas de una app, mensajes en soporte, comentarios en redes sociales, tickets de atención al cliente o conversaciones con usuarios. Dentro de ese ruido puede haber señales útiles: peticiones de funcionalidad, quejas recurrentes, problemas de usabilidad o errores que todavía nadie ha convertido en incidencia.

Existen herramientas IA que pueden ayudar a detectar y agrupar esas señales. Por ejemplo, algunos trabajos recientes estudian cómo identificar peticiones de funcionalidad a partir de reseñas de apps móviles, y cómo hacer que esa clasificación sea más explicable para quienes deben tomar decisiones a partir de ella[6]. Otros enfoques comparan quejas de usuarios con reseñas positivas de aplicaciones competidoras para sugerir posibles mejoras de producto[7].

5.5.2. IA para mejorar peticiones e informes

Otro uso muy natural consiste en ayudar a escribir mejores peticiones de cambio e informes de error.

Como hemos visto, una petición de funcionalidad escrita en lenguaje natural puede ser a veces ambigua o incompleta. Por ejemplo: “mejorar las notificaciones” puede significar agrupar mensajes, añadir filtros, cambiar el diseño visual, permitir silencios temporales o enviar correos. Un modelo de lenguaje puede actuar como primer lector crítico, de manera que pueda detectar que falta información y proponer preguntas de aclaración.

De hecho, trabajos recientes sobre peticiones de funcionalidad en proyectos de software libre estudian precisamente cómo usar LLMs para detectar ambigüedad e incompletitud para generar preguntas que ayuden a refinar la solicitud[8]. Esto encaja con lo que hemos discutido en este capítulo: antes de convertir una petición en trabajo planificado, tenemos que entenderla.

También se están estudiando herramientas para mejorar informes de error. Por ejemplo, hay trabajos que usan LLMs para transformar informes casuales y desestructurados en bug reports con plantilla[9], o para detectar y completar

[6] https://ieeexplore.ieee.org/abstract/document/10636143

[7] https://arxiv.org/abs/2409.15724

[8] https://arxiv.org/html/2507.13555v1

[9] https://arxiv.org/abs/2504.18804

secciones como pasos para reproducir, comportamiento observado y comportamiento esperado[10].

Básicamente, si sabemos que muchos informes fallan por falta de información, una herramienta puede guiar a la persona que reporta haciéndole preguntas: "¿qué esperabas que ocurriera?", "¿puedes indicar los pasos?", "¿qué versión estabas usando?", "¿apareció algún mensaje?". Así que quizá el futuro de los formularios de incidencia sea transformarse en una conversación guiada por un asistente IA.

5.5.3. IA para reproducir, diagnosticar y proponer cambios

Algunos trabajos recientes van un paso más lejos y plantean sistemas de gestión de incidencias asistidos por múltiples agentes. En esa visión, una persona informa de un problema en lenguaje natural; un agente conversa para completar el informe; otro intenta reproducirlo; otro lo clasifica; otro busca incidencias parecidas; otro localiza código sospechoso; otro propone una prueba o un parche; y el pipeline valida si el cambio parece correcto[11].

Suena impresionante, y es probable que a corto plazo veamos herramientas de este estilo integradas en los entornos de desarrollo, pero conviene no dejarse llevar por la emoción.

Porque reproducir una incidencia puede depender de datos privados, servicios externos, concurrencia, configuración de producción o comportamientos no deterministas. Y diagnosticar puede exigir entender decisiones de diseño, restricciones de negocio o historia del proyecto. Incluso si el sistema llega a proponer un parche, no está garantizado que el cambio sea correcto, mantenible o seguro.

Además, en un flujo multiagente los errores pueden propagarse. Si el sistema entiende mal la incidencia, puede intentar reproducir el fallo equivocado. Si reproduce mal, puede diagnosticar mal. Si diagnostica mal, puede proponer un parche que arregla un síntoma o introduce una regresión. Por eso, cuanto más automático sea el proceso, más importantes serán los puntos de validación, las señales de confianza y la supervisión humana.

Por tanto, aunque usemos este tipo de soluciones, la necesidad de criterio humano no desaparece. Las personas siguen teniendo que decidir qué señales importan, qué información falta, qué riesgos son aceptables, qué pruebas demuestran la corrección y qué cambios pueden entrar en la línea principal.

[10] https://arxiv.org/abs/2604.26142

[11] https://arxiv.org/html/2510.08005v2

Para saber más

Un ejemplo muy interesante es `mod_exelearning`, un repositorio público y libre del proyecto eXeLearning. En este proyecto se usan ADRs (*Architecture Decision Records*) para documentar decisiones técnicas. Cada decisión se registra como un documento con las siguientes secciones: contexto, problema, opciones, evidencia, decisión, consecuencias, riesgos y validación. En el proyecto usan agentes IA para redactar las opciones y la evidencia (incluyendo fuentes), y luego la persona responsable estudia la propuesta y decide, fijando el estado a aceptada o rechazada. Para entender mejor el proceso consulta el ADR `DEC-29-01-deteccion-geogebra-auto-scorm`[a] e intenta reconstruir la trazabilidad completa de la incidencia:

- ¿Cuál es el *issue* que origina la decisión?
- ¿El informe de la incidencia contenía información suficiente?
- ¿Qué agente y modelo de IA se usó para asistir con la decisión?
- ¿Qué evidencia se aporta para justificar la decisión?
- ¿Cuál fue la decisión humana final? ¿Fue aceptada, rechazada o quedó como propuesta?
- ¿Qué *pull request* implementa la corrección?
- ¿Qué *commits* la componen?
- ¿Qué ficheros se modificaron?
- ¿Qué pruebas se ejecutaron? ¿Se puede comprobar en el flujo de CI si pasaron?

[a] https://github.com/exelearning/moodle-mod_exelearning/blob/main/research/decisiones/adr/DEC-29-01-deteccion-geogebra-auto-scorm.md

5.6. Reflexión final: gestionar tareas e incidencias es gestionar la configuración... y el aprendizaje

En este capítulo hemos visto que gestionar tareas e incidencias consiste en hacer visible el trabajo, ordenar el cambio, coordinar al equipo y conservar trazabilidad entre necesidades, decisiones, código, pruebas y releases.

El cambio es inevitable y por ello tenemos que gestionarlo. Cuando una petición de funcionalidad se registra, se analiza, se prioriza y se transforma en work items concretos, el equipo trabaja con más claridad. Cuando una incidencia se informa

bien, se reproduce, se diagnostica, se repara y se analiza, el sistema queda mejor preparado para el futuro (figura 5.10).

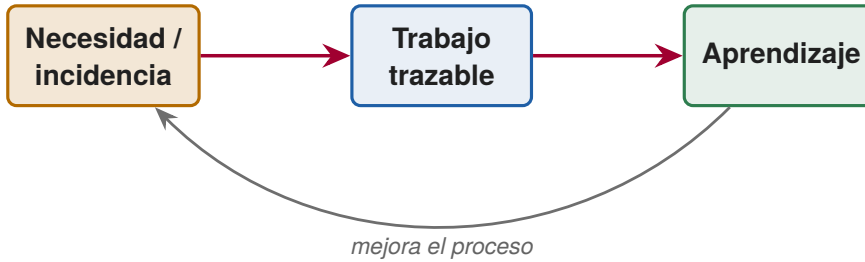


Fig. 5.10 Gestionar tareas e incidencias no solo ordena el cambio: con trazabilidad y análisis, cada ciclo deja al equipo y al sistema mejor preparados para el siguiente.

También hemos visto que la herramienta de gestión que utilicemos puede ayudarnos mucho, pero solo si el equipo lo usa con criterio. Estados, prioridades, tipos, roles, plantillas y tableros tienen valor porque ayudan a pensar y comunicar mejor. Aunque si solo se utilizan porque lo exige una política interna y las tratamos como burocracia vacía, dejan de ser útiles.

Esta idea conecta con todo lo que hemos visto hasta ahora. En integración continua, cada cambio debe validarse. En el repositorio, cada cambio forma parte de una historia. En pruebas, cada cambio debería aumentar o preservar nuestra confianza. Y en gestión de tareas e incidencias, cada cambio debería tener una razón comprensible y trazable.

