Código Sostenible

Cómo programar código fácil de mantener

Carlos Blé Jurado

Prólogo de Javier Ferrer

savvily

ypramos1986@hotmail.com

ypramos1986@hotmail.com

Código Sostenible

ypramos1986@hotmail.com

ypramos1986@hotmail.com

Código Sostenible

Cómo programar código fácil de mantener

Carlos Blé Jurado

ypramos1986@hotmail.com

Primera edición de esta colección: valueOfFirst-Edition
© Impresión
Grafiexpress S.L
Grafiexpress
ISBN: 978-84-09-36125-0
Depósito legal: TF155-2022

A todo el equipo de Lean Mind, por inspirarme, ayudarme y facilitarme que escribiera este libro.

ypramos1986@hotmail.com

Índice general

ypramos1986@hotmail.com

Prólogo

Cada vez escuchamos más eso de que en el desarrollo de *software* está todo inventado, pero apuesto a que al finalizar el libro compartiremos que publicaciones como esta aportan valor a la profesión.

En el libro encontrarás una recopilación de principios de *software* ilustrados con ejemplos concretos y al grano. Personalmente, me ha encantado ir picoteando conceptos: desde cómo usar el patrón de diseño *Notificación* en conjunto con el tipo *Either* para modelar errores, hasta ejemplos de aplicación de la Ley de Demeter.

Carlos es de esas personas que hace que sienta orgullo de estar en esta profesión. Me encanta ver cómo alguien puede contribuir a subir los estándares de calidad de una industria sin el mínimo aspaviento. Desde la nobleza, la humildad y, justamente, quitándole esoterismo al asunto.

Coincidí con Carlos en la *Software Crafters Barcelona* de 2014. Era la primera vez que asistía a una conferencia así. Fue un momento de revelación para mí porque pude conocer a un montón de gente con la que compartía intereses y la forma de ver el desarrollo de *software* que me retaban a seguir apostando por la mejora continua.

Recuerdo que por entonces yo estaba bastante frustrado por la complejidad que empezaba a suponer testear la aplicación donde trabajaba. Como no conocía a nadie de la comunidad y me daba corte, yo simplemente me acoplaba a las conversaciones y escuchaba. Estoy convencido que escuchar esas conversaciones donde se hablaba de por qué apostar por unas determinadas estrategias de *testing*, de cuestionarnos la forma de trabajar, fue de las cosas que más me inspiraron a seguir dándole caña; y de igual forma estoy convencido de que este libro te inspirará a ti también a seguir retándote y mejorando como profesional.

Fue en esa *Software Crafters Barcelona* donde aluciné por la cantidad de conocimiento que se compartía en un fin de semana. Charlas y conversaciones brutales. Conocimiento que de forma volátil iba a desaparecer pasadas esas 48 horas. Si no habías tenido la casualidad de cruzarte con algún *tweet* sobre la comunidad, como fue mi caso, ni llegarías a saber que había pasado. Eso me empujó a agrupar todo el material de las charlas en un repositorio de GitHub, y que en la siguiente edición pusiera mi granito de arena grabando las charlas.

Al final, esa motivación por hacer difusión de las pequeñas micro-alegrías de fantasía que

tenemos en la comunidad de desarrollo, es lo que empujó a que naciera CodelyTV. Básicamente darle difusión en formato vídeo a esas cosas para potenciar la comunidad. Por lo que Carlos me pidiera hacer este prólogo cierra un círculo bastante bonito.

En el libro encontrarás este tipo de aspectos que cuando los ves te sale la sonrisilla y hasta un poquito de emoción por saber que no eres la única persona que se preocupa por esos detalles. Siempre se habla de forma despectiva de invertir tiempo en darle nombre a variables o clases. Recursos como los que encontrarás en este libro ayudan a dejar de ver esa inversión como una pérdida de tiempo de la que sentirnos culpables y pasar a ponerlo en valor y justamente empezar a mejorar ese proceso con *tips* concretos.

Luego coincidimos en una *Software Crafters Pamplona*… y el final te sorprenderá. El buen hombre propuso una sesión de meditación. Efectivamente, conferencia *Software Crafters*, sesión de meditación en el *Open Space*. Sin duda es una persona auténtica y valiente a la que de verdad espero que le sigan yendo las cosas igual de bien que hasta ahora.

Necesitamos gente como Carlos en la industria que, lejos de crearse torres de marfil y tirar abajo la escalera por donde han subido, sigan remando por acercar esto de la calidad del código de una forma práctica a cada vez más gente. Sin aspavientos ni esoterismo.

Javier Ferrer, co-fundador de Codely.tv

Prefacio

Estamos en un momento histórico en el que se ha disparado la demanda de programadoras y programadores en todo el mundo, dicen que hace falta cubrir miles o quizás millones de vacantes. Para mucha gente, programar se ha convertido en una aspiración, en una profesión de ensueño, por lo menos hasta que consiguen su primer empleo. Cada día surgen nuevas escuelas que ofrecen formación en tiempo récord para intentar cubrir esta brecha laboral del sector. Me gusta la docencia y he tenido la suerte de conocer de cerca algunas de estas iniciativas, así que en la última década me he estado preguntando, ¿qué técnicas básicas deberíamos enseñar?, ¿qué clase de competencias son las más adecuadas para las nuevas generaciones?, ¿cómo queremos que entiendan la profesión?, ¿qué valores queremos transmitirles?, ¿cuáles son las actitudes y aptitudes que aportan valor a la sociedad? Este libro responde a algunas de esas preguntas. Una de mis principales motivaciones para escribirlo era disponer de una guía básica para cualquiera que trabaje en Lean Mind, sobre todo para las personas que están empezando su carrera profesional. Ahora que por fin lo tenemos como apoyo, está dando lugar a interesantes conversaciones y debates sobre principios, prácticas y valores, que nos están sirviendo para consensuar lo que significa desarrollar *software* con calidad. Conocer y entender el legado de las anteriores generaciones de profesionales nos proporciona una base sólida y nos abre las puertas de la innovación.

Esta es la guía que a mí me hubiera gustado tener y que, hasta donde yo sé, no existía en castellano. Es un libro que regalaría a mi «yo» del pasado. Si hay ideas que te aportan, aprovéchalas, y si hay otras que no tienen sentido para ti, ignóralas, por favor. Puede que algunas no las comprendas hasta que no las practiques; vuelve al libro cuando quieras. Con el paso de los años, yo mismo seguiré haciendo cambios en mi forma de programar y el libro se irá quedando obsoleto salvo que lo actualice. Mis consejos y observaciones no son una verdad absoluta, sino el fruto de mi visión y mi subjetiva experiencia. Si discutes con alguien sobre alternativas para implementar una solución, no es recomendable que utilices este libro como argumento de autoridad («porque lo pone en el libro de código sostenible») sino que, en todo caso, te apoyes en los razonamientos que aparecen en él.

Los ejemplos de código que aparecen en el libro han sido seleccionados para que resulten lo más reales posibles. Algunos pertenecen a proyectos de código abierto, tal como lo demuestran los enlaces a los repositorios originales. Generalmente, los proyectos de *soft-*

ware libre de éxito tienen un código más legible que los proyectos privados desarrollados por programadores aislados, porque el hecho de saber que otras personas van a verlo y mantenerlo, anima a escribirlo con cuidado[1]. Aun así, ciertos listados presentados en el libro siguen convenciones de nombres que yo no comparto.

Otros ejemplos de código de este libro son adaptados de proyectos privados, con simplificaciones para que sea más fácil de leer (no quería cansarte con funciones de cientos de líneas de código). También hay código inspirado en ejercicios de programación realizados por nuestros estudiantes en prácticas. El objetivo ha sido encontrar un balance entre realismo y facilidad de lectura. Hay código que deja mucho que desear, que está buscado así a propósito, y a la vez cabe en pocas páginas. Hay ejemplos de código en varios lenguajes de programación, porque algunos me ayudan más que otros a ilustrar o a explicar conceptos. He utilizado principalmente Java, C#, Kotlin, JavaScript, TypeScript y Python, para abarcar más problemas y comparar soluciones. Me consta que la mezcla de lenguajes disuade o intimida a algunas personas, y por eso la mayoría de los ejemplos están escritos en Java. Sin embargo, este no es un libro especializado en Java. Utilizo palabras como función, método, procedimiento o subrutina, como si fueran sinónimas, porque podría estar hablando de cualquier lenguaje. Si me estuviera ciñendo a Java, solo hablaría de métodos. Espero que la mezcla de lenguajes te brinde una perspectiva amplia y que puedas trasladar los conceptos a tu lenguaje favorito. Conocer diferentes lenguajes y paradigmas me ha abierto la mente y me ha proporcionado más herramientas para solucionar problemas.

Me alegra que hayas decidido estudiar *Código sostenible*, ojalá te aporte ideas prácticas y las puedas aprovechar en tus labores diarias. Disfruta de la lectura, de los ejercicios y de las conversaciones que surjan con colegas del gremio.

Cómo leer este libro

Si estás leyendo en un dispositivo pequeño como Kindle es recomendable que lo configures para leer en modo apaisado, porque es la mejor manera de preservar el indentado de los bloques de código fuente. La sintaxis no está coloreada en los formatos epub, mobi, ni en la versión impresa (para reducir el coste de impresión), por eso hemos creado también un pdf que sí tiene colores, diseñado para leer en pantallas a color.

Pienso que este libro no es el mejor candidato para practicar técnicas de lectura rápida,

[1] Sería fantástico que los equipos de desarrollo sintieran tal nivel de satisfacción con su trabajo que no tuvieran vergüenza en publicar el código cuando hiciera falta.

porque hay una gran cantidad de sutilezas en los bloques de código y en sus explicaciones. Se requiere tiempo para poder asimilar, reflexionar e incluso debatir con otras personas. Se trata de un texto para el estudio que exige concentración. Por si te sirve para relajarte y disfrutar, te diré que no me parecería nada mal, ni raro, que el estudio de este libro te lleve todo un año, o por lo menos 6 meses.

En repetidas ocasiones, menciono los test automáticos y la técnica de *Test-Driven Development* (TDD), aunque no la describo en detalle. Para profundizar en la temática escribí otro libro llamado *Diseño Ágil con TDD*, que complementa muy bien a este (escribí dos libros completamente distintos con el mismo título, así que te recomiendo la edición con fecha de 2019 o posteriores). En el libro hay comportamientos que se ilustran mediante test automáticos. Un test no es más que un método escrito para que sea ejecutado por un *framework* de *testing*. En el caso de JUnit, el método se decora con *@Test*. Las últimas líneas del método constan de una o varias aserciones que especifican el comportamiento esperado del código bajo prueba:

Java

```java
@Test
public void sum_numbers(){
    assertThat(sum(2,2)).isEqualTo(4);
}
```

Versión del mismo test con JavaScript y Jest (*framework*):

Javascript

```javascript
it("sums numbers", () => {
    expect(sum(2,2)).toBe(4);
})
```

Los test que aparecen en el libro suponen que la aserción se cumple y, por tanto, el test se vería en verde si los ejecutásemos. Utilizo los test como una especificación técnica del comportamiento del código que se ejecuta en cada uno de ellos. Es una forma de expresar verdades sobre el comportamiento.

El libro está pensado para que lo puedas leer en cualquier orden o incluso para que leas solamente los capítulos que más te interesen, aunque ciertamente unos refuerzan o complementan lo estudiado en otros. Si acompañas las sesiones de lectura del libro con sesiones de revisión de código del proyecto en el que trabajas ahora, podrás encontrar diferencias interesantes sobre las que reflexionar. Podrías aplicar *refactoring* en una rama nueva,

también sería posible hacer un *fork* del proyecto para modificarlo, con el único objetivo de aprender, sin preocuparte de que se pueda romper algo. La idea no es criticar el código ni a las personas, sino aprender. Si además tomas apuntes y escribes en tu blog o diario sobre lo que aprendes, mucho mejor aún. Eso sí, utiliza por favor tus propios ejemplos de código en los artículos públicos, en vez de los del libro. Buscar ejemplos te hace pensar y añade valor a la comunidad. Tus artículos servirán además como difusión, lo cual te agradeceré mucho.

Estoy seguro de que hay imprecisiones y errores en el libro, por lo que te pido paciencia y apertura para no abandonar la lectura en caso de encontrar un gazapo o un error conceptual o histórico. Si nos escribes para avisar del error será un placer corregirlo y mejorar cada edición futura.

1. ¿Qué es código sostenible?

El código sostenible es aquel que se puede mantener fácilmente a lo largo del tiempo. Para ello, es necesario que su diseño sea intuitivo, que su complejidad sea la mínima imprescindible y que esté bien cubierto por pruebas automatizadas. Mantener se refiere tanto al mantenimiento evolutivo como al correctivo. No existen recetas mágicas que podamos seguir para desarrollar *software* con estas características, si las hubiera sería un trabajo industrial realizado por máquinas. Hace décadas que se está trabajando con generación automática de código (en algunos contextos resuelve y funciona), sin embargo, en el desarrollo de *software* empresarial se trabaja de forma artesanal y no parece que vayamos a quedarnos sin trabajo en la próximas décadas.

Este libro expone una serie de razonamientos, principios, patrones y heurísticas, basadas en la experiencia práctica, que nos ayudan a producir código sostenible en el tiempo. Código cuyo desarrollo se sostiene en términos de rentabilidad económica, progreso, innovación, cadencia, salud del equipo y calidad del ambiente laboral. Programar para la sostenibilidad es mucho más que «picar código», es una labor de equipo que requiere disciplina, atención y respeto. En inglés se habla en términos de *maintainable code*, pero en castellano no existe el término «mantenible», por lo cual he elegido el término *sostenible* que además tiene mayor alcance y representa mejor la esencia de este libro.

El arte de escribir código para humanos

Escribir código que funciona es relativamente fácil, sin embargo, escribirlo para que otra persona lo entienda y sea capaz de modificarlo, es todo un arte. Arte, tal y como lo define Seth Godin[1], como un acto generoso que requiere una combinación de talento, habilidad, oficio y un punto de vista diferente. Es arte cuando somos capaces de crear una obra transformadora. Un buen *software* transforma la vida de los usuarios en mayor o menor medida. Un código simple, testado, expresivo y explícito, es capaz de transformar incluso la visión del lector o lectora sobre lo que implica programar. Nuestro trabajo es valioso para las personas que usarán el *software*, así como para las que se encargarán de su mantenimiento.

[1] Godin es un empresario y escritor americano que ha sido muy inspirador para mí en cuanto a escribir libros y producir podcasts. Su definición de arte aparece en el libro, *The Practice*.

Desarrollar *software* es sobre todo un trabajo de equipo. Un equipo muy particular formado por las personas que están trabajando hoy, más las que estarán en el futuro y las que estuvieron en el pasado. Las que estuvieron nos dejaron un *software* valioso, que genera ingresos y pese a que lo hicieron lo mejor que supieron, también nos legaron limitaciones y dificultades con las que tenemos que bregar hoy. Tenemos que aceptarlo y continuar prestando el servicio, además de mejorarlo para facilitar el trabajo de quienes nos releven en el futuro. El *software* no suele envejecer bien, tiende a degenerar, porque no conseguimos recubrirlo con test de calidad ni preservar la simplicidad.

Construir *software* sin que degenere en el tiempo, o incluso para que mejore con su paso, consiste en codificar para las personas, no para las máquinas. No va de actos heroicos individuales, sino de inteligencia colectiva.

En los programas formativos para desarrolladores de *software*, no se trabaja en equipo lo suficiente, ni se habla del mantenimiento por varios motivos. Uno es el apretado currículum que están obligados a impartir, en el que no hay cabida para prácticas que se extiendan en el tiempo lo suficiente como para experimentar problemas de mantenimiento. Otro es que la mayoría de docentes no son profesionales del desarrollo de *software*, sino de la docencia, con lo cual no han experimentado las calamidades del paso del tiempo en los proyectos. El sistema educativo tampoco otorga suficiente peso a las competencias interpersonales. Sigue existiendo una gran brecha entre el mundo académico y el profesional, aunque desde ambos se realizan grandes esfuerzos por reducirla. Durante nuestra vida académica estudiamos teoría y hacemos simulacros de soluciones, aunque no llegamos a trabajar en proyectos reales (con usuarios de verdad reportando incidencias en producción). Luego, en la vida profesional trabajamos en proyectos reales, pero ya no estudiamos. Hemos aceptado que estas dos etapas son excluyentes, pero ¿tiene sentido que sea así?

Las personas que se dedican a programar de manera profesional, no suelen dedicar tiempo a escribir libros ni artículos, ni a publicar cursos en internet o impartir talleres para alumnos de instituciones académicas (enseñar es una de las formas de aprendizaje más profundas). Además, hay profesionales que dejaron de formarse cuando terminaron sus estudios, con lo que su visión de la profesión y su conocimiento de la técnica se limita a la experiencia que hayan tenido en su empresa. Por eso, en nuestro sector hay muchos cultos del cargo; *esto se hace así porque aquí siempre se ha hecho así*. Como vivimos con una sensación de urgencia permanente, no encontramos el tiempo para el estudio. En un campo tan cambiante y tan joven, la clave para progresar es seguir estudiando de manera permanente, dentro de las posibilidades de cada persona, lógicamente. La participación en eventos de comunidades profesionales es tan importante como leer un buen libro.

El idioma no juega a nuestro favor. Pese a que el castellano es uno de los más hablados en el mundo, la gran mayoría de los recursos están en inglés y suele producir desidia consumir material que no esté en nuestro idioma. Hay libros excelentes sobre técnicas de programación, pero la experiencia apunta que las personas evitan leer en inglés. Por este motivo, he visto una oportunidad de contribución con la escritura de este libro.

Esta es mi visión de la realidad, subjetiva y condicionada, como cualquier otra. Este es el cuento que me hago para tratar de explicarme las barbaridades que hacemos en el código. No me cabe duda de que, tanto desde la academia como desde la empresa, cada persona hace lo que puede para contribuir con los recursos que tiene. No se trata de culpar a los académicos, ni a los empresarios, ni a los profesionales. Se trata de apoyarles, de apoyarnos mutuamente. Nuestra empresa lleva años colaborando con diversas instituciones académicas y podemos constatar que realizan una gran labor educativa y de acercamiento de los dos mundos. Progresamos en la medida en que cada persona contribuye o aporta a la comunidad. En vez de pasarme la vida quejándome del panorama que veo, he decidido aportar mi granito de arena con la escritura de este libro, que en un mundo ideal ya no debería hacer falta.

La degeneración del código

Nos encanta empezar un proyecto nuevo desde cero, un *greenfield project*. No tenemos que bregar con código heredado y eso nos permite ir muy rápido, desarrollando funcionalidades en cuestión de días. Además, podemos utilizar la última versión de nuestro *framework* o librerías favoritas y con suerte nuestro *stack* tecnológico preferido. Vamos a poder profundizar en lo que nos gusta, o quizás sea la oportunidad de aprender una tecnología o una arquitectura nueva, lo cual, también nos gusta. Desgraciadamente, el aire fresco del nuevo proyecto no dura mucho, a las pocas semanas o meses empieza a enrarecerse ¿Por qué se echa a perder el código? Porque la complejidad accidental se nos va de las manos.

Cada problema tiene su propia complejidad de naturaleza inherente o fundamental. Por ejemplo, calcular las rutas óptimas para una flota de ambulancias o de camiones de reparto es un problema difícil de resolver. Su complejidad fundamental es elevada y, por tanto, el código será complejo. Sin embargo, el código no degenera por la complejidad de estos algoritmos, sino por otro tipo de complejidad, la que introducen los accidentes que sufre el código. Por eso se le llama complejidad accidental. Hay muchos tipos de accidentes, desde nombres y metáforas desafortunadas hasta arquitecturas desproporcionadas, pasando

por inconsistencias, a veces fruto de la rotación de personal en el proyecto.

Los accidentes en el código ocurren día a día, estropeando la salud del mismo paulatinamente. Se les da poca importancia, salvo que haya una caída de un servicio que suponga una pérdida directa para el negocio. Muchas veces, ni siquiera somos conscientes de que estamos arruinando el código, porque no hemos adquirido el nivel de sensibilidad suficiente. Nos empezamos a dar cuenta el día que el problema se hace una gran bola de nieve y provoca una avalancha. Otras veces, somos conscientes de que no lo estamos haciendo lo mejor que sabemos, pero nos puede la presión, la sensación de urgencia, la falta de conocimiento, la desidia, la cultura del equipo… y lo dejamos pasar como *peccata minuta*[2].

Mientras que la complejidad fundamental crece linealmente a lo largo de la vida del proyecto (puesto que suelen llegar nuevas funcionalidades), la complejidad accidental puede crecer exponencialmente con el tiempo hasta llegar al punto en que el equipo pierda el control del proyecto. Cuanto más retorcida es la solución implementada, más degenera el código y más cuesta revertir la situación. Puede llegar el momento en que todo el equipo se dé por vencido y entonces decida que la mejor solución es tirarlo todo abajo y empezar el desarrollo desde cero, volviendo al glorioso estado de *greenfield*. Trabajé ayudando a varios equipos que estudiaban dicha estrategia y les planteé una situación que no esperaban: «Si volvéis a empezar de la nada, ¿qué vais a hacer de forma diferente para no veros en la misma situación dentro de un año y volver a pedir que se empiece de cero?». Este libro pretende poner luz a las fuentes habituales de complejidad accidental, para aumentar la sensibilidad de las personas que programan en su día a día o que programaron alguna vez y se dedican a la gestión. Para que seamos conscientes de lo que supone añadir ese nuevo *if*, a los quince que ya hay en el mismo bloque de código.

¿Hay alguna forma de evitar la degeneración del código?, ¿se puede sostener un *greenfield* a lo largo del tiempo? Sí, es posible. He tenido la suerte de experimentarlo en varios proyectos y es fantástico. Hace falta que el código sea sostenible, lo cual requiere, entre otras cosas, que tenga una cobertura de test decente[3]. Sin test no hay gloria.

Hace años trabajé ayudando a una joven empresa que estaba empezando a desarrollar su propio *software* de gestión del negocio. Ya tenían un volumen de facturación considerable y era insostenible seguir trabajando solamente con hojas de cálculo y procesos manuales. Para seguir creciendo necesitaban automatizar más. Decidieron hacer un sistema mini-

[2] Locución latina que significa literalmente «faltas pequeñas» y se usa como «error o falta leve».

[3] Es difícil dar un número orientativo de porcentaje de cobertura de test adecuado, porque la calidad de los test incide directamente sobre la seguridad que proporcionan. El porcentaje por sí mismo no es una métrica suficiente, este artículo (https://bit.ly/3uGeoUc) lo explica muy bien.

malista adaptado a las necesidades del momento y totalmente a la medida, con muy poco *software* de terceros (algunas librerías y un *framework* conocido). Mi trabajo consistió en enseñarles a hacer test en todas las partes del sistema y en desarrollar un frontal web (la intranet de clientes), que conectaba a su *backend*. Lo hicimos practicando *Test-driven development* (TDD) y la cobertura superó el 90 %. El director técnico quedó muy contento con el trabajo, el equipo (cuatro personas) acordó que escribir test era indispensable y se comprometieron a seguir haciéndolo, por lo que me pude marchar a otro proyecto. Cuatro años más tarde, volvieron a llamarnos para pedir ayuda; habían multiplicado su facturación a lo largo de los años, con el mismo equipo, y el *software* se les estaba quedando corto para tantas peticiones. La escalabilidad del sistema era insuficiente para alcanzar el siguiente nivel de facturación. Estuvimos trabajando en la optimización de algunos procesos, introduciendo cambios en el código y en la arquitectura del sistema, manteniendo siempre una sensación de estar en un *greenfield project*. El resultado fue un éxito en tiempo y forma. Curiosamente, a la vez que hacíamos esta mejora, nos contrató otra empresa del mismo sector, que operaba de manera muy similar, pero con un equipo de desarrollo de cuarenta personas (10 veces más). Esta empresa en sus inicios decidió contratar la licencia de un *software* propietario genérico, que supuestamente era fácilmente adaptable a sus procesos de negocio: nada más lejos de la realidad, lo que nos encontramos fue un infierno. El *software* era un lastre para la compañía, no les permitía avanzar al ritmo que el negocio necesitaba y por supuesto el coste era salvaje en cuanto a salarios. Tuvieron que contratar a expertos en soporte y mantenimiento de ese *software* propietario, para ayudar al resto del equipo de desarrollo. Nos habían llamado para domar a la bestia que les tenía secuestrados, una faena que costaría varios años ¡Qué gran diferencia comparada con la primera empresa! Nunca antes había podido comparar dos casos tan parecidos y a la vez tan diferentes. La diferencia en términos de costes económicos era como mínimo de diez veces y todo por un código indomable, insolente y catastrófico.

¿Es posible reconducir un proyecto en el que hemos perdido el control? Perder el control significa no tener ni idea de lo que vamos a tardar en hacer cambios, ni en lo que va a dejar de funcionar cuando los hagamos o tan siquiera dónde hay que aplicarlos. La respuesta corta es sí, algo se puede hacer. Ahora bien, lo que se convirtió en un monstruo con los años, no se va a convertir en oro de la noche a la mañana, sino que también tomará años de trabajo. Hace falta mucha paciencia y constancia, como con todo lo bueno en la vida. Existen diferentes estrategias para trabajar con código legado y en mi opinión pasan todas por entender primero cómo es un código flexible, fácil de cambiar. De eso trata este libro.

¿Por qué es tan importante la sostenibilidad?

Por muchas razones. Primero, porque se espera que hagamos un trabajo técnico de excelente calidad. Nadie que pague por un *software* está deseando que le entreguen un trabajo mal hecho. Trabajamos en uno de los sectores mejores pagados de todos, con lo cual deberíamos ofrecer unos niveles de calidad acordes ¿Alguna vez te has parado a calcular cuánto se invierte en sueldos en el proyecto en que trabajas?, si tuvieses el dinero, ¿invertirías tu propio dinero en el proyecto? Ojalá que la respuesta fuese que sí, implicaría confianza y satisfacción por el trabajo bien hecho. Esta pregunta se la he lanzado a muchos equipos; algunos esbozan una sonrisa pícara como respuesta.

Estamos al servicio de otros negocios y de otros profesionales que esperan una mejora en su productividad gracias al *software* que construimos para ellos. La mejor manera de ofrecer soluciones ágiles y adaptadas a sus problemas es mediante el asesoramiento, el acompañamiento y la entrega continua de *software* que funciona. Es imposible entregar *software* robusto con la cadencia adecuada, si su diseño y su código son insostenibles. Por más rápido que queramos ir, cuando alcanzamos varios miles de líneas de código que no están respaldadas por test, cuesta entender y cambiar; nos volvemos lentos e impredecibles. El tiempo se esfuma depurando *bugs* o intentando comprender código. Cuanto más queremos correr, más empeoramos el código y más tardamos en dar respuesta al negocio. Lo que al principio tomaba días, ahora supone meses, y cada modificación del código tiene efectos secundarios insospechados, rompiendo con funcionalidades existentes sin que nos demos ni cuenta de ello. Este es el panorama predominante que me he encontrado en determinados proyectos. En algunas organizaciones se ha normalizado la idea de que el *software* no funcione bien, de manera que tanto el equipo técnico como las personas que usan el *software*, se acostumbran a molestias constantes y tratan de restarles importancia. Esta actitud, sostenida en el tiempo, desmotiva a cualquiera que aspire a mejorar en su trabajo. Obviamente, hay veces que toca trabajar en tareas que no son de nuestro agrado y aun así lo hacemos con ganas, de la mejor forma posible, porque somos profesionales. Le ponemos tesón a lo que venga. Lo que causa frustración, irremediablemente, es que la precariedad se mantenga en el tiempo como si fuera la vía correcta.

Por suerte, he vivido otros escenarios en los que el proyecto huele a limpio y parece un «campo verde» a pesar del transcurso del tiempo. Por eso sé de primera mano que hay otra forma de entender el desarrollo, más allá de acumular líneas de código sin tino, poner parches y hacer ñapas con prisas. De hecho, cada vez presenciamos más casos de empresas jóvenes que entienden el valor de la excelencia técnica y que desbancan a otras

muy antiguas. Hoy los equipos de desarrollo de las tecnológicas de éxito programan con test automáticos de calidad, diseñan arquitecturas adaptadas a su contexto y escriben el código para que cualquier persona del equipo lo pueda modificar.

Trabajar en *software* representa una oportunidad única de contribuir al desarrollo de la humanidad y de solucionar muchos de los retos que tenemos por delante. Para poder innovar con soluciones disruptivas a los grandes problemas, hace falta dejar de tropezar en la misma piedra: «Locura es hacer la misma cosa una y otra vez esperando obtener resultados diferentes»[4]. Picar código de cualquier manera, subestimando la legibilidad o infravalorando los test de respaldo, nos lleva siempre al mismo lugar: «aquí no se puede», «ahora no se puede», «ya lo haremos en el próximo proyecto», «en el próximo *sprint*», «otro día»… En cambio, escribir código amistoso para las personas significa darles alas para que puedan embarcarse en misiones con mayor impacto. Para poder llegar al siguiente nivel, se necesita una buena base. Desarrollar *software* sostenible tiene como recompensa el tiempo libre para la innovación y la resolución de problemas más interesantes. Cuando escribo código con la intención de ser explícito, simple y conciso, siento que contribuyo al bienestar de la empresa, de las personas que trabajan en ella y de las que vendrán en el futuro. Sentir que hago bien mi trabajo es muy satisfactorio, lo cual a su vez hace que mantenga la motivación profesional para seguir haciéndolo. Cualquiera que trabaje programando tiene a su alcance la posibilidad de realizar esta contribución, independientemente de su rol y su nivel de *seniority*. Desde que abres tu entorno de desarrollo, tienes el potencial de facilitarle el trabajo a los demás, al progreso, o de ponerles la zancadilla.

Las personas primero

La calidad del código es importantísima para la sostenibilidad del desarrollo, de lo contrario no hubiera escrito este libro, aunque me gustaría resaltar que la calidad de la relación entre las personas es todavía más importante. Un grupo de personas que trabaja en equipo siempre llegará más lejos que un individuo, por más voluntad que este le ponga. El cuidado del código no puede estar por encima del cuidado de las relaciones y las personas. El rigor y el empeño que pongo en hacer un trabajo técnico de calidad, día tras día, puede hacerme olvidar que el código no es de mi propiedad, ni es una extensión de mí mismo. El código es de todo el equipo y mi trabajo no consiste realmente en programar, sino en resolver problemas. Existe el peligro de que mi esfuerzo y mi afán programando pudieran llevarme a

[4] Pensé que esta frase era de Albert Einstein pero parece ser de narcóticos anónimos (https://bit.ly/3uHcLFS).

justificar (en mi cabeza) que me vuelva rígido y dogmático. Quizás olvide que mi trabajo como desarrollador consiste, entre otras cosas, en contribuir a que seamos un equipo fuerte, sano y motivado, que aporte el máximo valor al negocio durante el mayor tiempo posible. Escribir y mantener código es una de nuestras contribuciones, pero no es la única. Pongo todo mi afán en escribir código de calidad de la mejor manera que sé, pero reconociendo que es mucho menos valioso que las relaciones. Mantengo la capacidad de dar un paso atrás, desapegarme de «mi código» y mirar las cosas con un poco de perspectiva para no confundirme. Programar con disciplina no implica ser radicales ni extremistas.

He visto fracasar muchos proyectos por problemas de comunicación o de estrategia, no por problemas en el código. La falta de inteligencia emocional, de empatía, de actitud, de conocimiento, de capacitación… son las principales amenazas de los proyectos; son incluso la principal amenaza para cualquier organización. Nuestra mejor baza para el éxito y la sostenibilidad, reside en las habilidades no técnicas, la inteligencia intrapersonal y la interpersonal. La segunda baza es la pericia técnica, aunque solo suma (o multiplica) cuando va acompañada de la primera.

El mayor reto de un proyecto de *software* de grandes dimensiones es el trabajo con humanos; lleva tiempo que personas con diferentes visiones, trayectorias, habilidades y conocimientos hagan piña y consume energía. Es toda una inversión que requiere cuidado, porque a diferencia del código (que no se arruina en un solo día), las relaciones entre personas podrían romperse en una discusión. Un lema que me gusta recordar es, «firme con el asunto, pero flexible con la persona». Me ocupo más de observar las tendencias que las excepciones esporádicas, es decir, no hay problema si alguien incumple alguna convención o acuerdo técnico una vez, si la mayoría de las veces se respeta. Prefiero preguntar, dialogar, confiar asumiendo buena voluntad, que acusar y reprochar. Hablar cara a cara es más barato, efectivo y rentable, que poner comentarios afilados en una revisión de código asíncrona (por ejemplo, *pull/merge request*). No me gustaría que los contenidos de este libro se convirtieran en un dogma impuesto de forma autoritaria a un equipo de desarrollo. Es una guía con recomendaciones útiles para quien las quiera seguir, no para quien las reciba por imposición. Imponer con autoridad no funciona, la gente hará otra cosa desde que la autoridad les dé la espalda y se las ingeniará para sacar de contexto las tácticas y estropear el código igualmente. La base del código sostenible es la empatía, este libro podría haberse llamado código empático, como una persona me sugirió mientras lo escribía. Las discusiones que se enfocan desde la perspectiva de que una persona va a ganar con sus argumentos y la otra va a perder, son dañinas para el equipo, porque a nadie le gusta perder. Si usas este libro o cualquier otro para ganar tus batallas dialécticas, estarás

enrareciendo el ambiente de trabajo. Intenta comprender genuninamente el punto de vista y las necesidades de la otra persona y verás como entonces se abrirá a nuevas opciones.

2. Refactorización

El código más intuitivo y elocuente es fruto de una serie de aclaraciones que aplicamos al mismo cada vez que volvemos a leerlo. Rara vez conseguimos darle toda la expresividad de la que somos capaces a la primera. Estas pequeñas mejoras de legibilidad que aplicamos al código se conocen como refactorización o *refactoring*. Las aplicamos todos los días, varias veces al día. Yo practico varios ciclos de refactorización que me toman muy poco tiempo y que proporcionan una alta rentabilidad.

- Conforme he terminado de escribir un bloque y funciona, lo vuelvo a leer buscando si se puede escribir de forma más explícita.
- Cada día al empezar la jornada, dedico diez minutos a leer el código del día anterior, y para mi sorpresa, a veces me da la sensación de que lo hubiera escrito otra persona. La distancia de una noche permite que aún recuerde lo que quería hacer ayer y, sin embargo, hace que me dé cuenta de que puedo mejorar los nombres, las abstracciones o dejar mi intención más clara. Lleva muy poco tiempo aplicar cambios pequeños y seguros con el apoyo de un IDE que tenga soporte para refactorización automática.
- Conforme he terminado la funcionalidad (tarea, requisito, historia de usuario…), repaso todo el código de la misma, buscando posibles sorpresas que haya podido dejar. Cuanto más días hayan pasado, mejor perspectiva tengo para identificar código sorpresivo o desconcertante (ojo, que si dejo pasar demasiados días, pierdo el contexto y ya no me acuerdo).
- Generalmente, programo con otra persona (*pair programming*), pero cuando no es así o cuando otro miembro del equipo trabajó en solitario, nos juntamos para releer el código. Si la persona que escribió el código necesita explicárselo a la otra, ella misma se da cuenta de cómo lo puede mejorar. Por otro lado, si la persona que está leyendo el código de la compañera no lo entiende, aprovecha para preguntar y juntas lo mejoran.

El proceso es muy parecido al de escribir un libro. Si estuvieses leyendo la primera versión de cada párrafo de este libro, te resultaría confuso, inconexo, ambiguo. Seguro que más que ahora. La versión que estás leyendo, ha sido primero releída por mí mismo párrafo a párrafo, conforme los escribía, luego al día siguiente, al final del capítulo, del libro… Des-

pués, otras personas lo han revisado y han hecho sugerencias de mejora, con lo cual lo he vuelto a leer y cambiar. He leído el libro muchas más veces de las que lo he escrito ¿Hay alguna razón por la que debiéramos escribir código fuente con menos cuidado que cuando escribimos un libro? Si es un código que va a prestar servicio durante años, yo no la encuentro. Esos meses y años posteriores, el código será leído muchísimas veces, es decir, se va a emplear mucho más tiempo a leerlo que a crearlo y modificarlo. Razón de peso para escribirlo con esmero. No como si fuera una obra artística, un cuadro o una escultura, sino para que se entienda como un libro. En realidad, no lleva mucho tiempo volver a leer y retocar, no se trata de quedarnos paralizados buscando la perfección. Si en un proyecto nuevo practicas refactorización a diario, podrás seguir dedicando la mayor parte del tiempo a implementar la funcionalidad. El tiempo refactorizando no será significativo. Si por contra olvidas o pasas por alto la refactorización, llegará pronto el día en que dedicarás la mayor parte del tiempo a tratar de entender el código y cuando en el equipo os propongáis hacer refactorización, estaréis días o semanas sin escribir código nuevo. Hay que evitar llegar a esta situación y eso se consigue poniendo un poco de orden y claridad cada día. En la práctica, este libro te servirá de muy poco si no te habitúas a refactorizar unos minutos todos los días.

La táctica que recomiendo para aplicar refactorización es priorizar aquellos cambios que producen el máximo retorno de inversión con el menor riesgo. Renombrar una variable es un ejemplo de esta táctica, sobre todo si lo hacemos automáticamente con la ayuda de un IDE que se asegura de cambiar todas las apariciones del nombre y ninguna más. Sustituir un nombre por otro más apropiado aumenta significativamente la expresividad, además de ser un cambio trivial con mínimo riesgo. Mucha gente piensa que refactorizar consiste en buscar código feo y complicado, como enfrentarse a un reto, cuando en realidad se trata de buscar lo más fácil, rápido y seguro. Piensa en la refactorización como en una tarea cotidiana tal como dejar tu escritorio recogido, en lugar de abordarla como una macro reforma de una casa.

Para refactorizar con seguridad, lo ideal es contar con el respaldo de una buena batería de test automáticos que verifiquen que todo sigue funcionando tras aplicar cambios al código. Si no hay test, la recomendación es añadirlos antes de ponerse a refactorizar, tejiendo una red de seguridad que cubra las funcionalidades que puedan verse afectadas por la refactorización. En Java y C#, los IDE son capaces de hacer refactorizaciones automáticas potentes, con garantías, con lo cual no son tan críticos en todos los casos, pero en el resto de lenguajes son imprescindibles. Incluso en Java y C# hay cambios que son imposibles de verificar sin test. Los compiladores y los IDEs tienen sus límites. El código que ha sido

desarrollado sin el apoyo de test, es típicamente difícil o imposible de testar, con lo cual entramos en un círculo vicioso de ausencia de refactorización y de test. Podemos romper el círculo utilizando el ingenio, probando estrategias de *testing* de integración o incluso armando planes de pruebas manuales. Lo importante es minimizar la probabilidad de introducir defectos en el código al refactorizar y no dejar de hacerlo. Hacerlo sin miedo pero con cabeza, a ser posible trabajando en pares.

Al igual que a veces resulta más fácil borrar por completo el texto de un email que no termina de encajar y volverlo a empezar, hay veces que borrar una función o una clase es la táctica que menos problemas da en el corto, medio y largo plazo. Tenemos mucho apego al código que escribimos, como si añadir líneas fuera algo grandioso, cuando a veces lo mejor que podemos hacer es borrarlas. Borrar y volver a escribir hasta que el código sea legible es mucho más barato que seguir adelante con un código enrevesado. Suele ser enrevesado cuando tiramos millas con la primera solución que se nos ocurre, pues casi siempre hay otras más sencillas. Uno de los motivos por los que trabajamos en pares es para identificar esas soluciones sencillas dialogando, antes de ponernos a escribir código.

Code smells

La recopilación de «tufos» del código o *code smells* apareció por primera vez en el libro original de Martin Fowler, *Refactoring*. Se trata de una metáfora popularizada por Kent Beck a finales de los años noventa. Sin duda, este texto es de mis libros de programación favoritos, fue el primero que me hizo reflexionar sobre las sutilezas del diseño de *software*. El catálogo de malos olores ofrece ejemplos de código potencialmente problemático, a menudo relacionado con la cohesión y el acoplamiento. Código que podría dejar entrever carencias del diseño en cuanto al mantenimiento. La metáfora dice, «vaya, hay algo en este código que me huele mal» y a cada «hedor» le dan un nombre. Debemos tener en cuenta que son heurísticas y no axiomas, por lo tanto, no es verdad que en todos los casos sean antipatrones. Habrá situaciones en las que una supuesta pestilencia sea en realidad la manera más simple de resolver un problema y, por ende, la más conveniente. Las herramientas de análisis estático de código son capaces de reconocer algunos de estos olores, sin embargo, no siempre tienen la capacidad de ofrecernos mejores alternativas. Mi recomendación es prestar atención a las advertencias de las herramientas de análisis, sin despreciar el juicio de las personas que están programando, porque tienen mayor contexto y por supuesto capacidad de raciocinio para determinar la mejor solución (que podría ser mantener el código tal y como está). El código perfecto no existe. Estas herramientas tal vez tengan más

criterio sobre calidad del código que un humano que está comenzando en el oficio, pero a día de hoy no toman mejores decisiones que uno con experiencia y conocimientos sobre desarrollo sostenible. En la actualidad, el apoyo de las máquinas o de la automatización es una ayuda al programar, no un reemplazo del intelecto humano. Si en el futuro la inteligencia artificial da un giro a nuestro trabajo como desarrolladores, la mayor parte de este libro quedará obsoleta y tendremos otro tipo de problemas a los que enfrentarnos.

El libro de *refactoring* presenta cada olor, explica por qué podría resultar inconveniente y ofrece una receta para transformar el código eliminando su peste. Para profundizar en el tema, recomiendo estudiar dicho libro, así como otras fuentes que muestren ejemplos aplicados con código en diferentes lenguajes de programación. Es muy enriquecedor conocer que existen nombres y explicaciones coherentes para hablar de código que tiene «mala pinta».