Una incidencia bien gestionada puede convertirse en una prueba nueva, una mejora de diseño, una plantilla más clara, un mensaje de error más útil, un log mejor pensado o una decisión documentada.

Por tanto, gestionar tareas e incidencias es gestionar la configuración, y también el aprendizaje del equipo. El objetivo es que, cuando aparezcan problemas, y seguro que lo harán, el equipo sea capaz de entenderlos, resolverlos y salir de ellos con un sistema un poco mejor.

5.7. Lecturas y recursos recomendados

- Capítulo 9, *Going on-call*, del libro *The Missing README: A Guide for the New Software Engineer* (Chris Riccomini y Dmitriy Ryaboy).
- Libro *Debug It! Find, Repair, and Prevent Bugs in Your Code* (Paul Butcher).

- Libro *Debugging by Thinking: A Multidisciplinary Approach* (Robert Charles Metzger).
- Libro *Configuration Management Best Practices: Practical Methods that Work in the Real World* (Bob Aiello y Leslie Sachs).
- Artículo *What Makes a Good Bug Report?* (Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj y Thomas Zimmermann).
- Libro *Sprint: How to Solve Big Problems and Test New Ideas in Just Five Days* (Jake Knapp, John Zeratsky y Braden Kowitz).
- Artículo *Demystifying Feature Requests: Leveraging LLMs to Refine Feature Requests in Open-Source Software*.
- Artículo *Past, Present, and Future of Bug Tracking in the Generative AI Era*.

Capítulo 6

Procesamiento y compilación aisladas

Un contenedor no es un ente: es un conjunto de namespaces y cgroups aplicado a un proceso.
Jessie Frazelle.

Resumen En este capítulo estudiaremos por qué el entorno en el que se construye, prueba, empaqueta y ejecuta el software es una pieza esencial de la gestión de la configuración. Para entenderlo recorreremos cómo ha evolucionado la industria desde instalaciones manuales y servidores configurados artesanalmente hasta enfoques basados en entornos declarativos, virtualización, contenedores y entornos de desarrollo reproducibles. Discutiremos conceptos como aislamiento, reproducibilidad, hermeticidad o inmutabilidad, y de su relación con CI/CD. Y, tras ver el impacto de la IA en este campo, cerraremos el capítulo con una idea principal: que el procesamiento aislado nos ayuda a reducir incertidumbre y a construir software de manera más fiable.

6.1. Introducción: cuando automatizar no basta

En el Capítulo 2 vimos que la automatización es una pieza fundamental para construir, probar, empaquetar, entregar y desplegar software de forma más rápida y fiable. Una pipeline de integración continua puede compilar el proyecto, ejecutar pruebas, analizar el código, generar un artefacto y avisar al equipo cuando algo falla. Pero hay una pregunta que quizá quedó en segundo plano: ¿dónde ocurre todo eso?

Porque construir, probar, entregar o desplegar ocurre en una máquina concreta, con un sistema operativo determinado, con unas versiones específicas de bibliotecas, herramientas, variables de entorno, permisos, servicios auxiliares y configuraciones. Y, si esas condiciones cambian, el resultado también puede cambiar.

Es probable que hayas vivido alguna versión de esta situación. Trabajas en una práctica, consigues que funcione en tu ordenador, la compartes con otra persona

del equipo y, de repente, no arranca. Después de un rato mirando el código, alguien descubre que una persona tiene Python 3.10 y otra Python 3.12. O que a una le falta una biblioteca del sistema. O que una instaló un servicio hace meses y ya no recordaba que arrancaba automáticamente y se quedaba escuchando en el mismo puerto que requiere la aplicación. Y entonces llega la frase clásica, que ya hemos comentado antes en este libro: "Pues en mi máquina funciona".

Esta frase a veces se dice en tono un poco defensivo, pero en realidad es una pista. Nos está diciendo que el software no depende solo del código fuente, sino que también lo hace del entorno en el que se ejecuta (figura 6.1).

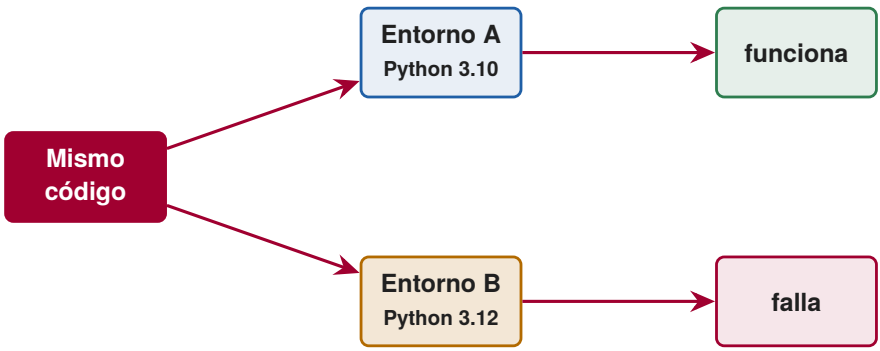


Fig. 6.1 "En mi máquina funciona". El mismo código puede comportarse de forma distinta según el entorno, porque el software no depende solo del código fuente.

Y esto afecta directamente a la fiabilidad. Una pipeline puede estar perfectamente automatizada y, aun así, ser poco fiable si depende de una máquina configurada a mano que nadie sabe reconstruir. Un script puede ejecutar siempre los mismos pasos, pero si los pasos se ejecutan sobre entornos distintos, los resultados pueden ser distintos. Automatizar el qué no resuelve por completo el dónde ni el cómo.

Podemos verlo con una analogía. Imagina una cocina profesional que quiere preparar el mismo plato todos los días. No basta con escribir la receta. También importa la temperatura del horno, el estado de los utensilios, la organización de la cocina y la forma de conservar los alimentos. Si en dos cocinas se usan elementos distintos, la receta puede estar escrita correctamente y, aun así, el plato no salir igual.

Con el software ocurre algo parecido. La receta puede ser la pipeline. Pero la cocina es el entorno. Y si la cocina no está bajo control, la receta no garantiza el resultado (figura 6.2).

Por eso, en este capítulo vamos a dar un paso más. Además de preguntarnos cómo automatizar la construcción, las pruebas o el despliegue, vamos también a

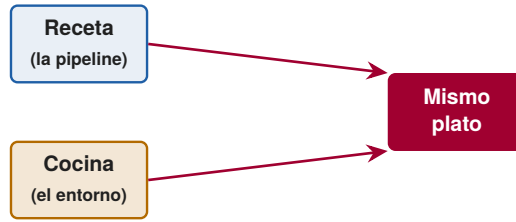


Fig. 6.2 La analogía de la cocina: una buena receta (la pipeline) no basta para obtener siempre el mismo plato si la cocina (el entorno) no está bajo control.

intentar conseguir que esas acciones ocurran en entornos previsibles, reproducibles y, en la medida de lo posible, aislados de accidentes externos.

6.1.1. El entorno también tiene historia

Durante mucho tiempo, muchas organizaciones gestionaron sus entornos de forma bastante artesanal. Una persona instalaba un servidor, copiaba ficheros, ajustaba permisos, instalaba paquetes, cambiaba configuraciones y dejaba el sistema funcionando. A veces esos pasos se documentaban en una guía. A veces se recogían en scripts. Y a veces, sencillamente, quedaban en la memoria de quien había hecho la instalación.

Este enfoque podía funcionar razonablemente bien cuando los sistemas cambiaban poco, había pocas máquinas y las entregas se hacían cada mucho tiempo. Si una aplicación se desplegaba dos veces al año en un servidor concreto, quizá era aceptable tener un proceso lento, manual y muy controlado. No era la situación ideal, desde luego, pero podía funcionar.

En la industria se usa a veces la expresión *servidor copo de nieve*[1] para referirse a una máquina especial, única, irrepetible. Como un copo de nieve, parece tener una forma propia que no se puede reproducir exactamente. Quizá empezó siendo una máquina estándar, pero durante meses o años se le fueron aplicando pequeños cambios: un paquete instalado para resolver una incidencia urgente, un fichero modificado directamente, un parche que no llegó al repositorio, un certificado copiado a mano, una variable añadida en una consola...

El resultado es un entorno que funciona, pero que nadie se atreve a tocar demasiado. Y, si la máquina se pierde o hay que crear otra igual, aparece el problema: sabemos que el sistema funcionaba, pero no sabemos reconstruir con seguridad

[1] https://martinfowler.com/bliki/SnowflakeServer.html

las condiciones que lo hacían funcionar. Esto es un problema claro de gestión de la configuración. Porque, como venimos apuntando durante los capítulos previos, la configuración de un sistema no está formada únicamente por el código fuente. También forman parte de ella las dependencias, las herramientas de construcción, los servicios necesarios, las versiones, los parámetros, los artefactos, los scripts y los entornos en los que ocurre todo el proceso.

Esta idea ha ido ganando importancia a medida que la industria ha pasado de cambios lentos y manuales a cambios frecuentes y automatizados. Cuando una organización quiere entregar software varias veces al día, no puede permitirse que cada entorno sea una artesanía irrepetible. Necesita que los entornos puedan describirse, reconstruirse y verificarse.

Y es por ello que han surgido tantas soluciones relacionadas con la automatización de infraestructura, la virtualización, los contenedores y los entornos de desarrollo reproducibles que vamos a conocer en este capítulo.

> **Para saber más**
>
> Busca información sobre el concepto *environment drift*. ¿Qué significa que dos entornos *deriven* con el tiempo? Relaciónalo con la idea de servidor copo de nieve. ¿Qué pequeños cambios pueden hacer que dos máquinas que empezaron siendo iguales acaben comportándose de forma distinta?

6.1.2. Gestionar el entorno como configuración

En capítulos anteriores hemos insistido en que el código fuente debe estar versionado, que las pruebas deben poder ejecutarse de forma automática y que los artefactos deben ser trazables. Ahora añadimos otra pieza: el entorno también debería poder describirse de forma explícita.

Una configuración reproducible intenta que el entorno pueda reconstruirse a partir de información que el equipo conserva y mantiene (figura 6.3). Esto puede incluir ficheros de dependencias, scripts, documentación, plantillas de infraestructura, ficheros de contenedores, configuraciones de servicios, variables de entorno esperadas o instrucciones para preparar el IDE. Lo que se busca es que el conocimiento necesario para construir y ejecutar el sistema no esté solo en la cabeza de una persona, o en documentación dispersa, o en una máquina que nadie sabe volver a crear.

En el Capítulo 2 ya vimos algunos ejemplos sencillos. Un fichero `requirements.txt`, `pyproject.toml`, `package.json`, `pom.xml` o `build.gradle` permite declarar dependencias del proyecto. Un fichero de bloqueo,

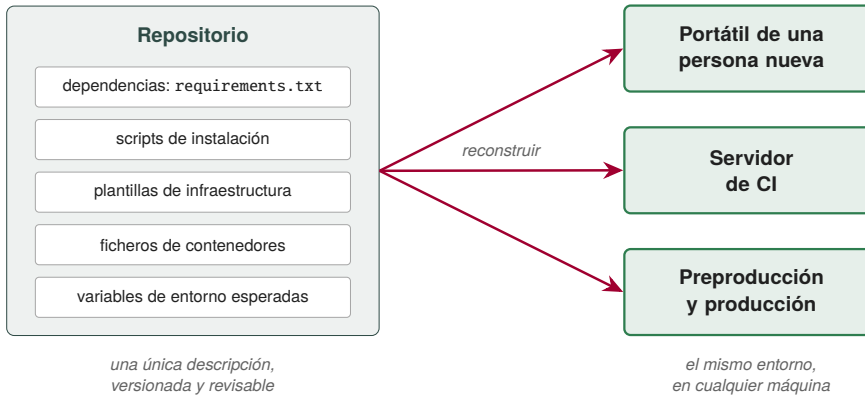


Fig. 6.3 Tratar el entorno como configuración: en lugar de montarlo a mano, se describe en ficheros versionados que permiten reconstruirlo de forma repetible en cualquier máquina.

como `poetry.lock` o `package-lock.json`, ayuda a fijar versiones concretas para que distintas máquinas resuelvan las mismas dependencias.

Sin embargo, las dependencias del lenguaje no siempre son suficientes. Una aplicación puede necesitar una base de datos, una cola de mensajes, una versión concreta de una librería nativa del sistema, una herramienta externa, un puerto disponible, un certificado, una estructura de carpetas o determinadas variables de entorno. Si solo declaramos las dependencias Python, JavaScript o Java, seguro que nos estamos dejando fuera piezas importantes.

Por eso, poco a poco, la industria ha tendido a tratar los entornos como algo que también puede expresarse mediante código o mediante ficheros de configuración. Esta idea se conoce como *infraestructura como código*. En lugar de crear o modificar infraestructura únicamente mediante clics, comandos manuales o scripts, se intenta describir el estado esperado en ficheros que pueden versionarse, revisarse y automatizarse.

Esto tiene varias ventajas. Un entorno descrito de forma explícita puede ser revisado por el equipo, igual que revisamos código. Además permite conservar su historia en Git y hacer comparaciones entre versiones. Puede ejecutarse en CI. Puede ser un buen apoyo cuando incorporamos a una persona nueva al proyecto. Y puede ayudar a responder preguntas como qué cambió en el entorno desde la última release, por qué esta versión falla en preproducción, o qué dependencias necesita realmente el sistema.

Por tanto, un buen entorno de trabajo debería ser, en la medida de lo posible, autocontenido, declarativo, desechable, idéntico, predecible, transparente y versionable. Autocontenido, porque debería especificar todo lo que necesita. Declarativo, porque

debería expresar qué estado esperamos, no solo una colección de pasos a seguir. Desechable, porque deberíamos poder destruirlo y reconstruirlo sin miedo. Idéntico, porque dos personas deberían poder obtener condiciones equivalentes. Predecible, porque el resultado no debería depender de casualidades locales. Transparente, porque el equipo debería entender qué contiene. Y versionable, porque los cambios en el entorno también son cambios del proyecto.

> **Para saber más**
>
> Que un entorno sea desechable puede llamar la atención en primera instancia. Busca información sobre la expresión *pets vs cattle*. Aunque sea una metáfora algo dura, ¿por qué crees que se defiende que un entorno desechable se debe parecer menos a una mascota y más a ganado?

6.2. Aislamiento y reproducibilidad

Para controlar mejor el entorno, la industria ha utilizado diferentes técnicas de reproducibilidad y distintos niveles de aislamiento.

Por un lado, la reproducibilidad busca que si se repite un mismo proceso en dos máquinas distintas, en dos momentos diferentes o por dos personas, podamos obtener el mismo resultado. Por otra parte, aislar significa separar una parte del sistema para que no dependa de lo que ocurre fuera, o para que no afecte a otras partes, de forma que se evitan interferencias y, por tanto, se mejora la reproducibilidad.

Una primera forma de aislamiento consiste en gestionar las dependencias del lenguaje de programación. En Python podemos usar entornos virtuales para que un proyecto tenga sus propias bibliotecas sin mezclarlas con las de otro proyecto. Por ejemplo, un proyecto puede depender de una versión moderna de Django y otro proyecto puede usar otra versión distinta, más antigua, sin interferencias. Esto ayuda a evitar situaciones en las que una aplicación necesita una versión de una biblioteca y otra aplicación necesita otra versión incompatible. Pero un entorno virtual de Python no aísla todo. No aísla la base de datos, por ejemplo, o una herramienta instalada globalmente en el sistema operativo.

Otro nivel lo proporcionan las máquinas virtuales, con las que seguro que ya tienes experiencia. Una máquina virtual permite ejecutar un sistema operativo completo sobre una capa de virtualización. Esto proporciona un aislamiento más fuerte, porque cada máquina virtual puede tener su propio sistema operativo completo, sus propios servicios y su propia configuración.

Y a un nivel intermedio de aislamiento entre entornos virtuales y máquinas virtuales están los contenedores. Como vamos a ver en detalle en las siguientes secciones, en lugar de aislar solo paquetes de un lenguaje, un contenedor permite empaquetar una aplicación con muchas de sus dependencias de ejecución. Pero un contenedor comparte el kernel del sistema operativo del sistema anfitrión, por lo que si bien permite controlar mucho mejor el entorno que los entornos virtuales, tiene un nivel de aislamiento menor que las máquinas virtuales.