Tiempo después de este libro, Joshua Kerievsky publicó otro llamado *Refactoring To Patterns* (https://bit.ly/34xOirB), que añade más *smells* y complementa muy bien al libro de Fowler.

Las técnicas de refactorización son tan beneficiosas y variadas que me he propuesto escribir un próximo libro sobre ello en castellano. Mientras tanto, puedes encontrar más ejemplos de refactorización en mi anterior libro, *Diseño Ágil con TDD*. Te recomiendo que además leas este completo artículo de Raquel M. Carmena.

3. Fundamentos

Una de las características fundamentales que debe tener el código para que sea sostenible el desarrollo de un proyecto con miles o millones de líneas de código, es la facilidad con la que puede adaptarse a cambios en los requisitos, ampliaciones o correcciones. Cualquier producto *software* de éxito requiere de actualizaciones. En algunos casos, se producen actualizaciones varias veces al día, mientras que en otros, es una vez al año, o incluso cada par de años, pero siempre existe la necesidad de hacer cambios, aunque solo sea para adaptar el producto a las nuevas versiones del sistema operativo, el navegador, el dispositivo móvil o cualquiera que sea la plataforma de ejecución. El coste de no modernizar el *software* ha puesto en jaque a más de una empresa que, un día, de repente, se encuentra con un sistema que deja de funcionar y nadie sabe arreglar.

Diseñar código para el presente

En el mundo de la programación existe una idea muy arraigada y extendida, que es la de escribir el código tan genérico, abstracto y complejo, que pueda resolver los problemas del futuro sin tenerlo que cambiar. Es decir, anticipar los posibles escenarios futuros, incluso los desconocidos, dejando el *software* preparado para funcionar de forma distinta mediante parámetros de configuración.

Esta es la receta perfecta para conseguir que un desarrollo se vuelva insostenible, ya que el código va a ser escrito justamente para que no requiera cambios, en lugar de para ser flexible ante los cambios. La experiencia nos dice que, en realidad, es imposible trabajar en un producto que está vivo y que aporta valor, sin tocar su código. Escribir código para el futuro es un concepto mal entendido.

El código más preparado para el futuro resulta ser el que se ciñe a lo estrictamente necesario para cumplir con los requisitos del presente. Es minimalista, simple, conciso, concreto, explícito y está bien respaldado por baterías de test automáticos. Estas cualidades hacen que resulte más fácil de entender y de modificar. Reduce la complejidad accidental y los efectos secundarios indeseados e inesperados cuando realizamos modificaciones.

Esto no significa que queramos reinventar la rueda[1] ni que nos olvidemos de la arquitectura del *software*, sobre todo de sus atributos de calidad. Los requisitos no funcionales deben cumplirse también en el presente y soportar lo que se sabe con certeza que sucederá en el futuro próximo. Si el negocio necesita que el producto soporte un millón de usuarios concurrentes, realizando una determinada operación en línea, no podemos olvidarnos de este requisito por estar haciendo las pruebas con cinco usuarios, sino que debemos cumplir con lo que se pide. No vamos a esperar a llegar al millón y que el sistema colapse para implementar una solución. Otra cuestión es que sea acertada la predicción de alcanzar el millón de usuarios en la fecha prevista, pero eso ya tiene que ver más con la estrategia de negocio que con el diseño de *software*. Introducir atributos de calidad que no son necesarios para el negocio, añade complejidad accidental en el producto y dispara su coste de mantenimiento. La ingeniería prematura supone, además, un coste de oportunidad, ya que el equipo deja de trabajar en otras características que podrían ser más ventajosas para el negocio. Es difícil encontrar el punto óptimo de inversión en las características no funcionales del *software*; se necesita que las áreas de negocio y de tecnología estén muy alineadas y entiendan mutuamente las consecuencias de sus decisiones.

Por otro lado, están los requisitos funcionales, que tienen una naturaleza distinta a los atributos de calidad. La mejor oportunidad para escribir código minimalista la tenemos cuando implementamos escenarios concretos de un campo de especialización específico. Aquí, no corremos el riesgo de quedarnos cortos, por eso hay que evitar las líneas de código *por si acaso*. Hay una táctica que arruina el código: «Ya que estoy escribiendo estas líneas por aquí, implemento algo más que no me han pedido, por si acaso me lo piden luego». Parece buena idea, pero no lo es. Sale mucho más barato consensuarlo con las personas que entienden el negocio, que inventarnos código, porque las palabras quizás se las lleve el viento, pero el código no se lo lleva.

Podría poner muchos ejemplos de generalizaciones prematuras y cargantes, uno que he visto varias veces es la validación de fortaleza de una contraseña. Pongamos que hay una aplicación que exige a los usuarios que se registren utilizando una clave muy segura, que contenga ocho caracteres, entre los cuales haya algún número, alguna letra minúscula, otra mayúscula y algún símbolo. Una solución podría ser algo como:

Java

```
public boolean isStrongPassword(String password){
```

[1] Es más que recomendable reutilizar piezas transversales de terceros como las librerías de comunicaciones, de monitorización, de seguridad, de acceso a datos, etc., que hoy en día suelen estar integradas en proyectos open source. También es posible que haya que desarrollar artefactos transversales propios.

```
    return isLongEnough(password) &&
           containsLowercaseChar(password) &&
           containsUppercaseChar(password) &&
           containsSymbols(password);
}
```

Es un código sencillo, que no obliga a quien lo lee a estar computando mentalmente, sino que se lee como si fuera un libro ¿Qué pasa si en el futuro quiero quitar alguna de estas reglas o añadir nuevas? Creo que no es ningún drama tocar este código para eliminar una línea o añadir otra, sobre todo, si está respaldado por test. Aun así, hay quien decide que, de cara al futuro, es mejor implementar un motor de reglas de validación con inyección de dependencias, configurable mediante ficheros *xml* o *yaml*, para cambiar el comportamiento de la validación sin tocar el código. Se debe al afán por diseñar la solución última, la que aguante el paso de los siglos. En realidad, lo que conseguimos con estas soluciones cargantes es complicarle la vida a las personas que toman el relevo (que podría ser uno mismo pasados unos meses o años).

Construir una carretera, un puente o un edificio, es muy diferente a construir un proyecto de *software* de grandes dimensiones. Todas estas disciplinas comparten el nombre de ingeniería, pero la naturaleza de los problemas que resuelven no tiene nada que ver. Hacer obras de mejora o rectificación de una carretera, tiene un coste muy elevado y consecuencias graves, como cortes de tráfico o grandes atascos durante meses o años. Hay que anticipar cómo va a comportarse el tráfico a la hora de diseñar una nueva rotonda, puede que incluso haya que sobredimensionarla más allá del uso que va a tener los primeros años. Nunca he diseñado ni construido una carretera, pero como ingeniero me puedo imaginar que tengan que anticipar muchos problemas para evitar rehacer la obra en poco tiempo. Al *software* le ocurre lo contrario, si el código es simple, cuesta muy poco hacer rectificaciones o ampliaciones. No tienen por qué causar molestias ni cortes notables en el servicio, ni en la productividad. Trabajamos con unos materiales extremadamente maleables, el cerebro humano y el código fuente. Cuanto más flexibles sean estos materiales, más fáciles y baratas resultarán las obras. Cuanto más sostenible sea el diseño del *software*, más valor tendrá en el presente y en el futuro. Como veremos a lo largo del libro, desarrollar *software* sostenible no es programar sin pensar ni planificar, es un trabajo de ingeniería y de artesanía.

Diseñar para el uso concreto, no para reutilizar

El otro gran malentendido es el de escribir el código pensando en que pueda ser reutilizado. Puede ser productivo pensar en la reutilización cuando el código ya ha sido usado, una vez que se ha puesto en producción y está prestando un servicio a usuarios reales, pero hacerlo antes de que se haya usado nunca es arriesgarse a complicar innecesariamente el diseño. Este mantra de la reutilización provoca que diseñemos soluciones excesivamente complicadas, como el motor de reglas del caso anterior. Pensamos que si diseñamos una pieza genérica que es capaz de validar cualquier cosa, la podremos reutilizar muchas veces. Lo que suele suceder en realidad es que tardamos mucho más en programar ese código genérico y luego es muy incómodo de usar donde sea que lo intentemos encajar. Si nos empeñamos, conseguiremos reutilizar código, aunque perderemos flexibilidad a cambio. Lo mejor que puede pasar es que no lleguemos a reutilizar ese motor de reglas, porque así la complejidad accidental se quedará en un único lugar y no por todo el proyecto.

Cuando el código se escribía en tarjetas perforadas, tenía sentido quererlo reutilizar, porque esas tarjetas salían caras hasta de almacenar. En cambio, hoy en día, escribir líneas de código nuevas tiene muy poco coste, el cuello de botella de los proyectos no está en el teclado. Lo que supone más coste es comprender y modificar las líneas de código existentes. En proyectos grandes, pasamos mucho más tiempo leyendo código (y depurando) que escribiendo código nuevo. Sale más caro escribir código «reutilizable» que código específico, justamente porque el primero es más difícil de entender y modificar que el segundo.

Incluso en proyectos donde la cobertura de test era elevada, el código estaba hecho con esmero y los *bugs* que llegaban a producción se podían contar con los dedos de una mano, desechamos la idea de reutilizar piezas para otros proyectos, porque siendo honestos, sabíamos que no entraba ni con calzador. Lo que hace que el código sea más o menos fácil de entender, son las metáforas que tiene escritas, la forma en que modela el problema y la solución. Si utiliza palabras (abstracciones) relacionadas con el dominio del problema, nos resultará familiar conociendo el dominio, y será más fácil encontrarle el sentido. Quien programa, deja más clara su intención si el vocabulario que usa en el código es concreto y expresivo. Quien luego lo lee, lo entiende mejor si averigua cuál era la intención de quien lo programó. Si la intención no está clara, o directamente se escribe sin intención, resulta muy confuso entender el código.

Aspiramos a diseñar y escribir código que sea flexible, como para seguir usándolo cuando aparezcan nuevas necesidades dentro de su dominio, pero no como para que pueda ser

usado en otros dominios. Si escribimos un *software* para un taller de mecánica, es mejor no hacerlo pensando en que nos pueda valer también para una clínica dental.

He conocido empresas donde necesitaban más personas expertas en la configuración de su producto, que personas programando. El código era tan intratable, que solo unos pocos se atrevían a cambiarlo y cuando lo hacían, rompían otras funcionalidades existentes (y eran los clientes quienes les avisaban de las roturas). El área de negocio de la empresa quería avanzar rápido desarrollando nuevas funcionalidades, pero el equipo técnico no podía ir más rápido, vivían estresados y con miedo. Lo que antaño se tardaba en hacer un par de días, ahora ocupaba muchos meses y los clientes se cansaban de esperar. Hay varias razones por las que se había llegado a este punto; la complejidad accidental era una de ellas.

Las reglas del código sostenible

Escribir código sostenible es más difícil de lo que parece y por más esmero que pongamos, pasará que cuando leamos el código que escribimos meses atrás, nos parecerá tosco y no tan claro como pensábamos. No hay un conjunto de recetas que podamos seguir para conseguir desarrollar *software* fácil de mantener, este es un oficio artesanal que requiere estar tomando pequeñas decisiones de diseño todos los días. Utilizamos conceptos como *Domain-driven design* (DDD), patrones de diseño, patrones de implementación, etc., para aumentar la expresividad de nuestro código, y aún así, a veces las buenas intenciones terminan añadiendo complejidad accidental. Damos demasiadas vueltas de tuerca y acabamos diseñando sistemas excesivamente complicados.

¿Existe el mínimo común denominador del código sostenible? Probablemente no, pero para mí fueron muy inspiradoras las cuatro reglas del diseño simple de Kent Beck, que dicen que el código debe:

1. Pasar los test: obviamente, para que los test puedan pasar, debe haber test. Todavía hay demasiados proyectos que no tienen test y demasiados equipos que los ven como un extra idílico, inalcanzable o prescindible.
2. Revelar la intención: el código debe estar escrito con una intención notable.
3. No contener duplicidad.
4. Tener el menor número de elementos posible.

Las últimas dos reglas se malinterpretan con frecuencia. Duplicidad no significa necesa-

riamente que un bloque de código aparezca dos veces, sino más bien que el conocimiento no esté repetido. Significa no duplicar abstracciones ni artefactos, pero el código podría estar repetido de casualidad. Sandi Metz tiene una frase muy sabia que dice «La duplicación es de lejos, más barata que la abstracción incorrecta». La cuarta regla habla de reducir lo máximo posible la cantidad de elementos que añadimos, con el propósito de simplificar el diseño, de evitar la sobreingeniería. Corey Haines lo explica muy bien en su libro *Understanding the Four Rules of Simple Design*, dice que él se hace preguntas cuando termina de implementar las primeras tres reglas, tales como *¿hay código que ya no se usa?*, *¿me he pasado extrayendo piezas?* Es una buena regla para tener en mente cuando refactorizamos.

Las reglas de Beck, han sido para mí una guía muy útil en los momentos en que me costaba especialmente tomar una decisión de diseño. Con el paso de los años y las experiencias en diferentes proyectos, las he ido adaptando tanto para mí, como para explicarlas a otras personas. Parto de ellas para darles una vuelta de tuerca y definir las reglas del código sostenible, que son las siguientes.

1. El código está cubierto por test

Modificar código que no está respaldado por una sólida batería de pruebas automatizadas, entraña un elevadísimo riesgo de estropear múltiples de sus funciones. Lo peor es que quizás no lo advertimos en el momento, sino días, semanas o meses más tarde, cuando una persona está usando el producto y sufre una pérdida de sus datos, su dinero o su tiempo. No existe un umbral mínimo universal aceptado como porcentaje de cobertura, ni es necesario testar por igual todas las funciones, porque algunas son más críticas que otras. Las funcionalidades más importantes deberían contar con un porcentaje de cobertura cercano al 100 %. Luego hay código de métodos simples que quizá no necesite test, porque incluso puede que ya esté respaldado por el compilador. Si hacemos la media de estos dos escenarios, podemos pensar que la cobertura deseable podría estar por encima del 50 %. En mi experiencia, cuando desarrollamos productos con TDD, la cobertura suele estar en torno al 90 %. Una cobertura global del 100 % no tiene sentido, porque terminaríamos con test contraproducentes o innecesarios. Llega un punto en que mayor cobertura de test no es mejor, sino peor. Cualquier IDE hoy en día sirve para medir la cobertura, pero también están las herramientas de análisis estático de código, que miden este y otros parámetros.

Los test automáticos son imprescindibles en todo código que se considere de calidad. Si no se ha desarrollado una cobertura de test adecuada, no hay ninguna garantía de que funcione. No existe el código de calidad sin test.

2. Los test son sostenibles

Sí, esta definición es recursiva; simplemente quiere decir que el código de los test tiene la misma importancia que el código que se ejecuta en producción, por lo que su sostenibilidad es igualmente indispensable. Un proyecto grande con una sólida cobertura de test acumula decenas de miles de ellos. Si el código de esos test es intratable, serán una trampa, en lugar de una ayuda.

No importa si vamos escribiendo cada test antes del código de producción (*test-driven* o *test-first*), o si los escribimos *a posteriori*. Esto no tiene por qué afectar a la calidad del código, ni de sus test, aunque yo personalmente encuentro más productivo, ameno y simple, seguir el ciclo de TDD.

Lo que importa es que el código de prueba sea igual de conciso, expresivo y concreto, que cualquier otro código que escribamos; que los nombres de las pruebas hablen del caso de uso con un lenguaje de negocio; que cada test ejercite un único comportamiento del sistema, para que solo falle por un motivo. Hay toda una serie de recomendaciones sobre cómo escribir los test para que aporten valor y sobre todo para que sean fáciles de mantener en el tiempo. Es un error muy común tratar los test como código de segunda clase, como si no importase su calidad, pero resulta que también han de ser mantenidos en el tiempo. En el libro *Diseño Ágil con TDD*, dedico varios capítulos a hablar sobre los test, independientemente de que se escriban practicando TDD o no.

3. Las abstracciones tienen sentido

Cuando utilizamos un lenguaje de programación, cada palabra del código cuyo nombre podemos elegir es un concepto que introducimos. No tenemos control sobre las palabras reservadas del lenguaje, ni sobre sus símbolos, pero sí nos vemos comprometidos a elegir nombres para conceptualizar el resto del código. Los conceptos son el modelo simplificado con el que entendemos la realidad que nos rodea. La palabra *mesa* no es una mesa, sino cuatro letras juntas, pero en tu mente acabas de imaginar una estructura de madera con un tablero y cuatro patas, o quizás de metal con dos patas… la palabra *mesa* es una abstracción[2]. Las abstracciones son poderosas a la par que imprecisas, de hecho, uno de los problemas clásicos de la toma de requisitos para un nuevo proyecto es que sean demasiado abstractos y cada persona tenga ideas distintas de lo que se quiere. Una de las

[2] En lingüística, Saussure habla de significado y significante para diferenciar la palabra escrita de lo que representa mentalmente - Saussure , F. (1945). *Curso de lingüística general*. Buenos Aires: Editorial Losada.

definiciones de abstraer, según la RAE es: «Separar por medio de una operación intelectual un rasgo o una cualidad de algo para analizarlos aisladamente o considerarlos en su pura esencia o noción».

Poner nombres es una de las tareas más duras de la programación y todo el mundo piensa que se le da mal. Es una habilidad que se entrena como cualquier otra, se aprende a base de esforzarse por buscar buenos nombres. Te invito a que deseches la idea de que siempre se te va a dar mal poner nombres, porque entonces no te esfuerzas por buscarlos, asumes que eso no es lo tuyo y lo sigues haciendo sin cuidado. Si te convences de que nombras mal, puede que la profecía se cumpla y nombres cada vez peor. El impacto que los nombres tienen en el código es radical, ya que cada nombre es una abstracción que determina el concepto de solución que se crea la persona que luego lo lee.

Es paradójico que a la hora de programar no haya ninguna otra habilidad con tanto impacto en la calidad del código, como la de poner nombres, y que a la vez esté tan infravalorada o sea tan ignorada, pese a ser bien conocida.

Uno de los motivos por los que introducimos abstracciones es para sacar factor común de código repetido. Cuando extraemos un método o función para eliminar duplicidad, nos vemos en el compromiso de poner un nuevo nombre. Si no encontramos un nombre obvio para esta nueva abstracción y le ponemos cualquier cosa, estaremos haciendo que el código sea más duro de entender. Cuando no somos capaces de nombrar un nuevo elemento (sea un método, una variable, una clase o cualquier otra cosa), puede que en realidad esa abstracción no tenga sentido y que sea mejor no crearla (o sea, no extraer ese método, variable, clase…). Esto suele ser así cuando no tenemos ni idea de qué nombre podríamos usar. En otras ocasiones, tenemos la intuición de que la abstracción tiene sentido, pero no encontramos un nombre que nos termine de gustar, por eso recomiendo practicar refactorización más tarde o al día siguiente, cuando se vuelva a leer el código.

La mejor forma para no equivocarse al poner nombres es no ponerlos, por eso, a veces conviene unir expresiones en una misma línea y así nos quitamos de encima un compromiso.

Ejemplo de típicas variables que no dicen nada (*“aux”, “tmp”, “str”,…*):

Java

```java
String aux = firstTransformation(payload);
String result = secondTransformation(aux);
return result;
```

Podemos eliminarlas perfectamente:

Java

```java
return secondTransformation(firstTransformation(payload));
```

Esto no significa que siempre debamos tener todo en una línea, cuidado con sacar las cosas de contexto, que hay mil situaciones en las que el código se lee mejor partiendo la operación en varias líneas, mediante el uso de variables explicativas con los pasos intermedios. Por ejemplo:

Esto me obliga a computar en la cabeza:

Java

```java
if (getAppearancesCount(numbers, theNumber) % 2 == 1)
```

Mientras que esta alternativa me hace pensar menos:

Java

```java
int appearancesCount = getAppearancesCount(numbers, theNumber);
boolean hasOddNumberOfAppearances = appearancesCount % 2 == 1;
if (hasOddNumberOfAppearances)
```

Cuando el nombre de una función/método es adecuado, no tengo la necesidad de entrar a leer su contenido para entender lo que va a hacer. Por tanto, otra pista sobre la idoneidad de una abstracción es si somos capaces de encontrarle sentido sin tener que ir más allá en el código, buscando información adicional.

Ejemplo de información insuficiente:

Java

```java
return reduce(e); // ???
```

Mejor explicado así:

Java

```java
return calculateTotalCostOf(expenses);
```

Si escribimos código pensamos en los posibles escenarios del futuro o en reutilizarlo, tal

como decía al principio del capítulo, utilizamos generalizaciones muy abstractas. La generalidad no respeta nombres concretos del dominio, porque pretendemos que «el motor de reglas» valide cualquier cosa o que «el sumatorio de costes sume cualquier matriz». Entonces empezamos a inventar nombres que no existen en ese dominio, que las personas expertas en el negocio no utilizan, ni conocen. Cuando meses después se incorpora alguien al equipo de desarrollo, se encuentra con un negocio que seguramente no conoce (en cuanto a que no sabe cómo funciona el sector) y con un código que le habla de otras metáforas que no son las de ese negocio. Cuanto más extraña sea la metáfora elegida, más se va oscureciendo el código con el paso de los días. Si además falta consistencia en los nombres, por ejemplo, si a un mismo concepto le crían alias esparcidos por los diferentes ficheros del proyecto y se le llama de cuatro formas diferentes, todavía es mucho más confuso. Estos dos ingredientes del caos son muy frecuentes, metáforas alienígenas con nombres mutantes pululando por el código. Por si no hubiera ya suficientes palos metidos en la rueda del equipo de desarrollo, a veces se aliña todo con arquitectura e infraestructura de magnitudes totalmente desproporcionadas. Aquello que nace con la idea de resistir el paso del tiempo, se come el tiempo de todo el equipo.

Cualquier abstracción que diseñemos va a tener sus límites y sus agujeros, tal como bien explica Joel Spolsky en este artículo. Cuanto más temprana sea la creación de una abstracción, más posible es que erremos con ella, sobre todo cuanto más grande sea el artefacto. Por eso, algunas personas preferimos apoyarnos en el diseño emergente, basado en la refactorización, y esperar a que el código haya alcanzado buen grado de funcionalidad para introducir una abstracción. Hasta que el código no revele una estructura que case bien con una posible abstracción, de manera sugerente, prefiero no hacerla. No importa que el tamaño del método en el que estoy trabajando llegue a las cincuenta líneas o las que hagan falta. De acortarlo siempre hay tiempo. No importa que la clase vaya teniendo muchos métodos, de partirla en otras clases siempre hay tiempo. Digo que siempre hay tiempo si partimos de la premisa de que la cobertura de test sea adecuada. Las abstracciones prematuras introducen una complejidad accidental tremenda. Refactorizar cuando se tiene una gran cobertura de test, es seguro y productivo. Es como esculpir la piedra para darle la forma adecuada. El proceso contrario, el de diseñar todas las abstracciones con diagramas antes de programar, es como hacer castillos en el aire. La maestría en el diseño está en saber cuánto diseño debe hacerse *a priori* y cuánto debe emerger con refactorización, porque en ese balance está la virtud. Un diseño 100 % emergente sería un desastre y lo contrario también lo es.

En la última década se ha demonizado la duplicidad en el código, pero en realidad el prin-

cipio DRY (*Don't repeat yourself*)[3], no se refiere a líneas de código que son iguales, sino a evitar resolver problemas por duplicado. Puede haber líneas o bloques de código que casualmente sean iguales y que en realidad estén resolviendo problemas diferentes. De ser así, con el tiempo es probable que ambos fragmentos de código evolucionen de manera distinta. Si desde el instante en que vemos dos bloques o líneas de código iguales eliminamos la «duplicidad» aparente, tendremos problemas para que el código se adapte a los requisitos futuros.

Los problemas de repetirnos son mayores cuanto más grandes/complejos son los artefactos y cuantas más veces se repita la solución por diferentes módulos del código. Por ejemplo, si se ha implementado un validador de datos de entrada/salida para un módulo del código, evita escribir otro para los otros módulos. Que en el proyecto haya dos validadores diferentes es confuso para quien los tenga que usar o modificar, genera una incertidumbre que puede disuadir a cualquiera de hacer cambios. Si les da miedo tocar, no harán tareas de mejora de la legibilidad y tendremos un monstruo de dos cabezas. Cuando se detecte un problema, habrá que corregirlo en dos o más sitios. Cuanto más crezcan los artefactos repetidos, peor. Cuanto más dispersa esté la repetición, peor (mejor tres repeticiones en un fichero que una repetición en tres ficheros diferentes). Un motivo típico por el que se repiten las soluciones en los proyectos, es que la persona que tiene que meter mano en la solución existente no la entiende o no se siente con la confianza de modificar sin romper y decide que es mejor implementar su propia solución. Por tanto, DRY aplica a conocimiento repetido, no necesariamente a código repetido.

Me ha tocado trabajar con códigos heredados de todos los tipos: en un extremo, código con complejas generalizaciones, aparentemente sin líneas de código repetidas. En el otro, código con bloques de cuatro o cinco líneas, copiados y pegados (literalmente) uno debajo de otro, hasta ocupar cuarenta líneas. En este segundo caso, era muy obvio que el código estaba realizando una operación concreta, que estaría mejor si fuera una función, porque así la llamamos varias veces y quitamos una duplicidad muy obvia. Era feo, no pude explicarme por qué alguien había preferido copiar y pegar, en vez de extraer una función. Lo cierto es que no me costó mucho entender el patrón, ni tampoco refactorizarlo, porque su estructura era muy simple, era programación estructurada estilo *script*. Lo peor en este tipo de códigos de la vieja escuela suelen ser las abreviaturas de los nombres (que son crípticas) y los números mágicos (*magic numbers*).

Java

[3] La mejor explicación sobre DRY es la de los autores que acuñaron el término y está en la edición del 20 aniversario del libro *The Pragmatic Programmer*.

```
int ctx = zhk + 45; // ???
```

A pesar de esta dificultad, un viejo *script* ha sido menos duro para mí que un sistema con complejas abstracciones. Cuando he tenido que bregar con artefactos genéricos super configurables, ahí sí he pasado fatiga. Primero, para entender las abstracciones; y luego, para identificar los distintos usos que podía tener, tratando de entender si podía estar rompiendo cualquier cosa (ya que la gran mayoría de las veces no hay test en el proyecto).

Generalizar código es más fácil que deshacer una generalización. Si tengo diez líneas de código iguales consecutivas, resulta muy fácil introducir un bucle que contiene una sola línea e itera diez veces. Lo contrario ya no es tan fácil, porque la carga cognitiva de comprender el bucle, con sus intervalos, es mayor. Implica mayor probabilidad de equivocarme e introducir un *bug*. Cuanto más compleja sea la estructura que hemos construido, cuanta más complejidad ciclomática tenga, más esfuerzo cognitivo nos exigirá, lo que se traduce en mayor resistencia al cambio. Esto no significa que dejemos de utilizar bucles, sino que seamos conscientes del coste de mantenimiento que añadimos al *software* con cada generalización. Si acertamos con ella, conseguimos ahorro, pero si nos equivocamos, aumentamos el coste. A menudo, hay mucho que perder y poco que ganar cuando generalizamos.

Ante la duda, es preferible posponer las generalizaciones y la creación de nuevas abstracciones. El mejor momento para hacerlo suele ser cuando hemos terminado de implementar un requisito, todos los test pasan y volvemos a leer el código en busca de obviedades que puedan ser refactorizadas.

4. Hay una intencionalidad explícita

Cuando heredamos código que hay que modificar, bien para añadir funcionalidad o bien para corregir defectos, lo primero que hacemos es intentar comprenderlo. Típicamente, las personas que lo escribieron ni siquiera trabajan allí ya o no están disponibles para explicarlo, ni para resolver dudas. Inconscientemente, buscamos patrones, convenciones, consistencia, tratando de averiguar cómo pensaban las personas que lo escribieron. Inspeccionamos el código como Jessica Fletcher inspeccionaría la escena de un crimen, buscando irregularidades, prestando atención a todo lo extraño.

Por ejemplo, si todos los nombres de las clases empiezan con letra mayúscula y de re-

pente encontramos una con minúscula, la marcamos mentalmente como sospechosa y asumimos que debió ser escrita en minúscula a propósito. Le damos unas vueltas al asunto, *¿por qué la habrán escrito en minúscula?, ¿qué querrían expresar con esto?* Si existe un mecanismo para el acceso a datos que se usa en toda la aplicación, salvo en un punto, entonces nos hacemos la misma pregunta sobre ese punto, *¿por qué habrán dejado ese código accediendo a la base de datos de otra forma?* Cada irregularidad que encontramos nos va produciendo un poquito más de incertidumbre y de miedo a tocar el código. Al fin y al cabo, *¿quién sabe por qué ese código no sigue la convención de acceso a datos?, debieron de tener una buena razón para ello, así que no les vamos a contradecir, lo mejor será dejarlo así*.

Es curioso cuanto menos que inspeccionemos el código de otros como si cada línea estuviera escrita con intencionalidad, con un propósito. Desgraciadamente, la realidad dista mucho; no acostumbramos a poner intención al programar, con lo cual, esa clase estaba en minúscula por despiste y ese código accedía a los datos, porque la tarea fue implementada por otra persona que ignoraba la existencia de la capa de acceso a datos. En realidad, el código está plagado de accidentes fruto de las prisas, el desconocimiento, la inconsciencia, el menosprecio, los problemas de comunicación, la falta de atención, de concentración…

La consecuencia de los accidentes es el empobrecimiento paulatino del código. No arruinamos un proyecto en un solo día, sino con la suma de pequeños accidentes a lo largo de semanas, meses y años. Los accidentes disuaden a quien mantiene el código de realizar mejoras (sobre todo cuando no hay test de respaldo). Ya cuando el caos termina reinando, la gente se acostumbra y se deja llevar por la inercia del «todo vale». Llegados a ese punto, al equipo le parece normal escribir código sin cuidado, haciendo ñapas por todos lados como si fuera lo correcto. Lo habitual es que el código vaya a peor cuando empieza a ser caótico.

De las muchas formas que hay de resolver un problema con código, procura elegir aquella en la que tu intención quede lo más clara posible, que resulte evidente para futuros lectores del código. Escribe cada línea de código deliberadamente, incluso cuando utilizas líneas en blanco entre bloques. Luego, sé consistente con ese estilo, no tomes decisiones arbitrarias, porque el proyecto se arruina o se mejora línea a línea, día a día.

Echarle la culpa a las personas de los desastres en el código es como escupir para arriba; no te lo recomiendo, no soluciona el problema y estropea las relaciones. Cada persona y cada equipo hace lo que puede con los recursos que tiene en ese momento, no sacamos nada culpando a los otros. Alguna vez llegué a pensar que quien había escrito cierto código

debió haberlo hecho de mala gana, hasta que descubrí que había sido yo mismo años/meses atrás. . . Cuando nos toca trabajar con un código heredado, tenemos la oportunidad de limpiarlo, al menos hasta dejarlo un poquito mejor de lo que lo encontramos. Lo mejor que podemos hacer para sumar valor es escribir el código con intencionalidad, ayudando a otras personas a darse cuenta de lo importante que es mediante el ejemplo.

4. Técnicas para elegir nombres

Nombrar es muy difícil, sobre todo cuando no dispones de ninguna guía orientativa. Este capítulo recopila una serie de recomendaciones que te servirán de punto de partida. Los nombres son adecuados cuando consiguen que el diseño quede intuitivo y simple ¿Qué características debe reunir un nombre para escribir código sostenible? Veamos algunas:

Nombres fáciles de pronunciar

Los nombres de las variables y del resto de elementos deberían ser pronunciables en el idioma que sea que hayamos escogido. En la mayoría de equipos en los que he trabajado decidimos usar el inglés, mientras que en otros casos fue muy problemático traducir términos de negocio del castellano al inglés. En parte, porque el nivel de inglés no era suficiente, con lo cual se decidió poner los nombres en castellano y funcionó mejor para la legibilidad del código. Utilizar el inglés va muy bien si el equipo encuentra rápidamente las palabras a la hora de nombrar. Resulta casi obligatorio usar inglés si el proyecto es internacional, si trabajan personas que no entienden el castellano. En otros contextos, es más práctico poner los nombres en el idioma nativo del equipo (considero que las personas expertas en el negocio son parte del equipo), porque elegir buenos nombres es ya de por sí un problema suficientemente complicado.

Un nombre pronunciable no empieza por símbolos como el guión bajo (*underscore*), ni utiliza convenciones/notaciones que se usaban en otra era (como la húngara), cuando el código se enviaba a la impresora matricial. Si utilizamos la regla de nombres pronunciables, no podemos comernos letras de las palabras para acortarlas. Los nombres han de ser tan largos como sea necesario para representar el concepto adecuado. Sería extraño y seguramente inadecuado un nombre con ochenta caracteres, pero no hay problema si tiene diez, veinte o incluso más. Ya no existen las limitaciones de hace décadas en cuanto a la memoria y a los compiladores, que obligaban a utilizar variables con pocas letras. Que sea un nombre excesivamente largo, podría indicar que la abstracción no es buena, pero no supone un problema de rendimiento. Ciertas comunidades alrededor de un lenguaje de programación, a veces insisten en acortar los nombres al máximo utilizando códigos y abreviaturas como *addr* para *address*, pero eso no significa que sea una buena idea. Es cier-

to que el ámbito de una variable influye en la elección de su nombre, ya que si es un ámbito muy reducido el nombre puede ser corto, aunque yo insisto en que sea pronunciable y tenga más de una letra. Los índices de los bucles pueden ser la excepción de la regla, a los que típicamente nombramos con letras, *i, j, k*, y queda más legible que usar palabras completas. Otra excepción podría ser el parámetro de una expresión *lambda* corta. El peligro de usar nombres cortos, aun para ámbitos acotados, es que esos ámbitos podrían crecer y entonces la información del nombre ya no sería suficiente. Por ejemplo, una función de una, dos o tres líneas, se presta a contener variables con nombres cortos, pero ¿cómo sabemos que no va a aumentar el número de líneas en el futuro? Es preferible no quedarse cortos con los nombres, estirarse lo que haga falta. Lógicamente, sin ser redundantes ni exagerados, porque si las líneas llegan a doscientos caracteres, tendremos que desplazarnos a la derecha para leer y será fastidioso (aunque con el tamaño de las pantallas y las resoluciones que se manejan hoy en día, no suele faltar ancho). Puede resultar molesto leer nombres largos si el código está en papel o en un libro electrónico, pero generalmente el código se escribe para ser leído en una computadora con un IDE (*Integrated Development Environment*). En este libro he tomado algunas decisiones con respecto a ciertos nombres, que no serían iguales si los hubiera escrito para ser leídos en un IDE (también respecto a la indentación y al tamaño de las líneas). El contexto tiene un gran impacto sobre el estilo. Si el código no es para un libro y no hay una restricción real que impacte en el consumo de recursos o en algún otro atributo de calidad como el rendimiento, no hay necesidad de abreviar, a no ser que queramos presumir de escribir algo que nadie más puede entender. Alardear de código oscuro era típico de los programadores en los ochenta y noventa, en comunidades como la de Perl. El lenguaje Perl no tenía la «culpa» de que lo usaran para escribir código ofuscado, fueron sus usuarios quienes terminaron por arruinar su popularidad. Que se pueda programar con un cierto estilo no significa que se deba. Las máquinas se tragan cualquier programa que compile, mientras que los humanos pensamos de manera abstracta.

Sin información técnica

El hecho de que un cierto *framework* utilice unas convenciones de nombrado, no significa que cuando lo utilizamos para construir nuestras aplicaciones debamos seguirlas. El *framework* tiene un propósito general, es una pieza horizontal, mientras que las aplicaciones tienen un propósito específico, son piezas verticales. Un ejemplo típico es el *framework* .*Net*, cuyas interfaces empiezan con una letra «I» mayúscula, es decir, la primera letra de la palabra *Interface* (*IList, ISet…*). Puede tener sentido en un contexto tan genérico como es

el de este *framework* tan ámplio y versátil, aunque en el *framework* de colecciones de Java y en otras librerías no se utiliza esta convención y no hay ningún problema por ello. Que sea una convención usada por el *framework* .*Net* no significa que debamos aplicarla en nuestras aplicaciones .*Net*. En general, debemos evitar reflejar información sobre tipos en los nombres. Por ejemplo, evitar que diga que el tipo de una variable *string*, interfaz, clase abstracta, implementación de una interfaz…

- *sSurname, IShoppingCart, AbstractShoppingCart, ShoppingCartImpl, IAsyncUserFinder,…*

La información sobre los tipos era útil cuando leíamos código en papel continuo, pero hoy en día ya no es necesaria y de hecho es contraproducente. La razón para desaconsejar nombres que contienen información técnica es que nos impiden elegir nombres con un nivel de abstracción adecuado. Esto tiene un profundo impacto en el diseño ¿Qué sucede cuando tienes una interfaz y solamente una implementación de la interfaz? Lo típico con C# es usar el prefijo «I» en la interfaz, mientras que lo típico en Java es añadir el sufijo *Impl* a la implementación. Ahora bien, que lo hayamos visto muchas veces no significa que sea la mejor táctica. Si te cuesta encontrar dos nombres diferentes para interfaz e implementación y te conformas con seguir este estilo, quizá te pierdas la posibilidad de encontrar un diseño mejor. La dificultad a la hora de nombrar, a menudo, nos brinda pistas sobre la calidad de nuestras abstracciones y sobre el diseño del *software*. Es como si el código nos hablase, los nombres nos orientan para escribir código sostenible y para diseñar intuitivamente. Con frecuencia, la dificultad para ponerle nombre a una interfaz te está diciendo que no necesitas que exista una interfaz. En C#, Java, TypeScript y tantos otros lenguajes, una clase también es una interfaz y está definida por sus métodos públicos. La palabra reservada *Interface*, se utiliza cuando varias clases comparten una misma interfaz, en cuyo caso es fácil encontrar nombres para dicha interfaz y sus implementaciones (ej: *UserRepository, PostgresUserRepository, MongoUserRepository*).

Un argumento que me he encontrado varias veces al respecto de tener una interfaz separada para cada clase (única implementación), es el de que reduce el acoplamiento. Mi experiencia trabajando con código legado rechaza esa teoría. Hay casos en los que sí y casos en los que no. No cabe duda de que quien escribió el código así, lo hizo con su mejor intención, sin embargo, lo cierto es que resulta muy molesto tener que navegar saltando de interfaz a clase cada vez que necesitas ver la implementación. La cantidad de indirección que se introduce hace más difícil seguir y entender el código, debido a todos los saltos que obliga a dar.

Al emplear nombres que incluyen información técnica sobre tipos (por ejemplo, usando prefijos/sufijos como *Abstract* o *Base*), tenemos la falsa sensación de estar haciendo el código más legible, cuando en realidad le estamos dando información redundante que ya provee el IDE (con el resaltado y coloreado de sintaxis, las sugerencias y los metadatos que muestra al señalar los elementos con el ratón). Entender los tipos de las variables es fácil, lo complicado es entender los conceptos que representan el dominio. Lo importante no es saber que una variable es de tipo cadena, sino entender para qué se usa, qué función cumple en el conjunto del programa. La información sobre los tipos o incluso sobre los patrones utilizados (usar sufijos *Singleton*, *Facade*…), no añade abstracción con semántica de negocio, simplemente ofrece una descripción técnica. A donde quiero ir es a que al aplicar la restricción de no añadir información técnica, va a resultar más difícil poner nombres y es muy posible que incluso terminemos cuestionando el uso que hacemos de ciertos patrones. No obstante, la elección de nombres no es una ciencia exacta y habrá ocasiones en las que lo más intuitivo sea utilizar prefijos como *Abstract*; lo que propongo es que no se haga sin pensar, no asumir que este tipo de recursos son una buena práctica. Hay sufijos que pueden ayudar a entender el contexto de una variable, como, por ejemplo, las siglas *dto* para los objetos de transporte de datos, el sufijo *repository* para las clases que implementan dicho patrón de *Domain-driven design* o el sufijo *viewModel* para los objetos que modelan vistas. La información acerca de *para qué es*, resulta más interesante que la información sobre *cómo está implementado*. Como regla general, evita prefijos, sufijos y cualquier otra palabra con información sobre la naturaleza técnica de la abstracción; recurre a estas salidas cuando no te quede más remedio o como solución temporal. Recuerda que también puedes apoyarte en los espacios de nombres para contextualizar los elementos que tienen dentro.

Nombres concretos

La cuarta restricción habla de que los nombres sean concretos ¿Cuál es el nivel adecuado de concreción? Un truco es pensar si un nombre es aplicable a muchos elementos a la vez, en cuyo caso quizás debamos descartarlo. Ejemplos de nombres demasiado genéricos:

- *Helper, manager, generator, engine, tool, service, utils, process, execute, input, …*

Estas palabras pueden aparecer tal cual, como prefijos o sufijos (coletillas). Son nombres que lo aguantan todo y a la vez no dicen nada. Por ejemplo, el nombre *input* para el parámetro de una función es redundante, ya que los parámetros son de por sí *input* de las funciones

(¿te imaginas que se llamara *output*?). Cualquier parámetro de cualquier función podría llamarse *input*, así que es mejor buscar un nombre que sea exclusivo. La lista de arriba son solo algunos ejemplos y como tal, no contiene todas las palabras sospechosas. Además, hay casos donde algunas de estas palabras de ejemplo tienen sentido, como la palabra *execute*, cuando implementamos un patrón *command*; la palabra *service*, cuando nos apoyamos en *Domain-driven design*. No se trata de una lista de palabras prohibidas, sino de una idea; evitar los nombres que valen para todo.