Podemos verlo, por tanto, como una escala (figura 6.4). En un extremo tenemos instalaciones directas en la máquina, con poco aislamiento y mucha dependencia del entorno local. Después aparecen entornos virtuales del lenguaje, que aíslan dependencias de una plataforma concreta. A un nivel mayor están los contenedores, que encapsulan aplicaciones y muchas dependencias del sistema. Y, finalmente, máquinas virtuales, que aíslan sistemas operativos completos.

Pero con cada nivel de aislamiento, crece también el coste en recursos consumidos. Una máquina virtual ocupa mucho más espacio en disco y es más lenta de arrancar que un contenedor, por ejemplo. Por ello, como vamos a ver a lo largo del capítulo, cuando queramos decidir el nivel de aislamiento que necesitamos, lo importante es entender qué parte del problema queremos aislar y decidir qué coste podemos asumir.

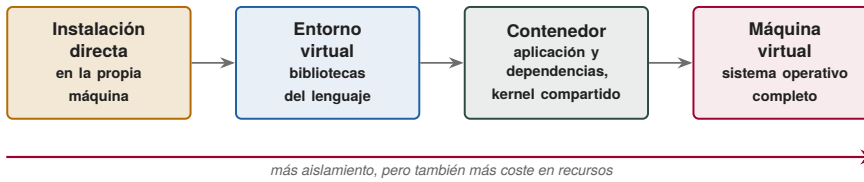


Fig. 6.4 La escala de aislamiento: desde instalar directamente en la máquina hasta una máquina virtual con su propio sistema operativo. Cada paso aísla más, pero también consume más recursos.

6.2.1. Procesamiento aislado y hermeticidad

Llamamos procesamiento aislado a la ejecución de una tarea dentro de un entorno controlado, separado en cierta medida del entorno general de la máquina. Esa tarea puede ser compilar, ejecutar pruebas, empaquetar una aplicación, generar documentación, lanzar un análisis estático o preparar un artefacto. Y la idea central

es que el resultado de la tarea dependa de lo que hemos declarado y preparado para ella, no de accidentes o situaciones no planificadas del entorno local.

El procesamiento aislado está relacionado con la *hermeticidad*, que es la capacidad de un proceso para depender solo de sus entradas declaradas y no de elementos externos no controlados. Un proceso hermético se parece a una caja bien cerrada: lo que ocurre dentro está determinado por lo que hemos metido en ella, no por lo que casualmente había alrededor (figura 6.5).

Fig. 6.5 Un proceso hermético depende solo de sus entradas declaradas, no de lo que casualmente haya en la máquina. Por eso produce el mismo resultado en cualquier sitio.

En la práctica, conseguir hermeticidad completa puede ser difícil. Muchas construcciones descargan dependencias de Internet, algunas pruebas dependen de servicios externos, puede haber procesos que usan la hora del sistema o rutas absolutas, por ejemplo. Pero, incluso si no conseguimos una hermeticidad perfecta, pensar en ella nos ayuda a detectar posibles dependencias ocultas.

El procesamiento aislado es especialmente importante en CI/CD. Una pipeline debería comportarse como una línea de montaje. Cada vez que entra un cambio, la línea ejecuta pasos conocidos en condiciones conocidas. Si la línea de montaje depende de restos de ejecuciones anteriores, paquetes instalados a mano o configuraciones invisibles, deja de ser una línea fiable y se convierte en una fábrica con un resultado difícil de predecir.

Además el procesamiento aislado nos ayuda a razonar mejor sobre los fallos cuando se producen. Porque si el entorno está controlado, hay menos lugares donde buscar. Y si el entorno se puede reconstruir, otras personas pueden reproducir el problema y trabajar en él con garantías, como vimos en Capítulo 5.

6.3. Contenedores: cajas estándar para software

Una de las metáforas más conocidas para explicar los contenedores software procede del transporte marítimo, como explica Gabriel N. Schenker en su libro *Learn Docker*.

Durante mucho tiempo, transportar mercancías era una tarea compleja porque cada mercancía tenía una forma, tamaño y cuidado distinto. Había cajas, sacos, barriles, piezas sueltas y objetos frágiles. Mover todo eso entre camiones, trenes y barcos exigía mucho trabajo manual y mucha coordinación. Cada carga era casi un caso especial.

La estandarización del contenedor marítimo cambió el problema. Desde fuera, el contenedor tiene una forma conocida. Quien lo transporta no necesita saber todos los detalles de lo que hay dentro. Necesita saber cómo mover un contenedor. Los barcos, grúas, trenes y camiones pueden adaptarse a esa unidad estándar.

Con el software ocurre algo parecido. Antes de los contenedores, mover una aplicación entre desarrollo, pruebas y producción podía implicar muchas instrucciones específicas: instala esta versión del lenguaje, después esta biblioteca, luego este paquete del sistema, copia este fichero, configura este servicio, abre este puerto, no olvides esta variable de entorno. Cada aplicación podía tener su propia forma de instalarse y desplegarse.

Un contenedor intenta convertir esa aplicación y sus dependencias en una unidad más estandarizada. Desde fuera, las herramientas pueden tratar muchos contenedores de forma parecida, aunque dentro ejecuten aplicaciones distintas. Esto reduce fricción entre entornos y equipos.

La metáfora no es del todo perfecta, pero es ilustrativa. En realidad un contenedor software no es una caja completamente aislada del mundo. Comparte el kernel del sistema anfitrión y puede comunicarse con otros procesos, redes y volúmenes si lo configuramos para ello. Pero sí proporciona una forma más controlada y portable de ejecutar una aplicación.

> **Para saber más**
>
> Esta metáfora la utilizó también el propio Solomon Hykes, creador de Docker, cuando presentó en público el proyecto en el año 2013[a]. Se trata de una charla cortita muy recomendable, porque explica muy bien la motivación detrás de los contenedores.
>
> [a] https://www.youtube.com/watch?v=3Kn6_b-1mK4

Esta idea encaja muy bien con CI/CD. Si nuestra pipeline es una fábrica, el contenedor puede actuar como una unidad estandarizada que entra y sale de la línea de montaje. La fábrica no tiene que aprender de nuevo cómo instalar cada aplicación desde cero en cada entorno. Puede construir una imagen, probarla y moverla a través del proceso.

¡Uy, pero ya estamos mezclando conceptos! Porque en el párrafo anterior hemos diferenciado entre contendor e imagen. Vamos a explicar la diferencia para que quede clara.

Una imagen es una plantilla preparada para crear contenedores. Contiene un sistema de ficheros con la aplicación y sus dependencias, junto con cierta información de configuración. Podemos pensar en una imagen como un artefacto construido. Un contenedor, por su parte, es una ejecución concreta de una imagen. Y podemos crear tantos contenedores como queramos a partir de una misma imagen. Si una imagen fuera un clase, un contenedor sería una instancia de esa clase (figura 6.6).

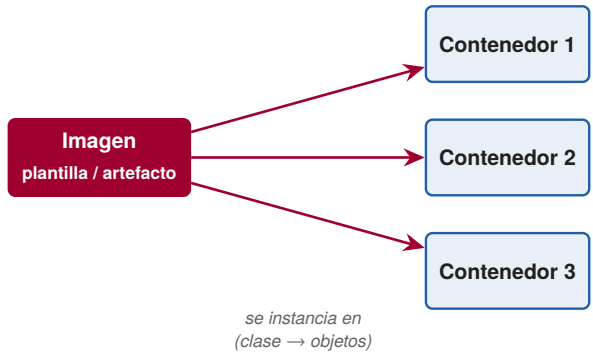


Fig. 6.6 Una imagen es la plantilla (el artefacto que se versiona y almacena); un contenedor es una ejecución concreta de esa imagen. De una imagen pueden nacer muchos contenedores.

Esta diferencia es importante y tenemos que tenerla clara. Cuando construimos una imagen estamos preparando un artefacto. Cuando arrancamos un contenedor estamos ejecutando ese artefacto. Una imagen puede almacenarse en un registro y versionarse con etiquetas. Un contenedor tiene ciclo de vida: se crea, se ejecuta, puede detenerse y puede eliminarse.

Y esto conecta con la idea de inmutabilidad que ya discutíamos en el Capítulo 2. Una imagen debería tratarse como algo que no se modifica a mano después de construirse. Si necesitamos cambiar la aplicación o sus dependencias, se debe construir una nueva imagen con una nueva etiqueta. Esto ayuda a evitar cambios invisibles y a mantener trazabilidad. Básicamente, esta forma de trabajar nos empuja

a una idea sana: en lugar de "arreglar" entornos a mano, es más apropiado cambiar la definición que permite reconstruirlos.

6.3.1. Un poco por debajo: qué aíslan realmente los contenedores

No vamos a convertir este capítulo en una explicación avanzada del kernel de Linux, que para eso has cursado ya una asignatura de Sistemas Operativos, pero sí conviene entender un par de ideas importantes al hablar de contenedores: *namespaces* y *cgroups*.

Los namespaces permiten limitar lo que un proceso puede ver. Por ejemplo, un proceso dentro de un contenedor puede tener su propia visión de los procesos, de la red, de los puntos de montaje o del nombre de la máquina. Desde dentro, parece vivir en un entorno propio, aunque en realidad está compartiendo el sistema anfitrión con otros procesos.

Los cgroups, o control groups, permiten limitar cuánto puede usar un proceso o grupo de procesos. Por ejemplo, pueden ayudar a controlar consumo de CPU, memoria o entrada/salida. Esto es importante porque, sin límites, una aplicación podría consumir demasiados recursos y afectar a otras aplicaciones que se ejecutan en la misma máquina.

Dicho de forma simplificada, los namespaces aíslan vistas del sistema y los cgroups controlan recursos. En la práctica, el aislamiento de un contenedor también depende de otros mecanismos y de su configuración. Pero lo importante es comprender que cuando usamos contenedores en realidad estamos ejecutando procesos usando mecanismos del sistema operativo Linux que proporcionan aislamiento lógico y control de recursos.

Y esto también permite entender una diferencia importante entre contenedores y máquinas virtuales. Una máquina virtual ejecuta un sistema operativo invitado completo sobre un hipervisor. Un contenedor, en cambio, ejecuta procesos aislados que comparten el kernel del sistema anfitrión, gestionados por un motor de contenedores. Por eso los contenedores suelen ser más ligeros y arrancar más rápido. Pero también por eso no ofrecen el mismo nivel de aislamiento que una máquina virtual (figura 6.7).

> **Para saber más**
>
> Busca información sobre qué es un hipervisor y qué papel desempeña en la virtualización. Después, investiga la diferencia entre hipervisores de tipo 1 y

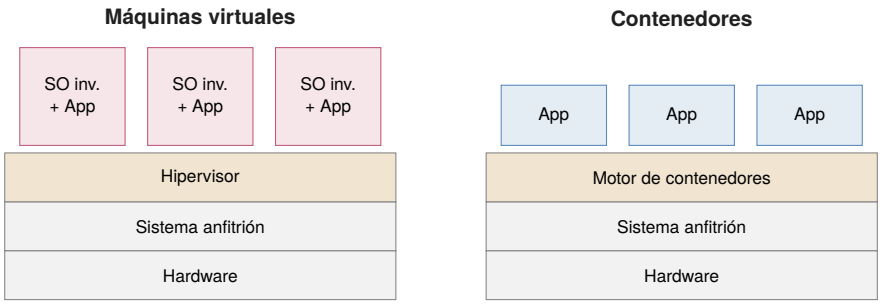


Fig. 6.7 Máquinas virtuales frente a contenedores. Cada VM incluye su propio sistema operativo invitado; los contenedores comparten el *kernel* del anfitrión, por lo que son más ligeros.

de tipo 2. ¿Qué significa que un hipervisor se ejecute directamente sobre el hardware físico (*bare metal*)? ¿Y qué cambia cuando el hipervisor se ejecuta sobre un sistema operativo anfitrión? ¿De qué tipo es el hipervisor que usa VirtualBox?

Por tanto, insistimos en que no se trata de decidir que una tecnología sustituye siempre a la otra. Si necesitamos ejecutar sistemas operativos distintos o aislar un sistema completo, una máquina virtual puede tener mucho sentido. Si queremos empaquetar una aplicación y muchas de sus dependencias de ejecución de forma ligera, un contenedor puede ser más adecuado.

Para saber más

Busca información sobre MicroVMs. ¿Por qué crees que se ha creado este nuevo tipo de virtualización? ¿En qué nivel de la escala de aislamiento que discutíamos en la sección anterior la situarías?

6.4. Docker, Vagrant y otras respuestas al problema

Llegados a este punto, podemos situar mejor algunas herramientas de este ecosistema que seguramente ya has escuchado, como Docker, Vagrant, Terraform o Ansible[2].

Docker es una de las tecnologías que popularizó el uso de contenedores en el desarrollo y despliegue de software. Permite construir imágenes, ejecutar contenedores y trabajar con registros de imágenes, como vas a realizar en las clases de prácticas. Y su objetivo es empaquetar una aplicación y sus dependencias en una unidad portable, reproducible y controlada.

Un Dockerfile permite describir cómo construir una imagen. En lugar de instalar dependencias manualmente en una máquina, podemos dejar instrucciones en un fichero versionado. En las prácticas verás cómo es su sintaxis, pero conceptualmente lo importante es que ese fichero convierte parte del conocimiento del entorno en algo revisable y repetible.

> **Para saber más**
>
> Aunque, como decimos, en las clases de prácticas tendremos una sesión más aplicada para trabajar con contenedores, puedes echar un vistazo a algún Dockerfile real para que esta sección no suene tan abstracta. Por ejemplo, revisa este Dockerfile del proyecto NodeShift de Red Hat[a]. ¿Puedes intuir el objetivo de la mayoría de sus líneas? ¿Ves por qué es un fichero que puede versionarse, revisarse y evolucionar junto con el resto del proyecto?
>
> [a] https://github.com/nodeshift/docker/blob/main/Dockerfile

Como veremos también en las prácticas, Docker Compose aparece cuando una aplicación necesita varios servicios. Por ejemplo, una aplicación web puede necesitar un servidor de aplicación, una base de datos y quizá una caché. En un entorno manual, cada persona tendría que instalar y configurar esos servicios. Con un enfoque basado en contenedores, podemos describir con Docker Compose un pequeño ecosistema de contenedores que colaboran entre sí. Y verás que resulta muy sencillo de escribir para levantar, por ejemplo, un entorno de desarrollo muy similar a producción.

[2] De hecho, si escuchaste los dos episodios del podcast recomendado en el Capítulo 2, ya te sonará cómo se utilizan todas estas soluciones en la industria. Si no lo escuchaste en su momento, quizá ahora sea una buena oportunidad para hacerlo.

Para saber más

Después de ver un Dockerfile aislado, también puede ser útil observar un ejemplo de Docker Compose. Revisa este fichero `compose.yaml` del repositorio `awesome-compose` de Docker[a]. No hace falta que entiendas todavía todos los detalles, pero intenta identificar cuántos contenedores forman este pequeño ecosistema, qué papel cumple cada uno y qué información se necesita para que puedan comunicarse. ¿Ves cómo este fichero no describe solo una imagen o un contenedor, sino una aplicación formada por varias piezas que deben arrancar y trabajar juntas?

[a] https://github.com/docker/awesome-compose/blob/master/wordpress-mysql/compose.yaml

Por otra parte, Vagrant ofrece funcionalidades relacionadas pero para máquinas virtuales. Mediante un fichero de configuración el equipo puede describir una máquina virtual, indicar qué sistema usar y qué recursos hardware se le deben asignar. Y Vagrant permite utilizar los mismos ficheros de configuración para trabajar con múltiples proveedores de máquinas virtuales.

El objetivo principal de Vagrant suele ser facilitar la creación y gestión de máquinas virtuales para tener entornos de desarrollo reproducibles. Pero esta misma idea puede usarse para otro tipo de infraestructura. En proyectos reales, muchas veces no necesitamos solo máquinas, sino también redes, balanceadores de carga, reglas de seguridad, almacenamiento, clústeres o servicios en la nube. Herramientas como Terraform u OpenTofu permiten describir esa infraestructura mediante ficheros de configuración, para que pueda ser creada, modificada o destruida de manera más controlada y repetible.