Nombres que forman frases

La quinta restricción tiene como objetivo que cada línea de código sea lo más parecida a una frase con sentido en lenguaje natural, para lo cual el nombre estará influenciado por la posición que ocupa la palabra dentro de la frase:

Java

```
if (isPaidInvoice)
...
if (containsNumbers)
...
if (areThereItems)
...
if (isNotBlank(page))
...
while (maximumCapacityHasNotBeenReached)
...
```

Una sana convención extendida es la de utilizar prefijos de tipo pregunta, para expresiones o variables tipo *boolean*: *is, has, does, are, contains, will, should*… Incluso a veces se utilizan las negaciones *isNot*, *doesnt, hasnt*…, porque puede resultar más claro leer en lenguaje natural, que los signos de negación combinados con otros operadores lógicos (solo a veces). En cambio, los operadores lógicos *and* y *or* no son recomendables para los nombres, es mejor utilizar los propios operadores del lenguaje, ya que cuando son parte de un nombre denotan exceso de responsabilidad. Por ejemplo, un método que se llame *saveUserAndSendEmail*, está claramente realizando dos acciones a la vez. Si las dos acciones han de realizarse en bloque, seguramente existe un nombre de dominio más adecuado para la operación (ej: *registerUser*), pero si no lo hay, el propio nombre nos indica que el diseño incumple el principio de responsabilidad única. Cuando estoy refactorizando código legado que me cuesta entender, puedo llegar a renombrar un método recurriendo a la conjunción

and, justamente para poner de manifiesto que el código huele mal, como paso intermedio hacia su futura mejora.

Las convenciones para los nombres tipo pregunta están muy extendidas en gran variedad de lenguajes de programación, por eso confunde verlas en operaciones que no sean *boolean*, como en este ejemplo:

Java

```java
try {
   File file = isExistingFile(path);
} catch (...)
```

La función devuelve un fichero si existe en la ruta y lanza excepción en otro caso. No es una buena idea por varios motivos: el primero, por usar excepciones donde no hay un caso excepcional; y el segundo, porque la función hace dos cosas en una. Si respetamos la convención del prefijo *is* para las expresiones *booleanas*, quizá nos demos cuenta de que este diseño no es apropiado ¡Qué buenas pistas sobre el diseño nos da la búsqueda concienzuda de nombres!

Sin alias

Evita los sobrenombres. Utilizar nombres distintos para el mismo concepto resulta confuso, cuando no redundante. Ejemplo de redundancia:

Java

```java
public String[] partirPorComas(String texto){
    return texto.split(',');
}
```

Cualquiera que trabaje con cadenas habitualmente sabe que *split* significa *partir*, no aporta nada traducirlo o utilizar un sinónimo para la operación. Otro ejemplo de redundancia:

Java

```java
public void throwException(String message){
    throw new RuntimeException(message);
}
```

Los sinónimos y las traducciones son innecesarias, introducen una indirección que dificulta la lectura.

Los alias son especialmente peligrosos cuando nos inventamos nombres nuevos para referirnos a un único concepto. Por ejemplo, si en términos de negocio se habla de «pagar una factura», el código no debería hablar en términos de «procesar una factura». Hay alias que son sinónimos precisos y pese a que no es lo correcto, permiten que el código medio se entienda, pero luego hay alias que introducen conceptos nuevos que ya no son sinónimos (más que en la cabeza de quien se lo inventa). Por ejemplo, si como experto de negocio hablase de *filtrar* facturas que no tienen puesto un email de contacto, el código fuente no debería hablar en términos de *extraer* facturas. Filtrar y extraer son operaciones parecidas según el contexto, pero no son lo mismo. El siguiente nivel son los alias marcianos o alienígenas, conceptos totalmente artificiales que no tienen nada que ver con el negocio. Para el ejemplo de filtrar facturas, un alias marciano podría ser *escanear* facturas; para la persona que decide usar ese nombre, está claro su significado en el momento, pero para los demás o para ella misma, pasado el tiempo será confuso.

Debemos tener cuidado de no utilizar nombres parecidos para operativas que tienen objetivos diferentes. Por ejemplo, la función *parseInt* en JavaScript, convierte un tipo cadena en un tipo numérico, un comportamiento diferente al que me encontré en un proyecto:

Javascript

```
function parseInteger(expression: string) {
  return /^\d+$/.test(expression) ? Number(expression) : 0;
}
```

Esta función convierte a número cuando el argumento es de verdad numérico y devuelve cero en otro caso, para no alterar el total de la suma que realiza otra función de este módulo. Los números negativos los está evaluando como cero. Tiene un comportamiento distinto al de *parseInt*, aunque parece ser el mismo por su nombre. Esto provoca sorpresa o confusión. La verdad es que el comportamiento de *parseInt* es ya bastante sorprendente de por sí ¿Qué devuelve esto?

Javascript

```
parseInt("3tristesTigres");
```

¡Devuelve el número 3! Para evitar este comportamiento prefiero usar la función *Number*:

Javascript

```
Number("3tristesTigres"); // <--- devuelve NaN
```

Mejor así:

Javascript

```
function getValueOf(expression: string) {
  return /^\d+$/.test(expression) ? Number(expression) : 0;
}
```

Nombres que se apoyan en el contexto

El nombre de un método debe apoyarse en el contexto proporcionado por el nombre de su clase, así como por los nombres de sus argumentos y sus tipos. Sucede lo mismo si hablamos de funciones dentro de módulos, en lenguajes como Python, JavaScript, Ruby o Kotlin, tanto el nombre de la función como el del módulo se pueden complementar. Lo que debemos pensar es cómo se leerán las líneas que invocan a dicho método o función:

Java

```
int sum = calculator.sumNumbersIn(expression);
UserData userData = mapper.toDataTransferObject(user);
action.execute(command);
Order order = shoppingCart.checkout()
```

El primer ejemplo nos muestra que es muy interesante apoyarse en los parámetros para darle más expresividad a las llamadas al método. Los tipos también suman información al contexto, aunque no se refleje explícitamente en el nombre, por eso cuando definimos tipos específicos en lugar de los *built-in* que trae el lenguaje, podemos ahorrarnos cierta redundancia en los nombres. Al usar los tipos del lenguaje para los parámetros, tenemos menos probabilidades de que las variables se nombren como nos gustaría por parte de quien invoca a las funciones:

Java

```
public int sumNumbersIn(String expression){/*...*/}
/*...*/
int sum = calculator.sumNumbersIn(str);
```

Pongamos que la función fue diseñada por una persona, mientras que la llamada a la función la escribió otra persona distinta. El nombre *str* estropea la frase, por eso, sería conveniente incluir toda la información en el nombre de la función, para asegurarnos que la llamada queda clara (aunque pueda ser redundante):

Java

```java
public int sumNumbersInExpression(String expression){/*...*/}
/*...*/
int sum = calculator.sumNumbersInExpression(str);
/*...*/
int sum = calculator.sumNumbersInExpression(expression);
/*...*/
var contact = mapper.fromUserToContact(user);
```

Aunque trabajemos con módulos y funciones, como podría ocurrir con JavaScript, podemos importar los módulos de manera que su nombre se vea en la llamada a la función (en el ejemplo, *calculator*, podría ser el módulo).

Al usar tipos específicos del dominio, es más probable que las variables se nombren igual que sus clases:

Java

```java
public class Expression {/*...*/}
/*...*/
public class Calculator {
    public int sumNumbersIn(Expression expression){/*...*/}
}
/*...*/
Expression expression = Expression.parse("1,2,3");
int sum = calculator.sumNumbersIn(expression);
```

Los tipos específicos atraen nombres de variables acordes a ellos, además de comportamiento.

Los métodos o funciones que tienen más de un parámetro, ya no se prestan tanto a construir frases similares al lenguaje natural. Si hay varios parámetros, lo mejor será que el nombre del método no dependa de ellos, para que la llamada sea muy clara. Cuantos menos parámetros tenga un método, más probabilidad hay de que su implementación quede simple y de que tenga una única responsabilidad. He aquí otro indicio de que el énfasis por encontrar nombres significativos enriquece el diseño del *software*, además de la legibilidad.

Que un método sea de tipo *void* o que devuelva alguna respuesta, también influye en la forma en la que podemos nombrarle. Al ver la llamada, enseguida entendemos si devuelve algo, porque aparece en una asignación o como subepxresión en la misma línea:

Java

```
Amount amount = invoice.amount();
/*...*/
Amount total = amount.sum(otherInvoice.amount());
```

Sin haber visto la firma de los métodos *amount*, ni *sum*, sabes que devuelven valores. De aquí que no haga falta ningún prefijo como *get* (*getAmount*). Este prefijo tan extendido era necesario en el pasado para ciertas herramientas que utilizaban metaprogramación y/o que se apoyaban en la convención del prefijo (más información en el capítulo sobre principios mailinterpretados). Hoy en día, la mayoría de las veces lo usamos como una muletilla que en verdad no aporta nada. Mi consejo es tratar de evitarla, para ver si así se nos ocurren mejores nombres.

En cuanto a los métodos que no devuelven nada, es recomendable poner nombres con un tono imperativo, evidenciando que acarrean efectos secundarios; si un método no devuelve nada, es porque va a alterar el sistema, ya sea modificando los argumentos, el estado interno de la clase, la base de datos, la cola de mensajes…

Java

```
messenger.send(email);
eventBus.notifySubscribers(event);
shoppingCart.add(item);
userRepository.delete(user);
```

Distinguir sustantivos, verbos y adjetivos

Típicamente, usamos sustantivos para nombres de clase/módulo/paquete y verbos para nombres de método/función. Es una convención aceptada independientemente del lenguaje, que con poco esfuerzo nos aporta mucha coherencia. Las clases o módulos expresan conceptos, por eso se recurre a los sustantivos. Los métodos o funciones expresan acciones, por eso se recurre a los verbos:

Java

```
messenger.sendEmail(email);
   ^           ^       ^
sustantivo   verbo    sustantivo
```

Como de costumbre, hay excepciones a la regla. En un ejemplo anterior, decidí rechazar el prefijo *get*, lo cual convierte al nombre del método en un sustantivo, y, sin embargo, no se reduce la legibilidad:

Java

```
Amount amount = invoice.amount();
```

Al final, lo que valida la calidad de los nombres es la facilidad con la que entendemos las frases que los contienen. Por eso, practicar TDD ayuda a nombrar, porque hacemos uso de cada método antes de que exista; nos permite darnos cuenta de manera anticipada de la pinta que tendrá nuestro código cuando terminemos la faena.

Otra excepción a la regla son las clases con un solo método, que se utilizan para representar acciones y que, por tanto, llevan la acción verbal en el nombre de la clase:

Java

```
public class PayOrder {
    /*...*/
    public void execute(Order order){/*...*/}
}
/*...*/
public class SendEmail {
    /*...*/
    public void execute(Email email){/*...*/}
}
```

Depende de la arquitectura que utilicemos puede resultar más o menos natural nombrar las clases como acciones. Si no tienes un motivo específico para saltarte la convención o no te has planteado usar una convención, ahora es un buen momento para empezar a utilizar sustantivos para las clases y verbos para los métodos; mejorará la experiencia de lectura del código que escribas. Mantener una cierta homogeneidad en la forma de implementar, suele producir mejores resultados que programar de manera arbitraria cada bloque de código. Recuerda que cuando leemos código antiguo, intuitivamente buscamos patrones de implementación para comprender la estructura del proyecto y la forma de pensar de quienes lo programaron.

Los verbos responden a la pregunta de *«¿qué hace?»*, mientras que los sustantivos responden a la pregunta de *¿qué es?*, y a veces también a la pregunta de *¿para qué es?* Ejemplos de nombres y de lo que sugieren, incluso sin especificar tipos:

Javascript

```
let invoicesCount;  // <- tipo numérico, usado como contador
let widthInCentimeters;  // <- numérico, para guardar ancho en centímetros
let populationInMillions;   // <- numérico, las unidades son millones
let percentageWithTwoDecimals; // <- precisión de dos decimales
let netAmount;      // <- numérico, (monto neto) con decimales
let invoices;       // <- coleccion de facturas
let contactInfo;    // <- objeto con varias propiedades
```

¿Para qué utilizamos los adjetivos? Para describir comportamiento. Los adjetivos describen muy bien a las interfaces, así como a los metadatos que utilizamos en metaprogramación. En las interfaces que trae integradas Java o C#, podemos ver buenos ejemplos de adjetivos, sobre todo en las que tienen un solo método. Una interfaz que puede correr o ejecutar, es ejecutable:

Java

```
public interface Runnable {
    void run();
}
```

El método es un verbo y la interfaz que lo contiene es un adjetivo. Más ejemplos típicos de nombres de interfaces son, *Serializable*, *Comparable*, *Disposable*, *Formattable*, *Throwable*, palabras que funcionan como adjetivos para describir interfaces (de uno o más métodos). Si tenemos una interfaz con un solo método *update*, puede quedar muy bien que su nombre sea *Updatable*. Si una interfaz tuviera un método *cancel*, quizá podría llamarse *Cancellable*. Cuantos más métodos tenga una interfaz, más dificil será que podamos encontrar adjetivos para ella, con lo que terminaremos recurriendo a sustantivos. No hay ningún problema en utilizar sustantivos con capacidad descriptiva como *Observer*, sin embargo, es interesante hacer el ejercicio de buscar adjetivos para reflexionar sobre el propósito de cada interfaz. Puede ayudarnos a cuidar el principio de segregación de interfaces, es decir, a que cada interfaz tenga un propósito único y exclusivo. En cuanto a la metaprogramación, la definición de metadatos también suele hacerse mediante adjetivos, por ejemplo para las anotaciones o atributos: *@Autowired, @Deprecated, [Conditional], [WebMethod], [JSONInclude], [Json Property]*... La metaprogramación es típica de librerías y *frameworks*, se sale del alcance de los contenidos de este libro. Es código que se alimenta de otro código

para enriquecer o alterar su comportamiento.

Darle nombre a los valores literales

Anteriormente, escribí que lo mejor para no tener problemas con los nombres es no ponerlos, sin embargo, esto no aplica a las constantes literales. Los números mágicos y otros literales mágicos son una incógnita para quien está leyendo el código. Cualquier número o cadena de caracteres literal debe tener un nombre que explique lo que significa ese valor:

Java

```
if (invSt == 3) // ???
...
if (rtCd == "C1005") // ???
```

Mejor así:

Java

```
if (invoiceStatus == Status.Paid)
...
if (returnCode == Error.invalidPostErrorCode)
```

Los tipos enumerados son ideales para agrupar constantes, no obstante, si resulta artificial o prematuro crear una abstracción para alojar un único valor literal, podemos recurrir a constantes simples:

Java

```
final int paid = 3;
```

Las primeras versiones del compilador de C no proporcionaban ninguna forma de indicar que un elemento fuera constante y por eso se extendió la convención de ponerlas en mayúscula. En lenguajes donde podemos utilizar *const* o *final*, no hay necesidad de ponerlo todo en mayúsculas, pero si el equipo prefiere mantener esa vieja costumbre, tampoco es un problema, solamente una cuestión de gustos. Este es otro ejemplo donde el nombre contiene información del tipo, utilizando las mayúsculas para decir que es constante, y, por tanto, es redundante si el lenguaje y el IDE ya lo dicen. Las recomendaciones de Oracle para Java siguen siendo las de utilizar mayúsculas para constantes, y yo no me opongo a

ello, simplemente digo que no me parece necesario porque el compilador y el IDE avisarán de que algo es constante.

Renombrar al día siguiente

Las restricciones que acabamos de ver son heurísticas, es decir, estrategias para descubrir mejores nombres, no reglas exactas. Me temo que no existe la fórmula mágica, ya que el nombre perfecto tampoco existe. Por eso, recurro al *refactoring* a diario, porque los pequeños ajustes en los nombres me resultan más productivos que bloquearme buscando el nombre ideal a la primera. Pongo conciencia al nombrar, pero sin llegar a la parálisis por análisis. Uno de mis lemas favoritos es; atención y pragmatismo en vez de dogmatismo.

Las dificultades para encontrar buenos nombres nos dan pistas valiosas sobre nuestro diseño. Es muy rentable detenerse a indagar sobre la dificultad y hacerse preguntas, *¿por qué nos cuesta tanto encontrar consenso sobre el nombre de un método?* Las respuestas son muy interesantes; igual resulta que el problema de fondo es otro, que el nombre de la clase/variable es inadecuado y por eso nos cuesta poner un buen nombre al método…

Vamos a continuar estudiando ejemplos de buenas y malas ideas para poner nombres a lo largo de todo el libro, sin ir más lejos, en el próximo capítulo. Algunos ejemplos podrían parecer contradictorios, la clave para entenderlos será siempre el contexto ¿Te has dado cuenta de lo importantes que son los nombres en el código fuente?

5. Principio de menor sorpresa

No está claro el origen del también llamado *principio de menor asombro*, pero Eric Raymond ya lo utilizaba al comienzo de la era Unix y ha sido muy difundido por Ward Cunningham, co-autor de *Extreme Programming* y del/la wiki. La idea es sencilla, procura que el código se comporte como cabe esperar. Las sorpresas en el *software* no le gustan a nadie, porque casi nunca son buenas.

Java

```
User user = findUserBy(Id userId);
```

¿Podría esperarse que esta línea de código envíe un correo electrónico?, ¿o siquiera que haga alguna escritura en la base de datos? Yo no lo esperaría, me llevaría una sorpresa si lo hiciera. Puede que lo de enviar un email aquí sea exagerado, pero alguna vez sí que he visto código que decidía actualizar datos en una tabla cuando se hacía una búsqueda. Alguien debió pensar que era más óptimo aprovechar a cambiar datos en ese momento, quizás intentó terminar rápido dos tareas en una o simplemente tomó una decisión arbitraria, no lo sé. La cuestión es que me llevé un chasco.

La brújula del código sostenible

De entre todos los principios de diseño, el de menor sorpresa es para mí el más importante. Cuando tengo dudas sobre mi diseño, lo que me pregunto es si otra persona que llegue después se sorprenderá con lo que lea. Me hago la pregunta cuando estoy escribiendo el código, luego cuando ya lo he escrito y también en los días posteriores. Nada mejor que releer tu código del día anterior o de la semana anterior, para verlo desde la distancia y darte cuenta de si es un código que causa sorpresa o es intuitivo. El código debería comportarse tal y como cualquiera esperaría cuando lo lee, sin tener que entrar a mirar el detalle de todas las funciones o métodos privados, ni todas las definiciones de variables. Cada abstracción o cada artefacto debe comportarse como una caja negra, con una entrada y una salida claras, con un comportamiento obvio.

Si no aplicamos bien este principio, sembraremos la desconfianza, perderemos los be-

neficios de la encapsulación y terminaremos con niveles de acoplamiento excesivos. La confianza es un factor clave para la productividad de cualquier organización o equipo de personas; avanzamos mucho más rápido cuando confiamos en los demás. Lo mismo sucede cuando confiamos en el código, nos permite ser verdaderamente ágiles. Un código de confianza marca la diferencia, nos ayuda a encontrar soluciones más coherentes y sencillas que responden mejor a las necesidades del negocio.

La mayoría de lenguajes de programación modernos están pensados y diseñados para reducir la cantidad de sorpresas que se pueden dar con ellos, lo cual reduce la carga cognitiva al programar. Por ejemplo, el tipo de dato *string* es inmutable en muchos lenguajes, para evitar mutaciones perversas. El comportamiento de valores y referencias es también un estándar independiente del lenguaje por el mismo motivo. El lenguaje C, perdió popularidad debido a la aparición de otros lenguajes con gestión de memoria automática, con sistemas de tipos fuertes y con tipos integrados más potentes (como cadenas y colecciones), entre otras cosas porque se reducían las sorpresas. Durante muchos años, hemos sufrido las deficiencias de diseño de las primeras versiones de JavaScript o de PHP, justamente porque nos daban sorpresas (y se ha hecho un gran esfuerzo por corregirlo en cada nueva versión). Gary Bernhardt lo cuenta con mucho humor en su famosa presentación «Wat». Esto no significa que dichos lenguajes sean malos, opino que el diseño de JavaScript como lenguaje multiparadigma es genial, especialmente por las influencias de Scheme y AWK, pero el hecho de que su creador se viera obligado a desarrollarlo en apenas dos semanas tuvo sus consecuencias. Los lenguajes evolucionan, además de que van apareciendo otros nuevos que hacen mayor énfasis en la reducción de la probabilidad de errores y de sorpresas, sin embargo, todavía no consiguen frenar el ingenio humano a la hora de escribir código asombrosamente ofuscado. Por tanto, la responsabilidad sigue estando en quien utiliza el lenguaje, más que en quien lo diseña. Con cualquier lenguaje se puede escribir código sin sorpresas o código muy sorpresivo, aunque unos te hacen pensar más que otros.

El código intuitivo combina las abstracciones con los tipos de datos del lenguaje y con el resto de construcciones del mismo, de manera coherente. Además, las abstracciones se comportan como cabe esperar. Ejemplos de incoherencias podrían ser:

Java

```java
void findTheBiggestNumber(List<Integer> numbers);
```

El nombre dice «busca el número más grande» y recibe una lista de números, pero luego

no devuelve ningún número. Me va a obligar a investigar dónde será que deja el número. Me resultaría más intuitiva esta otra firma:

Java

```java
Integer findTheBiggestNumber(List<Integer> numbers);
```

Sin embargo, la sorpresa podría estar encerrada dentro del método:

Java

```java
Integer findTheBiggestNumber(List<Integer> numbers){
    Integer biggestNumber = Collections.max(numbers);
    numbers.remove(biggestNumber);
    return biggestNumber;
}
```

Efectivamente, se devuelve el número más grande, pero también se elimina del vector de entrada. Sorpresa total, le pides a una función que te busque el máximo valor de una lista de números y resulta que borra un elemento de la misma. Poner un comentario en el código no lo va a mejorar:

Java

```java
// Ojo que esta función borra elementos de la lista.
Integer findTheBiggestNumber(List<Integer> numbers){
    Integer biggestNumber = Collections.max(numbers);
    numbers.remove(biggestNumber); // elimina elemento de la lista.
    return biggestNumber;
}
```

Cuando nos encontremos con la llamada a la función en alguna parte del código, no veremos el comentario, solamente la llamada. Queremos evitar que la lectora o lector tenga que escanear el 100 % del código para comprenderlo y para no llevarse chascos. Es mejor rediseñar el código reemplazando sorpresas por obviedades. Hagamos un refactor para que el código haga lo que dice y solamente lo que dice (quitamos la función):

Java

```java
Integer biggestNumber = Collections.max(numbers);
numbers.remove(biggestNumber);
```

Te aseguro que no pondría este tipo de ejemplos en un libro, si no me los hubiera tropezado

en proyectos reales (además de haber escrito cosas así yo mismo, por supuesto).

Los lenguajes que admiten diferentes tipos de dato para el retorno de una función, pueden prestarse a inconsistencias sorpresivas:

Javascript

```
function findTheBiggestNumber(numbers){
    if (numbers == null){
        return "Numbers can't be null";
    }
    if (numbers.length == 0){
        return "0";
    }
    return Math.max(...numbers);
}
```

En este ejemplo, la función devuelve en unos casos una cadena de caracteres; en otros, el cero como cadena; y en los casos restantes, un tipo numérico. Definitivamente, es una forma de poner en un apuro a quien vaya a utilizar esta función, y por supuesto, una sorpresa para quien se encuentra llamadas a esta función por el código. El efecto que estas inconsistencias provocan en el proyecto es la proliferación de código defensivo por todas partes:

Javascript

```
if (typeof(number) != "undefined"){
  if (typeof(number) == "number" && !isNaN(number)){
    ...
  } else if (typeof(number) == "string" && number !== null){
    ...
  } else {
    ...
  }
}
```

Al igual que no queremos tener que defender cada propuesta que hacemos a nuestro equipo, como si estuviéramos ante un tribunal de justicia, tampoco queremos vernos obligados a programar a la defensiva por todo el código. La programación defensiva tiene sentido en los límites de nuestro sistema, es decir, en aquellas partes que interconectan nuestro código con el de terceros, porque no tenemos control sobre el exterior. Para código que está a nuestro cargo, lo deseable es tener el control y confiar en la consistencia de los artefactos que construimos.

Los lenguajes dinámicos admiten el reemplazo de partes del sistema base durante la eje-

cución, por ejemplo, podemos sustituir un método del objeto *String* para que tenga otro comportamiento:

Javascript

```
String.prototype.toUpperCase = function(){
   return this.toLowerCase();
}
let text = "AbcDeF";
console.log(text.toUpperCase());
// prints "abcdef"
```

¡Qué confuso!, ¡qué sorpresa! Esta técnica se conoce como *monkey patching*, y en la mayoría de los casos está desaconsejada, justamente por su factor sorpresa. Cuando se recurre a esta técnica es porque nos encontramos ante una situación complicada; seguramente hemos perdido el control. La usamos para testar un código que no tiene un diseño testable, ni test, o para modificar el comportamiento de un *software* de terceros cuyo código fuente no tenemos. Está bien saber que existe la técnica para el día en que no quede más remedio que usarla, pero mientras tanto, evítala.

Por otra parte, tanto lenguajes dinámicos como estáticos, suelen ofrecer la posibilidad de extender el sistema base con nuestros propios extras. Suena tentador añadir nuestros propios métodos al objeto *String*, pero ¿quién se va a esperar que estén alojados ahí?, ¿cómo de compatible será con otros módulos de nuestro propio *software*?, y ¿qué sucede si en el futuro el propio sistema añade una función con el mismo nombre? Nos arriesgamos a perder la confianza en el propio sistema base, o sea, todavía peor que desconfiar de nuestro código. De nuevo, esta técnica es un recurso que nos sirve cuando necesitamos alterar el comportamiento de un código que no podemos cambiar. Es un recurso útil en la construcción de *frameworks* y librerías, sin embargo, no la recomiendo dentro del código que contiene la lógica de dominio. Los métodos y funciones de extensión en lenguajes modernos como C# o Kotlin, son interesantes para construir librerías de propósito general, no tanto para aplicaciones. Nuestro código es más expresivo si utilizamos nuestros propios tipos específicos con la semántica más precisa y explícita posible.

En general, ante la duda sobre si elijo una determinada implementación u otra, me decanto por la que pienso que causará menos sorpresa, aunque la alternativa me pueda parecer más elegante, ingeniosa o novedosa.

Las sorpresas suelen tener relación con efectos secundarios inesperados como cambios de estado ¿Ves algún problema en el siguiente bloque de código?

Java

```
for (File file: files){
    if (hasValidFormat(file)){
        validFormatFiles.add(file);
    }
}
```

Parece un código que explora una serie de ficheros y guarda en una variable aquellos con un formato válido. Debería ser inocuo, pero cuando entramos a mirar los detalles descubrimos lo siguiente:

Java

```
boolean hasValidFormat(File file){
    List<String> lines = read(file);
    for(String line: lines){
       if (!hasValidFormat(line)){
           delete(file);
           return false;
       }
    }
    return true;
}
```

¡Agüita! Una función que debería limitarse a decirnos, «verdadero o falso», ¡está borrando ficheros del disco!, ¡sooorpresaaaaa!

En un intento por evitar que nuestro código mienta o sorprenda, podríamos pensar que la mejor opción es poner nombres muy abstractos, muy genéricos:

Java

```
for (File file: files){
    if (process(file)){
        resultFiles.add(file);
    }
}
```

Recordemos que nombres como *process* lo aguantan todo, o sea, que no se mojan nada. El otro extremo es eliminar la abstracción mediante nombres que dicen exactamente lo mismo que el código:

Java

```
boolean deleteAllFilesWithInvalidFormatAndReturnTrueInCaseAllOfThemAreValid(
```

```
    File file);
```

El problema no es tanto la longitud del nombre sino la cantidad de información que revela. Otro ejemplo:

Java

```java
void throwExceptionIfWordIsBlank(String word){
    if (word.equals("")) {
        throw new ArgumentException("Word can't be empty");
    }
}
```

El nombre del método dice en «inglés» lo que el cuerpo del método dice en «Java». No hay nivel de abstracción adicional en el nombre, con lo cual no aporta nada. El problema es que si mañana cambia cualquier detalle del método, su nombre ya no le valdrá. Son nombres que al no abstraer, no sintetizan la información del bloque de código y que, por tanto, aumentan la carga cognitiva, a la par que eliminan la preciada encapsulación. Nombres que sí añaden abstracción podrían ser:

Java

```java
void checkWord(Sring word);
void validateWord(String word);
void assertValidWord(String word);
```

El código no debe guardar sorpresas, sino guardar secretos, que son aquellos detalles de implementación que no incumben a nadie más.

Hasta ahora, hemos visto ejemplos de sorpresas encerradas en funciones, sin embargo, también se ocultan dentro de artefactos de mayor calado como clases, paquetes, módulos, librerías e incluso aplicaciones enteras. Un sistema compuesto por múltiples subsistemas puede albergar comportamientos asombrosos, fruto de la interconexión de sus piezas. Las condiciones de carrera o los bloqueos mutuos son sorpresas clásicas en artefactos concurrentes de un sistema mal diseñado. Por eso, las arquitecturas y las infraestructuras *software* también deben ser cuestionadas en cuanto a su capacidad para sorprender. Los mecanismos de caché de datos, a menudo son motivo de sorpresa, sobre todo si los encontramos en sistemas en los que nadie sospecha que sea necesario almacenar en caché. La sorpresa puede estar a la hora de guardar, de borrar o de actualizar. Cuando exista la necesidad de apoyarse en memoria caché, conviene que el código sea lo más explícito posible expresando este comportamiento. Hay muchas formas de hacerlo, una de ellas

podría ser la llamada explícita:

Java

```
List<User> users = cache.retrieveAll<User>();
```

A lo largo del libro, continuaremos explorando ejemplos de código sorpresivo, así como principios que se basan en este para darnos más pistas sobre cómo diseñar de manera intuitiva.

Recuerda, el principio de menor sorpresa es la brújula que te guiará en el diseño de *software* fácil de entender y de modificar.

6. Cohesión y acoplamiento

La estrella polar del diseño sostenible

Dos conceptos muy sonados en el área de diseño del *software* son la cohesión y el acoplamiento. El lema es diseñar *software* con alta cohesión y bajo acoplamiento, pero ¿qué significa esto? A día de hoy, no disponemos de axiomas ni teoremas que nos permitan expresar de una manera precisa estos conceptos. Ni si quiera programadores que llevan décadas trabajando con ellos parecen ponerse de acuerdo en una definición.

Los conceptos de cohesión y acoplamiento van a guiarnos para navegar por los entresijos del diseño del *software*, como la estrella polar guiaba a los barcos en medio del mar.

Un artefacto con alta cohesión es aquel capaz de operar con un alto grado de autonomía o de autosuficiencia, mientras que un artefacto que presenta acoplamiento es aquel que depende de otros para operar. Ambos conceptos tratan la interdependencia entre elementos. Entendemos por artefacto o por componente cualquier elemento de *software* compuesto por otros elementos: un método/función, una clase, un paquete/módulo, un microservicio…

El acoplamiento es indispensable e inevitable, lo que procuramos es reducirlo al mínimo imprescindible. Una función por sí misma no resuelve ningún problema entero, salvo que se combine con otras funciones. Por tanto, dentro de un artefacto con alta cohesión, sus elementos manifestarán necesariamente un cierto nivel de acoplamiento entre ellos. Vigilaremos el nivel de acoplamiento cuanto más subamos en el nivel de abstracción, es decir, las consecuencias serán más graves conforme los artefactos interdependientes sean más complejos. Que un método de una clase esté acoplado a otro de la misma clase, es potencialmente menos problemático que si una clase está acoplada a otra clase.

El vagón de un tren es un módulo cohesivo, porque no necesita a otros vagones; al mismo tiempo que el tren es cohesivo, porque no depende de otro tren. Sin embargo, para componer un tren hay que engancharle vagones (acoplar vagones entre sí). Si bien el tren es cohesivo en el sentido de autonomía observado como un todo, por dentro tiene piezas acopladas. Si diseñamos módulos de *software* independientes e interconectables, conseguiremos un sistema más intuitivo y ahorraremos en mantenimiento considerablemente.

Acoplamiento

Los protocolos de comunicaciones que permiten la interconexión de sistemas heterogéneos están diseñados para que estos puedan comunicarse con el mínimo nivel de acoplamiento. Gracias a esto, un servidor web puede servir a un navegador web, a una aplicación iOS y a una aplicación Android a la vez ¿Te imaginas que cada vez que saliera una actualización de Firefox hubiese que actualizar el iPhone? Gracias a un buen trabajo de ingeniería, no es así. En el *software* debemos diseñar cada artefacto, de manera que cuando requiera cambios, no tengamos que cambiar también otros artefactos. Varios autores llaman ortogonalidad[1] a esta capacidad de poder realizar cambios con independencia.

Tengo la sospecha de que la proliferación de los sistemas basados en microservicios es consecuencia (en parte) de que las restricciones que impone el protocolo de comunicación reducen las posibilidades de acoplamiento entre los mismos. Una arquitectura de microservicios, en teoría, limita la variedad de formas en que se pueden acoplar los artefactos, por ejemplo, manteniendo la independencia para compilar y desplegar. La contrapartida es el aumento de complejidad en la infraestructura, y como no, la posible reducción de cohesión. Si nos excedemos reduciendo el acoplamiento dentro de un artefacto, corremos el riesgo de perder cohesión y de trasladar ese acoplamiento afuera, convirtiendo el sistema en una maraña. Cuando un microservicio no sirve absolutamente para nada sin los otros, probablemente no deba ser un microservicio. Un exceso de acoplamiento entre artefactos a nivel de conocimiento tácito es más difícil de detectar que una referencia directa.

Un sistema monolítico, diseñado a conciencia, puede estar compuesto por artefactos cohesivos sanamente desacoplados entre sí. Para ello, disponemos de multitud de herramientas que vamos a ir explorando, tales como la encapsulación, los tipos específicos del dominio, el paso de mensajes… Por tanto, el acoplamiento dentro de un monolito o entre microservicios, es causado por las decisiones que se toman (o que se ignoran) y no tanto por la tecnología en sí misma.

El acople no es malo ni bueno *per se*, lo que dificulta el mantenimiento del *software* es el diseño inconsciente. El problema es la falta de conocimiento. Desde el momento en que nos damos cuenta de la cohesión y del acoplamiento en los artefactos que programamos, podemos tomar mejores decisiones sobre su diseño para evitar consecuencias indeseadas.

[1] Dave Thomas y Andrew Hunt lo detallan en *The Pragmatic Programmer*, mientras que Eric Raymond lo explica en *The Art of Unix Programming*.

Hay tanta variedad de formas de acoplamiento, que cada día descubrimos más. Cada paradigma, cada lenguaje y cada sistema, es propenso a sufrir problemas de gestión de interdependencias, así sean más o menos explícitas. Cuanto más evidente es la interdependencia, más fácil es advertir los posibles efectos secundarios que la realización de cambios puede acarrear en el sistema. Por el contrario, cuanto más subyacente sea la interdependencia, mayor será el riesgo de romper el sistema en lugares insospechados. Además de la visibilidad de la interdependencia, podemos hablar de diferentes grados.

Java

```java
class A {
    private B dependency = new B();
    /*...*/
}
class B {
    /*...*/
}
```

En el ejemplo es evidente que la clase *A*, depende de la clase concreta *B*, porque se encarga de instanciarla. La dependencia podría ser aún de mayor grado si, a su vez, *B* dependiese de *A*:

Java

```java
class A {
    public B dependency = new B();
    /*...*/
}
class B {
    public A dependency;
    /*...*/
}

public class Tests {
    @Test
    public void circular_dependencies_are_not_json_serializable() {
        A a = new A();
        a.b.dependency = a;
        ObjectMapper objectMapper = new ObjectMapper();
        assertThatThrownBy(() -> objectMapper.writeValue(new File("test"), a)
            )
                .isInstanceOf(JsonMappingException.class)
                .hasMessageContaining("StackOverflow");
    }
```

Es bien sabido que las dependencias cíclicas son problemáticas por muchos motivos; porque cuesta comprender el grafo, porque no se puede serializar cualquier formato, etc. El

test del ejemplo demuestra que si intentamos convertir el objeto *A* en una estructura JSON, se produce una excepción por desbordamiento de pila, ya que su estructura es recursiva. Los circuitos cerrados de dependencias complican el mantenimiento.

El grado del acoplamiento puede ampliarse o reducirse aplicando cambios en el diseño. En el primer ejemplo, cuando *A* dependía de *B*, podríamos optar por lo siguiente:

Java

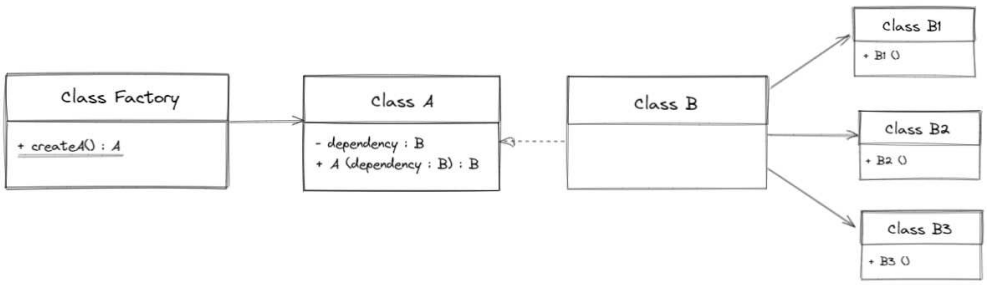
```java
class A {
    private B dependency;
    public A(B dependency){
        this.dependency = dependency;
    }
}
```

Si bien es cierto que la clase *A* depende de una interfaz *B* (que podría estar definida como *interface*, *abstract class* o *class*), desconoce su implementación exacta, ya que la recibe como dependencia inyectada:

Java

```java
class B1 extends B {
   /*...*/
}
class Factory {
    public static A createA(){
        return new A(new B1());
    }
}
```

Así hemos reducido el grado de acoplamiento entre *A* y las implementaciones de *B*. Esta técnica se conoce como inyección de dependencias, una implementación del principio de inversión de dependencias. Si dibujamos la relación con flechas para indicar el sentido de la dependencia, veríamos algo como esto:

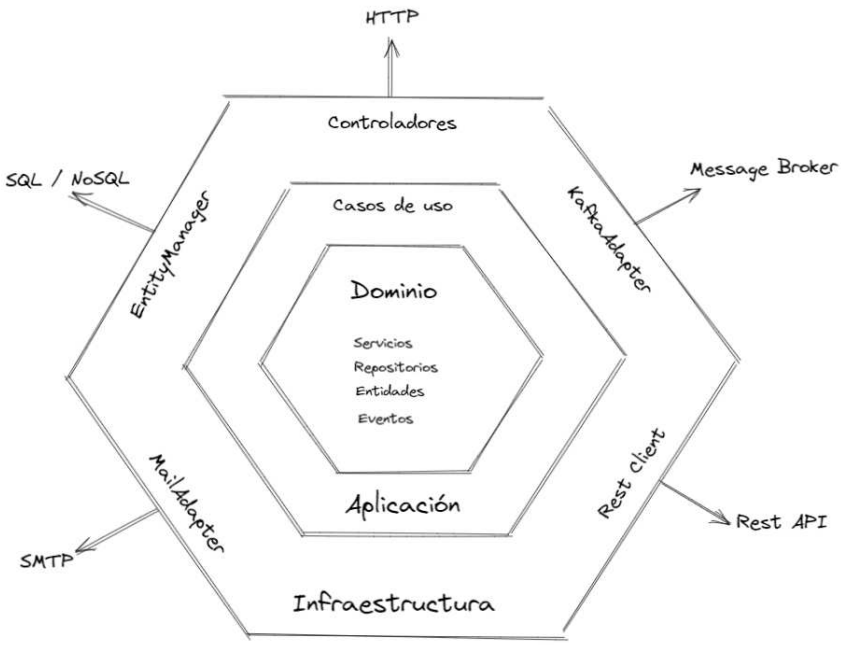


Tanto *B* como cualquier subtipo que cumpla su contrato, desconocen por completo la existencia de *A*. La dependencia es de *A* hacia *B*, aunque los detalles concretos sobre las implementaciones solo los conoce la factoría.

Las herramientas que generan automáticamente un diagrama de clases a partir de un repositorio de código, son tremendamente prácticas para visualizar el acoplamiento de un sistema. Si el diagrama está plagado de flechas que apuntan a todos lados, aparentemente sin ton ni son, vamos a tener problemas de mantenimiento serios por exceso de acoplamiento.

Arquitectura Hexagonal

En la propuesta de arquitectura limpia de Robert C. Martin, que en cierto modo está basada en la arquitectura hexagonal de Alistair Cockburn, se da mucha importancia al orden de las dependencias, como puede observarse en la figura.



Si miramos el sistema como una cebolla con diferentes capas y ponemos la lógica de dominio (reglas de negocio, código vertical específico) en el corazón de la misma, entonces el orden de las dependencias será de fuera hacia adentro. Por ejemplo, el controlador web conoce el protocolo HTTP para gestionar peticiones y respuestas, a la vez que conoce la siguiente capa hacia adentro para pedirle que resuelva el problema de negocio. Sin embargo, la capa de negocio no conoce nada del protocolo de comunicaciones ni tampoco del sistema de persistencia de datos, sino que está desacoplada del entorno. Esta arquitectura facilita, por ejemplo, la modernización de las librerías y *frameworks* utilizados en las capas exteriores del sistema, ya que no afectan al núcleo duro que entiende el negocio. Lo normal es que las versiones de librerías y *frameworks* cambien con mucha más frecuencia que las reglas de negocio (si es al revés, probablemente sea porque no entendemos el negocio o no estamos tomando adecuadamente los requisitos).

Ley de Demeter

Uno de los principios más relevantes para mí a la hora de reducir el acople entre partes del sistema, es el de mantener la privacidad. Programo cada artefacto como si desconociera por completo el comportamiento interno de los demás, limitándome a consumir su API en caso de haber una relación directa. Digamos que, de la API hacia adentro, un artefacto no quiere saber nada del otro. Recordemos que por API puede entenderse el conjunto de métodos públicos de una clase y no solamente a REST (*representational state transfer*). Cuanto menor sea el número de dependencias y su grado, mejor preparado estará el sistema para cambiar en el futuro. Me refiero tanto a las referencias directas y explícitas, como a dependencias subyacentes, las que tienen que ver con el conocimiento. Pongamos el ejemplo de un método o incluso de una API REST que devuelve un listado de elementos. El artefacto cliente o consumidor de ese listado descubre que los elementos vienen ordenados alfabéticamente por su nombre, bien porque se fija en los resultados repetidas veces, o bien porque directamente conoce el código interno del proveedor. No hay nada en la API que garantice este comportamiento, es decir, la ordenación no es parte del contrato, porque no aparece reflejada en el nombre del método o del *endpoint*, ni tampoco en ninguno de sus parámetros. Es un detalle de implementación interno que por algún motivo quedó así y que solo debería incumbir al implementador. En el momento en que los consumidores asumen dicho comportamiento y toman decisiones en base al mismo, quedan acoplados de manera implícita. Si mañana cambia la implementación del proveedor y los elementos dejan de estar ordenados, es muy posible que dejen de funcionar clientes de manera insospechada. Ni el compilador, ni las herramientas de análisis de estático de código, pueden alertar de este tipo de acoplamiento subyacente, lo cual pone en riesgo la estabilidad del sistema. Con test automáticos pueden detectarse algunos de estos problemas, pero no todos, incluso aunque tuviéramos un 100 % de cobertura de test. Como regla general, trato a los artefactos como cajas negras, cuidando que su API no exponga ningún secreto, ninguna intimidad (que luego puede ser utilizada en su contra).

La ley de Demeter, nos ayuda a diseñar de manera modular, respetando la encapsulación. En los lenguajes orientados a objetos, típicamente se explica diciendo que evitemos poner más de «un punto por línea» de código. Las siguientes líneas violarían este principio:

Java

```
var diameter = vehicle.wheel().dimension().diameter(); // <- 3 puntos
var street = user.address().street(); // <- 2 puntos
```

Se habla del punto, porque es el símbolo que representa el acceso a una propiedad de un objeto, con lo cual si hablásemos de PHP, sería la flecha (->). En realidad, habrá situaciones en las que convenga tener más de «un punto por línea» por una cuestión de simplicidad, por ejemplo, cuando trabajamos con *Data Transfer Objects* o cuando utilizamos el patrón *Builder*, y no por ello estamos violando esta ley:

Java

```
var device = builder
                .withId(1234)
                .withName("Kinesis Keyboard")
                .withDescription("Ergonomic keyboard")
                .withModel("Advantage")
                .buildDevice();
```

Los principios pueden colisionar unos con otros, sobre todo si los malinterpretamos. Lo importante no son los puntos por línea, sino la idea de guardar la privacidad para reducir el acoplamiento. Como dice el *paper original* de 1988, esta ley trata sobre encapsulación y modularidad; lo que pretende es simplificar el proceso de modificación de un programa, así como su complejidad. La propuesta es que cada método conozca solamente la estructura inmediata de la clase que lo contiene. En los ejemplos del *paper*, se muestra incluso cómo abogaban por evitar el acceso directo a campos heredados, utilizando *getters* en su lugar.

Enunciado de la ley: *La implementación de un método M, que pertenece a una clase C, solo podrá invocar a métodos de objetos que ya sean atributos de C, a otros métodos de C, o bien a los métodos de los objetos que M pueda recibir como argumentos*.

Los autores recomendaban, además, minimizar la duplicación, el número de argumentos pasados a los métodos y el número de métodos por clase.

Dile, no preguntes

¿Cuál es la alternativa para obtener el diámetro de la rueda, del vehículo del primer ejemplo, sin quebrar la ley de Demeter? Para responder a esta pregunta, hay que responder antes otra; ¿para qué necesitas el diámetro? En vez de preguntar por datos, ¿podrías pedirle al objeto que resuelva directamente el problema por sí mismo? Esto sugiere el principio de *Tell, don't ask*, pero si esta no fuera la mejor solución, entonces podrías añadir métodos de fachada:

Java

```
var diameter = vehicle.getWheelDiameter();
```

Es una manera de evitar que los consumidores de *vehicle* conozcan su estructura interna.

Ahora volviendo a *Tell, don't ask*, el principio propone que los datos y la lógica que opera sobre ellos estén unidos en el mismo objeto, para darle cohesión y reducir el acoplamiento con los otros objetos (que ya no necesitan preguntarle por datos, sino pedir soluciones). Ejemplo de diseño en el que los datos y la lógica están separados:

Java

```
class Service {
    public Amount calculateNetAmount(Invoice invoice, DirectTax tax){
        return new Amount(
                   invoice.gross().subtract(
                       invoice.gross().multiply(
                           new BigDecimal(tax.percentage()/100))));
    }
}
class Invoice {
    /*...*/
    private BigDecimal gross;
    public Invoice(BigDecimal gross /*,...*/){
        this.gross = gross;
        /*...*/
    }
    public BigDecimal gross(){
        return this.gross;
    }
    /*...*/
}
class DirectTax {
    /*...*/
    public double percentage(){
        return this.percentage;
    }
    /*...*/
}
class Amount {
    private BigDecimal amount;
    public Amount(BigDecimal amount){
        this.amount = amount;
    }
    public BigDecimal getAmount(){
        return this.amount;
    }
}
```

Tanto *Invoice* como *Tax*, son modelos anémicos, en el sentido de que tienen datos, pero les falta el músculo de las funciones. Una clase que tiene una serie de campos privados, *setters* y *getters* para todos ellos, y nada más, es lo que se denomina modelo anémico. *Service* se dedica a saquearles, porque no posee ningún dato propio (les envidia). Además, *Amount* expone el valor interno con un *getter*, así que no funciona realmente como envoltura y por eso *Invoice* recibe *BigDecimal* en lugar de *Amount*. Ejemplo alternativo que une lógica y datos:

Java

```java
class Invoice {
    private DirectTax tax;
    private List<InvoiceLine> lines;
    /*...*/
    public Invoice(List<InvoiceLine> lines, DirectTax tax){
        this.lines = lines;
        this.tax = tax;
    }
    public Amount calculateNetAmount(){
        return gross().minus(tax.applyTo(gross()));
    }
    private Amount gross(){
        /*... calculo del total bruto sumando líneas ...*/
    }
    /*...*/
}
class DirectTax {
    private double percentage;
    /*...*/
    public DirectTax(double percentage /*,...*/){
        this.percentage = percentage;
        /*...*/
    }
    public Amount applyTo(Amount amount){
        return amount.calculatePercentage(percentage);
    }
    /*...*/
}
class Amount {
    private BigDecimal amount = new BigDecimal(0);
    /*...*/
    public Amount(BigDecimal amount){
        this.amount = amount;
    }
    public Amount calculatePercentage(double percentage){
        return new Amount(amount.multiply(new BigDecimal(percentage/100)));
    }
    public Amount minus(Amount other){
        return new Amount(amount.subtract(other.amount));
    }
}
```

Como pauta para reducir el acoplamiento entre objetos, limitaremos el uso de *setters* lo máximo posible y reduciremos el uso de *getters*, tratando de dotar al objeto de su propia capacidad de resolución. No se trata de eliminar por completo los *getters*, puesto que seguramente se introduciría complejidad excesiva en el diseño, lo importante es darse cuenta de que cada *getter* constituye un potencial acoplamiento subyacente.

Cohesión

Cualquiera que haya tenido que mantener el código de un mismo producto en producción, durante meses o años, acaba tomando conciencia de que el acoplamiento improcedente es dañino. Por otra parte, ¿qué sabemos acerca de la cohesión?, ¿cuáles son sus beneficios? Principalmente son tres:

- Disminución de la carga cognitiva requerida para comprender el sistema.
- Menos efectos secundarios indeseados desencadenados por los cambios.
- Mayor libertad para realizar cambios en el diseño, sobre todo para practicar técnicas de diseño emergente (el que se realiza a base de refactorizar).

Otro supuesto beneficio es la reutilización de código, aunque ya hemos visto que solo tiene sentido en código transversal sin ninguna relación con un negocio concreto.

Si dibujamos un mapa con los diferentes artefactos de un *software* y las dependencias entre ellos, podremos señalar las zonas cohesivas y los acoplamientos. Cuando hay cohesión las piezas están cerca unas de otras, con lo cual entender esa parte del mapa demanda menos esfuerzo mental. Siempre es más fácil comprender una función que cabe en una pantalla y que no tiene dependencias de otras funciones, que una función con cientos o miles de líneas de código realizando llamadas a decenas de otras funciones. Ahora bien, si nos quedamos solo con la idea de que las funciones breves son positivas y las funciones extensas negativas, perdemos de vista que quizás la función extensa nos ofrezca mayor cohesión y, por tanto, mayor capacidad comunicativa. Por ejemplo, si una función tiene cuatro líneas de código, de las cuales dos son llamadas a funciones cuyo cometido no está claro, tendré que navegar hacia ellas para entender el conjunto.

Java

```
public void execute(Action action){
    /*...*/
    prepare(action);
    process(action);
    /*...*/
}
```

A su vez, si esas funciones contienen llamadas confusas a otras funciones volveré a navegar. Así, puede llegar el punto en que me pierda por el camino, sin comprender el objetivo principal. Tanto la estructura de dependencias anidadas como la falta de información en los nombres, pueden suponer un problema para comprender el entramado. Por este motivo, cuando diseño funciones/métodos o cuando refactorizo, espero a dividir el código en funciones más pequeñas hasta que me doy cuenta de cuál es la cohesión que quiero darle al conjunto. Típicamente, aguanto hasta tener esa funcionalidad terminada para extraer funciones a partir de bloques de código. No me preocupa que un método llegue a alcanzar el alto de una pantalla mientras estoy trabajando en él, porque como desarrollo con test, sé perfectamente que lo puedo refactorizar sin riesgo (si es código nuevo lo escribo aplicando TDD). También es cierto que el tamaño de letra que utilizo es bien generoso. Al terminar el requisito funcional, extraigo los métodos necesarios para reducir la carga cognitiva que conlleva el método principal. Quizá llegue a las treinta líneas (el número concreto es irrelevante) antes del refactor y termine con cuatro funciones de siete líneas cada una después del mismo. Dependerá de que las funciones resulten cohesivas, en cuyo caso no será costoso encontrar un buen nombre para ellas. Además, dependerá de que el acoplamiento entre ellas sea el mínimo imprescindible. De nuevo, vemos cómo los nombres que elegimos son tan importantes que incluso nos ayudan a calibrar los niveles de cohesión y acoplamiento (si les prestamos atención).

La cohesión no es una característica deseable exclusivamente de las funciones, sino para cualquier artefacto, sistema o incluso equipo humano.

La cohesión en un artefacto se reconoce cuando no le sobran ni le faltan piezas, cuando no parece haber ninguna agrupación lógica mejor para sus elementos. Si hablamos de una clase, resultará cohesiva cuando no parece que le sobre ninguna de sus variables de instancia, ni de sus métodos, ni que tenga sentido partir dicha clase en dos o más clases.

Connascence

Una de las propuestas más estructuradas sobre cómo medir el acoplamiento es la de *connascence*. Es una palabra poco común en inglés y pese a que algunos sitios web la traducen al castellano como «connascencia», el Diccionario de la Lengua Española no recoge el término. No me atrevo a decir cómo se traduce, por lo que utilizo el término en inglés.

Se trata de una métrica de calidad, aplicable a aquellas relaciones entre dos o más elementos del *software*. Cuando se producen cambios en uno de los elementos, los otros también se ven obligados a cambiar, para preservar el correcto funcionamiento del sistema. El término fue acuñado por el autor Meilir Pages-Jones, en un artículo publicado en 1992, que pasaría a formar parte de su posterior libro, y luego popularizado o divulgado por Jim Weirich, alrededor de 2012. Las métricas se basan en tres dimensiones fundamentales, el grado, la localidad y la fuerza:

- Grado: ¿Cuántos elementos están acoplados entre sí?
- Localidad: Cuanto más reducido sea el ámbito del acople, menores sus efectos secundarios.
- Fuerza: ¿Cómo de explícito es el acoplamiento? Pues de haber acople, cuanto más explícito sea mejor, ya que el acoplamiento implícito es difícil de detectar y de corregir. Por eso, se considera débil al acople por referencia directa, mientras que la dependencia encubierta es considerada acople fuerte. Las dependencias que son detectadas por compiladores o linters, dan menos sorpresas que las que no.

Las variantes de *connascence* se clasifican como estáticas o como dinámicas, en función de si pueden observarse analizando el código o ejecutándolo. Así, las estáticas suponen un acople más débil que las dinámicas, hablando en términos de fuerza. Por ejemplo, dentro de las dinámicas tenemos la variante *Connascence of Execution*:

Java

```java
class Invoice {
    private List<Line> lines;
    private Tax tax;

    public void setLines(List<Line> lines){
        this.lines = lines;
    }

    public void setTax(Tax tax){
```

```
        this.tax = tax;
    }

    public Amount calculateNetAmount(){
        Amount amount = Amount.zero();
        for(Line line: lines){
            /*...*/
            amount = amount.add(tax.applyTo(line.amount()));
            /*...*/
        }
        /*...*/
    }
    /*...*/
}
```

Para que el método *calculateNetAmount* pueda funcionar, alguien debe haber invocado primero a *setLines* y a *setTax*, de lo contrario, se producirá una excepción por acceso a un objeto nulo (*NullPointerException*). Una forma de evitar esta dependencia del orden de ejecución es pasar los argumentos al constructor, como en el ejemplo anterior a este.

Las tres dimensiones de *connascence* nos sirven de guía a la hora de refactorizar código si nos centramos en reducir el grado, el ámbito y la fuerza de los acoples. Para conocer las variantes de *connascence* y ver ejemplos de ellas en diferentes lenguajes de programación, basta con hacer una búsqueda por la red, ya que la peculiaridad de la palabra arroja resultados enseguida. Además, la fuente original siempre es más que recomendable.

Si el principio de menor sorpresa era la brújula, los conceptos de cohesión y acoplamiento son la estrella polar para navegar por los entresijos del diseño de *software*.

7. Principios SOLID

SOLID es un acrónimo propuesto por Robert C. Martin, en un artículo del año 2000, y popularizado con la ayuda de profesionales como Michael Feathers y Sandi Metz. Se trata de cinco principios de diferentes autores:

1. *Single responsibility principle* - SRP (principio de responsabilidad única)
2. *Open-closed principle* - OCP (principio abierto-cerrado)
3. *Liskov substitution principle* - LSP (principio de sustitución de Liskov)
4. *Interface segregation principle* - ISP (principio de segregación de interfaces)
5. *Dependency inversion principle* - DIP (principio de inversión de las dependencias)

Estos principios son útiles incluso más allá de la programación orientada a objetos, son parcialmente aplicables a otros estilos de programación.

Principio de responsabilidad única (SRP)

Robert C. Martin explica el principio de responsabilidad única diciendo que cada artefacto que diseñamos debería tener únicamente un motivo para cambiar. Incumplimos este principio si, por ejemplo, diseñamos una clase con métodos para formatear textos de una factura y métodos para realizar cálculos sobre los importes de la factura, ya que el aspecto de dicha factura puede cambiar independientemente de los cálculos del importe y viceversa. No hace falta adivinar el futuro para intuir que potencialmente existe más de un motivo para cambiar esa clase.

Las razones para cambiar un componente están directamente relacionadas con el funcionamiento del negocio. No podremos ser capaces de identificar si cumplimos este principio, mientras no conozcamos lo suficiente el dominio en el que estamos trabajando. En mi opinión, no puedes construir un buen *software* para una fábrica de automóviles si no tienes ni idea de cómo son las labores del día a día en la fábrica. Cuando aterrizo en un sector que desconozco, realizo una labor de investigación y de autoformación para conocer las particularidades de los roles desempeñados por las personas que luego usarán el *software*. Siempre que puedo, me desplazo al lugar de trabajo para conocerlas a ellas y a sus cir-

cunstancias. Hablo con la gente y les observo para entender cuáles son sus necesidades, sus aspiraciones, sus preocupaciones… La visita es un gran ejercicio para empatizar, aumentar la motivación y hacer piña. Su colaboración puede ser imprescindible para el éxito del producto. Uno de los mayores problemas que tuvimos para implantar un *software* en el que trabajamos hace años, fue que los usuarios no sentían que se hubiese contado con su opinión lo suficiente a la hora de diseñarlo y muchos se negaban a utilizarlo (preferían el *software* antiguo pese a que sus quejas sobre su funcionamiento eran constantes).

Es fundamental que todo el equipo de desarrollo conozca en detalle la operativa del negocio para que puedan programar un código fiel a la realidad, con sentido. Como es natural, si se incorpora un miembro nuevo al equipo tendrá que trabajar de la mano de los que ya conocen el dominio, por ejemplo, practicando *pair programming*. Desde luego, quienes desempeñan el rol de PO (*product owners*) conocen perfectamente la idiosincrasia del grupo al que está dirigido el *software*. De lo contrario, en vez de construir un *software* adaptado a las personas, terminan siendo ellas quienes se ven forzadas a adaptarse al funcionamiento de un *software* que les complica la labor. Lo ideal es que entre los miembros del equipo de desarrollo se aglutine el conocimiento necesario como para ejecutar las acciones del día a día del negocio al que servimos. Pongamos que creamos una aplicación para un restaurante, cuyo personal anota las comandas en un dispositivo inalámbrico: si el equipo de desarrollo se trasladase al restaurante, debería ser capaz de explicar con total precisión cómo se trabaja allí, desde que un cliente aparece hasta que se marcha. He visto más proyectos fracasar por desconocimiento del sector y por problemas de comunicación entre personas, que por cualquier otro motivo. Una gestión inadecuada de los recursos disponibles para afrontar un proyecto, puede forzar al equipo a empezar a escribir código antes de conocer los procesos de negocio: «no entendemos cómo funciona esto, pero id empezando a programar aunque sea el registro». Resulta escandalosamente caro descubrir el grueso del funcionamiento de un negocio mediante la entrega de *software* con reglas inventadas y las interminables rondas de rectificaciones sucesivas ¿Pasará esto porque las partes interesadas cambian de opinión cada día o será porque no hemos entendido a qué se dedican? El descubrimiento del problema, la toma de requisitos y el resto de procesos de análisis de una solución, dan como para escribir varios libros completos (en inglés existe literatura muy interesante al respecto, en el apéndice los tienes).

Una vez que comprendemos el negocio, somos capaces de identificar los motivos de cambio de cada componente. El nivel de abstracción de los motivos de cambio varía según hablemos de una función/método, una clase, un paquete/módulo, un microservicio, etc. En artefactos pequeños como las funciones, los posibles cambios tendrán una relación más

directa con decisiones técnicas que con decisiones de negocio, aunque todo tenga sentido dentro del negocio. Volvamos al ejemplo de facturación: es posible que sea apropiado disponer de un paquete de *software* compuesto por los cálculos de la factura y por las preferencias de formato. Donde es definitivamente inapropiado, es en un mismo método o en una misma clase, puesto que ambos artefactos deberían tener un uso específico y exclusivo.

En las funciones o los métodos, a menudo es posible identificar diferentes responsabilidades separadas por líneas en blanco o por líneas de comentario:

Java

```
public void removeTabAt(int position) {
    final int selectedTabPosition = mSelectedTab != null ? mSelectedTab.
        getPosition() : 0;
    removeTabViewAt(position);

    final Tab removedTab = mTabs.remove(position);
    if (removedTab != null) {
        removedTab.reset();
        sTabPool.release(removedTab);
    }

    final int newTabCount = mTabs.size();
    for (int i = position; i < newTabCount; i++) {
        mTabs.get(i).setPosition(i);
    }

    if (selectedTabPosition == position) {
        selectTab(mTabs.isEmpty() ? null : mTabs.get(Math.max(0, position -
            1)));
    }
}
```

Lo primero que me llama la atención del bloque son las líneas en blanco, ya que parecen delimitar subtareas de la función. La primera línea está desconectada del resto, no tiene sentido hasta que alcanzamos a leer el último bloque. Un cambio aparentemente inocuo, a la vez que esclarecedor, consiste en reubicar líneas (ojo, que sin ejecutar test automáticos no puedo garantizar que sea inocuo):

Java

```
public void removeTabAt(int position) {
    removeTabViewAt(position);
    final Tab removedTab = mTabs.remove(position);
    if (removedTab != null) {
        removedTab.reset();
```

```
        sTabPool.release(removedTab);
    }

    final int newTabCount = mTabs.size();
    for (int i = position; i < newTabCount; i++) {
        mTabs.get(i).setPosition(i);
    }

    final int selectedTabPosition = mSelectedTab != null ? mSelectedTab.
        getPosition() : 0;
    if (selectedTabPosition == position) {
        selectTab(mTabs.isEmpty() ? null : mTabs.get(Math.max(0, position -
            1)));
    }
}
```

Ahora observamos que el proceso está compuesto por tres pasos: eliminar pestaña (*tab*), recolocar las pestañas restantes y actualizar la selección de pestaña si hiciera falta. Extraigo métodos y, además, aprovecho para hacer algunos cambios de nombres de variables, de manera que sean pronunciables:

Java

```
public void removeTabAt(int position) {
    destroyTab(position);
    shiftTabsFrom(position);
    updateTabSelection(position);
}
private void destroyTab(int position){
    removeTabViewAt(position);
    Tab removedTab = tabs.remove(position);
    if (removedTab != null) {
        removedTab.reset();
        tabPool.release(removedTab);
    }
}
private void shiftTabsFrom(int position){
    for (int i = position; i < tabs.size(); i++) {
        tabs.get(i).setPosition(i);
    }
}
private void updateTabSelection(int position){
    if (selectedTabPosition() == position) {
        selectTab(tabs.isEmpty() ? null :
                  tabs.get(Math.max(0, position - 1)));
    }
}
private int selectedTabPosition(){
    return selectedTab != null ? selectedTab.getPosition() : 0;
}
```

El método principal no puede dejar de encargarse de invocar a los demás, lo que sucede es que ahora su única responsabilidad es la de gestionar esa secuencia de llamadas. Ya no conoce los detalles de implementación de los otros métodos, cuya responsabilidad es única también. Los cuatro métodos privados desconocen el orden en el que se les invoca, porque es algo que no les concierne. Esto nos lleva a un diseño modular que reduce la carga cognitiva al leer el código y reduce el acoplamiento.

Evita la responsabilidad compartida

Si nos pasamos dando vueltas de rosca al principio de responsabilidad única, podemos acabar en el extremo opuesto, repartiendo una misma responsabilidad entre dos o más artefactos. Perderíamos la cohesión y generaríamos un acoplamiento públicamente visible. Podemos identificar este problema cuando, por ejemplo, dos clases van a todas partes juntas, porque una no tiene sentido sin la otra. Veamos un ejemplo:

Java

```
class EncryptorConfiguration {
    private final SecretKey key;
    private Cipher cipher;

    public EncryptorConfiguration(SecretKey key){
        this.key = key;
    }
    public void configureForEncryption(String transformation) {
        try {
            cipher = Cipher.getInstance(transformation);
            cipher.init(Cipher.ENCRYPT_MODE, key);
        } catch (NoSuchPaddingException |
                 NoSuchAlgorithmException |
                 InvalidKeyException e) {
            throw new CipherConfigurationException(e);
        }
    }
    public Cipher getCipher(){
        return cipher;
    }
}

class Encryptor {
    private EncryptorConfiguration config;

    public Encryptor(EncryptorConfiguration config) {
        this.config = config;
    }

    public void encryptFile(String content, String filename, String
```

```
        transformation) throws IOException {
        config.configureForEncryption(transformation);
        byte[] iv = config.getCipher().getIV();
        try (var fileOut = new FileOutputStream(filename);
             var cipherOut = new CipherOutputStream(fileOut, config.getCipher
                 ())
        ) {
            fileOut.write(iv);
            cipherOut.write(content.getBytes());
        }
    }
}
```

La segunda clase no puede encriptar sin la primera que, a su vez, no sirve para nada sin la segunda. No es verdad que en este contexto haya una responsabilidad de configurar y otra de encriptar. Aunque se ha hecho una inyección de dependencias, no se pueden reemplazar las implementaciones de ninguna de las clases, porque la inicialización de *cipher* es específica para la forma en que luego se usa. Más tarde o más temprano, nos daremos cuenta de estos problemas de diseño por lo molestos que son, por lo que nos cuesta entender el código y por los *bugs* que acarrean.

Componentes ortogonales

La metáfora de la ortogonalidad tal como se explica en el libro *The Pragmatic Programmer*, puede darnos pistas a la hora de diseñar nuestros componentes: en una gráfica con dos ejes de coordenadas, un punto puede desplazarse por uno de los ejes sin moverse por el otro (sin afectar al otro). Resumiéndolo en una frase, dice que un componente debe poder ser modificado sin que afecte a otro. Si los cambios en un artefacto obligan a cambios en otro para mantener el sistema funcionando, no sería un diseño ortogonal. Para conseguir tal diseño, es obvio que cada componente debe cumplir con el principio de responsabilidad única. Cuando los componentes son ortogonales pueden intercambiarse, reemplazarse y componerse, como las piezas de un lego. Los comandos típicos de cualquier sistema Linux son ortogonales (por herencia de Unix) en cuanto a que cumplen una función útil por sí solos y, además, se pueden combinar con otros de muchas maneras, cada uno tiene su responsabilidad:

Bash

```
cat ip_addresses | grep 192.168 | sort -n | uniq -c | sort -nr
```

Pensar en términos de ortogonalidad puede ser útil para diseñar clases, módulos o micro-

servicios, tan enfocados como los comandos de un sistema Linux.

Si tuviéramos que desarrollar funcionalidad para leer lotes de ficheros con formato XML y guardar cierta información en una base de datos, ¿cuántos elementos combinables podrías dibujar? Barajemos varias opciones. Pongamos que empezamos con un diseño de una sola clase:

Java

```java
class BatchXmlImporter {
    public void importXmlintoDatabaseFrom(Path folderPath){ /*...*/ }
}
```

Posiblemente quede una clase con un método muy grande o quizá con muchos métodos privados pequeños. En cualquier caso, es una clase que tiene el objetivo de leer ficheros de un directorio con una cierta estructura y guardarlos en una base de datos. No queda ningún margen para combinar funcionalidades alternativas, como, por ejemplo, importar otros formatos diferentes o leer de otras fuentes, como una unidad de red o un servicio de FTP.

Cuando hablo de alternativas combinables, no estoy refiriéndome a escribir código abstracto y genérico para soportar cualquier necesidad futura. Recordemos que el código sostenible se escribe para los requisitos de hoy y no las suposiciones sobre el mañana. A lo que me refiero con tener opciones es a separar responsabilidades, para que cada clase sea simple, tenga una única misión y la cumpla perfectamente. Dejo el ejercicio para ti, aunque tienes una posible solución con tres clases en este repositorio.

El principio de responsabilidad única tiene muchas lecturas, dado que es bastante abstracto, lo cual lo convierte más bien en un conjunto de heurísticas. Diseñar artefactos que cumplen con este principio a la primera es algo que conseguimos pocas veces, sin embargo, aplicando refactorización diaria con esta meta en mente, lo lograremos progresivamente. No refactorizamos para conseguir un código *limpio* ni *bonito*, sino para que cumpla con unos principios sobre los que somos capaces de razonar.

Principio abierto-cerrado (OCP)

Bertrand Meyer escribió por primera vez sobre este principio (*Open-closed Principle*) en su libro *Object Oriented Software Construction* (año 1988). Meyer consideraba que era código

abierto aquel que admitía ampliaciones y que estaba cerrado aquel que se usaba por otros módulos (por su compromiso con ellos).

Hoy se dice que un artefacto debe admitir modificaciones en su comportamiento sin necesidad de cambiar su código, es decir, debe estar abierto a extensión y cerrado a modificación. En lenguajes orientados a objetos, típicamente se utilizan la composición o la herencia para tal fin (¡mejor la composición!). El patrón estrategia, es un ejemplo de aplicación de este principio. Ojo, que si lo aplicamos al pie de la letra en todos los artefactos que diseñemos, vamos a terminar con una complejidad infernal, sobre todo si lo planteamos como otra forma de programar código genérico que aguante toda la vida sin modificaciones. En los primeros capítulos, escribí sobre los problemas de pretender escribir código intocable. Sellar los artefactos de antemano supone autolimitarse desde antes de que conozcamos siquiera los problemas futuros. Un claro ejemplo reside en una leve diferencia entre Java y C#: mientras que en Java los métodos de una clase están por defecto abiertos a ser sobreescritos (*overrided*) por clases hijas, en C# son finales por definición. Ambos lenguajes permiten la herencia y la reescritura. Si en Java queremos evitar que se pueda sobreescribir un método en una clase hija, hay que especificarlo explícitamente mediante el uso de la palabra reservada «final» en la definición del método. Si en C# queremos conceder a las clases hijas la posibilidad de reescribir un método heredado, hay que especificarlo explícitamente con la palabra reservada «virtual». Dos caras de la misma moneda. En mi experiencia, nunca tuve problemas cuando dejaba los métodos abiertos al trabajar con Java, pero sí los tuve dejándolos cerrados a la hora de trabajar con C#. No problemas graves, sino molestias al suplantar elementos con *mock objects*. Este pequeño ejemplo no significa que un lenguaje sea mejor que otro, personalmente, C# me parece una genialidad, es sin duda de mis lenguajes favoritos (es más moderno que Java y sintácticamente más cómodo). El trabajo de Anders Hejlsberg y su equipo es brillante en mi opinión, tanto en el diseño de C# como en el de TypeScript.

El lugar más habitual donde encontramos implementaciones de este principio es el código de los *frameworks* y librerías de propósito general. Hay una buena razón para ello y es que son herramientas que deben adaptarse a nuestras necesidades sin modificar su código. Quienes desarrollan y mantienen un *framework*, quieren ofrecer flexibilidad en sus prestaciones sin que personas externas al proyecto modifiquen su código. Los mecanismos habituales de extensión son la configuración (*xml, yml, json*, programática…), la herencia y la composición. Los asuntos a tener en cuenta cuando se desarrolla código horizontal son diferentes a los que debemos considerar al trabajar con código vertical. Entre otras cosas, porque el código horizontal no tiene que ser mantenido por los consumidores.

Los *frameworks* y librerías de propósito general, disociados de cualquier dominio vertical o específico, son muy útiles para no reinventar la rueda y para no olvidarnos de atributos de calidad, como la seguridad o la internacionalización. Van evolucionando y cada vez son menos invasivos, menos prescriptivos, más ligeros, más sencillos, más humildes… Por otra parte, los «*frameworks* a medida», diseñados dentro del marco de un proyecto orientado a un sector profesional específico (turismo, sanidad, empleo, automoción…) me han resultado cargantes, limitantes y contraproducentes. Si el código es de mi equipo, prefiero tener la capacidad de cambiar cualquier parte del mismo cada vez que necesite algún ajuste. Prefiero que no haya mecanismos de extensión rimbombantes.

Considero un mal olor que para una aplicación vertical se construya un *framework*, sobre todo si no es *open source*. Primero, porque seguramente para el negocio es más prioritario lanzar al mercado cuanto antes funcionalidades de la aplicación que disponer de un *framework*. Segundo, porque ya hay *frameworks* de propósito general que seguramente resuelven mejor, que son libres, gratuitos y que están avalados por una gran comunidad (o por una gran empresa con décadas de experiencia en el desarrollo de *frameworks* y con capacidad para contratar a profesionales especialistas). Tercero, porque liberar el código somete al *framework* al escrutinio de personas que no tienen conflicto de interés y que manejan otros criterios. Publicar el código del *framework* es un gesto de transparencia que demuestra que no hay nada que ocultar. Las veces que he tenido que decidirme por utilizar un nuevo *framework*, he leído su código fuente, he observado su cobertura de test y me he fijado en la actividad de su comunidad. Elegir un *framework* puede ser una decisión que aumente la productividad, pero también nos puede poner contra las cuerdas.

Entonces, ¿tiene beneficios el principio abierto-cerrado más allá de *frameworks*? Pues sí. Nos invita a pensar en aumentar la cohesión de los artefactos y en reducir su acoplamiento con los otros, mediante el diseño de mecanismos de extensión. No se trata de implementar esos mecanismos siempre, sino de considerar la posibilidad de preparar el diseño para futuros cambios de comportamiento, que resultan verosímiles o razonables con el contexto que conocemos hoy. Se puede ver incluso como una planificación concreta de una deuda técnica. Por simplicidad, me gusta quedarme un paso por detrás de la introducción de puntos de extensión, de tal manera que cuando de verdad necesite extender, puedo cambiar un poco el código y abrirlo. Sandi Metz, en sus libros sobre diseño orientado a objetos, nos brinda una pauta muy útil para conseguir este objetivo, que radica en encapsular el uso de las dependencias. Lo vemos con un código antes de dicha encapsulación:

Java

```
class Repository {
    public void delete(User user){
        System.out.println("Deleting user:" + user.Id); // <-- dependencia de
             System.out
        /*...*/
    }
}
```

Y después de refactorizarlo cuando se practica la encapsulación:

Java

```
class Repository {
    public void delete(User user){
        log("Deleting user:" + user.Id);
        /*...*/
    }
    private void log(String message){ // <-- encapsulación de la dependencia
        System.out.println(message);
    }
}
```

No ha quedado un diseño extensible, sin embargo, el día que necesite hacerlo extensible tengo tres opciones a mano. Una opción basada en la herencia para que una clase heredera pueda redefinir el comportamiento:

Java

```
protected void log(String message){  // <--- protected permite override en
    herederas
    this.logger.log(message);
}
```

Otra opción basada en la composición con inyección de dependencias:

Java

```
class Repository {
    private Logger logger;
    public Repository(Logger logger){
        this.logger = logger;
    }
    public void delete(User user){
        log("Deleting user:" + user.Id);
        /*...*/
    }
    private void log(String message){
        logger.log(message);
    }
```

```
}
```

Otra opción basada en la composición implementando el patrón decorador:

Java

```java
public interface Repository {
    public void delete(User user);
}

public class DatabaseRepository implements Repository {
    public void delete(User user){
        /*... sólo trabajo con base de datos ...*/
    }
}

/**
 * Repository decorator that logs operations before execution
*/
public class LoggedRepository implements Repository {
    private Repository repository;
    private Logger logger;
    public LoggedRepository(Repository repository, Logger logger){
        this.repository = repository;
        this.logger = logger;
    }
    public void delete(User user){
        log("Deleting user": + user.Id);
        repository.delete(user);
    }
    private void log(String message){
        this.logger.log(message);
    }
}

public class Factory {
    public static LoggedRepository createLoggedDatabaseRepository(){
        return new LoggedRepository(new DatabaseRepository());
    }
}
```

Cada una de estas alternativas implica diferente grado de complejidad, cohesión y acoplamiento. Para mí, los principios de simplicidad y de menor sorpresa son una guía muy práctica hasta para encontrar los niveles adecuados de cohesión y acoplamiento. Por eso, diseño y refactorizo en busca de la simplicidad. En este ejemplo, es posible que la solución más simple sea escribir en un *log* a la entrada del *endpoint* y/o delegarlo en el motor de base de datos, en vez de hacerlo en el repositorio.

Métodos de extensión

Ya que hablamos de extensión sin modificación, merece la pena que hablemos de los métodos de extensión, para entender sus beneficios y sus inconvenientes. Hay varios lenguajes que permiten añadir funcionalidad a clases/módulos existentes sin tener que modificar su código y sin necesidad de herencia. En C# y Kotlin, los métodos de extensión son un recurso muy utilizado por *frameworks* y librerías. En Java, también se puede hacer mediante *plugins* para el compilador. En Python, Ruby y JavaScript/TypeScript, se pueden alterar los objetos directamente, recurriendo a la técnica de *monkey patching*, con resultados similares. A modo de ejemplo, vamos a ver cómo se podría añadir el método *toCamelCase* a la clase *String* en diferentes lenguajes.

C#:

C#

```
public static class ExtensionMethods
{
  public static string ToCamelCase(this string theString)
  {
    string camelCased = /*... algoritmo ...*/
    return camelCased;
  }
}
/*..*/
string test = "some_example"
string camelText = test.ToCamelCase() // someExample
```

Kotlin:

Kotlin

```
fun String.toCamelCase(): String {
    string camelCased = /*... algoritmo ...*/
    return camelCased;
}
```

Ruby:

Ruby

```
class String
    def camelize()
        camelize = #... algoritmo ...#
    end
end
```

JavaScript/TypeScript:

Javascript

```
String.prototype.toCamelCase = function () {
   let camelCased = /*... algoritm ...*/
   return camelCased;
};
```

Los métodos de extensión aportan legibilidad a las líneas de código que los invocan, porque hacen que se parezcan a frases en lenguaje natural, ya que los métodos se encadenan:

Java

```
name.ToCamelCase();
```

Los métodos de extensión son un mecanismo potente para construir APIs fluidas mediante el encadenado de funciones, ideal para librerías de propósito general como las de aserciones para test. Como todo, tienen sus inconvenientes y se paga un precio cuando se recurre a ellos ¿Cómo podemos razonar sobre los pros y los contras de esta técnica? Pues antes que nada, debemos pensar en cómo será la experiencia (*developer experience*) de las personas que mantengan el código en el futuro. Ponerse en el lugar de los demás es básico para tomar buenas decisiones para el equipo. Pongamos por caso el método de extensión *toCamelCase* en el objeto *String*. Las necesidades y situaciones que podemos preveer que se encontrará quien continúe el proyecto son las siguientes:

- En algún momento, el autocompletado del IDE va a sugerir el método al teclear el punto detrás del nombre de una variable de tipo cadena. La duda que le puede surgir a cualquiera, o no surgir, es; ¿Kotlin implementa este método en las cadenas?, ¿cómo funciona exactamente? Quizás vaya a buscarlo por Internet, en cuyo caso se frustrará por no encontrarlo en la documentación oficial. Si pone el cursor encima del método, el IDE sí que le dice si es un método implementado en Kotlin o no, y en qué fichero está la implementación. La cuestión es que se le ocurra hacer eso y que sepa leer la información que da el IDE. Seguramente, le va a ayudar que haya un comentario, pues se mostrará en la burbuja del autocompletado:

Kotlin

```
/**
* Extension method to convert string to camel case
* Usage:
*  theString.toCamelCase()
*/
fun String.toCamelCase(): String {
  /*...*/
}
```

- Si en el futuro añaden al proyecto una librería que casualmente incluye también un método de extensión con el mismo nombre, habrá un conflicto, mientras que si el nombre es similar habrá confusión sobre cuál es cuál. Si además la implementación difiere, se obtendrán resultados diferentes, dando lugar a *bugs*.
- Si en el futuro el propio Kotlin implementase ese método, ya no sería trivial actualizar la versión del lenguaje utilizada en el proyecto.
- Cuando alguien aterrice en el proyecto y explore la estructura de directorios, módulos, clases, etc., ¿dónde tendrá sentido encontrar la implementación de los métodos de extensión?, ¿qué inquietud le surgirá si las encuentra (o no las encuentra), en esta o aquella carpeta?

A continuación, podemos razonar sobre el diseño, sobre cohesión y acoplamiento. Un método de extensión es público y accesible desde cualquier punto del proyecto, por lo que su potencial de acoplamiento es muy elevado. Además, son métodos estáticos, lo que implica que no pueden guardar estado, por lo que su potencial de cohesión es insuficiente frente a problemas que no se resuelven con un único método. Mientras que una clase aglutina datos y operaciones sobre dichos datos, un método de extensión solamente contiene operaciones. Si predominan los métodos de extensión sobre las clases, el estilo de programación predominante no será orientado a objetos. Las clases son abstracciones que, cuando aciertan en su cometido de representar conceptos, actúan como un campo magnético para atraer el comportamiento de los nuevos métodos que aparezcan. Hacen que resulte intuitivo saber dónde colocar las nuevas funciones y dónde ir a buscarlas. Resolver problemas complejos mediante funciones estáticas aumentará la complejidad de las implementaciones por falta de cohesión.

Tanto por la experiencia al desarrollar como por las complicaciones en el diseño del *software*, los métodos de extensión deben reservarse para problemas genéricos, transversales y no para problemas concretos de un dominio. Sobre todo si lo que extendemos son mó-

dulos o clases del sistema base. Por eso, donde más se utilizan es en librerías, como, por ejemplo, Kotest, (Kotlin) o FluentAssertions (C#):

C#

```
string actual = "ABCDEFGHI";
actual.Should()
    .StartWith("AB")
    .And.EndWith("HI")
    .And.Contain("EF")
    .And.HaveLength(9);
```

Imagina cómo sería la experiencia de desarrollo, así como los niveles de cohesión y acoplamiento, si recurrimos a métodos de extensión para resolver problemas concretos de un dominio, como, por ejemplo, buscar una entidad por su identificador:

Kotlin

```
fun updateUsername(id: String, username: String) {
    val user = id.findUser();
    /*...*/
}

fun String.findUser(): User {
    val user = findUsersById(this)
    /*...*/
    return user
}
```

El hecho de que un objeto de tipo cadena tenga un método *findUser* le rompe los esquemas a cualquiera.

Los métodos de extensión podrían ser un recurso interesante a la hora de trabajar con código legado que no tenemos la posibilidad de modificar, pero que necesitamos ampliar. Supongamos que nos hace falta añadir un nuevo método para calcular el total de una factura, pero que es imposible modificar el código de la clase *Invoice*. Podríamos hacer que parezca que el método existe en la clase, con este método de extensión:

Kotlin

```
fun Invoice.totalAmount(): Amount { /*...*/}
```

El inconveniente de este enfoque es que el método de extensión no permite el acceso a variables privadas de la clase, con lo cual añadimos *getters* para acceder al estado y

perdemos la encapsulación de las clases. Sucede lo mismo si extendemos colecciones:

Kotlin

```
fun List<InvoiceLine>.totalAmount() : Amount {
    return this.map { line -> line.amount}
            .reduce { accumulator, amount -> accumulator.sum(amount)}
}
```

Además de los *getters*, otro riesgo de programar así es que al separar datos y operaciones, puede que el concepto *Invoice* nunca llegue a emerger (y quizá *Invoice* fuese una buena abstracción para envolver una colección de *InvoiceLine*, entre otras cosas). Los métodos de extensión son menos abstractos que las clases, para lo bueno y para lo malo.