Ahora bien, una vez tenemos creada la infraestructura, quizá necesitamos instalar paquetes, crear usuarios, copiar ficheros de configuración, activar servicios, desplegar certificados o preparar una aplicación para que pueda ejecutarse. A esta tarea se le suele llamar aprovisionamiento.

Herramientas como Ansible, Chef o Puppet ayudan precisamente a automatizar ese aprovisionamiento para que no dependa de una persona conectándose por SSH y lanzando un script. En el caso de Ansible, esa configuración se describe normalmente mediante playbooks: ficheros escritos en YAML que indican qué tareas deben aplicarse sobre qué máquinas. Un playbook puede expresar, por ejemplo, que un servidor debe tener instalado un paquete, que un servicio debe estar arrancado o que cierto fichero de configuración debe tener un contenido concreto.

Así, utilizando este tipo de soluciones, tanto la creación como la configuración de una máquina se escribe en ficheros versionables, revisables y repetibles, por lo

que dejamos de depender de instrucciones manuales y reducimos el riesgo de que cada entorno acabe configurado de una forma ligeramente distinta.

6.4.1. Dev containers: el entorno del desarrollador también cuenta

A veces los equipos se preocupan mucho por aislar CI, preproducción y producción, pero dejan el entorno local de cada desarrollador un poco a su suerte.

Esto genera una situación curiosa. La pipeline puede estar muy cuidada, los despliegues pueden estar automatizados y producción puede estar perfectamente monitorizada. Pero luego una persona nueva tarda dos días en levantar el proyecto en su portátil. O una prueba falla solo en el ordenador de alguien. O cada miembro del equipo usa versiones distintas de las herramientas. Es decir, hemos mejorado la fábrica, pero tenemos personas que siguen fabricando piezas en un taller distinto.

El entorno local es importante porque es donde se escriben, ejecutan y validan muchos cambios antes de llegar al repositorio compartido. Si ese entorno es demasiado diferente del entorno de CI, aparecerán sorpresas. Y si cada persona tiene un entorno muy distinto, resolver problemas se vuelve más difícil.

Por supuesto, no todo tiene que estar absolutamente fijado. Hay preferencias personales razonables: tema del editor, atajos, algunas extensiones, forma de organizar ventanas. Pero hay otras decisiones que forman parte del contrato del proyecto: versión del lenguaje, herramientas de formateo, linters, dependencias, servicios necesarios, variables esperadas, comandos de construcción y estructura de carpetas.

Los Dev Containers intentan dar respuesta precisamente a este problema. La idea consiste en abrir el proyecto desde el editor, pero haciendo que las herramientas de desarrollo se ejecuten dentro de un contenedor preparado para ese proyecto. Es decir, el código puede seguir estando en tu máquina, pero el intérprete, las bibliotecas, los linters, las herramientas de construcción o incluso algunas extensiones del editor pueden vivir dentro de un entorno descrito por el propio proyecto. Así, el entorno de desarrollo deja de ser una configuración puramente personal de cada ordenador y pasa a ser una parte más explícita y compartida del proyecto (figura 6.8).

> **Para saber más**
>
> Revisa el tutorial de Dev Containers de Visual Studio Code[a]. Fíjate en la idea general: el editor se conecta a un contenedor preparado para el proyecto, y el fichero devcontainer.json permite describir parte del entorno de desarrollo.

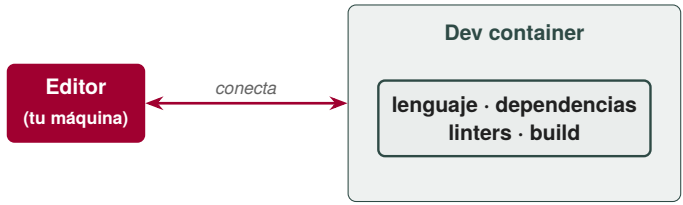


Fig. 6.8 Un *dev container* lleva la idea de entorno reproducible al puesto de desarrollo: el editor se conecta a un contenedor que define el lenguaje, las dependencias y las herramientas del proyecto.

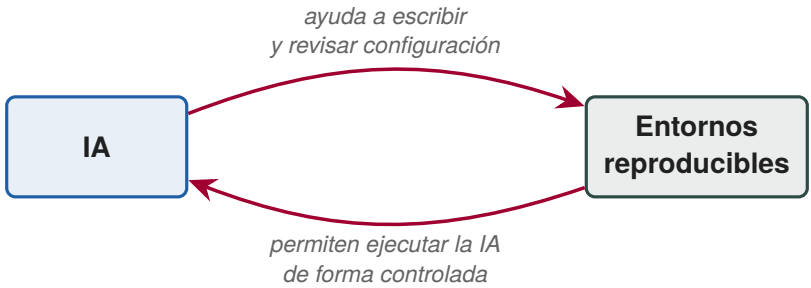


Fig. 6.9 La relación entre IA y reproducibilidad es doble: la IA ayuda a escribir y revisar configuraciones de entornos y, a su vez, los entornos aislados y reproducibles permiten desarrollar y ejecutar sistemas de IA de forma más controlada.

> Mientras lo lees, trata de identificar qué elementos del entorno quedan definidos explícitamente y cuáles siguen dependiendo de tu máquina.
>
> [a] https://code.visualstudio.com/docs/devcontainers/tutorial

6.5. ¿Y cómo afecta la IA a todo esto?

Como en los capítulos anteriores, merece la pena preguntarse cómo afecta la inteligencia artificial a las prácticas que estamos estudiando. En este caso, la relación es doble. Por un lado, la IA empieza a utilizarse para ayudar a escribir, revisar y comprender configuraciones de infraestructura, contenedores y entornos. Por otro lado, las propias técnicas de aislamiento y reproducibilidad se están volviendo importantes para desarrollar y ejecutar sistemas basados en IA de forma más controlada (figura 6.9).

Empecemos por la primera dirección. Si describir infraestructura, contenedores o entornos de desarrollo exige escribir ficheros de configuración, parece natural que aparezcan herramientas capaces de generar o revisar esos ficheros con ayuda de modelos de lenguaje. Por ejemplo, ya existen asistentes que permiten generar fragmentos de infraestructura como código a partir de descripciones en lenguaje natural, como AIaC[3]. Una persona puede pedir algo como "crea una máquina virtual con una base de datos PostgreSQL y una regla de seguridad para permitir conexiones desde esta red", y el sistema propone una configuración inicial.

Esto puede ahorrar tiempo, especialmente cuando una persona está empezando con una herramienta o quiere preparar un primer borrador. Pero conviene tener cuidado. Que un sistema de IA genere un fichero de Vagrant, un Dockerfile o un playbook de Ansible no significa que esa configuración sea correcta, segura, mantenible o adecuada para producción. Puede usar versiones obsoletas, abrir permisos demasiado amplios, ignorar costes, introducir dependencias innecesarias o no ajustarse a las políticas del equipo. Como hemos repetido varias veces, automatizar no elimina la necesidad de criterio.

También están apareciendo herramientas que no solo generan configuración, sino que ayudan a revisarla. Por ejemplo, Terracotta[4] analiza propuestas de cambio en infraestructura como código para señalar posibles problemas antes de aplicarlas. Esta idea encaja muy bien con lo que ya hemos estudiado sobre revisión de código y trazabilidad: si la infraestructura se expresa como código, entonces también puede revisarse, probarse y analizarse antes de aplicar los cambios.

Docker ha publicado Gordon[5], un asistente que puede ayudar a explicar errores, proponer cambios o navegar por tareas relacionadas con contenedores. Este tipo de herramientas pueden servir de apoyo para interpretar mensajes largos, entender por qué falla la construcción de una imagen, sugerir mejoras en un Dockerfile, detectar variables de entorno que faltan o resumir qué cambia en una configuración.

Pero la relación funciona también en sentido contrario. A medida que usamos más agentes y herramientas de IA capaces de ejecutar código, leer ficheros, lanzar comandos o modificar proyectos, se vuelve todavía más importante controlar el entorno en el que actúan. Si un agente puede ejecutar tareas sobre tu máquina, conviene preguntarse qué puede ver, qué puede modificar, a qué red puede acceder y qué recursos puede consumir. Es decir, volvemos a las mismas preguntas que discutíamos al hablar de aislamiento: ¿cuánto puede ver este proceso? ¿Cuánto puede usar? ¿Qué ocurre si se equivoca?