Para hacer el código más fácil de mantener es imprescindible pensar en las situaciones que los demás se van a encontrar cuando les pasemos el testigo. No todo el mundo tiene la cabeza amueblada igual. Cuanto más esotérico sea nuestro estilo, más dura será la curva de aprendizaje para quienes que se incorporen al proyecto.

Principio de sustitución de Liskov (LSP)

Barbara Liskov es una científica de la computación americana, cuyo trabajo en las décadas de los 70 y los 80 ha tenido un profundo impacto en la programación orientada a objetos y en el diseño de lenguajes como Java o C#. B. Liskov y su compañero S. Zilles (en el MIT), fueron pioneros en la definición de ciertas reglas sobre abstracciones y tipos de datos que hoy en día asumimos como universales, publicando un *paper* en 1974 en la ACM Sigplan Conference. Ya a finales de 1973, Liskov había empezado a trabajar en el desarrollo del lenguaje CLU, que implementaba características revolucionarias para la época. CLU incluía, por ejemplo, tratamiento de excepciones, iteradores y lo que hoy conocemos como *generics* en Java y C# (*templates* en C++), concepto al que ella denominaba polimorfismo en aquel momento. El principio de sustitución (*Liskov Substitution Principle*), empezó a gestarse en 1974, aunque fue presentado con ese nombre en una ponencia en 1988 y, posteriormente, respaldado con un *paper* en 1994, escrito con la colaboración de Jeannette Wing. Estos datos los he extraído de una ponencia de B. Liskov en QCon London 2013.

Para situarnos un poco en el eje temporal, quiero destacar que el nacimiento de Java fue un año después, en 1995 (creado por James Gosling), y el de C# fue en el año 2000 (creado por Anders Hejlsberg), que también creó los legendarios Turbo Pascal y Delphi, y que ha

creado el popular lenguaje TypeScript).

En su libro *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*, Liskov y Guttag explican con todo tipo de detalles el principio de sustitución, que dota de coherencia a los tipos y reduce las sorpresas a la hora de entender el código. El principio dice que dado un tipo base o *supertipo*, cualquier *subtipo* suyo debe ser intercambiable por dicho *supertipo*. Esto aplica a cualquier artefacto que cumpla con una determinada interfaz, entendiendo como interfaz a un conjunto de métodos públicos (se utilice la palabra reservada *interface* o no), es decir, un contrato. Lo entendemos mejor con un ejemplo de código que incumple este principio:

Java

```
    /*...*/
    public void attack(Fighter fighter){
        if (fighter instanceof Ryu){
            ((Ryu)fighter).haduoken();
        }
        else if (fighter instanceof ChunLi){
            ((ChunLi)fighter).hakkei();
        }
        else if (fighter instanceof MrBison){
            ((MrBison)fighter).psychoInferno();
        }
        /*...*/
    }
}
class Fighter { /*...*/ }
class Ryu extends Fighter { /*...*/ }
class Ken extends Fighter { /*...*/ }
class ChunLi extends Fighter { /*...*/ }
class MrBison extends Fighter { /*...*/ }
```

El código toma decisiones en función del subtipo del argumento recibido. Concretamente, invoca a métodos de subclases mediante conversión de tipos (*type casting*) para poder acceder a cada método específico de la subclase. Recurrir a la conversión de tipos es un síntoma de debilidad en el diseño. Preguntar por subtipos con sentencias tipo *instaceOf, getType, getClass*, etc., (variaciones según el lenguaje de programación), es también un mal síntoma, salvo que estemos programando una librería basada en metaprogramación.

Como consumidor de cualquier método público, debo ser capaz de intuir lo que pretende con solo mirar su firma y no su contenido:

Java

```
public void attack(Fighter fighter);
```

La firma me da a entender que el ataque (*attack*) es aplicable a cualquier tipo de luchador (*Fighter*). No hay indicios que me hagan pensar que si mañana aparece un nuevo tipo de luchador, el código no va a funcionar. Me causaría una gran sorpresa descubrir que el método conoce a cada luchador específico y lo maneja diferente. Cada vez que haya cambios en los luchadores o se introduzca uno nuevo, estaré obligado a recordar que el método requiere modificaciones, porque el compilador no tiene la capacidad de avisarme.

Vamos a cambiar el diseño para cumplir con este principio. Podríamos definir el método de ataque en la interfaz matriz y que cada luchador lo implemente como más le convenga:

Java

```
    /*...*/
    public void attack(Fighter fighter){
        /*...*/
        fighter.attack();
        /*...*/
    }
}

interface Fighter {
    void attack();
}
class Ryu implements Fighter {
    public void attack(){
        /*...*/
        hadouken();
        /*...*/
    }
    private void hadouken(){/*...*/}
}
```

Este principio nos ayuda a conseguir esos bloques cohesivos, y a la vez intercambiables, que citaba con el ejemplo del tren y los vagones. En lenguajes interpretados, podemos recurrir a la técnica de *duck typing* si respetamos este principio:

Python

```
def attack(fighter):
    #...
    fighter.attack()
    #...
```

De lo contrario, tendríamos que preguntar si el método existe o capturar excepciones:

Python

```
def attack(fighter):
    try:
       fighter.attack()
    except (AttributeError, TypeError):
       print "The given argument is not a real fighter"
       #...
```

Entre otras ventajas, un diseño que respeta el LSP es más portable a otros lenguajes de programación, porque tal como estamos viendo, el resultado es parecido. No todos los lenguajes tienen el mismo soporte de tipos, pero casi todos utilizan funciones o métodos.

LSP dice que los subtipos de un sistema deben de cumplir tres reglas:

- Regla de la firma: los métodos de un subtipo han de tener la misma firma que los del supertipo. Obvio para quienes aprendieron sobre los tipos con Java/C#, puesto que el compilador fuerza a que cualquier implementación de una cierta interfaz tenga todos sus métodos con la misma firma. Sin embargo, en la época en que se definió esta regla, no era nada obvio ni universal.
- Regla de los métodos: cada método de un subtipo debe preservar el comportamiento del supertipo. Por ejemplo, si un supertipo que representa una colección de elementos tiene un método para añadir un elemento que siempre incrementa el tamaño de la colección, la implementación de dicho método por parte de cualquier subtipo también debe incrementar el tamaño de la colección (esto es una postcondición). Por este motivo, un conjunto no debe ser subtipo de una lista, puesto que la lista siempre crecerá al añadir, mientras que el conjunto no será alterado si el elemento que se intenta añadir ya está contenido. La regla de los métodos no puede ser validada por el compilador porque es cuestión de semántica.
- Regla de las propiedades: cada tipo mantiene una serie de propiedades invariantes que definen su esencia; los subtipos deben mantener las invariantes de los supertipos. Por ejemplo, si el tipo base es un conjunto, una propiedad invariante del mismo, es que no puede haber elementos repetidos. Ningún subtipo del conjunto admitirá elementos repetidos, porque estaría violando la invariante del supertipo y causará sorpresa a quien consuma dicho subtipo. Otro ejemplo que incumple la invariante es el de un rectángulo que pretende ser subtipo de un cuadrado; no estaría respetando la propiedad de que todos los lados han de ser iguales.

Tanto Barbara Liskov como Bertrand Meyer hablan de precondiciones, postcondiciones e

invariantes en el diseño de subtipos. Una precondición es una propiedad de los parámetros de un método o del estado del tipo previo a la ejecución del método, mientras que una postcondición es una propiedad del valor retornado o del estado del tipo al terminar la ejecución.

Un método del subtipo podría tener menos restricciones que el supertipo en sus precondiciones, pero debe mantener todas las postcondiciones. Incluso podría tener postcondiciones más estrictas que el supertipo. El patrón *null object*, puede ser un buen ejemplo para ilustrar pre y postcondiciones. Ejemplo:

Java

```java
class Player {
    private final Nickname nickname;
    private int points = 0;
    private boolean isAlive;

    public Player(Nickname nickname, int points){
        isAlive = true;
        this.nickname = nickname;
        this.points = points;
    }
    /*...*/
    protected boolean isAlive(){
        return isAlive;
    }
    /*...*/
    protected int points(){
        return points;
    }
}

class NullPlayer extends Player {
    public NullPlayer() {
        super(new Nickname("Irrelevant"), 0);
    }
    /*...*/
    @Override
    protected boolean isAlive() {
        return false;
    }
    /*...*/
    @Override
    protected int points() {
        return 0;
    }
}
```

La clase *Player* toma por constructor un nombre de usuario y unos puntos que luego la

clase hija no necesita para nada, es decir, las precondiciones del subtipo son más laxas que las del supertipo. Por otra parte, el supertipo devuelve un boolean y un número, mientras que el subtipo siempre devuelve falso y cero, es decir, las postcondiciones del subtipo son más fuertes que las del supertipo.

Lo que no podría suceder es que el subtipo tenga unas precondiciones más específicas que el supertipo. Por ejemplo:

Java

```
class Rectangle {
    private double high;
    private double wide;

    public Rectangle(double high, double wide) {
        this.high = high;
        this.wide = wide;
    }
    /*...*/
    public void resize(double high, double wide){
        this.high = high;
        this.wide = wide;
    }
}
class Square extends Rectangle {
    private double length;

    public Square(double length){
        super(length, length);
    }
    /*...*/
    @Override
    public void resize(double high, double wide) {
        if (high != wide){
            throw new IllegalArgumentException("All sides of a square must be
                  equal");
        }
    }
}
```

El método para redimensionar tiene unas precondiciones más estrictas en el subtipo para tratar de mantener sus invariantes, por lo cual esta implementación del cuadrado no puede sustituir a la del rectángulo en cualquier parte del código, incumpliendo el LSP. La consecuencia de esta jerarquía es la proliferación de programación defensiva por todo el código, con preguntas del tipo, *¿es esto de verdad un rectángulo o es un cuadrado?* Si la implementación fuera la siguiente, sí cumpliría con el principio, puesto que cumple todas las reglas:

Java

```
class Rectangle {
    private double high;
    private double wide;

    public Rectangle(double high, double wide) {
        this.high = high;
        this.wide = wide;
    }
    public double area() {
        return high * wide;
    }
    public double diagonal(){
        return Math.sqrt(high * high + wide * wide);
    }
}
class Square extends Rectangle {
    private double length;

    public Square(double length){
        super(length, length);
    }
}

public class Tests {
    @Test
    public void a_square_is_also_a_rectangle(){
        Rectangle rectangle = new Rectangle(100, 100);
        Square square = new Square(100);
        assertThat(rectangle.diagonal())
            .isEqualTo(square.diagonal());
        assertThat(rectangle.area())
            .isEqualTo(square.area());
    }
}
```

Diseñar tipos y subtipos es complicado, la manera en que me resulta más natural encontrar jerarquías o familias de tipos es refactorizando. Cuando tengo dos o tres tipos que parecen compartir propiedades, estudio cómo de factible es agruparlos aplicando LSP.

Principio de segregación de interfaces (ISP)

Para mí, este principio es un caso específico dentro del principio de responsabilidad única. Dice que las interfaces deberían ser minimalistas, como para que todos sus métodos sean utilizados por cada consumidor. Si una interfaz tiene varios consumidores que utilizan sub-conjuntos de métodos diferentes de dicha interfaz, seguramente deberíamos dividirla en

tantas interfaces como subconjuntos se estén usando en la práctica. Divide y vencerás. Las interfaces infladas acoplan a los consumidores entre sí, en el sentido de que cuando se actualice una interfaz para cumplir con un requisito procedente de uno de los consumidores, los demás también tendrán que actualizarse, aunque no les afecte el cambio a nivel funcional. Este problema cobra especial relevancia cuando las interfaces se usan en artefactos que se despliegan de manera independiente, como sucede con los microservicios, ya que no queremos tener que redesplegar varios cuando solamente uno necesita actualización.

Principio de inversión de las dependencias (DIP)

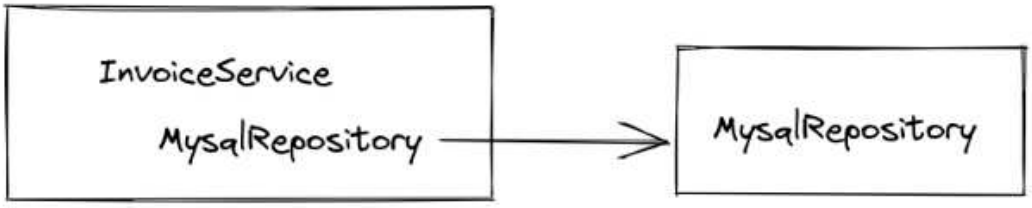
En el capítulo sobre cohesión y acoplamiento vimos un ejemplo de cómo inyectar dependencias, pero no quedó explicado en qué consiste el principio de inversión. Este dice que una clase con un nivel de abstracción alto no debería depender de una clase con un nivel de abstracción más bajo. Nivel alto significa más cercano al negocio, mientras que bajo significa más cercano a la infraestructura, «al hierro».

Java

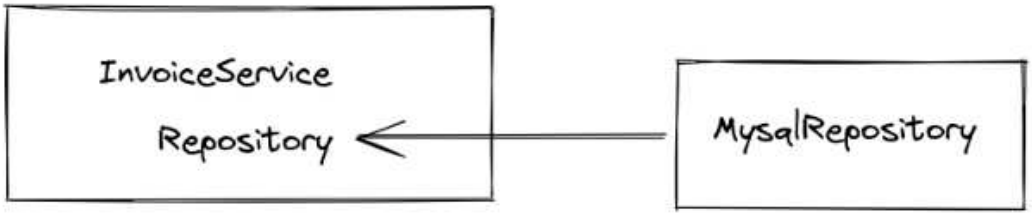
```
class InvoiceService {
    MysqlRepository repository = new MysqlRepository();

    public List<Invoice> findInvoiceBy(InvoiceNumber number){
        /*...*/
    }
}
```

Aquí se está violando el principio, porque la clase *InvoiceService* tiene mayor nivel de abstracción que *MysqlRepository* y es dependiente de ella. En el momento en que *InvoiceService* decide instanciar a *MysqlRepository*, se establece el sentido de la dependencia que es contrario a lo que nos dice el principio:



Invertir las dependencias significa invertir el sentido de la flecha:



Para conseguirlo, lo que aparece en el lado derecho de la asignación no puede ser elegido por la clase de más alto nivel, sino por una tercera:

Java

```
class InvoiceService {
    private final InvoiceRepository repository;

    public InvoiceService(Repository repository){
        this.repository = repository; // <- inversión mediante inyección
    }

    public List<Invoice> findInvoiceBy(InvoiceNumber number){
        /*...*/
    }
}

interface InvoiceRepository {
    List<Invoice> findInvoiceBy(InvoiceNumber number);
}

class InvoiceRepositoryMysql implements InvoiceRepository {
    public List<Invoice> findInvoiceBy(InvoiceNumber number){
        /*...*/
    }
}
```

Ahora *InvoiceService* desconoce el nivel de abstracción de *InvoiceRepository*, ya que en cierta forma lo hemos elevado. Hemos «abierto un puerto» en el servicio para que sea posible inyectar cualquier clase que implemente el contrato de su dependencia. La inversión no significa que el repositorio dependa del servicio, realmente es natural que la clase con mayor alcance se componga de otras de menor calado, por tanto, el servicio sigue dependiendo del repositorio. Sin embargo, ya no depende de un repositorio concreto, ha cambiado el sentido de la flecha. La clase *InvoiceRepositoryMysql* depende de *InvoiceRepository*, que será una interfaz o una clase base. No siempre hay por qué introducir interfaces, *InvoiceRepository* podría ser una clase también, pero lo importante es que el servicio no invoca a

su constructor. Así es como la dependencia se mantiene como un artefacto abstracto para el servicio. La inyección de dependencias la realiza típicamente una clase encargada de la construcción de objetos, o bien un *framework* que haga de contenedor de dependencias:

Java

```
public class Factory {
    public static InvoiceService createInvoiceService(){
        return new InvoiceService(new InvoiceRepositoryMysql());
    }
}
```

8. Implementación sostenible

Para evitar que el código se nos eche a perder tenemos que cuidarlo a diario. Hay que ponerle atención al detalle para mantener cada línea simple y legible. Si escribimos cada bloque de código con atención y tomando cada microdecisión de manera deliberada, seremos capaces de mantenerlo bajo control a lo largo del tiempo. Un jardín se mantiene bello cuando las hierbas se arrancan cada día y las plantas se riegan regularmente, en la justa medida. El jardinero observa permanentemente a las plantas y conoce cuándo necesitan cuidados especiales. En el código no buscamos la belleza sino el pragmatismo, y justamente lo más práctico es mantenerlo cuidado. Un código que ha sido escrito y refactorizado con esmero, resulta bello para quien sabe reconocer la generosidad y la humildad que hay en él.

Todos los capítulos anteriores exponen técnicas o principios para cuidar del código a diario, lo que hacemos en este es revisar técnicas cotidianas muy concretas que a menudo se infravaloran. Vamos a hablar del poder comunicativo del código, de la estructura de los bloques y los artefactos, así como de características de los lenguajes que suelen menospreciarse. Este capítulo está especialmente inspirado en el libro de Kent Beck, *Implementation Patterns*, si bien es cierto que contiene ideas de muchos otros profesionales. A continuación se explican recomendaciones, heurísticas y principios, para cuidar de los detalles que marcan la diferencia. También expongo antipatrones de implementación que me encuentro a menudo, en multitud de proyectos, porque la gente no sabe que lo son. Cuando hablo de patrones no me estoy refiriendo necesariamente a los patrones de diseño, sino al sentido más general de patrón, como forma que se replica en diferentes lugares.

Indentación consistente

Una de las características que más nos gusta del lenguaje Python, es que nos obliga a indentar cada bloque en la misma columna acorde a su nivel de anidamiento. Una correcta indentación es fundamental para identificar rápidamente las distintas partes de un listado de código y evitar confusiones. Hoy en día, los editores de código ofrecen la posibilidad de corregir la indentación automáticamente con un atajo de teclado e incluso de poder elegir plantillas con nuestro estilo de indentación preferido. Lo importante no es si las llaves (*curly*

braces) van en la misma línea o en la siguiente, sino acordar un criterio y cumplirlo siempre. Es beneficioso seguir las convenciones de la comunidad de usuarios de cada lenguaje en cuanto a la indentación, sobre todo para facilitar la incorporación de otras personas al equipo. No obstante, cada equipo es libre de configurar su estilo. Lo más importante es la consistencia a lo largo y ancho del repositorio de código.

Cuando trabajo con nóveles que me piden ayuda para encontrar un *bug*, típicamente han dedicado mucho tiempo rebuscando en un código mal indentado. Nos ponemos manos a la obra a trabajar juntos y lo primero que sugiero es que arreglemos la indentación línea a línea empezando por arriba. Realizamos los cambios manualmente para entender cada línea. En muchos casos encuentran el *bug* sin mi ayuda, solo con el ejercicio de corregir la indentación.

Cuando hago las veces de entrevistador para un puesto de trabajo, me fijo en la importancia que la gente le da a la indentación, tanto si programamos juntos en vivo como si me muestran código ya hecho. No le doy importancia a que alguna línea se haya descuadrado (nos puede pasar a todos), sino a que en general quiera mantener el código indentado. Interpreto la falta de sensibilidad hacia la indentación como falta de horas de experiencia batallando contra *bugs*.

Si tenemos que corregir la indentación de uno o varios ficheros de código fuente y están siendo versionados con alguna herramienta de control de versiones (no debería ser de otra manera a estas alturas), es importante no mezclar cambios funcionales con cambios de indentación en la misma versión (hacerlo en *commits* separados si hablamos de *Git*). Si la corrección del indentado la ejecuta el IDE automáticamente, no hay posibilidad de introducir errores y tampoco habrá necesidad de revisar los cambios en esa versión (ese *commit*), porque no contienen alteraciones en la funcionalidad.

Cuando era niño, el profesor del colegio que nos explicaba matemáticas, D. Alfonso, hacía mucho hincapié en que las líneas horizontales de las ecuaciones y de las facciones estuvieran alineadas; yo siempre lo agradecí. Cuando era estudiante universitario, el profesor que nos enseñaba teoría de lenguajes formales, Kiko, hacía mucho hincapié en que el código estuviera bien indentado; siempre lo agradecí, porque me ha permitido encontrar/evitar muchos *bugs*.

Reducir las líneas en blanco intercaladas

He aquí un antipatrón universal disfrazado de buena práctica; las líneas en blanco entre bloques de código de una misma función. No me refiero al principio de la función, o entre funciones, sino en medio de un mismo método ¿Para qué dejas líneas en blanco? Típicamente, la respuesta de la gente es, «porque así el código se ve más claro o más limpio». Es una respuesta demasiado abstracta y subjetiva. Con frecuencia, la verdadera razón es *que el bloque de encima tiene una responsabilidad y el de abajo tiene otra distinta* ¡Ajá!, casi inconscientemente, somos capaces de discernir que un bloque de código está resolviendo varios problemas a la vez y de manera sutil los delimitamos con líneas en blanco. Veamos un ejemplo:

Java

```java
public void onScale(ScaleGestureDetector detector) {
    float scale = detector.getScaleFactor();

    // Speeding up the zoom by 2X.
    if (scale > 1f) {
        scale = 1.0f + (scale - 1.0f) * 2;
    } else {
        scale = 1.0f - (1.0f - scale) * 2;
    }

    float newRatio = getZoomRatio() * scale;
    newRatio = rangeLimit(newRatio, getMaxZoomRatio(), getMinZoomRatio());
    setZoomRatio(newRatio);
}
```

Si aplicamos el principio de programar denotando una intención clara, podemos eliminar la línea en blanco y sustituir los bloques por llamadas a funciones. Además, este código tiene una línea de comentario que igualmente puede ser eliminada, creando una función con un nombre explicativo:

Java

```java
public void onScale(ScaleGestureDetector detector) {
    float scale = doubleZoomSpeedScale(detector.getScaleFactor());
    setZoomRatio(calculateZoomRatioFor(scale));
}
```

El código ya no necesita líneas en blanco para marcar la separación de responsabilidades. Si te gustan las líneas en blanco, ahora podrías volver a añadir una, aunque ya no hace falta

(acerca de los gustos procuro no discutir):

Java

```java
public void onScale(ScaleGestureDetector detector) {
    float scale = doubleZoomSpeedScale(detector.getScaleFactor());

    setZoomRatio(calculateZoomRatioFor(scale));
}
```

Aunque no tengo nada en contra de la línea en blanco, mi consejo es realizar el ejercicio de eliminarlas primero, como para sentir cuánto nos incomoda su ausencia, y de esa forma, detectar posibles mejoras aplicables al diseño o a la legibilidad. Una vez hecho el ejercicio, se pueden introducir líneas en blanco, pese a que su presencia podría incitar al abuso por parte de los demás (o de mí mismo cuando tenga un mal día). Personalmente, prefiero evitar la proliferación de líneas en blanco, esforzándome para que las funciones sean concisas y con una única responsabilidad.

Para eliminar líneas en blanco, no solo está el recurso de extraer funciones, en ocasiones se recurre a la técnica *inline variable* (reemplazar la declaración de variable por su valor), entre otras técnicas de refactor. Antes:

Java

```java
public void toggleCamera() {
    Set<Integer> availableCameraLensFacing = getAvailableCameraLensFacing();

    if (availableCameraLensFacing.isEmpty()) {
        return;
    }
    /*...*/
}
```

Después:

Java

```java
public void toggleCamera() {
    if (getAvailableCameraLensFacing().isEmpty()) {
        return;
    }
    /*...*/
}
```

Alternativa con extracción de método:

Java

```java
public void toggleCamera() {
    if (hasNoCameraLensAvailable()) {
        return;
    }
    /*...*/
}
private boolean hasNoCameraLensAvailable(){
    return getAvailableCameraLensFacing().isEmpty();
}
```

En el libro utilizo una línea de comentario con puntos suspensivos para expresar que en el código real habría más líneas (las he dejado fuera para acortar el listado). Desde ahora, usaré esta marca para denotar que ahí se ubican líneas irrelevantes para el caso que estudiamos:

Java

```java
/* ... */
```

Dicho todo esto, me gustaría señalar que existen ciertos usos de la línea en blanco, que están muy extendidos y que son recomendables. Por ejemplo, tras una cláusula guarda, es típico dejar una línea en blanco, de hecho, la línea en blanco es una ayuda visual para cualquier punto de salida de función:

Java

```java
public int sumNumbers(Expression expression){
    if (expression == null){
        return 0;
    }

    /*...*/
}
```

En el código de los test, también es muy típico separar las tres partes del test con líneas en blanco (*arrange, act, assert*).

No pretendo negar el uso de la línea en blanco de manera dogmática, sino poner de manifiesto que más líneas en blanco no se traducen en un código más claro, ni legible. Al contrario, el abuso de la línea en blanco se puede llegar a convertir en una excusa para no crear abstracciones, ni separar responsabilidades, sin reducir el acoplamiento. El concepto de limpieza es muy subjetivo, no se presta a ser razonado, para mí más aire en el código

no significa mayor limpieza.

Igualar el nivel de abstracción dentro de cada bloque

Kent Beck habla de la simetría como principio para diseñar código más fácil de entender. Uno de los patrones que implementan este concepto de simetría consiste en igualar el nivel de abstracción de las líneas de código que van juntas en un bloque. Ejemplo de disparidad en el nivel de abstracción:

Java

```java
public static URLClassLoader buildUserCodeClassLoader(
        List<URL> jars, List<URL> classpaths,
        ClassLoader parent, Configuration configuration) {
    URL[] urls = new URL[jars.size() + classpaths.size()];
    for (int i = 0; i < jars.size(); i++) {
        urls[i] = jars.get(i);
    }
    for (int i = 0; i < classpaths.size(); i++) {
        urls[i + jars.size()] = classpaths.get(i);
    }
    return FlinkUserCodeClassLoaders.create(
        FlinkUserCodeClassLoaders.ResolveOrder.fromString(
            configuration.getString(CoreOptions.CLASSLOADER_RESOLVE_ORDER)),
        urls, parent);
}
```

Las primeras líneas de código tienen un nivel de abstracción bajo, es decir, cercano a la máquina. Operan con tipos integrados, estructuras de datos básicas y utilizan sentencias de control de flujo elementales. Para entender el código tengo que computar en mi cabeza. El resto del código consiste en llamadas a funciones creadas por alguien de nuestro equipo o del *framework*, las cuales tienen un nivel de abstracción mayor (cercano al humano). Si igualamos el nivel de abstracción de todas las líneas, al más alto del bloque, conseguimos un código que se entiende sin necesidad de computar mentalmente:

Java

```java
public static URLClassLoader buildUserCodeClassLoader(
        List<URL> jars, List<URL> classpaths,
        ClassLoader parent, Configuration configuration) {
    return FlinkUserCodeClassLoaders.create(
        FlinkUserCodeClassLoaders.ResolveOrder.fromString(
            configuration.getString(CoreOptions.CLASSLOADER_RESOLVE_ORDER)),
        getAllUrlsFrom(jars, classpaths),  // <--- "extract method" aplicado
```

```
        parent);
}
private URL[] getAllURlsFrom(List<URL> jars, List<URL> classpaths){
    /*...*/
}
```

Hemos reducido la carga cognitiva requerida para entender el método. Aún podríamos mejorar el código, por ejemplo, reduciendo el número de parámetros y quizás convirtiéndolo en un método de instancia en lugar de ser *estático*. Pese a que necesitaría más contexto para tomar estas decisiones (sobre todo la segunda), hagamos el cambio para que sirva como ejemplo. Como regla general, no conviene que un método o función tenga más de dos o tres parámetros (cuantos menos mejor):

Java

```
public URLClassLoader buildUserCodeClassLoader(List<URL> urls,
                    ClassLoader parent, Configuration configuration) {
    return FlinkUserCodeClassLoaders.create(
              resolveOrder(configuration)), toArray(urls), parent);
}

private ResolveOrder resolveOrder(Configuration configuration){
    FlinkUserCodeClassLoaders.ResolveOrder.fromString(
            configuration.getString(CoreOptions.CLASSLOADER_RESOLVE_ORDER);
}

private URL[] toArray(){ /* ... */ }
```

Limitar el uso del modificador *static* o similares

Cuando trabajamos en el código vertical, el que implementa las reglas de negocio, no hay motivos para utilizar la palabra clave *static* o cualquiera que sea la alternativa en el lenguaje que utilicemos. Tampoco hay motivo para utilizar *singletons*. Ambas herramientas pueden tener sentido dentro de librerías, de *frameworks* y de código que se ocupa de resolver requisitos no funcionales, mientras que suelen ser contraproducentes cuando aparecen en código que se ocupa de los requisitos funcionales. En lenguajes como Java o C#, los métodos estáticos no pertenecen a las instancias de las clases, por lo que no tienen acceso a las variables de instancia, es decir, que no forman parte de los objetos. Si quieres programar en Java/C# como si fuera Pascal o Cobol, solo tienes que añadir *static* a todos los métodos que escribas. Por otra parte, el uso de artefactos estáticos (variables, métodos, clases…), reduce el grado de testabilidad del código, ya que resulta muy complejo suplantarlos para

realizar simulaciones en los test. Como no hay ningún mecanismo de indirección entre las dos partes, se incrementa considerablemente el grado de acoplamiento.

La combinación de palabras *public static* es la causante de todos estos problemas, sin embargo, hay un beneficio interesante en la combinación *private static*. Cuando un método de una clase es privado y a la vez estático, estamos diseñando funciones que no tienen dependencias (siempre que no accedan a variables estáticas externas), funciones que son autocontenidas. El modificador *static* sirve incluso para chequear rápidamente si un método accede a variables de instancia que no debería (se puede poner y quitar para hacer la prueba). Como de costumbre, el problema de usar estáticos aun siendo privados es que otras personas del equipo que no sepan discernir entre estáticos públicos y privados, se sientan animadas a utilizar estáticos a granel. Procuro usarlos teniendo en cuenta el contexto del equipo con el que trabajo en cada momento.

El patrón *singleton* (en la mayoría de casos es un antipatrón), consiste en compartir una única instancia de una clase por toda la aplicación. Sufre de los mismos problemas que los estáticos, y, de hecho, se implementa típicamente utilizando estáticos o similares (en lenguajes como Kotlin existen mecanismos específicos para ello, como la declaración *object*). Típica implementación de *singleton* en C#:

C#

```
public sealed class State {
    public static State Instance {get; } = new State();
    /* ... */
    private Singleton(){}
    public void addItem(Item item){ /* ...*/ }

}
public class Cart {
    public void addItemToCart(Item item){
        State.Instance.addItem(item);
    }
}
```

En este ejemplo, la clase *Cart* accede directamente al *singleton State*. El carrito conoce más detalles sobre la gestión del estado de los que necesita (acoplamiento excesivo). Incluso aunque haya un único objeto de datos que represente parte del estado de la aplicación, y que necesite ser compartido por varios artefactos, es preferible que dichos artefactos desconozcan la singular naturaleza de dicho objeto. Es preferible que no accedan directamente al *singleton*, sino que les venga dado (por ejemplo, como argumento en un constructor). Hay varias formas de corregir los problemas del código anterior, esta podría ser una de

ellas:

Java

```java
public class ShoppingCartState {
    private static final List<Item> sharedItems = new ArrayList();

    public void addItem(Item item) {
        sharedItems.add(item);
    }

    public void removeItem(Item item) {
        sharedItems.remove(item);
    }

    public Item[] allItems() {
        return sharedItems.toArray(Item[]::new);
    }
}

public class GroceryCart {
    private final ShoppingCartState state;

    public GroceryController(ShoppingCartState state) {
        this.state = state;
    }

    public void addGrocery() {
        /*...*/
        state.addItem(new Item());
    }
    /*...*/
}

public class BooksCart {
    private final ShoppingCartState state;

    public BooksCart(ShoppingCartState state) {
        this.state = state;
    }

    public void addBook() {
        /*...*/
        state.addItem(new Item());
    }
    /*...*/
}
```

Las clases que reciben el estado como argumento en su constructor, desconocen que está siendo compartido por otras clases. Así lo demuestra el siguiente test, en el que una instancia de *ShoppingCartState* es compartida.

Java

```
public class StateTests {
    @Test
    public void the_state_is_shared_among_carts(){
        var state = new ShoppingCartState();
        var groceryCart = new GroceryCart(state);
        var booksCart = new BooksCart(state);
        groceryCart.addGrocery();
        booksCart.addBook();
        state.addItem(new Item());
        assertThat(state.allItems().length).isEqualTo(3);
    }
}
```

El método *allItems* de la clase que maneja el estado, devuelve un *array*, en lugar de una lista, para dejar claro que solamente dicha clase puede añadir o quitar elementos del carrito. Podríamos plantearnos devolver una lista de solo lectura:

Java

```
public List<Item> allItems() {
    return Collections.unmodifiableList(sharedItems);
}
```

El problema es que si alguien por accidente intenta añadir o quitar elementos de la colección devuelta, se producirá una excepción en tiempo de ejecución que posiblemente le sorprenda. Teniendo en cuenta que el propósito del método es que se puedan leer los elementos (iterar), devolver una estructura que ya posee una interfaz que no deja añadir ni quitar elementos (como es un *array*), es menos arriesgado y más explícito. Habrá otras veces en las que tenga más sentido recurrir a *unmodifiableList* o incluso crear nuestro propio tipo inmutable.

Limitar la accesibilidad a métodos y clases

Cada vez que elegimos el modificador de acceso *public* para una función/método/clase, adquirimos un compromiso. Abrir el acceso a estos elementos es como establecer un contrato entre proveedor y consumidor, que además no tiene fecha de caducidad. Los contratos tienen la ventaja de que establecen un acuerdo de colaboración preciso entre las partes y la desventaja de que limitan la libertad para cambiar esos acuerdos en el futuro. Los métodos públicos de una clase componen su API pública (*Application Programming Interface*),

su interfaz. Desde el momento en que hay otro punto del código que consume esa API, se crea una dependencia y un acoplamiento que potencialmente puede suponer un lastre. El código legado que resulta más complicado de cambiar es el que tiene interdependencias entre artefactos. Una función que no tiene dependencia de nada exterior es más fácil de cambiar, en parte gracias a la ausencia de dependencias. Las dependencias tienen el peligro de provocar el típico efecto dominó; modifico el artefacto *A* y se rompe el artefacto *K*. Existen múltiples formas de dependencia más allá del modificador de acceso *public*. Lo cierto es que limitar la visibilidad o accesibilidad de los elementos no cuesta nada, basta con poner intención al diseñar las APIs. Podemos aplicar este principio tanto a una API dentro de nuestro código, como a una API HTTP o a cualquier otro protocolo de comunicación. No expongas lo que no quieres que otros sepan, porque tarde o temprano vendrán los *hacks* (apaños o ñapas). Si diseñásemos cada clase/módulo estableciendo su contrato deliberadamente, en muchos escenarios evitaríamos recurrir a complejas arquitecturas basadas en microservicios.

Como regla general, defino todos los elementos como privados y les subo el nivel de visibilidad conforme se requiere. No defino métodos públicos solo para poderlos atacar directamente con test unitarios, sino que los métodos privados son ejercitados por los test de manera indirecta, atacando a métodos que de verdad tiene sentido que sean públicos por diseño.

Es importante conocer los modificadores de acceso del lenguaje de programación que utilizas y sus sutilezas. Por ejemplo, en Java existen los siguientes modificadores: privado, por defecto, protegido y público ¿Sabrías decir cómo afecta cada uno de ellos a la propia clase, al paquete, a otros paquetes, a las subclases, etc.?

Mejor composición que herencia

La herencia es uno de los conceptos clásicos que se utiliza para explicar la orientación a objetos, cuando realmente debería explicarse el paso de mensajes. Para el gran Alan Kay, uno de los padres de la programación orientada a objetos, la idea clave es el paso de mensajes entre objetos como si fueran computadoras conectadas en red o células que interaccionan entre sí. No se trata de estructuras de datos dentro de clases desprovistas de métodos (los *getters* y *setters* no cuentan como métodos en el sentido de que no realizan ninguna operación sobre los datos). Los datos y los métodos que operan sobre los datos, deben estar juntos de manera que cada clase sea una cápsula con sus partículas cohesionadas.

Al igual que pasaba con los elementos estáticos, dentro del código que implementa las reglas de negocio, son contadas las ocasiones en las que tiene sentido recurrir a la herencia. En esta capa del sistema, los objetos son representaciones conceptuales del dominio y es inusual que un dominio concreto contenga jerarquías de conceptos. Quizás por eso sea tan frecuente recurrir a las figuras geométricas cuando se explica la herencia. Nunca en mi vida he construido un *software* que trabajase con formas geométricas. La herencia tiene más sentido en las capas del sistema relacionadas con el mecanismo de entrega, es decir, con la plataforma tecnológica donde se ejecuta el *software*. En esos puntos es más común recurrir a la herencia como mecanismo para evitar duplicidad, de hecho, si abrimos el código de algún *framework web* puede que encontremos buenos ejemplos de ello. Se aprende mucho leyendo código fuente de proyectos de *software* libre.

Lo cierto es que siempre podemos transformar una composición en herencia y viceversa, de hecho, algunos IDEs pueden realizar la conversión automáticamente. El ejemplo del estado y los carritos podría implementarse con herencia (fíjate que anteriormente se usaba composición con *ShoppingCartState*) :

Java

```
public class ShoppingCart {
    private static final List<Item> sharedItems = new ArrayList();
    public void addItem(Item item) {
        sharedItems.add(item);
    }
    public void removeItem(Item item) {
        sharedItems.remove(item);
    }
}
public class GroceryCart extends ShoppingCart {
}
public class BooksCart extends ShoppingCart {
}
```

Quizás nos permita ahorrar algunas líneas de código, pero el precio a pagar es demasiado alto. Ahora tenemos un acoplamiento más fuerte entre las tres clases, hay más riesgo de que si tocamos una de ellas, rompamos las otras. Al mismo tiempo aumenta la probabilidad de que cuando modifico una de las tres clases, me vea obligado a modificar las otras ¿Qué sucede si *GroceryCart* y *BooksCart* tomasen caminos diferentes en cuanto a la gestión del estado?, ¿qué tan fácil sería adaptar el *software* a esos cambios? Otra desventaja tiene que ver con la expresividad de los nombres; el código que utilizaba composición tenía nombres concretos ceñidos al contexto como *addGrocery* o *addBook*, respectivamente, mientras que ahora por compartir la interfaz se necesita un nombre más genérico como

addItem.

En líneas generales, empleo la herencia como último remedio. Es preferible probar primero a componer que a heredar. Curiosamente, las veces que termino usando herencia suele ser como consecuencia de aplicar refactorización para eliminar duplicidad y no suele suceder en código de negocio, sino en código transversal.

Si sueles pensar en herencia al modelar el dominio, no te costará mucho cambiar a composición; el truco está en mover a otras clases aquellos métodos que serían implementados por clases hijas (típicamente, son los métodos protegidos y los abstractos), y conectar las clases mediante inyección de dependencias. Veamos un ejemplo. Primero con herencia:

Java

```
public abstract class Escaparate {

    public List<ArticuloALaVenta> exponerArticulos(Temporada temporada){
        var articulos = listarCatalogo(temporada);
        var nivelesDeStock = leerNivelDeStockDe(articulos);
        var articulosExistentes = descartarArticulosAgotados(articulos,
            nivelesDeStock);
        return prepararOferta(articulosExistentes, nivelesDeStock);
    }

    protected abstract List<ArticuloCatalogado> listarCatalogo(Temporada
        temporada);

    protected abstract List<NivelDeStock> leerNivelDeStockDe(List<
        ArticuloCatalogado> articulos);

    private List<ArticuloCatalogado> descartarArticulosAgotados(List<
        ArticuloCatalogado> articulos, List<NivelDeStock> nivelesDeStock) {
        /*...*/
    }

    private List<ArticuloALaVenta> prepararOferta(List<ArticuloCatalogado>
        articulos, List<NivelDeStock> nivelesDeStock) {
        /*...*/
    }
}
```

Los dos métodos abstractos pasarían a ser métodos de otras clases que componen a esta:

Java

```
public class Escaparate {
    private final Catalogo catalogo;
    private final Stock stock;
```

```
    public Escaparate(Catalogo catalogo, Stock stock) {
        this.catalogo = catalogo;
        this.stock = stock;
    }

    public List<ArticuloALaVenta> exponerArticulos(Temporada temporada){
        var articulos = catalogo.listarArticulosDe(temporada);
        var nivelesDeStock = stock.leerNivelDe(articulos);
        var articulosExistentes = descartarArticulosAgotados(articulos,
            nivelesDeStock);
        return prepararOferta(articulosExistentes, nivelesDeStock);
    }

    private List<ArticuloCatalogado> descartarArticulosAgotados(List<
        ArticuloCatalogado> articulos, List<NivelDeStock> nivelesDeStock) {
        /*...*/
    }

    private List<ArticuloALaVenta> prepararOferta(List<ArticuloCatalogado>
        articulos, List<NivelDeStock> nivelesDeStock) {
        /*...*/
    }
}
```

Cláusulas guarda y simetría de bloques

En lenguajes con gestión de memoria automática y basados en el paradigma orientado a objetos, o el funcional, no supone ningún problema que una función tenga varios puntos de salida (varias sentencias *return*). De hecho, es recomendable salir de la función lo antes posible, tan pronto como hallemos un resultado válido, para reducir la probabilidad de error que podría ocurrir en la ejecución de las líneas restantes. La línea de código que no falla es la que no se ejecuta. Por ejemplo, cuando el valor de alguno de los argumentos supone un callejón sin salida, podemos concluir de inmediato:

Java

```
public int sumNumbers(String expression){
    if (expression == null or expression.isBlank()){
        return 0;
    }

    /* ... */
}
```

Esta comprobación inicial se conoce como cláusula guarda y se encarga típicamente de

los casos límite, cuando no hay nada más que se pueda hacer:

Java

```java
public float divide(float divisor){
    if (divisor == 0){
        throw new IllegalArgumentException("Divisor can't be zero");
    }

    /* ... */
}
```

El primer beneficio de las cláusulas guarda, es que reducen la probabilidad de errores; y el segundo, es que reducen el nivel de indentación del código, al menos en uno. Nótese que la comprobación contraria introduciría un nivel de indentación hacia adentro:

Java

```java
public float divide(float divisor){
    if (divisor != 0){
       /* ... */
    } else {
       throw new IllegalArgumentException("Divisor can't be zero");
    }
}
```

En los IDEs modernos como IntelliJ, la propia herramienta tiene la capacidad de invertir la condición por nosotros de manera automática. Simplemente, hay que poner el cursor en la palabra *if* y aplicar el atajo de teclado que abre las sugerencias del IDE, o bien hacer *click* en la típica bombillita que se dibuja en pantalla ¡*Voilà*! Mágicamente se invertirá la condición y se recolocarán los bloques de código.

Ahora bien, cuando tenemos un martillo parece que todo son clavos, por eso quiero destacar que no todas las condiciones son buenas candidatas para ser cláusulas guarda. No todas las estructuras condicionales con sentencias de salida de función son cláusulas guarda. El objetivo de las cláusulas es señalizar explícitamente situaciones terminales, no se trata de «eliminar la palabra *else*» sin más.

Para expresar condiciones en las que hay dos caminos válidos, mutuamente excluyentes y necesarios para el negocio, es más explícito un código con estructura simétrica, con sentencias *if-else* o con el operador ternario (para expresiones cortas):

Java

```
public Discount calculateDiscountFor(User user){
    Discount discount;
    if (user.hasStandardAccount()){
        discount = Discount.percentageOf(20);
    }
    else if (user.hasPremiumAccount()){
        discount = Discount.percentageOf(50);
    }
    else {
        discount = noDiscount();
    }
    return discount;
}
```

La estructura simétrica pone de manifiesto que hay tres caminos alternativos válidos y mutuamente excluyentes. El código se complica cuando hay combinaciones condicionales y anidamiento, ya que nos exigen computar mentalmente mientras leemos el código:

Java

```
public Discount calculateDiscountFor(User user){
    if (user.hasStandardAccount()){
        if (user.isEmployee()){
           return Discount.percentageOf(20);
        }
    }
    if (user.hasPremiumAccount()){
        return Discount.percentageOf(50);
    }
    return noDiscount();
}
```

La sentencia *return* de la cuarta línea, está implícitamente representando que, sin ser empleado, no hay descuento para las cuentas estándar. Las condiciones anidadas son duras de entender, sobre todo cuando hay ramas asimétricas. Como norma general, evitaremos tener más un nivel de anidamiento de condicionales:

Java

```
public Discount calculateDiscountFor(User user){
    Discount discount;
    if (user.hasStandardAccount() && user.isEmployee()){
        discount = Discount.percentageOf(20);
    }
    else if (user.hasPremiumAccount()){
        discount = Discount.percentageOf(50);
    }
    else {
        discount = noDiscount();
```

```
    }
    return discount;
}
```

La simetría mediante sentencias *if-else* no deja lugar a dudas. Aunque no sea estrictamente necesaria, nos ayuda a aclarar las ideas. Veamos otro ejemplo:

Java

```
public Discount calculateDiscountFor(User user){
    if (user.hasStandardAccount() && user.isEmployee()){
        return Discount.percentageOf(50);
    }
    if (user.hasStadardAccount()){
       return Discount.percentageOf(10);
    }
    if (!user.hasStandardAccount()){
       return Discount.percentageOf(5);
    }
    return noDiscount();
}
```

Este código presenta varios problemas. Primero, que cuesta entenderlo por su falso aspecto simétrico. Segundo, que el orden de los bloques es interdependiente, es decir, que existe un sutil acoplamiento indeseado entre ellos. Si alguien por error coloca el segundo bloque *if* en primer lugar, el comportamiento del método cambia por completo, introduciendo un defecto o *bug*. Este tipo de accidentes que consisten en mover un bloque (cuyo orden importa) por encima o por debajo de otro, ocurren porque el acoplamiento está camuflado. El tercer problema es que el resultado *noDiscount()* no se da nunca, aunque pareciera ser el valor retornado por defecto. Por suerte, hoy en día los IDEs nos dan algunos avisos cuando encuentran código como este, aunque no todo el mundo sabe reconocerlos. Podemos resolver casi todos los problemas siendo explícitos:

Java

```
public Discount calculateDiscountFor(User user){
    Discount discount;
    if (user.hasStandardAccount() && user.isEmployee()){
        discount = Discount.percentageOf(50);
    }
    else if (user.hasStandardAccount()) {
        discount = Discount.percentageOf(10);
    }
    else {
       discount = Discount.percentageOf(5);
    }
```

```
    return discount;
}
```

Ahora es menos probable que se intercambien bloques, porque son todos parte de una misma estructura y el código es más evidente. Aun así, un intercambio en el orden de las condiciones volvería a introducir un *bug*. Otra alternativa clara y simétrica, pese a tener un nivel más de indentación, es la siguiente:

Java

```
public Discount calculateDiscountFor(User user){
    Discount discount;
    if (user.hasStandardAccount()){
        if (user.isEmployee()){
            discount = Discount.percentageOf(50);
        }
        else {
            discount = Discount.percentageOf(10);
        }
    }
    else {
       discount = Discount.percentageOf(5);
    }
    return discount;
}
```

Ahora ya no existe la tentación de intercambiar el orden de los bloques. Desde aquí, no quedaría nada mal insertar un operador ternario:

Java

```
public Discount calculateDiscountFor(User user){
    Discount discount;
    if (user.hasStandardAccount()){
        discount = user.isEmployee() ?
            Discount.percentageOf(50) : Discount.percentageOf(10);
    }
    else {
       discount = Discount.percentageOf(5);
    }
    return discount;
}
```

Desde este punto sí que podemos refactorizar, e incluso eliminar el bloque *else*, sin perder expresividad:

Java

```
public Discount calculateDiscountFor(User user){
    if (user.hasStandardAccount()){
       return user.isEmployee() ?
            Discount.percentageOf(50) : Discount.percentageOf(10);
    }
    return Discount.percentageOf(5);
}
```

Lo que pretenden ilustrar estos ejemplos es que programar es mucho más que aplicar reglas matemáticas, más bien consiste en buscar un buen balance entre expresividad, carga cognitiva, concisión y reducción de la probabilidad del error humano (entre otras cuestiones). Si bien es cierto que como pauta evito los operadores ternarios y la anidación de bloques de código, no significa que los tenga prohibidos o que no sean la mejor solución en algunos casos. Hemos visto que los bloques *if-else* no son malos ni buenos, sino que son herramientas que el lenguaje nos ofrece para expresarnos. Cuanto mejor sepamos expresar los requisitos del negocio en código, más sostenible será el desarrollo del *software*.

Reducir al máximo el ámbito

Cuanto más local sea el ámbito de los diferentes elementos de un artefacto, más cohesión tendrá. Esto se traduce en que, a la hora de hacer cambios, encontraremos las piezas pronto por estar cerca las unas de las otras. También significa que el impacto de los cambios será más acotado, lo cual reduce la posibilidad de efectos secundarios indeseados en otras partes del código (el efecto dominó o reacción en cadena). Reducir al máximo el ámbito de las variables, constantes, métodos, clases, etc., producirá código más fácil de entender y menos peligroso de modificar. Las consecuencias serán locales.

En lenguajes modernos con gestión automática de la memoria, el mejor lugar para definir una variable o una constante, es justo donde se va a usar:

Java

```
class Parser {
    public List<SearchTerm> parseSearchTermsFrom(String text){
        final var separators = ",|\\s";
        return Arrays.stream(text.split(separators))
                .map(SearchTerm::new)
                .filter(SearchTerm::isKeyword)
                .collect(Collectors.toList());
    }
}
```

```
class SearchTerm {
    private String term;

    public SearchTerm(String term) {
        this.term = term;
    }

    public boolean isKeyword() {
        if (term.isBlank()){
            return false;
        }
        final var stopWords = Arrays.asList("from", "to", "the", "a", "an");
        return !stopWords.contains(comparableTerm());
    }

    private String comparableTerm() {
        return term.toLowerCase().trim();
    }
}
```

Las dos constantes *separators* y *stopWords*, en las clases *Parser* y *SearchTerm*, respectivamente, están definidas en el lugar más cercano a su utilización. Esto será chocante para quien siga la vieja costumbre de definir las constantes lo más arriba posible en una clase o incluso en un paquete. Los inconvenientes de moverlas arriba como campos de la clase, son varios:

- Quedan expuestas a los demás métodos con el consecuente riesgo de acoples indeseados entre ellos. Si ningún otro método hace uso de esa constante, lo mejor es que no sepan ni que existe.
- Los métodos que albergaban las constantes locales pierden cohesión, porque pasan a depender de constantes que están afuera.
- La persona que lee el código empieza de arriba a abajo y se encuentra con una constante que no sabe para qué se usa. Esto, por un lado, le genera ruido o duda, y, por otro, le obliga a memorizar para entender luego el bloque de código que la usa.
- Es más costoso aplicar *refactorings* como *extraer método* o *extraer clase*. Cuando hay partes sueltas, la refactorización se tiene que hacer manualmente o dando muchos pasos intermedios con el IDE, mientras que si todo está en un mismo lugar, el propio IDE es capaz de realizarla automáticamente, garantizando que todo sigue funcionando.

Espera un momento, ¿y si la lista de palabras vacías (*stop words*) se pasa por constructor o *setter* para que pueda ser configurable? (la mentalidad de programar el *software* para que sea configurable y no haya que tocarlo nunca más). Esto introduciría otro problema,

un acoplamiento a nivel de algoritmo (*connascence* de algoritmo), puesto que obliga al código consumidor a conocer el detalle de implementación (el tema de las palabras vacías o *stopwords*). Para reducir el acoplamiento, debemos trabajar con cada artefacto como si desconociéramos todos los detalles de implementación ¿Y qué tal si movemos las constantes al paquete/clase *Utils*? Pues esto introduce aún más problemas, como el síndrome de Diógenes, porque todo lo que no encuentra su lugar o es de dudosa utilidad termina amontonado ahí.

Lenguajes como Java, C#, TypeScript o Kotlin (entre otros), implementan ámbito a nivel de bloque (*block-scoping*), es decir, si una variable se define dentro de un bloque, solo es visible ahí. Su ciclo de vida está contenido en el del bloque, siendo eliminada por el recolector de basura al terminar su ejecución. El inicio y fin de bloque suele estar delimitado por las llaves ({ y }). Existen palabras reservadas de los lenguajes que añaden azúcar sintáctico a la gestión del ciclo de vida de las variables, como *const* y *let* en TypeScript, *using* en C# o *try()* en Java:

C#

```
UnicodeEncoding uniencoding = new UnicodeEncoding();
byte[] result = uniencoding.GetBytes(textField);
using (var fileStream = File.Open(filename, FileMode.OpenOrCreate))
{
    fileStream.Seek(0, SeekOrigin.End);
    await fileStream.WriteAsync(result, 0, result.Length);
}
```

El tipo *FileStream* accede a recursos que están fuera del *framework .Net* (ficheros del disco), y, por tanto, se requiere una llamada específica a «cerrar fichero» al terminar de usarlo. En este ejemplo, la palabra reservada *using*, se encarga de que la llamada para liberar el recurso (*Dispose*) se haga automáticamente, tanto si el bloque termina normalmente, como si se produce una excepción no controlada.

Si en lugar de programar así, definimos una variable de tipo *FileStream* arriba del todo de la clase, estaremos obligados a liberar los recursos manualmente en todas las partes de la clase que los utilicen. Además, tendríamos que añadir bloques *try-finally* para estar seguros que si ocurre una excepción no controlada, liberamos el recurso:

C#

```
var fileStream = File.Open(filename, FileMode.OpenOrCreate);
try {
    fileStream.Seek(0, SeekOrigin.End);
```

```
    await fileStream.Write(result, 0, result.Length);
}
finally {
    fileStream?.Dispose();
}
```

Todas estas razones nos invitan a declarar y utilizar variables/constantes en el ámbito más local posible. Sácale partido al ámbito a nivel de bloque en los lenguajes que lo implementan.

Las versiones de JavaScript previas a la aparición de las palabras reservadas *const* y *let*, solo implementaban ámbito a nivel de función, aunque la sintaxis de los bloques era la típica de llaves (copiada de Java como estrategia de *marketing*); esto nos dio muchos quebraderos de cabeza, siendo fuente de numerosos *bugs*.

Mantener los constructores simples

Imagina que la siguiente línea de código lanza una excepción al ejecutarse:

Java

```
Credentials credentials = new Credentials("username", "password");
```

Resultaría sorprendente, porque tenemos muy asumido que los constructores están desprovistos de funcionalidad. El único efecto secundario esperado de un constructor es reservar memoria para el objeto, nada más. Sería asombroso que una llamada a un constructor alterase el estado del sistema escribiendo datos en disco, mutando variables globales o lanzando excepciones.

La única operación que debe realizar un constructor es guardar los argumentos que recibe:

Java

```
class Credentials {
    private final String username;
    private final String password;
    public Credentials(String username, String password){
        this.username = username;
        this.password = password;
    }
}
```

Todos los datos que un objeto requiere para ser consistente deben ser enviados al constructor de su clase. De esta manera, sabemos que el objeto está completo desde el momento de su construcción. Utilizar *setters* para los datos no es recomendable, porque perdemos esta consistencia y porque el objeto pierde el control sobre su propio estado (encapsulación). Además, la falta de simetría resultaría muy confusa:

Java

```java
class Credentials {
    private final String username;
    private final String password;
    public Credentials(String username){
        this.username = username;
    }
    public void setPassword(String password){
        this.password = password;
    }
}
```

Cualquier persona que lea la clase se preguntará por qué uno de los datos se pasa por constructor y el otro por *setter*. La ausencia de simetría puede dar la impresión de que hubo intencionalidad en ello, lo cual da respeto a la hora de hacer cambios.

Aunque la mayoría de lenguajes orientados a objetos permiten la sobrecarga de constructores, no es una práctica que recomiende. Como son métodos con el mismo nombre, la persona que los va a usar puede tener dificultades para determinar cuál de ellos invocar («¿cuál será el bueno?»). No obstante, puede haber sobrecarga sin confusión si los constructores se limitan a ofrecer alternativas para la comodidad de los consumidores:

Java

```java
class Group {
    private List<User> users;
    public Group(List<User> users){ /*...*/}
    public Group(User[] users){ /*...*/}
    /*...*/
}
class Credentials {
    private final String username;
    private final String password;
    public Credentials(String username, String password){/*...*/}
    public Credentials(Username username, Password password){/*...*/}
}
```

Las alternativas consisten en utilizar diferentes tipos de datos puesto que todo lo demás

es igual, incluyendo el número y nombre de los parámetros.

Entonces, si otro tipo de sobrecarga no es aconsejable, ¿cómo puedo construir objetos con diferentes características partiendo de una misma clase?, ¿cómo puedo hacer validaciones durante la construcción de un objeto? La respuesta está en la siguiente sección, los constructores con nombre.

Los constructores tienen otra implicación sutil que a menudo se pasa por alto. Se trata del acoplamiento que producen cuando se utiliza la herencia. No todo el mundo entiende que, a diferencia de los métodos públicos y protegidos de una clase, los constructores no se heredan. El acoplamiento reside en la restricción de que el constructor de una clase descendiente está obligado a invocar al de su clase antecesora inmediata. Para entender esto, primero debemos saber que todas las clases tienen implícitamente un constructor por defecto (sin argumentos), que invoca al constructor de su clase base. Las siguientes dos definiciones son totalmente equivalentes:

Java

```
public class SomeClass {}
```

Java

```
public class SomeClass {
    public SomeClass(){
        super();
    }
}
```

Si no definimos ningún constructor para una clase, entonces el propio compilador añade esta implementación por defecto. Otro ejemplo:

Java

```
class Parent {
    protected string name;
    public Parent(){
        name = "foo"
    }
    public string name(){
        return name;
    }
}
class Child extends Parent {} // <---- sin constructor

public class Tests {
```

```
    @Test
    public void default_constructor_is_predefined(){
        var child = new Child();
        assertThat(child.name()).isEqualTo("foo");
    }
}
```

Desde el momento que defino un constructor con uno o más argumentos, el compilador ya no añade ese constructor por defecto. Sin embargo, las clases hijas siguen teniendo la obligación de invocar al constructor de su clase base. El compilador no puede hacer la invocación por sí mismo, porque no tiene forma de saber cuál es la manera correcta. Por eso, el programa no compila hasta que no hacemos la llamada explícita al constructor base:

Java

```
class Parent {
    protected int counter = 0;
    public Parent(int initialCount){
        counter = initialCount;
    }
}
class Child extends Parent {
    public Child(int initialCount) {
        super(initialCount); // <-- llama a constructor base
    }
}
```

Si la clase *Child* no realiza la llamada al constructor de *Parent* en su constructor, el código no compila. Además, debe de ser la primera línea del constructor, si estuviera en segundo lugar, tampoco compilaría. No solo es así en Java, sino también en todos los lenguajes que conozco y que soportan orientación a objetos, si bien la sintaxis puede cambiar ligeramente entre lenguajes. Por ejemplo, en C# sería así:

C#

```
class Parent {
    protected int counter = 0;
    public Parent(int initialCount){
        counter = initialCount;
    }
}
class Child : Parent {
    public Child(int initialCount): base(initialCount) {}
}
```

Dado que los constructores no se heredan, este mecanismo asegura que la inicialización del objeto es consistente en toda la jerarquía. Cuando las clases de la jerarquía difieren en cuanto al conjunto de datos que necesitan, esta restricción expone visiblemente la inconsistencia:

Java

```
class Credentials {
    /*...*/
    public Credentials(String username, String password){/*...*/}
}
class AnonymousCredentials extends Credentials {
    public AnonymousCredentials(){
        super("anonymous", "");
    }
}
```

El ejemplo muestra como la clase heredera se ve forzada a inventarse valores para invocar al constructor de su precursora. Esta puede ser una pista que nos ayude a reflexionar sobre el uso del constructor o incluso de la herencia. Las inconsistencias son indicadores sobre la aptitud del diseño del *software*.

Constructores con nombre

Una forma de especificar las cualidades diferenciales del objeto que se va a crear, es darle nombre al método o función que lo construye:

Java

```
Credentials credentials = Credentials.createWithStrongRandomPasswordFor("
    someUsername");
```

Es evidente que no hace falta enviar la contraseña como argumento, sino que se infiere enseguida que el método de creación se encarga. La implementación podría ser algo como esto:

Java

```
class Credentials {
   /*...*/
   public static Credentials createWithStrongRandomPasswordFor(String
       username){
```

```
      var password = generateStrongRandomPasswordFor(username);
      return new Credentials(username, password);
   }
   private Credentials(String username, String password){/*...*/}
   /*...*/
}
```

El constructor de la clase queda privado, solamente puede ser usado dentro de ella para no sembrar la duda en los consumidores, sobre cuál es la forma adecuada de crear nuevas instancias. Este tipo de constructores con nombre, se conocen también como métodos de factoría. En los patrones del *Gang of Four*, existe el *Factory Method*, pero su implementación es más compleja, porque su objetivo es más bien desacoplar tipos. Aquí, simplemente uso métodos de factoría con la intención de aportar claridad y evitar sorpresas, a la vez que mantengo el código lo más simple posible. La mayoría de IDEs modernos son capaces de transformar un constructor en un método de factoría, cambiando automáticamente todas las llamadas de los consumidores para adaptarlas a la nueva API. En el menú *Refactoring* de tu IDE favorito, busca algo como *Replace constructor with factory method*.

Cuando leo la llamada a un constructor con nombre, puedo concebir que realice validaciones, normalizaciones o incluso que lance excepciones. Al fin y al cabo, es un método como otro cualquiera:

Java

```
class EmailAddress {
   /*...*/
   public static EmailAddress parseFrom(String emailAddress){
      if (isNotValidAddress(emailAddress)){
          throw new IllegalArgumentException("Invalid email address:" +
              emailAddress);
      }
      return new EmailAddress(emailAddress.toLowerCase());
   }
   private EmailAddress(String emailAddress){/*...*/}
   /*...*/
}
```

También puedo expresar las variantes de inicialización de estado de un objeto mediante estos métodos de construcción:

Java

```
GameBoard board = GameBoard.createEmptyBoard();
GameBoard board = GameBoard.createBoardWithRandomCells();
GameBoard board = GameBoard.createBoardWith(livingCell(), deadCell());
```

Aprendí esta idea de recurrir a múltiples constructores con nombre en el libro de Corey Haines, *Understanding the Four Rules of Simple Design*. Es un libro muy instructivo para quien ya haya implementado un par de veces soluciones al juego de la vida de John H. Conway.

De acuerdo, y… ¿cómo podemos resolver el problema de no disponer de todos los datos que el objeto necesita al mismo tiempo?, ¿de qué formas se los podemos añadir después? Existen diversas soluciones que son más o menos adecuadas, según el contexto. Una de ellas consiste en posponer la construcción del objeto hasta el momento en que tenemos todos sus datos, mientras los vamos recopilando en otro lugar. Este sería el patrón *builder*:

Java

```
GameBoardBuilder builder = new GameBoardBuilder();
builder.withCells(livingCell());
/*...*/
builder.withDimensions(5,5);
/*...*/
GameBoard board = builder.buildBoard();
```

Otra solución aplicable a aquellos conceptos que representan valores, son los *Value Objects*. Aquí tenemos un ejemplo:

Java

```
Triangle regularTriangle = Triangle.equilateral(Dimensions.centimeters(20));
/*...*/
Triangle irregularTriangle = regularTriangle.increaseSizeOfSideA(10);
```

En un momento dado, necesitamos construir un polígono regular de tres lados, aunque más adelante en el tiempo resulta que necesitamos hacer más grande unos de los lados. La solución ha sido dotar al triángulo de la capacidad de construir otro triángulo, de manera que cada uno de los objetos está bien formado desde el momento de su construcción.

Control de flujo separado de la lógica de negocio

El paradigma de la programación imperativa consiste en especificar la secuencia ordenada de instrucciones que la máquina va a ejecutar. La mayoría de los lenguajes más extendi-

dos, incluidos los que soportan orientación a objetos, están basados en este paradigma. Por eso, quizás te resulte extraño que haya otros lenguajes donde no existen sentencias de control tipo bucle (*for, while,…*). Se trata de lenguajes que siguen un paradigma declarativo como Haskell, Lisp o Prolog. Los lenguajes declarativos, en concreto los que siguen un paradigma funcional, son más parecidos a las fórmulas matemáticas. Se centran en las reglas de negocio minimizando el ruido que producen las sentencias de control (condicionales, bucles, saltos, excepciones…). Para una persona que conoce las matemáticas y no conoce la programación, seguramente resulta más claro un estilo declarativo que uno imperativo. En lenguajes como Erlang, se utiliza además *pattern matching*, para que sea el propio sistema el que se encargue del control de flujo, ahorrándonos escribir condicionales:

Erlang

```
loop_through([Head | Tail]) ->
  io:format('~p~n', [Head]),
  loop_through(Tail);

loop_through([]) ->
  ok.
```

Este código recursivo imprime en pantalla los elementos de una lista. Para ello, define dos funciones con el mismo nombre. La primera será invocada cuando el argumento consista en una lista, con al menos un elemento en ella. Además de expresar la forma de su argumento como un patrón, hace una especie de desestructuración del mismo (parecida a JavaScript), dando nombre al primer elemento (*Head*) y al resto de la lista sin ese elemento (*Tail*). La segunda función será invocada cuando el argumento sea una lista vacía. La tercera línea del listado es una llamada a una de estas dos funciones. El propio sistema de ejecución de Erlang es quien decide a cual de las dos funciones invocar, según con qué patrón case el argumento, es decir, la condición la gestiona internamente Erlang.

Los lenguajes más usados hoy en día son multiparadigma. No es verdad que con Java podamos codificar de forma puramente funcional, pero sí podemos aprovechar muchos de los principios y herramientas de la programación funcional. Lo mismo pasa con C#, JavaScript, TypeScript, Ruby, Python…

El código se hace más fácil de leer cuando hacemos una separación entre control de flujo y lógica de negocio, técnica que podemos aplicar con cualquiera de estos lenguajes. Dado un código que mezcla el qué y el cómo, podemos aplicar refactorización de manera que haya un método que orquesta o coordina el flujo y otros que resuelven problemas específicos.

Ejemplo de código que mezcla flujo y «lógica»:

Java

```
private void addOrReplace(MutablePropertySources propertySources,
                          Map<String, Object> map) {
   MapPropertySource target = null;
   if (propertySources.contains(PROPERTY_SOURCE_NAME)) {
      PropertySource<?> source = propertySources.get(PROPERTY_SOURCE_NAME);
      if (source instanceof MapPropertySource) {
         target = (MapPropertySource) source;
         for (String key : map.keySet()) {
            if (!target.containsProperty(key)) {
               target.getSource().put(key, map.get(key));
            }
         }
     }
   }
   if (target == null) {
      target = new MapPropertySource(PROPERTY_SOURCE_NAME, map);
   }
   if (!propertySources.contains(PROPERTY_SOURCE_NAME)) {
      propertySources.addLast(target);
   }
}
```

Si extraemos métodos, se apreciará mejor el flujo:

Java

```
private void addOrReplace(MutablePropertySources propertySources,
                          Map<String, Object> map) {
   MapPropertySource target = null;
   if (propertySources.contains(PROPERTY_SOURCE_NAME)) {
      target = getMapPropertySourceFrom(propertySources);
      if (target != null){
         populateSourceFromMap(target, map);
      }
  }
  if (target == null) {
     target = new MapPropertySource(PROPERTY_SOURCE_NAME, map);
  }
  if (!propertySources.contains(PROPERTY_SOURCE_NAME)) {
     propertySources.addLast(target);
  }
}

private MapPropertySource getMapPropertySourceFrom(MutablePropertySources
    propertySources){
   PropertySource<?> source = propertySources.get(PROPERTY_SOURCE_NAME);
   if (source instanceof MapPropertySource) {
      return (MapPropertySource) source;
```

```
    }
    return null;
}

private void populateSourceFromMap(MapPropertySource source,
                                   Map<String, Object> map){
    for (String key : map.keySet()) {
        if (!source.containsProperty(key)) {
            source.getSource().put(key, map.get(key));
        }
    }
}
```

Generalmente, no es buena práctica devolver el valor *null*, al igual que no lo es recurrir al operador *instanceof*; no me convence el método *getMapPropertySourceFrom*. Es cierto que ha mejorado la legibilidad del método principal, así que parece ser un paso intermedio en la buena dirección. Los pasitos pequeños en la refactorización tienen menor riesgo de introducir *bugs* que los grandes. Además de avanzar en pequeños incrementos, procuro ser ordenado actuando por partes (típicamente con *commits* separados), para trazar y auditar mejor los cambios. Por ejemplo, aunque el tipo *MapPropertySource* sería más correcto en inglés como *PropertySourceMap* (el primero tiene forma de verbo, mientras que el segundo parece un sustantivo), decido no renombrarlo ahora, para no mezclar el refactor local del método con cambios en muchos otros ficheros.

Haber llegado a este punto hace que me dé cuenta de que el flujo podría ser simplificado, así que refactorizo la estructura y algunos nombres locales:

Java

```
private void addOrReplace(MutablePropertySources propertySources,
                          Map<String, Object> map) {
    if (propertySources.contains(PROPERTY_SOURCE_NAME)) {
        MapPropertySource source = getMapPropertySourceFrom(propertySources);
        if (source != null){
            populateSourceFromMap(source, map);
        }
     } else {
            propertySources.addLast(
                new MapPropertySource(PROPERTY_SOURCE_NAME, map));
     }
}
```

Esta simetría del bloque *if-else* es más sugerente que cuando parecía una especie de cláusula guarda. Gracias al cambio, puedo discernir que uno de los caminos es el de añadir, mientras que el otro es reemplazar, así que extraigo un nuevo método y elimino el que no

me convencía:

Java

```
private void addOrReplace(MutablePropertySources propertySources,
                          Map<String, Object> map) {
    if (propertySources.contains(PROPERTY_SOURCE_NAME)) {
        replace(propertySources, map);
    } else {
        add(propertySources, map);
    }
}

private void add(MutablePropertySources propertySources,
                 Map<String, Object> map){
   propertySources.addLast(
       new MapPropertySource(PROPERTY_SOURCE_NAME, map));
}

private void replace(MutablePropertySources propertySources,
                     Map<String, Object> map) {
   PropertySource<?> source = propertySources.get(PROPERTY_SOURCE_NAME);
   if (source instanceof MapPropertySource) {
      populateSourceFromMap((MapPropertySource) source, map);
   }
}

private void populateSourceFromMap(MapPropertySource source,
                                   Map<String, Object> map){
   for (String key : map.keySet()) {
      if (!source.containsProperty(key)) {
         source.getSource().put(key, map.get(key));
      }
   }
}
```

Ya no existe el método que devolvía *null* (no es conveniente devolver *null* en métodos públicos). El método de añadir (*add*) apenas aporta abstracción, la justa para igualar el nivel de la otra bifurcación, aportando simetría al conjunto. Aunque ya la línea se comprendía bien, el método revela los dos caminos inmediatamente, sin hacer pensar a quien lee y sin necesidad de comentarios en el código.

Hay un detalle que no me termina de cuadrar, una posible sorpresa que estamos dejando en el código: el método *replace* solo reemplaza cuando se cumple la condición que tiene dentro. No comprendo por qué el tipo de *source* podría ser distinto de *MapPropertySource*, ¿qué otra cosa podría ser si no? Si lo supiera, dejaría al menos un comentario en el código explicando la razón. Lo que sí puedo hacer es renombrar el método:

Java

```java
private void addOrReplace(MutablePropertySources propertySources,
                          Map<String, Object> map) {
    if (propertySources.contains(PROPERTY_SOURCE_NAME)) {
        tryToReplace(propertySources, map);
    } else {
        add(propertySources, map);
    }
}

private void tryToReplace(MutablePropertySources propertySources,
                          Map<String, Object> map) {
    PropertySource<?> source = propertySources.get(PROPERTY_SOURCE_NAME);
    // source could also be of type X, because ...
    if (source instanceof MapPropertySource) {
        populateSourceFromMap((MapPropertySource) source, map);
    }
}
```

Por simplicidad, lo deseable sería que el método siempre reemplazara, evitándonos la condición y el comentario, pero la realidad es que desconocemos el origen y la razón de este desafortunado acoplamiento. Quien escribió el código lo sabía y se quedó con ese conocimiento para sí, seguramente ignoraba que dejó una incógnita para los demás.

El objetivo de la refactorización es simplificar el código para facilitar su lectura. A su vez, facilitar la lectura tiene como objetivo reducir la probabilidad de equivocarnos (introducir *bugs*) al modificar código. También reduce el tiempo que tardamos en comprender el código al leerlo (y en depurarlo en caso de buscar fallos). Pasamos la mayor parte del tiempo leyendo y depurando código, por eso es tan rentable reducir los tiempos de asimilación. Quiero enfatizar que refactorizamos por motivos prácticos, para maximizar el retorno de la inversión, no para hacer florituras por amor al arte. Refactorizar no es como pintar un cuadro ni como escribir una canción, es un ejercicio de ingeniería. Si tuviéramos un artefacto con un código horrible que lleva años funcionando sin fallos, sin alterarse (porque no ha requerido cambios) y sin leerse, no tendría sentido refactorizarlo. No se trata de buscar el código más feo y de pelearnos con él para pasar el rato, sino de modificarlo cuando nos vemos obligados a dedicar demasiado tiempo a su lectura, a realizar modificaciones o a corregir fallos. Cuando hemos tropezado suficientes veces con el mismo peldaño o hemos tropezado una sola vez y nos hemos hecho daño, entonces quitamos los atrabancos de la escalera, la limpiamos bien, hacemos una reforma o instalamos un ascensor. La experiencia es la que nos dice cuándo ha llegado el momento de actuar y qué táctica proporciona la mejor relación coste/beneficio para que nadie se caiga por la escalera (y para que no se venga abajo la propia escalera).

Esencialmente, la separación de control de flujo y lógica se implementa en lenguajes como Java, extrayendo métodos de manera que contengan sentencias de control de flujo (*if-else, for, while, do, throw,…*) y llamadas a otros métodos que consisten en operaciones de asignación, cálculos, transformaciones, mutaciones…

Java

```
public void addFeeToItems(List<Item> items, Fee fee){
    for(Item item: items){          // <-- iteración sobre el flujo
        if (item.hasPrice()){       // <-- bifurcación de flujo
            item.increasePrice(fee);// <-- delegar lógica en el objeto
        }
    }
}
```

Como alternativa o complemento a los bucles y a las sentencias condicionales tradicionales, existen funciones de estilo declarativo incorporadas tanto en lenguajes modernos como en librerías. Java nos las proporciona mediante los *streams*, mientras que C# provee *LINQ* (*Language-Integrated Query*). Los lenguajes que nacieron con un estilo más funcional como JavaScript, incorporan las funcionas *map, filter, reduce*, etc. Veamos una alternativa al código anterior en Java:

Java

```
public void addFeeToItems(List<Item> items, Fee fee) {
    items.stream()
            .filter(Item::hasPrice)
            .forEach(item -> item.increasePrice(fee));
}
```

Es muy parecido con JavaScript:

Javascript

```
function addFeeToItems(items, fee){
    items.filter(item => item.hasPrice())
         .forEach(item => item.increasePrice(fee));
}
```

El método *increasePrice* tiene efectos secundarios, seguramente mutación del estado y, por tanto, no podemos utilizar un estilo tan funcional como lo haríamos con métodos de tipo consulta. Veamos el ejemplo de una consulta donde se utiliza un estilo declarativo con Java:

Java

```java
public BigDecimal calculateTotalPrice(List<Item> items) {
    return items.stream()
                .filter(Item::hasPrice)
                .map(Item::price)
                .reduce(new BigDecimal(0), BigDecimal::add);
}
```

El método filtra la colección de elementos quedándose con aquellos que devuelven verdadero al invocar a su método *hasPrice*. Luego, se queda con el precio de cada elemento y, finalmente, suma todos estos precios, que son de tipo *BigDecimal*, utilizando el método *add*. La versión C# de este método sería así:

C#

```csharp
public decimal CalculateTotalPrice(List<Item> items)
{
    return items.Where(item => item.HasPrice())
                .Select(item => item.Price())
                .Aggregate((price, subtotal) => subtotal + price);
}
```

El tipo más utilizado para cantidades de dinero en Java es *BigDecimal* por cuestiones de precisión, mientras que en C# es *decimal*. En C# apostaron por utilizar una nomenclatura similar a SQL, que es un lenguaje declarativo. De hecho, el lenguaje soporta unas construcciones llamadas *language-level query syntax*, que son tal que así:

C#

```csharp
var prices = from item in items
             where item.HasPrice()
             select item.Price();
```

Aunque los nombres estén inspirados en SQL, las operaciones son las típicas y sus correspondencias son:

- *Where* significa *filter*
- *Select* significa *map*
- *Aggregate* significa *reduce*
- *SelectMany* significa *flatMap*

Los IDEs modernos son capaces de transformar automáticamente bucles tradicionales en construcciones declarativas de este tipo. Por ejemplo, en IntelliJ y en Rider (productos de JetBrains) basta con poner el cursor encima de la palabra reservada *for* o *while* y aceptar la propuesta sugerida por el IDE. También realizan automáticamente el cambio inverso, lo cual es interesante para familiarizarse y experimentar con este tipo de sentencias.

¿Deberíamos prohibir los bucles clásicos y usar siempre *map/reduce/filter*, etc.? Opino que no, ¿cuál es el dogma? Justo antes vimos que es perfectamente posible escribir código si separamos control de flujo y lógica, aun cuando usamos bucles. Además, esta tampoco es una regla que debamos aplicar el 100 % de las veces. Cada bloque de código que programamos nos hace sopesar pros y contras de cada técnica, manejando principios que incluso pueden llegar a estar enfrentados según el contexto. Para mí, la simplicidad es uno de los principios que predomina en mis decisiones, y en ese sentido, hay situaciones en las que un bucle con unas cuantas líneas adentro es más fácil de digerir que combinaciones de *map, filter, reduce, pipe, flatMap*… Si usamos un estilo declarativo, es precisamente para no tener que pensar en el flujo, con lo cual, en el momento en el que traduzco funciones a bucles mentalmente, deja de tener sentido para mí. Hasta cierto punto, entender estas construcciones es también una cuestión de entrenamiento. Más allá de ese cierto punto, nos encontramos con la complejidad accidental como de costumbre.

A nivel de rendimiento, estas construcciones no son tan eficientes como lo sería un bucle, pero la diferencia es despreciable, porque la latencia de red o de acceso a disco es varios órdenes de magnitud mayor. Además, estamos hablando de complejidad lineal, que en el peor caso la colección se recorre dos o tres veces en lugar de una, lo que es insignificante en la inmensa mayoría de los casos. Mi consejo es medir siempre que nos preocupe la eficiencia, en vez de hacer apuestas y suposiciones. En ausencia de requisitos de rendimiento precisos y medibles, priorizo que el código sea legible.

Dar preferencia a las funciones puras

Tirando del hilo de la programación funcional, nos encontramos con un concepto tremendamente interesante, útil y aplicable en cualquier lenguaje: las funciones puras. Una función se considera pura si no tiene efectos secundarios (mutaciones o alteraciones del estado/sistema) y si produce siempre el mismo resultado ante una misma entrada, independientemente del número de veces en que se ejecute o del orden de ejecución. Una lectura de base de datos o de un servicio externo, no sería pura, porque no se puede garantizar que se

obtenga el mismo resultado en todas las llamadas. Las funciones puras cumplen la propiedad de la transparencia referencial, que dice que la llamada a una función podría sustituirse por su valor sin alterar el comportamiento del programa. Ejemplo de función pura:

Java

```
public float areaOfTriangle(float base, float height){
    return (base * height) / 2;
}
/*...*/
var area = areaOfTriangle(3, 2); // <-- 3
```

Esta función es pura, puesto que no produce ninguna alteración en el sistema y cumple con la transparencia referencial: el programa se comportará igual si invocamos a la función con los argumentos 3 y 2, que si directamente asignamos el valor 3 a la variable *area*. Esto no sería cierto en el caso de un método que altera el estado:

Java

```
class Triangle {
   float area;
   public void setArea(float base, float height){
      this.area = (base * height) / 2;
   }
   /*...*/
}
```

Las funciones puras son beneficiosas para el diseño, porque facilitan la lectura del código que las invoca, porque lo que ves es lo que hay, no dan sorpresas. Son idempotentes, lo que significa que invocarla una o mil veces sigue produciendo el mismo resultado y esto se traduce en confianza para quienes las usan. Escribir test automáticos para diseñar o validar funciones puras es trivial. No están sujetas a problemas de concurrencia, por eso también son ideales para computación paralela. Además, sus propiedades favorecen la composición de funciones y el encadenamiento de llamadas mediante *pipelines*.

Cuando practicas refactorización, piensa si tiene sentido partir los métodos o funciones para separar las partes que mutan el estado de las que no, de manera que las funciones puras emerjan como resultado de la práctica.

Considerar las funciones autocontenidas

Una función puede ser pura aunque dependa de variables que son externas a ella, siempre y cuando no mute sus valores. Aquella que accede a variables externas se denomina claursura (*closure*) en lenguajes funcionales. Las clausuras se suelen usar para gestionar estado (en cuyo caso no serían funciones puras), sin embargo, hay operaciones como la currificación que se implementan de manera pura, puesto que no alteran el estado. Los métodos de una clase se comportan de manera similar, pueden realizar lecturas de variables de instancia de su clase o de constantes, y ser puros, o bien pueden mutar dichas variables.

Con frecuencia, podemos restructurar las funciones o métodos que acceden a variables externas para que dejen de hacerlo, utilizando parámetros. Podemos enviarles todo lo que necesitan mediante argumentos, de manera que no tengan ninguna dependencia del exterior. Se trata de un cambio pequeño en el diseño, pero muy significativo; en lugar de que la función pida datos, se los entregamos listos para usar. La función sería autocontenida, independiente, autosuficiente y si además fuese pura, entonces su cohesión sería muy elevada y su acoplamiento con el exterior muy reducido. Fíjate que podría ser autocontenida y no ser pura, si hablamos de una función que tienen efectos secundarios. Incluso cuando no son puras, las funciones autocontenidas son interesantes porque facilitan los cambios en un diseño modular, puesto que es trivial trasladarlas de módulo/clase/paquete. Además, suelen ser de tamaño reducido.

Aunque una clase pueda definir atributos/campos globales a todos sus métodos, estos no tienen por qué acceder a ellos directamente, sino que podemos definir métodos que los reciben como argumentos. Así conseguimos métodos autocontenidos, que tienen menor grado de acoplamiento con los otros elementos de la clase y mayor cohesión:

Java

```
class ProjectBillingService {
    private final CustomerContract contract;
    private final DeveloperHistoryRepository repository;
    /*...*/
    public ProjectBillingService(DeveloperHistoryRepository repository,
                                 CustomerContract contract) {
        this.repository = repository;
        this.contract = contract;
    }
    /*...*/
    public BigDecimal calculateCost(YearMonth yearMonth){
        List<DeveloperHistory> history = repository.findBy(
```

```
            contract.id(),
            getStartDate(yearMonth),
            getEndDate(yearMonth));
        return contract.applyFeeTo(
            developersCost(history, contract, yearMonth));
    }

    private BigDecimal developersCost(List<DeveloperHistory> history,
                                      CustomerContract contract,
                                      YearMonth yearMonth) {
        return history.stream()
                .map(h -> costPerDeveloper(h, contract, yearMonth))
                .reduce(new BigDecimal(0), BigDecimal::add);
    }

    private BigDecimal costPerDeveloper(DeveloperHistory developerHistory,
                                        CustomerContract contract,
                                        YearMonth yearMonth){
        return multiply(
            contract.dailyRateFor(
                developerHistory.seniorityLevelOn(yearMonth)),
            developerHistory.billableDaysOn(yearMonth, contract.id()));
    }

    private BigDecimal multiply(BigDecimal dailyRate, float
        numberOfWorkingDays) {
        return dailyRate.multiply(BigDecimal.valueOf(numberOfWorkingDays));
    }

    private Date getEndDate(YearMonth yearMonth) {
        /*...*/
    }

    private Date getStartDate(YearMonth yearMonth) {
        /*...*/
    }
}
```

Este artefacto tiene como propósito calcular la factura del trabajo de un equipo de desarrollo en un proyecto. Los campos de la clase o variables de instancia, son un repositorio y un contrato. Ambos atributos son recibidos mediante su constructor. El método público *calculateCost* no es autocontenido, puesto que accede a las variables de instancia de la clase. En cambio, los métodos privados del ejemplo sí son autocontenidos, porque obtienen todo lo necesario de sus parámetros. Además son funciones puras porque no provocan ningún efecto secundario. Estos métodos son de tipo consulta, es decir, se limitan a responder una pregunta. Fíjate que la función *developersCost* podría acceder al atributo *contract* de la clase, pero hemos reemplazado dicha dependencia externa por una local (parámetro). Los métodos privados de lectura/consulta son muy buenos candidatos para ser diseñados

como autocontenidos, y cuando sea posible, como funciones puras.