[3] https://www.firefly.ai/blog/introducing-aiac-ai-powered-iac-generator

[4] https://tryterracotta.com/

[5] https://www.docker.com/products/gordon/

Por eso están apareciendo soluciones que ejecutan agentes o tareas de IA dentro de entornos aislados. Docker, por ejemplo, ofrece Docker Sandboxes[6] para ejecutar cargas de trabajo de agentes en entornos locales, aislados y desechables. La idea es permitir que el agente tenga autonomía para compilar, probar, explorar o modificar código, pero sin darle acceso ilimitado al sistema anfitrión. Así, si genera un comando peligroso, instala dependencias incorrectas o modifica ficheros que no debería, el daño queda más contenido.

Además, los contenedores también empiezan a usarse para ejecutar modelos y aplicaciones de IA de forma reproducible. Igual que una aplicación web puede empaquetarse con sus dependencias, un entorno de IA puede necesitar versiones concretas de bibliotecas, aceleradores o modelos. Herramientas como Docker Model Runner[7] apuntan precisamente en esa dirección, facilitando la ejecución local de modelos dentro del ecosistema de Docker.

También aparece aquí otra conexión interesante con los agentes. Si un agente necesita usar herramientas externas, acceder a servidores MCP o integrarse con distintos servicios, vuelve a ser importante saber qué herramientas están disponibles, cómo se instalan y en qué entorno se ejecutan. Iniciativas como Docker MCP Catalog and Toolkit[8] muestran cómo el ecosistema de contenedores puede utilizarse también para distribuir y ejecutar herramientas que serán usadas por aplicaciones y agentes de IA.

Si cada vez delegamos más tareas en asistentes capaces de modificar código, generar configuración o ejecutar órdenes, parece evidente que la reproducibilidad, la trazabilidad y el aislamiento se vuelven aún más importantes.

6.6. Reflexión final: aislar es reducir incertidumbre

En este capítulo hemos recorrido una historia que empieza con instalaciones manuales, servidores copo de nieve y entornos difíciles de reconstruir, y llega hasta enfoques basados en configuración versionada, máquinas virtuales, contenedores y entornos de desarrollo reproducibles.

La idea principal es que el entorno importa. Importa para construir, probar, empaquetar, entregar y desplegar. E importa también para depurar, incorporar personas al equipo y entender por qué un sistema falla de una forma en una máquina y de otra forma en otra.

[6] https://www.docker.com/products/docker-sandboxes/

[7] https://www.docker.com/products/model-runner/

[8] https://www.docker.com/products/mcp-catalog-and-toolkit/

Ya sabíamos que automatizar nos ayuda a ejecutar pasos de forma rápida y repetible. Pero, si esos pasos dependen de un entorno accidental, la fiabilidad sigue siendo limitada. Por eso necesitamos describir, aislar y reconstruir entornos (figura 6.10).

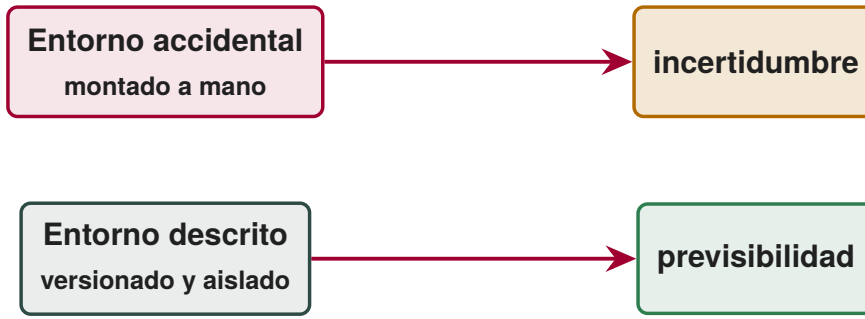


Fig. 6.10 Aislar es reducir incertidumbre: cuando el entorno se describe, versiona y aísla en lugar de montarse a mano, el comportamiento del software se vuelve mucho más previsible.

Hemos visto que hay distintos niveles de aislamiento. Un entorno virtual puede resolver conflictos entre dependencias de Python. Un contenedor puede empaquetar una aplicación con muchas de sus dependencias. Una máquina virtual puede aislar un sistema completo. Un Dev Container puede ayudar a que el entorno de desarrollo sea más compartido. La infraestructura como código puede permitirnos tratar servidores, redes y servicios como elementos versionables y revisables.

Pero ninguna herramienta elimina la necesidad de criterio. El equipo debe decidir qué parte del entorno debe fijarse, qué puede variar, qué estado debe persistir, qué dependencias se aceptan, qué riesgos se asumen y qué nivel de aislamiento merece la pena en cada caso.

Si Capítulo 3 nos mostró que gestionar código fuente es gestionar la historia del proyecto, este capítulo nos muestra que gestionar entornos es gestionar las condiciones en las que esa historia puede convertirse en software ejecutable de manera fiable.

Una historia cuidada ayuda a saber qué cambió. Unas pruebas cuidadas ayudan a saber si el sistema sigue comportándose como esperamos. Una gestión de incidencias cuidada ayuda a aprender de los problemas. Y unos entornos reproducibles ayudan a que todo ese proceso ocurra sobre una base más fiable.

El aislamiento, por tanto, nos ayuda a reducir incertidumbre. Así, cuando algo funciona, sabemos por qué funciona. Y cuando algo falla, podemos reproducirlo,

entenderlo y corregirlo sin depender de las circunstancias especiales de una máquina concreta.

> **Para saber más**
>
> Te recomendamos leer la entrevista ficticia y humorística que Alex Tatiyants realiza a un administrador de sistemas que se queja por cómo DevOps está arruinando su oficio y su artesanía[a].
>
> [a] http://tatiyants.com/devops-is-ruining-my-craft/

6.7. Lecturas y recursos recomendados

- Capítulos 1, 3, 8 y 9 del libro *Docker in Practice* (Ian Miell y Aidan Sayers).
- Capítulos 1, 2 y 9 del libro *Docker: Up & Running* (Sean P. Kane y Karl Matthias).
- Capítulo 1 del libro *Learn Docker - Fundamentals of Docker* (Gabriel N. Schenker).
- Capítulo 1 del libro *Infrastructure as Code*, (Kief Morris).
- Libro *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, (Jez Humble y David Farley).
- Libro *Continuous Integration: Improving Software Quality and Reducing Risk*, (Paul Duvall, Steve Matyas y Andrew Glover).

Apéndice A

Activación progresiva del software (*rollout*)

A lo largo de los diferentes capítulos hemos visto que desplegar consiste en llevar una release a un entorno concreto. Pero incluso cuando el software ya está desplegado en producción, eso no significa necesariamente que todos los usuarios lo estén utilizando.

En algunos casos es necesaria una fase que se conoce como *rollout* o activación. Podemos entender el rollout como el proceso de poner una versión desplegada a disposición de los usuarios. En algunos sistemas este paso puede ser casi inmediato, pero en otros requiere una política de activación más cuidadosa.

La idea principal es que desplegar software no siempre significa ponerlo en uso de inmediato. Cuando un cambio puede afectar a usuarios reales, el equipo puede necesitar decidir cómo activarlo, cómo observar su comportamiento y cómo reaccionar si algo va mal.

A.1. Activar no es lo mismo que desplegar

Aunque a veces se usan como sinónimos, creemos importante insistir en que desplegar y activar no son exactamente lo mismo. Desplegar significa que una versión está instalada o ejecutándose en un entorno. Activar significa que esa versión, o una funcionalidad concreta incluida en ella, empieza a ser utilizada por usuarios reales.

Por ejemplo, un equipo puede desplegar una nueva versión de una aplicación web en producción, pero mantener desactivada una funcionalidad nueva. El código está ya en producción, pero los usuarios todavía no lo ven. Más adelante, el equipo

puede activar esa funcionalidad para personas internas, después para un pequeño porcentaje de usuarios y finalmente para todo el mundo (figura A.1).