Hay ocasiones en que el diseño quedará más simple e intuitivo si dejamos que los métodos accedan a los atributos globales de la clase, no se trata de forzarlo para que todos los métodos privados sean autocontenidos siempre. Sucede lo mismo con las funciones de tipo claurusa (*closures*) en JavaScript, que son aquellas que acceden a variables externas, y que a menudo son una opción simple y cómoda.

A la hora de extraer métodos para que sean autocontenidos, no es suficiente con evitar las dependencias a nivel de compilación, sino también a nivel de conocimiento. Veamos un ejemplo, con una tentadora línea en blanco:

Java

```
String decrypt(String fileName) throws InvalidAlgorithmParameterException,
    InvalidKeyException, IOException {
    String content;
    try (FileInputStream fileIn = new FileInputStream(fileName)) {
        byte[] fileIv = new byte[16];
        fileIn.read(fileIv);
        cipher.init(Cipher.DECRYPT_MODE, secretKey, new IvParameterSpec(
            fileIv));

        try (
            CipherInputStream cipherIn = new CipherInputStream(fileIn, cipher
                );
            InputStreamReader inputReader = new InputStreamReader(cipherIn);
            BufferedReader reader = new BufferedReader(inputReader)
        ) {
            StringBuilder sb = new StringBuilder();
            String line;
            while ((line = reader.readLine()) != null) {
                sb.append(line);
            }
            content = sb.toString();
        }
    }
    return content;
}
```

¿Sería conveniente el siguiente refactor?

Java

```
String decrypt(String fileName) throws InvalidAlgorithmParameterException,
    InvalidKeyException, IOException {
    try (FileInputStream fileIn = new FileInputStream(fileName)) {
        byte[] fileIv = new byte[16];
        fileIn.read(fileIv);
```

```
        cipher.init(Cipher.DECRYPT_MODE, secretKey, new IvParameterSpec(
            fileIv));
        return decryptFile(cipher, fileIn);
    }
}
private String decryptFile(Cipher cipher, FileInputStream fileIn) throws
    IOException {
    try (
        CipherInputStream cipherIn = new CipherInputStream(fileIn, cipher);
        InputStreamReader inputReader = new InputStreamReader(cipherIn);
        BufferedReader reader = new BufferedReader(inputReader)
    ) {
        StringBuilder sb = new StringBuilder();
        String line;
        while ((line = reader.readLine()) != null) {
            sb.append(line);
        }
        return sb.toString();
    }
}
```

Yo no lo haría, porque la función *decryptFile* no va a funcionar si antes no se ha llamado a *cipher.init*, lo cual crea un acoplamiento cronológico entre las funciones. Dicho acoplamiento nos obliga a invocar a las funciones en un orden determinado. A nivel sintáctico, da la impresión de ser una función independiente, pero no lo es. No va a funcionar si se mueve a otra clase. Barbara Liskov llama a este tipo de métodos *procedimientos parciales*, porque solamente funcionarán para un subconjunto de los casos de entrada (en este ejemplo, solo cuando *cipher.init* haya sido invocado antes). La autora recomienda utilizarlos lo menos posible, reservándolos para contextos muy acotados y justificados por un aumento drástico de rendimiento. Además, recomienda añadir cláusulas guarda que lancen excepciones si no se cumplen las precondiciones imprescindibles (siempre que no penalicen el rendimiento). En su libro, pone como ejemplo un algoritmo de ordenación descompuesto en varios procedimientos parciales. Un procedimiento o función parcial, no es lo mismo que la aplicación parcial de funciones. La aplicación parcial de funciones, consiste en fijar algunos parámetros para producir otras funciones con aridad reducida, una herramienta muy útil, sobre todo en programación funcional.

Extraer métodos para conseguir un diseño modular no es trivial, no basta con agrupar líneas sin más. Además de los beneficios citados anteriormente, trabajar con funciones autocontenidas nos ayuda a identificar con más claridad si una función debería pertenecer a una clase o a otra. Cuando el diseño es modular y trabajamos con objetos que se comunican entre sí, pueden surgir dudas respecto del lugar en el que ubicar un determinado método ¿Quién debería tener la responsabilidad de realizar este cálculo?, ¿quién debería

tener el conocimiento? Que ciertas funciones sean independientes facilita la aplicación del *refactoring* «mover método», de una clase a otra. Este cuidado a la hora de ubicar el método busca reducir los motivos para cambiar de una clase, consiguiendo así mayor cohesión.

Funciones sin parámetros de configuración

Cuando añadimos a un método o función un parámetro de tipo *boolean*, esta adquiere automáticamente dos responsabilidades, una para cuando llega verdadero y otra para cuando llega falso:

Java

```
document.setWriteMode(true);  // Responsabilidad 1
```

Java

```
document.setWriteMode(false); // Responsabilidad 2
```

Java

```
component.setVisible(true);    // Responsabilidad 1
```

Java

```
component.setVisible(false);   // Responsabilidad 2
```

¿Cómo podemos evitar parámetros *boolean*?, creando funciones específicas. La alternativa al ejemplo anterior sería esta:

Java

```
document.switchToWriteMode();
```

Java

```
document.switchToReadMode();
```

Java

```
component.show();
```

Java

```
component.hide();
```

Ahora, cada método tiene un único propósito y los objetos gobiernan su estado internamente. Se consigue garantizar la consistencia del estado y reducir el acoplamiento, dado que los consumidores ya no tienen por qué saber cómo se implementa la gestión del estado. Queda más explícito lo que hace cada método.

Boolean es el tipo de parámetro de configuración más obvio y fácil de evitar, pero no es el único. Desde el momento en que dependemos del valor de uno de los parámetros para tomar decisiones dentro de la función, estamos potencialmente añadiéndole más de una responsabilidad:

Java

```
public void sort(List<Word> words, int order){
    if (order <= -1) {
        /*... responsabilidad 1 ... */
    } else if (order == 0) {
        /*... responsabilidad 2 ... */
    } else {
        /*... responsabilidad 3 ... */
    }
}
```

Esta oscura implementación codifica conceptos mediante valores (*magic numbers* en este caso), lo cual dificulta la lectura. Una alternativa de nuevo consiste en diseñar funciones específicas:

Java

```
sortAlphabeticallyAscending(words);
```

Java

```
sortAlphabeticallyDescending(words);
```

Estas funciones con responsabilidad única se han quitado de en medio los condicionales, reduciendo así la complejidad ciclomática y la accidental.

Los tipos de parámetros de configuración pueden ser variados: *boolean*, valores mágicos (números, *null*, códigos), enumerados, objetos de configuración,... Ejemplos de lo que recomiendo evitar:

Java

```
sort(words, SortCriteria.ascending); // <-- Enums
discountTax("IVA");                  // <-- Códigos de configuración
execute(command, null, null);       // <-- Null para representar conceptos
calculateTotal(items, config);       // <-- Combo de los anteriores
```

Si resultara insuficiente crear funciones específicas, porque la cantidad de variantes fuese abrumadora o porque se precisara abrir la puerta a otras implementaciones, podemos recibir un algoritmo como argumento:

Java

```
@Test
public void lambdas_my_be_used_as_arguments(){
    List<String> words = new ArrayList<>();
    words.add("short");
    words.add("esternocleidomastoideo");

    words.sort((item1, item2) -> Integer.compare(item2.length(), item1.length
        ()));

    assertThat(words.get(0)).isEqualTo("esternocleidomastoideo");
    assertThat(words.get(1)).isEqualTo("short");
}
```

El método *sort* de las listas en Java admite como parámetro una función anónima de comparación (o un objeto que implemente *Comparator*). Esta es una implementación del patrón estrategia. En el ejemplo, la estrategia consiste en ordenar primero las palabras con mayor longitud.

Definir parámetros de tipo estrategia tiene el beneficio de evitar los condicionales dentro del método, a la vez que abre el código a extensión sin necesidad de modificación. Como contrapartida, complica el código y produce un acoplamiento a nivel de algoritmo, al obligar a los consumidores a resolver parte del problema. No es una solución para todo. Es interesante en problemas muy genéricos como la ordenación de una lista donde la cantidad de soluciones posibles se escapa de la imaginación de quien diseña el método. También puede ser interesante en los métodos privados de una clase, para que el acoplamiento se limite a este contexto, sin que el exterior conozca estos detalles:

Java

```
class CaesarCipher {
    public String encryptToChars(String text) {
        return shiftCharacters(text, String::valueOf);
    }

    public String encryptToNumbers(String text) {
        return shiftCharacters(text, (aChar) -> String.valueOf((int) aChar));
    }

    private String shiftCharacters(String text, CharObfuscator obfuscator) {
        StringBuilder result = new StringBuilder();
        int shift = 10;
        for (char eachChar : text.toCharArray()) {
            char shiftedChar = (char) ((int) eachChar + shift);
            result.append(obfuscator.obfuscate(shiftedChar));
        }
        return result.toString();
    }
    interface CharObfuscator {
        String obfuscate(char character);
    }
}
```

La visibilidad actual de los métodos, junto con la probabilidad de que la mantengan en el tiempo, son factores con peso a la hora de tomar decisiones de diseño. Si no son públicos, ni parece ser que vayan a serlo, podemos permitirnos parametrizar métodos desvelando parte de su implementación:

Java

```
class QueryParser {
    public List<Token> parseTokens(String query){
        var tokens = getTokens(query, ",");
        tokens = removeIrrelevants(tokens, "a", "the", "from", "to");
        tokens = removeIrrelevants(tokens, "*", "\"", "'", ".", "-", "_");
        return tokens.collect(Collectors.toList());
    }

    private Stream<Token> getTokens(String expression, String separator) {
        return Arrays.stream(expression.split(separator))
                .map(token -> new Token(token.trim()));
    }

    private Stream<Token> removeIrrelevants(Stream<Token> tokens, String...
        irrelevants){
        return tokens.filter(token -> token.matchesAny(asList(irrelevants)));
    }
}
```

Estos métodos privados tienen dos argumentos, y en ambos casos el segundo de ellos desvela detalles de implementación. Obligan al método llamador a entender cómo funcionan por dentro. En este caso, no parece ser un problema, ya que el acoplamiento está encapsulado en la clase, manteniendo la cohesión de cara al exterior. Quizá alguien piense convertir esos métodos en públicos y moverlos a una clase *Utils* para poderlos reutilizar, pero no es buena idea. Cuando pasan a ser públicos, el conocimiento que debía estar encapsulado se disemina por toda la aplicación, acoplando piezas que deberían ser independientes y desconocidas entre sí. Además, estos pequeños métodos son triviales y se tarda muy poco en escribirlos (¿cinco minutos?). Es irrelevante que alguno de esos pequeños métodos privados se implemente varias veces en diferentes áreas del proyecto. No es un código que fuera a aportar gran cosa en una librería de propósito general, no es tan brillante como el algoritmo *quick sort* ni tan laborioso como el *problema del cartero chino*.

Como regla general, cuando hablamos de lógica de dominio, no es buena idea definir métodos públicos que reciben argumentos como estructuras de datos, constantes, estrategias o cualquier otra información que obligue al código consumidor a comprender (o intuir) el algoritmo interno.

Reducir la aridad y los parámetros opcionales

La aridad es el número de parámetros de una función. Hay una correlación entre la cantidad de parámetros de una función y la granularidad de su responsabilidad. Si una función tiene cero parámetros, y además no accede a ningún elemento externo a ella, lo único que puede hacer es devolver un valor. No tiene por qué ser trivial (*Math.random* no lo es), pero sí es más propensa a tener una única responsabilidad. Como es natural, a medida que incrementamos la aridad y las referencias a variables externas o a otras funciones, aumenta la posibilidad de introducir complejidad accidental en la función. Para diseñar funciones con una única responsabilidad, mi recomendación es definir el menor número posible de parámetros. A ojo de buen cubero, diría que el número de parámetros ideal está entre cero y tres. Esta recomendación es igualmente aplicable a los constructores, de manera que si una clase está recibiendo cinco dependencias inyectadas por constructor, puede ser que tenga demasiada responsabilidad (yo procuro que no haya más de dos).

Algunos lenguajes admiten la definición de parámetros opcionales, típicamente mediante la asignación de un valor por defecto:

Python

```
def build_invoice(id = None, concept = "", gross = 0.0, vat = 16.0):
    #...
```

Este mecanismo no es otra cosa que azúcar sintáctico para la sobrecarga de métodos, de hecho, es el uso habitual en lenguajes que no soportan la sobrecarga tradicional, como Python o JavaScript. En C# existen las dos posibilidades, sobrecarga tradicional y parámetros opcionales con valores por defecto. Ahí es donde me dí cuenta de que tenían la misma función, cuando el compilador me avisó de que había colisiones en la definición de los métodos. Definir un método con un parámetro por defecto es lo mismo que sobrecargar un método tradicional, con una versión que lleva el parámetro y otra que no. Si añadimos otro parámetro por defecto, la combinatoria aumenta. Este aumento de casos posibles tiene correlación con el número de casos que se nos va a olvidar dejar testados (con test automáticos), que a su vez implica mayor probabilidad de incurrir en *bugs*. Por eso, hay que tener cuidado y limitar su uso. Existen casos de uso especiales, como la construcción de objetos mediante *builders*, que se pueden beneficiar de los parámetros opcionales. Para el resto de situaciones mi recomendación es evitar los parámetros opcionales, o que haya solo uno opcional (esto depende de las características del lenguaje).

Separar en consultas y comandos (CQS)

Podemos clasificar los métodos o funciones en dos grandes grupos: comandos y consultas. Un comando sería un método que ejecuta una acción y no devuelve nada, mientras que una consulta responde a una pregunta.

Java

```java
public void deleteUser(User user);       // <-- Comando
```

Java

```java
public List<User> findByName(Name name); // <-- Consulta
```

Los comandos alteran el estado del sistema, mientras que las consultas no. Dicho de otra forma, los comandos son funciones mutadoras, mientras que las consultas pueden ser funciones puras.

Command-query separation es un principio acuñado por Bertrand Meyer. No es lo mismo que CQRS (arquitecta de *software*), aunque CQRS se apoya en CQS. La aplicación de CQS separa el código que produce mutaciones del que no. Por un lado, fomenta el uso de funciones puras que, como hemos visto, aumenta la cohesión. Por otro lado, delimita de manera explícita las partes del código donde se producen cambios de estado. Tales cambios son una de las principales causas de *bugs* en los programas, de manera que tenerlos bien localizados es de gran ayuda. La firma de un método que no devuelve nada (*void* en Java), hace explícito el hecho de que muta el estado (de lo contrario, no serviría para nada). Así que cuando un método tiene la intención de alterar sus argumentos, las variables de instancia o cualquier otro elemento del sistema, definirlo como *void* es la forma más explícita de hacerlo.

Lo contrario resulta confuso y causa sorpresas. Veamos como ejemplo el método *sort* de la clase *Array* en JavaScript:

Javascript

```
it("sorts the elements mutating the array", () => {
    let items = ['x', 'a', 'h', 'b'];
    let sortedItems = items.sort(); // ???
    expect(sortedItems).toEqual(['a', 'b', 'h', 'x']);
    expect(items).toEqual(['a', 'b', 'h', 'x']);
    expect(items).toBe(sortedItems);
})
```

La primera línea de las expectativas expone lo que cualquiera esperaría, puesto que estamos comprobando la respuesta del método *sort*. Ahora bien, la segunda es sorprendente, resulta que el vector original también ha sido modificado. Así que *sort* hace dos cosas en realidad, ordena y devuelve una referencia al propio *array*. Es confuso. El diseño sería más intuitivo si hiciera solamente alguna de las dos cosas: devolver los elementos ordenados sin modificar el original o no devolver nada y modificar el original. Sucede algo similar en Java con los iteradores, porque el método *next* devuelve el siguiente elemento de la colección, además de avanzar el iterador:

Java

```
while (iterator.hasNext()) {
    String next = iterator.next();  // ???
    System.out.println(next);
}
```

Sería más claro separar los dos comportamientos en métodos específicos, de manera que,

por un lado, pudiésemos obtener el elemento actual y, por otro, avanzar al siguiente:

Java

```java
while (iterator.hasNext()) {
    String nextItem = iterator.getNext(); // <-- query
    System.out.println(nextItem);         // <-- command
    iterator.moveToNext();                // <-- command
}
```

Siempre que sea posible en términos de simplicidad, rendimiento y consistencia de datos, es conveniente separar las operaciones en comandos y consultas.

A la hora de extraer métodos refactorizando, conviene agrupar los bloques para cumplir con este principio. En uno de los ejemplos anteriores teníamos el siguiente bloque de código:

Java

```java
PropertySource<?> source = propertySources.get(PROPERTY_SOURCE_NAME);
if (source instanceof MapPropertySource) {
    target = (MapPropertySource) source;
    for (String key : map.keySet()) {
        if (!target.containsProperty(key)) {
            target.getSource().put(key, map.get(key));
        }
    }
}
```

En lugar de extraer todo el bloque a un solo método, decidí extraer dos para cumplir con CQS:

Java

```java
MapPropertySource source = getMapPropertySourceFrom(propertySources); // <--
    query
if (source != null){
    populateSourceFromMap(source, map); // <-- command
}
```

Si fuera un solo método, estaría devolviendo *source* al mismo tiempo que mutaría dicha variable, es decir, la firma del método parecería engañosamente de tipo consulta. Además de la posible sorpresa que causaría a quien lea la llamada, me costaría trabajo encontrar un buen nombre para el método. Aquí tenemos otro ejemplo más de cómo la búsqueda de nombres apropiados nos ayuda a identificar el incumplimiento de los principios de diseño. Si cuidas los nombres, estarás cuidando los principios.

En un capítulo anterior trabajamos con un ejemplo de código que también incumple el principio CQS:

Java

```
@Override
public boolean onScale(ScaleGestureDetector detector) {
    float scale = detector.getScaleFactor();

    // Speeding up the zoom by 2X.
    if (scale > 1f) {
        scale = 1.0f + (scale - 1.0f) * 2;
    } else {
        scale = 1.0f - (1.0f - scale) * 2;
    }

    float newRatio = getZoomRatio() * scale;
    newRatio = rangeLimit(newRatio, getMaxZoomRatio(), getMinZoomRatio());
    setZoomRatio(newRatio);
    return true;
}
```

Claramente es un comando, pero su firma parece la de una consulta. Es un poco absurdo, además, porque siempre devuelve verdadero (el caso de falso no está ni contemplado). Teniendo en cuenta que se trata de un método heredado de una clase base, me puedo imaginar que es un caso de herencia metida con calzador. Alternativa:

Java

```
public void scale(ScaleGestureDetector detector) { // <-- command
    float scale = calculateScaleFactor(detector, Speed.x2); // <-- query
    setZoomRatio(calculateNewRatio(scale)); // <-- command
}
```

Quizá te está surgiendo la duda de, ¿cómo puede un comando avisar de que algo salió mal? Cuando un comando fracasa en su ejecución, la capa de código que inició la orden podría tomar la decisión de reintentarlo, informar a la usuaria, o lo que sea que tenga más sentido en un sistema robusto ¿Cuál es la forma más recomendable de informar sobre el éxito o fracaso de un comando? Esto lo veremos en el capítulo sobre gestión de errores. No ayuda devolver *booleans*, porque habitualmente queremos tener más información sobre el fallo.

En la medida de lo posible, hay que evitar que una consulta lance excepciones, y esto lo podemos conseguir separando responsabilidades, por ejemplo, separando los cálculos de las validaciones. En este ejemplo están mezcladas:

Java

```java
private List<Integer> parsePositiveNumbersFrom(String csv){
    List<Integer> numbers = new ArrayList<>();
    for (String value : csv.split(",")){
        if (value.matches("-?\\d+")) {
            int number = Integer.parseInt(value);
            if (number <= 0) {
                throw new NumberFormatException("Only positive numbers are
                    allowed");
            }
            numbers.add(number);
        }
    }
    return numbers;
}
```

Para evitar sorpresas, podemos separar la responsabilidad de transformación de la de validación:

Java

```java
private List<Integer> parseNumbersFrom(String csv){
    return Arrays.stream(csv.split(","))
        .filter(value -> value.matches("-?\\d+"))
        .map(Integer::valueOf)
        .collect(Collectors.toList());
}

private void checkForPositiveNumbers(List<Integer> numbers){
    var anyNegativeNumber = numbers.stream()
        .filter(number -> number <= 0)
        .findAny();
    if (anyNegativeNumber.isPresent()){
        throw new NumberFormatException("Only positive numbers are allowed");
    }
}
```

En el código desde donde se llame a estas funciones se entenderá cuál es consulta, cuál es comando y qué esperar de ambas:

Java

```java
/*...*/
List<Integer> numbers = parseNumbersFrom(csv);
checkForPositiveNumbers(numbers);
/*...*/
```

No siempre es posible cumplir con el principio CQS. El escenario de incumplimiento típico

es la creación de una nueva entidad en la base de datos:

Java

```java
public Id createUser(User user){
    String id = insertIntoUserTable(user);
    /*...*/
    return new Id(id);
}
```

Es muy común pedir el identificador de la entidad tan pronto como se crea en la base de datos, si es que delegamos en ella la creación de la clave primaria. Si lo hiciéramos en dos pasos, uno de creación y otro de consulta del identificador, no podríamos garantizar que no se insertan nuevas entidades por el medio (consistencia de los datos). La única manera de garantizar la consistencia de los datos sería realizar toda la operación de forma atómica[1].

Otras situaciones en las que compensa incumplir el principio, también tienen relación con los requisitos no funcionales como el rendimiento. Por ejemplo, si una determinada consulta es dura de computar y se realiza miles de veces, puede tener sentido guardar la pareja pregunta/respuesta en caché:

Java

```java
public Checksum getChecksum(String filename){
    if (checksumCache.containsKey(filename)){
        return checksumCache.get(filename);
    }
    MessageDigest md5 = MessageDigest.getInstance("MD5");
    md5.update(Files.readAllBytes(Paths.get(filename)));
    String checksum = DatatypeConverter
        .printHexBinary(md5.digest()).toUpperCase();
    checksumCache.put(filename, checksum);
    return checksum;
}
```

[1]También es cierto que existen otras formas de gestionar los identificadores de las entidades, por ejemplo, generando códigos UUID (https://bit.ly/34vFu5G) antes de persistir en la base de datos, aunque no reemplazan el id autogenerado de la base de datos, sino que lo complementan.

9. Gestión y prevención de errores

A día de hoy, es imposible escribir un programa sin errores, por más test automáticos que pongamos, ni por más *testing* de exploración que hagamos. Hay líneas de investigación y nuevos lenguajes de programación que buscan limitar la ocurrencia de errores, con la meta de llegar a eliminarla. Los lenguajes empresariales actuales están todavía lejos de ese punto, y encima los infrautilizamos programando como si estuviéramos con FORTRAN o C. Tanto este capítulo como el próximo, van de cómo sacarle partido a los lenguajes convencionales modernos como Java, C#, Kotlin o TypeScript, para diseñar artefactos con un código menos propenso a errores y más expresivo.

Si hay algún campo donde la ley de Murphy aplica de lleno, es sin duda la ingeniería, de hecho, la ley se originó cuando Edward A. Murphy Jr trabajaba experimentando con cohetes. Todo lo que piensas que puede salir mal cuando programas, saldrá mal. Por principio, debemos programar minimizando la cantidad de accidentes que pueden ocurrir, priorizándolo sobre cuestiones subjetivas como los gustos de cada persona. Es frecuente que los equipos discutan sobre gustos o cuestiones, como, por ejemplo, si las llaves ({}) van en la misma línea o en la siguiente, o si una sentencia *if* debe llevar llaves cuando solo afecta a una línea. Por suerte, las comunidades alrededor de los lenguajes recomiendan unos estándares de estilo que nos sirven de plantilla. Así, lo habitual en Java es que las llaves se coloquen en la misma línea, mientras que en C# van en la línea siguiente. En Python existe una popular guía de estilo llamada PEP-8. Las guías de estilo no lo abarcan todo, son unas recomendaciones básicas que siguen dejando margen para que cada equipo defina sus reglas o convenciones (y está bien que así sea). Digo equipo porque desearía que fuesen acuerdos de todo el equipo, y no decisiones arbitrarias de cada individuo, porque de lo contrario tendríamos multitud de estilos dentro del mismo proyecto y esto puede resultar confuso para quien lea el código. Sobre los gustos no se puede discutir, yo no le puedo debatir a nadie que le guste la alcachofa o el apio. Tampoco puedo estar en contra de que las llaves le gusten en una línea o en otra. Los gustos son irracionales. Para que las discusiones sean productivas y las diferencias nos enriquezcan, necesitamos razonar de manera objetiva con argumentos sólidos. Veamos un ejemplo:

Java

```
/*...*/
```

```
if (item.hasDiscount())
    offer.add(item);
/*...*/
```

Típicamente, en los lenguajes que utilizan llaves, estas son opcionales para una sentencia condicional tipo *if*. Si no hay llaves, se interpreta que solamente la siguiente línea está sujeta a la condicional, ninguna más. Por eso, cuando solo hay una línea, hay quien prefiere no poner las llaves. Aquí podríamos entrar a hablar de gustos sobre si a mí me gusta que estén las llaves y a ti no, o bien podemos agarrarnos a algún principio basado en el raciocinio. En este ejemplo, el principio es la reducción de la probabilidad del error: más de una vez he encontrado *bugs* debidos a que alguien no se dio cuenta de la ausencia de llaves y añadió líneas al «supuesto bloque» condicional pensando que caían dentro del ámbito del bloque *if*:

Java

```
/*...*/
if (item.hasDiscount())
    totalPrice += item.price();
    offer.add(item);
/*...*/
```

Es de humanos errar y así lo hizo la persona que creía estar añadiendo a la oferta, solo aquellos elementos con descuento, cuando en realidad los estaba añadiendo todos. Un despiste que en ausencia de test podría costar dinero a la compañía y que podría haberse ahorrado siendo consistentes con el uso de las llaves:

Java

```
/*...*/
if (item.hasDiscount()){
    totalPrice += item.price();
    offer.add(item);
}
/*...*/
```

Para personas que están acostumbradas a trabajar en Python, donde la indentación es la que marca los bloques, venir a Java, C# o Kotlin, y cometer este error es previsible. En realidad, también es probable para cualquier persona que trabaje todo el tiempo con Java, sobre todo cuanto más grande sea el código. Este tipo de errores son fastidiosos de encontrar, porque el cerebro asume que todas las líneas están dentro del bloque, y por más veces que uno pasa por ahí leyendo el código no lo advierte, hasta que ya obstinado abre

el depurador, revisa todo línea por línea, lo encuentra y termina sintiendo que ha perdido un tiempo precioso. Todo porque a alguien le pareció que quedaba más bonito sin las llaves (ese alguien pude haber sido yo). Te recomiendo que pongas siempre las llaves, así tienes una decisión menos que tomar y puedes concentrarte en lo importante. Si eres una persona que está convencida de que nunca comete estos errores tontos, piensa por favor que los demás sí los cometemos y que alguien que no eres tú vendrá en el futuro a mantener el código (que es propiedad de tu empresa o de tu cliente).

En una de sus conferencias Douglas Crockford[1] matiza muy bien la diferencia entre estilos y gustos. En las versiones de JavaScript anteriores a ES6, había un puñado de encrucijadas, por ejemplo, poner o no poner el punto y coma al final de la línea tenía unos efectos perversos, y pese a ello había gente que prefería no ponerlos por cuestión de gustos.

La mala indentación del código es otra fuente de errores humanos muy fácil de eliminar. Una vez que se acuerda cuál es la indentación que el equipo prefiere (en caso que sea diferente a la propuesta por el IDE), se configura el IDE para que todo el mundo tenga la misma. Conforme escribimos código, el IDE se encarga de la indentación y cuando encontramos un código antiguo con indentación diferente, podemos pedirle que aplique nuestra configuración de indentación de una tacada. Cuando programo con una persona que utiliza símbolos como el igual (=) sin un espacio a la izquierda y otro a la derecha, no puedo evitar interpretar que está poco sensibilizada con los errores tontos programando y me siento incómodo dejando las líneas así:

Java

```
totalPrice+=item.Price();   // sin espacios se lee peor
```

No cuesta nada separar cada parte de la expresión:

Java

```
totalPrice += item.Price(); // con espacios se lee mejor
```

Los nombres explicativos también ayudan a reducir la probabilidad de error humano comparados con los nombres de una sola letra: cuando en un bloque de código tengo varias variables con una sola letra, las posibilidades de que me confunda de variable aumentan proporcionalmente al número de variables que haya. Quiero decir que es menos probable

[1] Creador del formato JSON y autor del libro *JavaScript The Good Parts*, es uno de los programadores más influyentes en el ecosistema JavaScript.

que confunda la variable *offer*, que la variable *o*.

Los tipos específicos del dominio son otro recurso que reduce la probabilidad de error, puesto que si una variable es de tipo *Offer*, no puedo asignarle cualquier número o cadena. Cuando utilizamos los tipos integrados hay mayor riesgo de asignar valores incorrectos a variables.

Trabajar con un IDE que autocompleta y que hace otras sugerencias/advertencias «inteligentes», es otra excelente manera de identificar enseguida errores humanos. Personalmente, me encanta el editor Vim, es mi favorito para hacer ciertos *scripts* o para editar cualquier fichero cuando accedo a una terminal de comandos remota, sin embargo, no lo recomiendo para programar con Java o C#, porque la cantidad de errores que me ayuda a detectar es mucho menor (incluso con *plugins*). Si tengo que elegir prefiero IntelliJ (para Java y Kotlin) o Visual Studio (para C#).

Es por la capacidad de detectar errores humanos que me siento más tranquilo programando con lenguajes compilados que con los interpretados. Hace años era muy tedioso programar en Java comparado con Python, por lo verboso de la sintaxis (sigue siendo mucho más complejo para alguien que comienza en la programación), mientras que hoy en día los IDE autogeneran tanto código que termino tecleando menos con un lenguaje compilado que con uno interpretado.

A la hora de elegir lenguaje, influyen muchos más factores como la solvencia del equipo con el lenguaje, la inercia, el código legado, la comunidad que haya detrás, la estrategia de futuro, etc. Por ejemplo, la comunidad alrededor de Ruby fue pionera en la adopción de buenas prácticas como el *testing* automático o el diseño simple y modular. Algunos de mis libros de programación favoritos utilizan Ruby para los ejemplos de código. Hace diez años tenía mucho sentido usar Ruby, aunque hoy en día parece que está cayendo en desuso. Otros lenguajes que teóricamente reducen más la probabilidad de error, tienen la desventaja de que los conoce poca gente y cuesta encontrar trabajadores si el proyecto crece. La elección de lenguaje no es una decisión trivial ni está siempre a nuestro alcance.

Lo que vamos a ver en este capítulo es aplicable a la mayoría de lenguajes populares, independientemente del tipo que sean. Nuestra misión como profesionales es hacer el mejor trabajo posible con las herramientas que tengamos en cada momento, nos gusten más o menos; esto marca una diferencia entre profesional y amateur.

Comprender cómo funciona cada lenguaje

Para escribir programas consistentes necesitamos razonar, estudiar y comprender el comportamiento del lenguaje que utilizamos en el día a día como para tener el código bajo control. Al leer cada línea de código, debemos ser capaces de razonar cuál es el comportamiento esperado, sin titubear. Tener nociones teóricas sobre cómo se diseña un compilador ayuda a comprender mejor los lenguajes, aunque no es indispensable si tenemos claros los conceptos fundamentales de los lenguajes típicos. Vamos a hacer un repaso a conceptos generales que aplican a gran variedad de estos lenguajes.

Evaluación de expresiones complejas

Los paréntesis en el código tienen diferente significado según su orden de aparición en cada sentencia. Si lo que hay justo antes de unos paréntesis es un nombre de función/método, entonces indican definición o invocación de función. Será una cosa u otra, según la estructura de la frase (igual que ocurre con los lenguajes naturales), siendo generalmente definición, cuando hay llaves a continuación de los paréntesis:

Java

```java
public void action(){/*...*/}   // definición en Java
action();                       // invocación
```

Javascript

```javascript
function action(){/*...*/}        // definición con nombre en JavaScript
action();                         // invocación
() => {/*...*/}                   // definición anónima en JavaScript
otherAction = (args) => {/*...*/} // definición de variable que contiene
    función
otherAction(Math.PI);             // invocación
```

El otro uso de los paréntesis es el de agrupar expresiones y especificar su precedencia dentro de una sentencia compuesta. Los lenguajes que utilizan paréntesis y corchetes, evalúan las expresiones compuestas resolviendo desde la subexpresión más interna a la más externa. Por ejemplo:

Java

```java
(2 * (5 + (1 / (2 + 4)))) // -> 10
```

```
(2 * (5 + 1 / 2 + 4))     // -> 18
(2 * 5 + 1 / 2 + 4)       // -> 14
```

Primero se calcula el paréntesis más interno, luego el siguiente y así sucesivamente:

Java

```
(2 * (5 + (1 / (2 + 4)))) // -> (2 * (5 + (1 / 6))
(2 * (5 + (1 / 6))        // -> (2 * 5 + 0)
```

En ausencia de paréntesis la precedencia viene marcada por los operadores, por ejemplo, la multiplicación y la división tienen mayor precedencia (prioridad) que la suma y la resta:

Java

```
(2 * (5 + 1 / (2 + 4))) // -> 10
```

Para evitar cometer errores en expresiones complejas como esta, lo recomendable es poner los paréntesis, o sea, indicar explícitamente cómo queremos que se evalúe la expresión. Es una forma sencilla de prevenir fallos. Por otro lado, que haya demasiados paréntesis también puede ser fuente de *bugs*, ya que aumenta la probabilidad de equivocarnos abriendo y cerrando paréntesis en lugares incorrectos (afortunadamente, los editores de código señalan el paréntesis de cierre correspondiente al de apertura, cuando situamos el cursor justo en ese lugar y viceversa). A veces, aporta claridad partir la expresión en varias expresiones:

Java

```
partial = 1 / (2 + 4); // queda explícito que esta parte se resuelve primero
total = (2 * (5 + partial)); // y esta a continuación.
```

Los paréntesis y corchetes pueden ser complejos de entender cuando aparecen seguidos, incluso si no hay expresiones anidadas. Te planteo un ejercicio, ¿cómo tendrías que rellenar la función *koan* para hacer pasar el siguiente test? (lenguaje JavaScript):

Javascript

```
function koan(){
   // ?`qué código tendrías que escribir aquí?
}
it("may return arrays that contains closures and so on", function(){
    expect(koan()[0](1)[1]).toEqual(10);
    expect(koan()[0](2)[1]).toEqual(11);
```

```
    expect(koan()[0](3)[1]).toEqual(12);
});
```

Vamos a analizar la expresión paso a paso:

Javascript

```
koan()[0](1)[1]
```

Como no hay expresiones anidadas (paréntesis dentro de paréntesis), procedemos a leer la expresión de izquierda a derecha:

Javascript

```
koan() // esta parte podría ser invocación o definición
koan()[0] // la ausencia de llaves detrás de los paréntesis aclaran que es
    invocación
koan()[0] // los corchetes implican que la función ha devuelto un array
koan()[0](1) // los paréntesis indican que el elemento 0 del array es una
    función
koan()[0](1)[1] // el corchete indica que la función ha devuelto un array
```

Solución:

Javascript

```
function koan(){
    return [(number) => [0, number + 9]] // array con funcion adentro que
        devuelve array
}
```

Mi intención con este ejercicio es que comprendas que existe un orden lógico y determinista de evaluación de las expresiones, donde los símbolos y su orden producen un efecto u otro. Si no lo comprendes, aumenta la probabilidad de cometer errores. No hay magia ni misterio, aunque pueden llegar a ser tan difíciles de leer como las fórmulas matemáticas complejas. Se requiere entrenamiento para comprender, por ejemplo, funciones que devuelven funciones:

Javascript

```
let greet = (message) => (name) => {
    return `${message} ${name}!`;
}
let helloGreet = greet('Hello');
helloGreet('Peter');      // -> Hello Peter!
```

```
let goodDayGreet = greet('Good day');
goodDayGreet('Bob');      // -> Good day Bob!
```

Veamos un ejemplo avanzado con TypeScript, una función que devuelve una sublista de números en base a una lista de entrada, que contiene a aquellos que se repiten un número impar de veces:

Typescript

```
function extractNumbersRepeatedOddTimes(numbers: number[]): number[] {
    return [...new Set<number>(
        numbers
        .map(n => [
            n,
            numbers.reduce((repeatedTimes, y) => n === y
                ? repeatedTimes + 1
                : repeatedTimes, 0)
        ])
        .filter(([n, repeatedTimes]) => repeatedTimes % 2 !== 0)
        .map(([n, repeatedTimes]) => n)
    )]
}
extractNumbersRepeatedOddTimes([1,1,2,2,3,5,6,6,6]) // [3,5,6]
```

No es trivial entender esta función, nos obliga a computar mentalmente para resolver las operaciones desde dentro hacia afuera. Al igual que en el ejemplo de los paréntesis, podemos partir la expresión y darle nombres a las partes para explicarla:

Typescript

```
function extractNumbersRepeatedOddTimes(numbers: number[]): number[] {
    return removeDuplicates(
             filterNumbersRepeatedAnOddNumberOfTimes(
                 groupNumberRepetitions(numbers)));
}
function groupNumberRepetitions(numbers: number[]): Array<[number, number]> {
    return numbers
        .map(n => [
            n,
            numbers.reduce((repeatedTimes, y) => n === y
                ? repeatedTimes + 1
                : repeatedTimes, 0)
        ]);
}
function filterNumbersRepeatedAnOddNumberOfTimes(group: Array<[number, number
    ]>) : number[] {
      return group
             .filter(([n, repeatedTimes]) => repeatedTimes % 2 !== 0)
             .map(([n, repeatedTimes]) => n)
```

```
}
function removeDuplicates(numbers: number[]): number[] {
    return [...new Set<number>(numbers)];
}
```

Los occidentales leemos de izquierda a derecha, o sea que leemos las expresiones anidadas de fuera hacia dentro, en el orden opuesto al que utiliza la máquina para evaluar (de dentro hacia fuera). Si esto supone un obstáculo para la comprensión, disponemos de alternativas para expresarnos como la función *pipe*:

Typescript

```
function pipe(...functions: Function[]): (input: number[]) => number[] {
  return (input: number[]) => functions.reduce((result, fn) => fn(result),
      input)
}

let extractNumbersRepeatedOddTimes = pipe(
     groupNumberRepetitions,
     filterNumbersRepeatedAnOddNumberOfTimes,
     removeDuplicates
);

function groupNumberRepetitions(numbers: number[]): Array<[number, number]> {
    /*...*/
}
function filterNumbersRepeatedAnOddNumberOfTimes(group: Array<[number, number
    ]>) : number[] {
    /*....*/
}
function removeDuplicates(numbers: number[]): number[] {
    /*...*/
}
```

Si no tienes la costumbre de programar con lenguajes que trabajan con funciones de *orden superior*, es decir, que *admiten funciones como argumentos o que devuelven funciones*, igualmente te va a costar entender este código. No te frustres, es una cuestión de puro entrenamiento. Lo más importante es que entiendas el razonamiento y la forma en que el intérprete va a evaluar las expresiones. El objetivo de este apartado es que te quedes con algunas pistas para reducir la probabilidad de introducir errores de interpretación. Si utilizas JavaScript o TypeScript, entenderás que esta es la razón para usar promesas y otros edulcorantes sintácticos como los observables, *async/await*, etc., elementos que pueden simplificar la lectura cuando se usan bien, a la par que aumentan la complejidad de la implementación. El uso de estos recursos no exime de su comprensión, o sea que simplifican la lectura solo si entendemos su funcionamiento. La simplicidad y la legibilidad son cuali-

dades que pueden estar reñidas, cada equipo debe encontrar el balance adecuado en base a su nivel de conocimiento, rotación, etc. El hecho de que sepamos cómo funcionan las promesas o los observables, por ejemplo, no significa que los debamos usar si un simple *callback* (función pasada como argumento) es suficiente. El libro *JavaScript Allongé* de Reginal Braithwaite, es fantástico para comprender las funciones de orden superior.

Valores, referencias y objetos

No todo el mundo tiene claro cómo funciona la operación de asignación o el paso de argumentos a una función. Es sencillo, funciona igual en todos los lenguajes, y a pesar de ello, es fuente de *bugs* o de complicaciones innecesarias cuando no se comprende. Con este resumen espero que le pierdas el miedo a las operaciones básicas y que agudices tu olfato para la búsqueda de *bugs*.