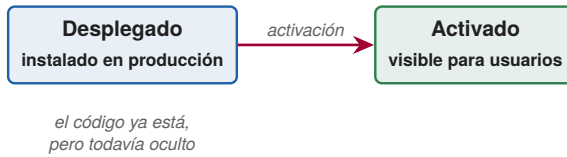


Fig. A.1 Desplegar no es activar: una versión puede estar instalada en producción (desplegada) con una funcionalidad aún oculta, que se activa después de forma controlada.

Esta separación es muy útil porque reduce el riesgo. Si cada despliegue activa automáticamente todos los cambios para todos los usuarios, cualquier error puede tener impacto inmediato. En cambio, si el despliegue y la activación están separados, el equipo tiene más margen para observar, aprender y actuar.

A.2. Activación progresiva y *feature flags*

Una técnica muy habitual para separar despliegue y activación son las *feature flags*. Una *feature flag* es un mecanismo que permite activar o desactivar una funcionalidad sin tener que cambiar de nuevo el código ni construir otra release.

La idea puede verse con un ejemplo muy simple:

Ejemplo simplificado de feature flag

```
if nueva_busqueda_activada:
    usar_nueva_busqueda()
else:
    usar_busqueda_actual()
```

Por un lado, gracias a una *feature flag*, el equipo puede integrar y desplegar código aunque una funcionalidad todavía no esté terminada. Esto permite trabajar con las técnicas de CI/CD estudiadas, con cambios pequeños y sin tener que aislar funcionalidades durante periodos largos de tiempo. Además, cuando la funcionalidad ya está preparada, la misma técnica nos ayuda a activarla de forma controlada (figura A.2).

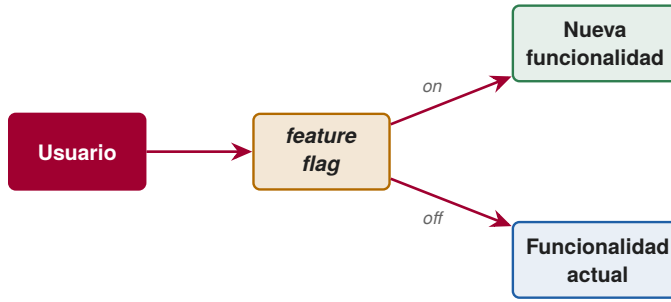


Fig. A.2 Una *feature flag* decide en tiempo de ejecución qué camino sigue la petición, sin necesidad de reconstruir ni volver a desplegar el software.

Por ejemplo, puede activarse primero solo para el equipo de desarrollo, después para usuarios beta, luego para un 5 % de usuarios y progresivamente llegar hasta el 100 %. O quizá puede hacerse la activación por tipos de usuarios, regiones o dispositivos (figura A.3).

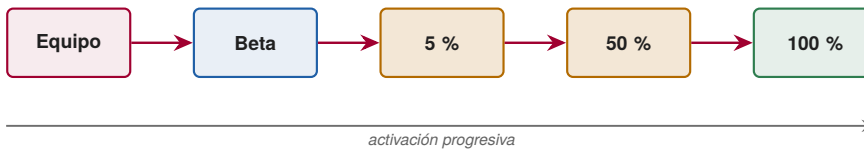


Fig. A.3 Activación progresiva: la funcionalidad se expone a grupos cada vez mayores de usuarios. Si algo va mal, se puede frenar o desactivar antes de llegar a todos.

Si durante esa activación progresiva aparecen errores, el equipo puede desactivar la funcionalidad sin tener que reconstruir o redesplegar todo el sistema. No obstante, las *feature flags* también deben gestionarse, porque si se acumulan flags antiguas, el código puede volverse más difícil de entender, probar y mantener.

Para saber más

Un grupo de investigación canadiense realizó un estudio sobre el uso de *feature flags* en el navegador Chrome a lo largo de 39 releases publicadas durante 5 años[a]. Consulta el estudio y localiza 2 datos: cómo evoluciona el número de feature flags entre la primera y la última versión analizada, y

cuántas se añadían y eliminaban de media en cada release. ¿Qué relación ves entre estos datos y la deuda técnica?

[a] https://dl.acm.org/doi/epdf/10.1145/2901739.2901745

A.3. No todo software llega igual a los usuarios

La fase de rollout depende mucho del tipo de software que estemos construyendo. En una aplicación web, el equipo suele controlar los servidores y puede mover tráfico progresivamente hacia una nueva versión. En una app móvil, en cambio, la publicación puede depender de una tienda y de que los usuarios actualicen.

Por otra parte, en una aplicación de escritorio puede haber distintos canales de actualización: un canal estable para la mayoría de usuarios, un canal beta para quienes aceptan probar versiones casi terminadas, y un canal *nightly* con versiones generadas con mucha frecuencia, pensadas para pruebas tempranas y no para uso general.

Y cuando hablamos de firmware o software embebido, la activación puede ser especialmente delicada porque un fallo puede dejar un dispositivo en mal estado.

Por eso no debemos pensar que todas las organizaciones pueden hacer rollout de la misma forma. Lo importante es que el equipo decida quién recibirá primero el cambio, establezca mecanismos para saber si está funcionando correctamente, pueda detener la activación si aparecen problemas y tenga la capacidad de volver a una situación segura.

A.4. Volver atrás: rollback, rollforward y seguridad de reversión

Cuando activamos software para usuarios reales, debemos asumir que alguna vez algo saldrá mal. Una prueba puede no haber cubierto un caso importante, un cambio puede comportarse de forma distinta con tráfico real o una dependencia externa puede responder de manera inesperada.

Ante un problema, hay dos estrategias habituales. La primera es hacer *rollback*, es decir, volver a una versión anterior que consideramos segura. La segunda es hacer *rollforward*, es decir, corregir hacia delante desplegando una nueva versión que soluciona el problema (figura A.4).

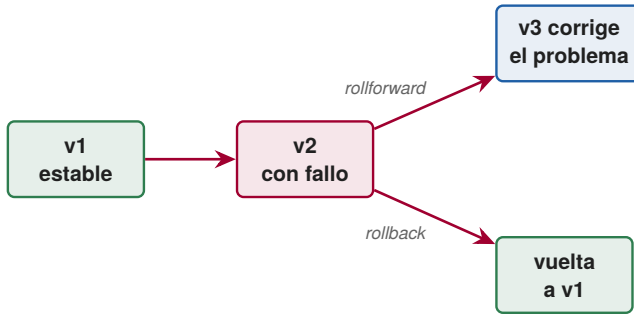


Fig. A.4 Ante un fallo en producción: *rollforward* (corregir hacia delante con una versión nueva) o *rollback* (volver a una versión anterior segura). La elección depende de la compatibilidad entre versiones.

Ninguna de las dos estrategias es siempre mejor. Un rollback puede ser muy rápido si la versión anterior sigue siendo compatible con el estado actual del sistema. Pero no siempre lo es. Imagina, por ejemplo, que una nueva versión empieza a escribir datos con un formato distinto y que la versión anterior no sabe leer ese formato. En ese caso, volver atrás podría provocar más problemas que continuar hacia delante con una corrección.

Por eso se habla de seguridad de reversión, o *rollback safety*. La idea es que, antes de desplegar, el equipo debe pensar si la nueva versión puede convivir con la anterior y si una reversión dejaría el sistema en un estado correcto. En sistemas distribuidos esto es especialmente importante, porque durante un rollout pueden convivir varias versiones al mismo tiempo. Unos servidores pueden estar ejecutando la versión nueva y otros la antigua. Si ambas versiones no son compatibles entre sí, el problema puede aparecer justo durante la transición. En algunos casos, esto obliga a dividir un cambio en varias fases: primero preparar el sistema para entender el formato nuevo y solo después activar su uso.

Para saber más

Otras técnicas muy habituales en la fase de rollout son: activaciones *blue-green*, *canary releases*, *dark launches*, *keystone interfaces* o *circuit breakers*. Lee sobre cada una de ellas y trata de entender qué problema intentan resolver y en qué situaciones podrían ser útiles. En todos los casos verás que la observabilidad es un requisito muy importante para poder tomar decisiones durante la fase de rollout. Investiga sobre cómo se estań usando en la industria

> soluciones como OpenTelemetry para ayudar a los equipos a observar si una nueva versión se comporta correctamente durante la activación.