La asignación y el paso de argumentos a función se comportan exactamente igual. Si se trata de valores se realiza una copia del valor; mientras que si se trata de referencias (a objetos) se realiza una copia de la referencia. Las variables con tipos primitivos representan valores, mientras que las de tipo no primitivo son referencias a objetos. Las cadenas son un tipo no primitivo especial, que suele tratarse como un valor en muchos lenguajes. Un objeto es el espacio de memoria al que apunta una referencia, es una instancia de una clase. Así es como distinguimos entre valor, referencia y objeto, tres conceptos diferentes que suelen confundirse. Las referencias son un concepto parecido a los punteros, aunque más sencillo y menos explícito por suerte.

Java

```
int length = 10;            // valor
int otherLength = length;   // copiado el 10
SomeClass instance = new SomeClass(); // referencia a un objeto en memoria
SomeClass other = instance; // copiada la referencia al objeto en memoria
someMethod(length);         // copiado el 10
otherMethod(instance);      // copiada la referencia
```

Los valores son inmutables, porque cualquier operación con ellos produce otro valor:

Java

```
@Test
public void values_are_immutable(){
    int length = 10;     // 10 siempre será 10
    int other = length;
```

```
    other = 20;
    asserThat(length).isEqualTo(10);
    assertThat(other).isEqualTo(20);
}
```

Aunque el lenguaje permita la reasignación de una variable, el valor original es imposible de cambiar:

Java

```java
int length = 10; // 10 es inmutable, siempre será 10
length = 20;     // length ha cambiado de valor
```

Los lenguajes funcionales prohíben la reasignación de variables, mientras que en los demás lenguajes existen palabras reservadas para definir variables no reasignables (*final* en Java, *val* en Kotlin, *const* en JavaScript y C#…):

Java

```java
final int length = 10;
length = 20; // error de compilación
```

Un *String* es un tipo integrado del lenguaje que no es primitivo, es un objeto, y en la mayoría de lenguajes se suele implementar como inmutable:

Java

```java
@Test
public void strings_are_immutable(){
    String text = "foo"; // foo siempre será foo
    String other = text;
    String another = text.substring(0,1);
    other = "bar";
    asserThat(text).isEqualTo("foo");
    assertThat(other).isEqualTo("bar");
    assertThat(another).isEqualTo("f");
}
```

Todos los métodos de la clase *String* producen una nueva instancia, jamás modifican la cadena original en Java, C# y en tantos otros lenguajes. Estos implementan los métodos de la clase *String* siguiendo el patrón *Value Object* (objeto valor). En Ruby una cadena sí se puede mutar, aunque hay que utilizar un operador explícito para hacerlo. En PHP, C++ y Swift, las cadenas también son modificables, así que ten cuidado.

Que una variable se pueda reasignar no significa que el contenido original mute, ni si quiera cuando se trata de referencias:

Java

```
@Test
public void references_are_copied_when_assigned(){
    Name first = new Name("Sarah");
    Name second = first;
    second = new Name("John");
    asserThat(first.toString()).isEqualTo("Sarah");
    assertThat(second.toString()).isEqualTo("John");
}
```

Veamos con detalle cada paso:

Java

```
Name first = new Name("Sarah");
```

- La parte izquierda de la asignación define una variable de tipo *Name*, es decir, una referencia.
- La parte derecha reserva memoria para un objeto de tipo *Name*.
- El operador de asignación (símbolo igual) hace que la referencia apunte a una zona concreta de la memoria.

Java

```
Name second = first;
```

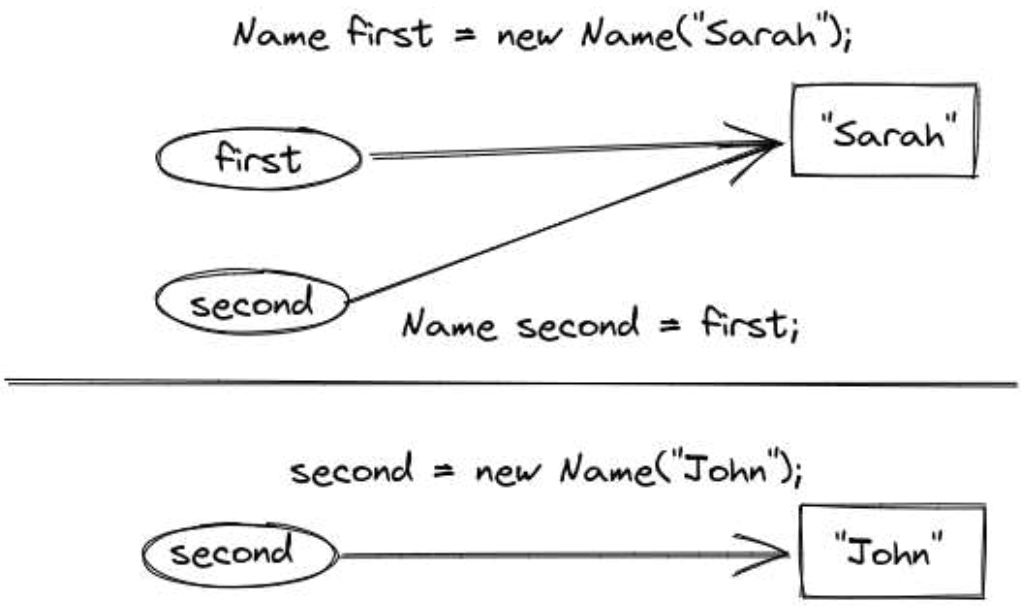
- De nuevo la parte izquierda define una referencia de tipo *Name*.
- La parte derecha es una referencia ya existente que apunta a un objeto en memoria.
- El operador de asignación (símbolo igual) hace que las dos referencias apunten a la misma zona de memoria concreta. Son como dos flechas apuntando al mismo lugar.

Java

```
second = new Name("John");
```

- La parte derecha de la asignación crea un nuevo objeto en memoria.

- La referencia *second* previamente definida ahora apunta a este nuevo objeto.
- La otra referencia (*first*) no ha cambiado, sigue apuntando a donde estaba.
- Ahora tenemos dos referencias, cada una apunta a un espacio de la memoria diferente, como puede observarse en el siguiente esquema.



Java

```
asserThat(first.toString()).isEqualTo("Sarah");
```

- Asumo que la implementación del método *toString* de la clase *Name* devuelve la misma cadena que recibe por constructor cuando se instancia. Este no es el comportamiento por defecto de la clase base *Object*, sino uno particular de la clase *Name*:

Java

```
class Name {
    private String name;
    public Name(String name){
        this.name = name;
    }
    @Override
    public String toString(){
```

```
        return name;
    }
}
```

Visto esto, te lanzo una pregunta, ¿es correcto el siguiente test?

Java

```
/*...*/
public void changeName(Name aName){
    aName = new Name("Changed!");
}
/*...*/
@Test
public void change_the_name(){
    Name name = new Name("Bob");
    changeName(name);
    assertThat(name.ToString()).isEqualTo("Changed!");
}
```

La respuesta es no, el test es incorrecto. El envío de argumentos a una función se comporta exactamente igual que la asignación, es decir, se produce una copia de la referencia. La función *changeName* es totalmente inútil y no tiene ningún efecto en absoluto (el IDE nos lo estaría chivando).

Ojo, que las mutaciones sí suceden cuando entramos al objeto y alteramos sus elementos internos, en cuyo caso lo siguiente es cierto:

Java

```
class Name {
    private String name;
    public Name(String name){
        this.name = name;
    }
    public void setName(String name){
        this.name = name;      // mutación
    }
    @Override
    public String toString(){
        return name;
    }
}
/*...*/
public void changeName(Name aName){
    aName.setName("Changed!");
}
/*...*/
@Test
```

```
public void change_the_name(){
    Name name = new Name("Bob");
    changeName(name);
    assertThat(name.ToString()).isEqualTo("Changed!");
}
```

Una cosa es cambiar la referencia para que apunte a objetos diferentes y otra bien distinta es entrar en el objeto y cambiar sus propiedades. Las palabras reservadas *final* o *const*, dan una falsa sensación de seguridad en el caso del trabajo con objetos, puesto que solamente pueden garantizar que la referencia no es reasignada, no que el objeto se mantenga intacto. El siguiente test es correcto:

Java

```
/*...*/
public void changeName(Name aName){
    aName.setName("Changed!");
}
/*...*/
@Test
public void change_the_name(){
    final Name name = new Name("Bob"); // final no evitará la mutación
    changeName(name);
    assertThat(name.ToString()).isEqualTo("Changed!");
}
```

El símbolo utilizado para acceder a las propiedades y a los métodos de un objeto es el punto (o la flecha en PHP), y su significado es: entrar a la posición de memoria en la que reside este objeto y acceder al elemento X del objeto.

Java

```
instancia.campo = "valor";
```

Sucede lo mismo con los corchetes en los *arrays* ¿Dirías que el siguiente test es correcto?

Java

```
/*...*/
public void changeArray(int[] numbers){
    numbers[0] = 777;
}
/*...*/
@Test
public void change_the_array(){
    final int[] numbers = {1,2,3};
    changeArray(numbers);
```

```
    assertThat(numbers[0]).isEqualTo(777);
}
```

Pues sí, es correcto, porque el corchete en un *array* es como el punto en un objeto, implica posicionarse en el lugar de la memoria donde reside el objeto.

Te lanzo una última pregunta: en lenguajes como Java, JavaScript, TypeScript o Python, ¿es posible escribir alguna función/método que tenga un *string* como parámetro y que sea capaz de mutar el *string* que le envíen como argumento?

Java

```
public void changeString(String aString){
    /*... ??? ...*/
}

@Test
public void change_the_string(){
    String aString = "hello";
    changeString(aString);
    assertThat(aString).isNotEqualTo("hello");
}
```

La respuesta es no. Es totalmente imposible, porque ninguno de los métodos de la clase *String* muta su estado. También es imposible con C#, excepto si defines el parámetro con la palabra reservada *ref* delante, la cual te desaconsejo salvo que sepas muy bien lo que estás haciendo y lo puedas justificar.

Espero que esta sección, junto con el capítulo de los principios malinterpretados, te ayude a escribir un código más sencillo y a que programes con seguridad.

Comparación de objetos

Cuando dos variables del mismo tipo primitivo tienen el mismo valor, no cabe duda de que son iguales a todos los efectos, lo que no está tan claro es ¿cómo se comparan las referencias?, ¿y los objetos?, ¿cuándo son iguales? Por defecto, dos variables del mismo tipo de objeto solamente son iguales si apuntan exactamente a la misma posición en la memoria:

Java

```
class Color {
```

```
    private int red;
    private int green;
    private int blue;

    public Color(int red, int green, int blue){
        this.red = red;
        this.green = green;
        this.blue = blue;
    }

    public static Color blue(){
        return new Color(240,248,255);
    }
}

@Test
public void references_are_equal_when_pointing_to_the_same_object(){
    Color color = Color.blue();
    Color color2 = color;
    assertThat(color == color2).isTrue();
}
@Test
public void identical_objects_are_not_the_same_by_default(){
    Color color = Color.blue();
    Color color2 = Color.blue();
    assertThat(color).isNotEqualTo(color2);
}
```

Cuando trabajamos con objetos que representan valores, es muy práctico redefinir la comparación para darle más sentido a las abstracciones. Esta redefinición se consigue mediante la implementación de dos métodos:

Java

```
class Color {
    private int red;
    private int green;
    private int blue;

    public Color(int red, int green, int blue){
        this.red = red;
        this.green = green;
        this.blue = blue;
    }

    public static Color blue(){
        return new Color(240,248,255);
    }

    @Override
    public boolean equals(Object other) {
        if (this == other){
```

```
            return true;
        }
        if (!(other instanceof Color)){
            return false;
        }
        Color color = (Color) other;
        return red == color.red &&
                green == color.green &&
                blue == color.blue;
    }

    @Override
    public int hashCode() {
        return Objects.hash(red, green, blue);
    }
}
/*...*/
@Test
public void objects_are_equal_when_equals_method_is_properly_redefined(){
    Color color = Color.blue();
    Color color2 = Color.blue();
    assertThat(color).isEqualTo(color2);
}
```

Estos dos métodos son *equals* y *hashCode* en el caso de Java, o bien *Equals* y *GetHashCode* en el caso de C#:

C#

```
class Color
{
    private int red;
    private int green;
    private int blue;

    public Color(int red, int green, int blue)
    {
        this.red = red;
        this.green = green;
        this.blue = blue;
    }

    public static Color Blue()
    {
        return new Color(240,248,255);
    }

    // Sobreescribir Equals:

    public override bool Equals(Object other)
    {
        return Equals(other as Color);
    }
```

```
    // En C# es típido sobrecargar Equals

    public bool Equals(Color other)
    {
        return other != null &&
               red == other.red &&
               green == other.green &&
               blue == other.blue;
    }

    public override int GetHashCode()
    {
        return HashCode.Combine(red, green, blue);
    }
}
/*...*/
[Test]
public void objects_are_equal_when_equals_method_is_redefined()
{
    var color = Color.Blue();
    var color2 = Color.Blue();
    Assert.AreEqual(color, color2);
}
```

Aunque hayamos sobreescrito estos dos métodos de la clase base *Object*, la comparación con el doble igual no cambia (porque se usa para comparar referencias) y seguirá diciendo que dos referencias son iguales solo si se trata de la misma posición en memoria:

Java

```
@Test
public void double_equal_compares_references(){
    Color color = Color.blue();
    Color color2 = Color.blue();
    assertThat(color == color2).isFalse();
}
```

Por este motivo, la forma adecuada de comparar objetos es invocando al método *equals* de manera explícita, incluso cuando utilizamos objetos de tipo *string*:

Java

```
@Test
public void objects_should_be_compared_using_equals_method(){
    Color color = Color.blue();
    Color color2 = Color.blue();
    assertThat(color.equals(color2)).isTrue();
    String text = "foo";
    String otherText = "foo";
```

```
    assertThat(text.equals(otherText)).isTrue();
}
```

Las librerías de aserciones para *testing* invocan al método *equals*, en vez de usar el operador doble igual, por lo que es preferible expresar lo anterior así:

Java

```
@Test
public void objects_should_be_compared_using_equals_method(){
    Color color = Color.blue();
    Color color2 = Color.blue();
    assertThat(color).isEqualTo(color2);
    String text = "foo";
    String otherText = "foo";
    assertThat(text).isEqualTo(otherText);
}
```

Con estas aserciones el *framework* proporciona más información sobre la comparación, que si utilizamos las aserciones *isFalse* o *isTrue*. Cuando un test falla se agradece cualquier información que nos ayude a comprenderlo. Cualquier librería que esté bien hecha va a comparar objetos invocando al método *equals*, salvo que explícitamente quiera comparar referencias (comprobar que se trata de la misma posición en memoria). En Kotlin es diferente, el significado del doble igual es como invocar al método *equals*, mientras que el triple igual es el equivalente al doble igual de Java o C#.

¿Para qué sirve el método *hashCode/GetHashCode*? Es para obtener la clave *hash* que representa a cada instancia de una clase, es decir, a cada objeto y se utiliza en aquellas operaciones que involucran tablas *hash*. Por ejemplo, cuando rellenamos un diccionario con objetos, este invocará a *hashCode* para obtener la clave de cada objeto y así poderlo guardar/recuperar:

Java

```
@Test
public void hash_value_is_used_for_hash_tables(){
    var set = new HashSet<Color>();
    set.add(Color.blue());
    set.add(Color.blue());
    set.add(Color.blue());
    assertThat(set.size()).isEqualTo(1);
}
```

Por tanto, los métodos *equals* y *hashCode*, siempre se implementan en pareja. Una bue-

na forma de implementarlos es pedirle al propio IDE que nos genere el código, ya que el asistente nos preguntará qué campos del objeto queremos emplear en la comparación y el código generado estará libre de fallos. Recurrir al IDE no significa aceptar su propuesta ciegamente, porque el código que genera no siempre tiene la forma que más nos conviene. En cualquier caso, los *bugs* en estos métodos son un auténtico quebradero de cabeza, porque pueden originar fugas de memoria y otros desastres. Tanto si lo hacemos a mano, como si autogeneramos, hay que comprender el código que dejamos escrito.

Entre otros usos, estos dos métodos se sobreescriben para implementar el patrón *Value Object* y también para evitar exponer *getters*, cuyo único objetivo es permitir la comparación de objetos. Adicionalmente, para facilitar la depuración de errores y evitar *getters*, se sobreescribe el método *toString*:

Java

```
class Color {
    /*...*/
    @Override
    public String toString() {
        return String.format("Color: rgb(%d,%d,%d)", red, green, blue);
    }
}
```

Esta definición de representación del objeto como cadena será utilizada por el *framework* de test cuando una aserción de comparación falle, de manera que podremos ver rápido qué contiene cada objeto.

Es tan habitual sobreescribir estos tres métodos que Kotlin ya incluye la implementación automática de los mismos, para casos cotidianos como los objetos que transportan datos (*data class* de Kotlin).

Copia y serialización de objetos

Ahora que comprendemos cómo funcionan las referencias a objetos y la comparación, podemos dar un paso más allá estudiando la copia de objetos (que no es lo mismo que la copia de referencias, la cual ya sabemos que se produce automáticamente al asignar o pasar argumentos). Hay dos formas de copiar objetos, la superficial y la profunda. Un objeto puede contener a otros objetos entre sus propiedades, que a su vez contengan más objetos, de manera que se pueden construir estructuras complejas como árboles o de grafos:

Java

```
class Node {
    private List<Node> children;
    private String name;
    private int weight;

    public Node(String name, int weight){
        this.name = name;
        this.weight = weight;
        children = new ArrayList<>();
    }

    public void appendChild(Node node){
        children.add(node);
    }
}
/*...*/
var tree = new Node("root", 0);
var level1a = new Node("level 1a", 1);
var level1b = new Node("level 1b", 2);
var level2a = new Node("level 2a", 3);
var level2b = new Node("level 2b", 4);
level1a.appendChild(level2a);
level1b.appendChild(level2b);
tree.appendChild(level1a);
tree.appendChild(level1b);
```

Si en algún momento tenemos que copiar la estructura a la que apunta la referencia *tree*, debemos preguntarnos si el objetivo es replicar la misma estructura al completo en su propio espacio en la memoria, o si una parte de la estructura puede constar de los mismos objetos. En el primer caso, tendremos que recorrer toda la estructura, nodo por nodo, clonando cada uno de los objetos por separado y generando la nueva estructura:

Java

```
class Node {
    /*...*/
    public Node deepCopy(){
        Node aNode = new Node(name, weight);
        for (Node child: children) {
            aNode.children.add(child.deepCopy());
        }
        return aNode;
    }
}
```

Este código produce una copia de todos y cada uno de los nodos, replicando el mismo árbol (copia profunda). Es muy diferente a este otro (copia superficial):

Java

```
class Node {
    /*...*/
    public Node shallowCopy(){
        Node aNode = new Node(name, weight);
        aNode.children = children;  // referencia apuntando al mismo lugar
        return aNode;
    }
}
```

Veamos el comportamiento con un test, añadiendo primero un *getter*:

Java

```
class Node {
    /*...*/
    public List<Node> children(){
        return children;
    }
}

@Test
public void shallow_copy_shares_references_with_the_original(){
    var tree = new Node("root", 0);
    var level1a = new Node("level 1a", 1);
    tree.appendChild(level1a);

    var copy = tree.shallowCopy();

    assertThat(copy != tree).isTrue();
    assertThat(copy.children().get(0)).isEqualTo(tree.children().get(0));
    assertThat(copy.children()).isEqualTo(tree.children());
}
```

La copia profunda genera objetos totalmente independientes, mientras que la copia superficial contiene subelementos compartidos. Ambos tipos de copia pueden dar lugar a *bugs*, fugas de memoria y otros comportamientos indeseados, si no se tiene constancia del funcionamiento de cada una. En Java, el método *clone* de la clase *Object* crea copias superficiales de los objetos, al igual que en C#, el método *MemberwiseClone*. Las copias profundas requieren implementaciones a medida.

La copia es un mecanismo usado típicamente para preservar la inmutabilidad de los objetos. Si quiero que la clase *Node* sea la única que gestiona su estado, no puedo implementar el *getter* tal cual lo he hecho, pues cualquiera alteraría la colección sin pedirle permiso:

Java

```
@Test
public void collections_are_objects_too(){
    var node = new Node("root", 0);
    var collection = node.children();
    assertThat(node.children().size()).isEqualTo(0);

    collection.add(new Node("child", 1));

    assertThat(node.children().size()).isEqualTo(1);
}
```

Si lo que me interesa es preservar la inmutabilidad de la colección, la solución más rápida es devolver una copia superficial de la misma:

Java

```
class Node {
    /*...*/
    public List<Node> children(){
        return new ArrayList<>(children); // shallow copy
    }
}

@Test
public void collections_are_a_kind_of_container(){
    var node = new Node("root", 0);
    var collection = node.children();

    collection.add(new Node("child", 1));

    assertThat(node.children().size()).isEqualTo(0);
}
```

Una copia superficial no quita que los elementos dentro de la colección puedan ser mutados por fuera:

Java

```
@Test
public void shallow_copy_is_not_deeply_immutable(){
    var node = new Node("root", 0);
    node.appendChild(new Node("child", 1));
    var collection = node.children();

    collection.get(0).setName("other name");

    assertThat(node.children().get(0).getName()).isEqualTo("other name");
}
```

Me he sacado de la manga un *setter* y un *getter* para mutar el campo *name*, lo cual, además, me sirve de refuerzo para recordarte que evites los *setter* en la medida de lo posible (para proteger el estado de los objetos). Si la inmutabilidad total fuera necesaria para el problema que trato de resolver, tendría que implementar el *getter* de esta otra manera:

Java

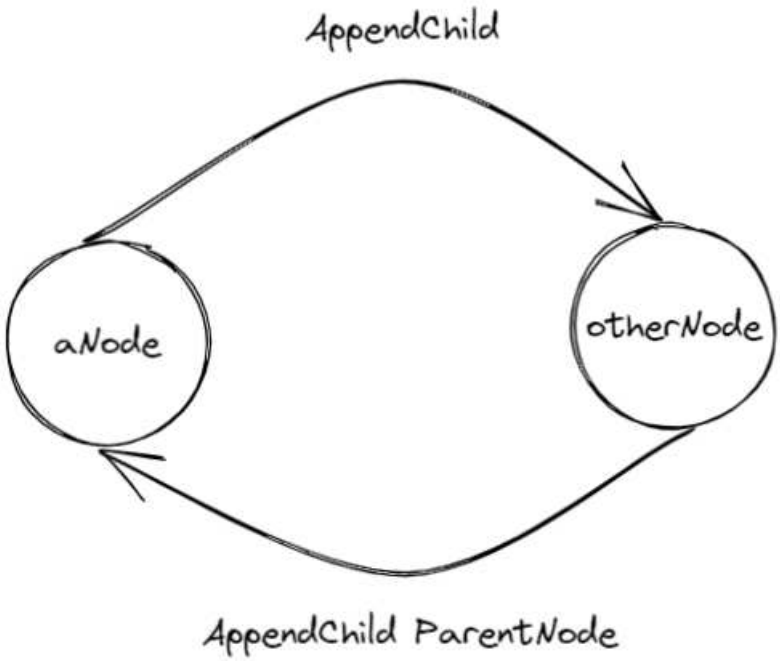
```
public List<Node> children(){
    return children.stream()
        .map(Node::deepCopy)
        .collect(Collectors.toList());
}
```

Lógicamente, la copia profunda tiene un coste; su implementación podría ser compleja o tediosa y se consume más memoria, por lo que no siempre debemos hacer copias profundas. Como de costumbre, tenemos que hacer balance de los beneficios y de los inconvenientes de cada técnica, según el contexto en el que nos encontremos.

Existe una implementación «sencilla» de las copias profundas que consiste en serializar los objetos. Serializar significa capturar una «foto» de la estructura de datos en forma de cadena de caracteres o de bytes, que permita guardarla en disco o enviarla a través de la red para su posterior reconstrucción (deserialización). La serialización más flexible es la binaria, porque aguanta cualquier tipo de estructura de datos; convierte el objeto en *bytes* que pueden volver a ser convertidos en objeto. Cada lenguaje incluye sus propias herramientas de serialización binaria, que típicamente pasa por implementar ciertas interfaces. El inconveniente del formato binario es que tanto la serialización como la deserialización debe hacerlas el mismo sistema, siendo imposible serializar en Java y deserializar en C# o Python. Por eso, para interoperar con sistemas heterogéneos, se utiliza la serialización a XML o a JSON, es decir, se convierte la estructura de datos en un formato XML o en un formato JSON, respectivamente. Estos formatos tienen una limitación expresiva, la incapacidad de representar estructuras circulares como la siguiente:

Java

```
Node aNode = new Node("a", 1);
Node otherNode = new Node("b", 1);
aNode.appendChild(otherNode);
otherNode.appendChild(aNode);
```



Si intentamos serializar *aNode* u *otherNode* a XML o a JSON, nos encontraremos una excepción en tiempo de ejecución que dice que es imposible representar una referencia circular con estos formatos. Tiene sentido si te paras a pensarlo, ¿cómo escribirías un XML o un JSON circular?

Es cierto que en el día a día no solemos encontrarnos con grandes retos en cuanto a copias de objetos, problemas de referencias o de serialización (sobre todo, porque los *frameworks* trabajan muy bien la fontanería habitual), sin embargo, conocer el funcionamiento aporta un valor diferencial cuando surge el problema (y para evitar el problema en primer lugar). Si quieres ser una persona resolutiva en los casos fáciles y en los difíciles, necesitas conocer estas bases de la gestión de memoria en lenguajes modernos. Las fugas de memoria (*memory leaks*) se producen cuando el recolector de basura no puede liberar la memoria ocupada por un objeto (aunque ya no se use), porque está siendo referenciado por una variable vigente. Podría ocurrir, por ejemplo, si tengo una variable *static* global que apunta a objetos. Es un defecto bastante común en código multihilo y en código asíncrono (en varios lenguajes), porque a veces quien programa no es consciente de cómo funcionan las

referencias o no se da cuenta.

Tramos de concentración de accidentes

En la señalización de tráfico existen advertencias de tramos de concentración de accidentes, típicamente zonas en las que los conductores nos confiamos demasiado o por los que circulamos a demasiada velocidad, y que acaban siendo un foco de trágicos accidentes. El título de este apartado es una metáfora que nos sirve para hablar de aquellas partes del código donde más solemos equivocarnos al programar, con la intención de concienciar sobre el riesgo que a menudo no es percibido. Es de especial riesgo trabajar con:

- Gestión del estado
- Expresiones booleanas combinadas
- Intervalos y rangos
- Asincronía y concurrencia

Gestión del estado

Los programas giran en torno a los datos, incluso una calculadora científica guarda datos en la memoria para operar con ellos. A los datos que están cargados en la memoria de la aplicación se les llama estado. Este sufre transformaciones durante la ejecución del programa como consecuencia de las acciones que se realizan. Los cambios de estado son un punto de concentración de *bugs* habitual, sobre todo, porque razonar sobre ellos es más difícil que razonar sobre un cálculo concreto. Por más que sea compleja la fórmula en la que se basa un cálculo, no está influenciada por los estados previos, mientras que una transición de estados a menudo tiene que ver con las transiciones anteriores. Los cambios de estado de una aplicación pueden ser tan complejos como la teoría de grafos. Incluso, algo tan aparentemente simple como programar la detección y gestión del *click* y del *doble click* de un ratón, puede sorprendernos en cuanto a su complejidad. A modo de ejercicio te invito a implementar los métodos comentados de este ejemplo, verás qué difícil es gestionar cambios de estado.

Java

```java
class Mouse {
    private List<MouseEventListener> listeners = new ArrayList<>();
```

```
    private final long timeWindowInMillisecondsForDoubleClick = 500;

    public void pressLeftButton(long currentTimeInMilliseconds){
        /*... debe notificar a los suscriptores ...*/
        /*... y gestionar el estado ...*/
    }
    public void releaseLeftButton(long currentTimeInMilliseconds){
        /*... debe notificar a los suscriptores ...*/
        /*... y gestionar el estado ...*/
    }
    public void move(Position from, Position to,long
        currentTimeInMilliseconds){
        /*... debe notificar a los suscriptores ...*/
        /*... y gestionar el estado ...*/
    }

    public void subscribe(MouseEventListener listener){
        listeners.add(listener);
    }

    private void notifySubscribers(MouseEventType eventType) {
        listeners.forEach(listener -> listener.handleMouseEvent(eventType));
    }
}

enum MouseEventType {
    SingleClick,
    DoubleClick,
    TripleClick,
    Drag,
    Drop
}

interface MouseEventListener {
    void handleMouseEvent(MouseEventType eventType);
}
```

La solución universal en el mundo de la informática es reiniciar: reiniciar la computadora, la *app*, el *router*, el *smartphone*,... lo primero que te pregunta cualquier técnico de microinformática o comunicaciones es, «¿ya probaste a reiniciar el ordenador?» ¿Qué sería de la humanidad si reiniciar no lo arreglase todo?, ¿por qué crees que llega el punto en que hace falta reiniciar? Pues, porque en algún momento se alcanza un estado al que teóricamente no debería haberse llegado si no fuera por los *bugs* en el código. Cuanto más complejo sea el sistema, habrá más combinaciones de transiciones entre estados, y, por tanto, mayor posibilidad de dejar escenarios sin testar. Reiniciar provoca que todo vuelva a un estado inicial que está bien probado, y por eso «lo arregla todo».

Cuando trabajamos con código que realiza cambios en el estado, el porcentaje de cobertura de test no es un dato que aporte suficiente fiabilidad, porque un par de test pueden

cubrir el 100 % del código y, sin embargo, no ejercitar cambios de estado mal implementados. Estamos ante problemas potencialmente combinatorios. Practicar TDD me ayuda a pensar en más casos de prueba que podrían fallar, ya que me obliga a implementar cada cambio de estado de manera focalizada y así termino con una batería de respaldo más sólida. Podemos recurrir a técnicas como los test basados en propiedades (con TDD o sin TDD), que son excelentes para encontrar fallos en problemas combinatorios. De todas formas, mi consejo es hacer un buen ejercicio de pruebas de exploración manuales en las partes de la aplicación que implican cambios de estado. Siempre es bienvenida la exploración manual y curiosa de una persona con buen olfato para romper *software*. Ten mucho cuidado con la refactorización de código que muta estado, no la hagas a la ligera, aunque el código te parezca trivial.

Cuando hablamos de gestión del estado nos referimos a código que asigna valores a variables, es decir, a las operaciones donde típicamente hay un operador de asignación, o que añaden o quitan elementos de colecciones:

Java

```
timeOfmostRecentClick = currentTimeInMilliseconds;
clickCount++;
listeners.add(listener);
items.remove(index);
```

Hay que tener especial cuidado con estructuras de datos complejas que tienen objetos anidados, porque ya vimos que las copias de dichos objetos no son profundas salvo que las hagamos explícitamente.

Java

```
order.description = anotherDescription; // <- peligro
employee.address.street = anotherStreetName; // <- más peligro
```

La situación empeora si las variables se asignan de manera concurrente o si hay algún artefacto que apunte a ellas para actualizar otras partes del sistema, como ocurre, por ejemplo, cuando utilizamos la técnica de *data binding* para las vistas. Hay bibliotecas como *React* que no actualizan la vista si el nuevo estado no es un objeto con una referencia distinta, pero incluso eso no quita que por dentro haya estructuras anidadas con varias referencias apuntando a ellas. Cuando se utilizan lenguajes funcionales es muy común evitar los cambios en el estado mediante la copia y la generación de nuevas estructuras a partir del estado anterior. Cuando trabajamos con orientación a objetos podemos perfectamen-

te aplicar estas mismas técnicas o también podemos hacer que los objetos se encarguen de su propio estado, siempre y cuando entendamos bien cómo funcionan las referencias y los valores.

Expresiones booleanas combinadas

El siguiente tramo de concentración de accidentes está en el álgebra de Boole; al igual que nos equivocamos constantemente entre la izquierda y la derecha, nos equivocamos entre verdadero y falso, es muy humano. A nuestro cerebro no se le da bien computar la combinación de operadores lógicos:

Java

```java
if (!isSouldOut || !isOutOfStock){/*...*/}
```

Este tipo de condicionales tienen que ser bien cubiertas por test automáticos, como mínimo tener uno para el caso en que la expresión evalúa como verdadera, y otro en el que evalúa como falsa. Computar mentalmente estas expresiones es cansino y propenso a errores, por lo que recomiendo ponerles un nombre mediante variables explicativas:

Java

```java
boolean isOnSale = !isSouldOut || !isOutOfStock;
if (isOnSale){
    /*...*/
}
```

O bien extrayendo métodos:

Java

```java
if (isOnSale()){
    /*...*/
}
```

De las dos estrategias, elijo la que me parezca más sencilla y aporte mayor legibilidad en cada caso. Un método oculta la implementación, posiblemente reduciendo el ruido en el lugar donde se invoca. Si es relevante que la implementación se vea, lo dejo como variable explicativa, mientras que si prefiero ocultar el detalle, utilizo un método. En ambos casos, el resultado al leer la sentencia *if* es mucho más humano. En este ejemplo me parece

más simple utilizar una variable explicativa, ya que no hay problema en dejar a la vista el cómputo (de hecho, me parece útil que se vea).

Cuando hay un método booleano que siempre se consume con el símbolo de negación delante, puede ser práctico cambiar su implementación y añadir la negación al propio nombre del método. Es decir, en lugar de esto:

Java

```java
if (!doesExist(user)){/*...*/}
```

Cambiarlo por esto:

Java

```java
if (doesNotExist(user))
```

Lo que conseguimos es eliminar el cálculo mental, reducir la carga cognitiva. Cuando una llamada a un método lleva siempre una negación por delante, hacer este cambio puede compensar. Para realizarlo la implementación del método debe ser invertida (si antes devolvía verdadero, ahora devuelve falso y viceversa), lo que implica riesgo de equivocarnos. Por suerte, el IDE puede invertir por nosotros de manera automática y sin equivocarse; poniendo el cursor en la palabra *if*, aparecen enseguida sugerencias del IDE, como invertir la condición. Si invertir la implementación no es tan sencillo como invertir un *if*, podemos crear un nuevo método que llama al anterior y lo niega, para encapsular la negación:

Java

```java
public void doesNotExist(User user){
    return !doesExist(user);
}
private void doesExist(User user){
    /*...*/
}
```

No digo que siempre evitemos usar el símbolo de negación en las expresiones booleanas, ni mucho menos. Todas estas ideas deben aplicarse con sentido común, cuando verdaderamente simplifiquen la lectura del código y reduzcan la probabilidad de error. Al igual que con los cambios de estado, si vas a refactorizar expresiones booleanas y condicionales, ponle cuatro ojos al asunto (*pair programming*) y mucho cuidado. Que sepas que en esto se equivoca todo el mundo, *juniors*, *seniors* y cualquier otra etiqueta que quieras poner.

Intervalos y rangos

Los índices, apuntando fuera de los límites de un intervalo o quedándose cortos, son otro clásico tramo de concentración de accidentes. Desde el lenguaje BCPL en adelante, los *arrays* han sido diseñados de manera que su primer elemento es el cero, algo muy conveniente en matemáticas, aunque no encaja con nuestra manera natural de contar. A menudo, confundimos el número total de elementos con el índice del último elemento del *array*, al igual que nos entran dudas sobre si la función *substring* utiliza intervalos abiertos o cerrados. Conviene asegurarse las veces que haga falta de que el comportamiento de las funciones que implican índices, rangos o intervalos, es el que esperamos. Una forma rápida de salir de dudas es hacer pruebas sobre un *REPL online* o de consola. Para evitar errores de índices, lo mejor es no tenerlos que utilizar, por eso los lenguajes modernos incluyen varios tipos de bucle como *forEach*, además de las formas declarativas como *map*. Es cierto que suelo utilizar el clásico bucle con índice cuando es la solución más rápida para ir de test rojo a test verde, practicando TDD, aunque luego en la etapa de refactor suelo cambiarlo por otra forma de bucle. No obstante, si un bucle con índice fuese la solución más simple, podría ser la más adecuada.

El lenguaje Python siempre ha sido muy intuitivo y cómodo para trabajar con listas, diccionarios, tuplas, etc., ya que su sintaxis es muy gráfica, sencilla y los bucles son potentes y expresivos. Cuando tengo que trabajar con *scripts* que utilizan listas u otras colecciones, Python es mi opción preferida.

He visto (y cometido) muchos *bugs* errando en los cálculos de intervalos y de índices, por lo que mi recomendación nuevamente es que probemos bien el código con test automáticos y manuales. Ten mucho cuidado al refactorizar, porque fallar en estos cálculos es mucho más habitual de lo que parece. He conocido casos de pérdidas económicas considerables por dejarse un elemento de una colección atrás. Los test basados en propiedades también pueden ayudarnos en estos casos.

Asincronía y concurrencia

Por último, pero no menos importante, me gustaría hablar de asincronía y concurrencia. Tanto en aplicaciones de escritorio nativo como en aplicaciones móviles (iOS, Android, etc.), es imprescindible utilizar varios hilos para que la GUI (*Graphic User Interface*) no se quede congelada, mientras se realizan operaciones que tardan en ejecutarse más de medio segundo. Esto es muy importante para la UX (*User Experience*). En la web, gracias a

la estandarización, cada vez más aplicaciones proporcionan una experiencia de usuario similar, para lo cual aprovechan los mecanismos de asincronía de los navegadores.

A nuestro cerebro no se le da bien pensar en tareas que se ejecutan en paralelo o simplemente en momentos del tiempo que no son consecutivos. Aquí, los fallos son más difíciles de entender y de reparar. Los lenguajes han ido añadiendo capas de azúcar sintáctico para simplificar la forma de expresar construcciones asíncronas, por ejemplo, mediante *async/await*, tanto en C# como en JavaScript/TypeScript (el significado es parecido, pero no es el mismo en ambos lenguajes). No todo el mundo conoce lo que ocurre bajo la superficie cuando se utiliza *async* y *await*, es decir, quizás se intuye el comportamiento del bloque que usa dicha construcción, pero no se comprende lo que sucede en realidad. A menudo, no se sabe cómo gestionar las excepciones que se producen en funciones asíncronas y las consecuencias de una mala gestión pueden acarrear fallos de los que no somos ni conscientes (los sufren los usuarios). La falta de comprensión profunda de la asincronía, provoca *bugs* difíciles de reproducir y desperdicio de recursos de los procesadores multinúcleo que tenemos hoy. Al igual que el uso de un *framework* no exime de entender lo que hace por debajo, el uso de construcciones asíncronas tampoco. Su cometido es facilitar la lectura/escritura, no eliminar el raciocinio ni el conocimiento. Mi recomendación es que te tomes el tiempo que necesites para experimentar con los mecanismos de asincronía que te ofrece el lenguaje o la librería que uses, hasta asegurarte de que controlas lo que ocurre y ganes la confianza necesaria para usarlos en tus programas. En su día, para aprenderlo en C#, me fue muy útil el libro de Alex Davies. Aunque la concurrencia y los algoritmos paralelos son algo que se escapa del alcance de este libro, mi recomendación es que leas sobre problemas típicos como interbloqueos (*deadlock*) o condiciones de carrera, al menos para saber lo que te puedes encontrar. También te recomiendo leer sobre la programación reactiva y las extensiones reactivas, para que conozcas el poder de los observables ante problemas asíncronos.

Código resiliente

Escribimos código tanto para el uso esperado y deseado del *software* (*happy path*), como para el uso inadecuado o accidental (*edge cases*), porque como vimos al comienzo de este capítulo, todo lo que puede salir mal, saldrá mal. Si no tenemos en cuenta los casos extremos, nuestras aplicaciones se estrellarán o se morirán congeladas ante el uso inesperado de las mismas. Por otra parte, si diseñamos librerías o *frameworks* y no tenemos en cuenta el uso incorrecto, se producirán errores indescifrables o desconcertantes para quienes los

están usando. La tolerancia a fallos de un sistema y la resiliencia, son atributos de calidad que forman parte de la arquitectura del *software*, por lo que deben formar parte del diseño inicial, o por lo menos deben estar en los planes del equipo de desarrollo. Este bloque no trata de la arquitectura a alto nivel, sino de la resiliencia en las pequeñas decisiones que tomamos al escribir cada línea de código, cada función.

Cuando una línea de código evalúa una variable y descubre que tiene un valor inadecuado, hay varias opciones para afrontar la situación. Una opción es la defensiva, que típicamente consiste en abortar la ejecución del camino en curso. Otra opción es la resiliente, que consiste en recuperarse de la situación cuando es posible y seguir el curso de ejecución normal. Dependerá de si el valor es «un caso imposible» o «un caso aceptable». La consideración de valor imposible o aceptable, se hace en base al principio de menor sorpresa, el cual se verá afectado por el contexto, ya que no es lo mismo diseñar *software* vertical (aplicaciones de propósito específico) que *software* horizontal (librerías, *frameworks*, lenguajes, … de propósito general).

Una de las mejores formas de evitar valores incongruentes es definir y utilizar nuestros propios tipos específicos. Aun así, habrá ocasiones en las que una referencia pueda ser nula y esto pueda interpretarse como un caso imposible, o como un caso aceptable, según el contexto.

Pongamos por caso una función que recibe un texto y lo parte en tantas líneas como sea necesario, para que ninguna ocupe un ancho mayor de *N* caracteres. Es una función que está dentro del código de un editor de texto, como podría ser *NotePad*, *gedit* o *TextEdit*:

Java

```java
public String wordWrap(String text, int columnWidth){
    /* ... */
}
/*...*/
@Test
public void wraps_the_text_adjusting_lines_to_column_width(){
    asserThat(wordWrap("hello world", 6)).isEqualTo("hello\nworld");
    asserThat(wordWrap("otorrino", 2)).isEqualTo("ot\nor\nri\nno");
}
```

Como usamos tipos integrados para sus parámetros, podrían llegar argumentos incongruentes:

- Que *text* sea nulo o vacío.
- Que *columnWidth* sea negativo.

- Que *columnWidth* sea cero.

La peor solución a este problema, la de mayor sorpresa para cualquiera que invoque a la función, sería devolver una cadena con un mensaje de error adentro: *Error: text no puede ser nulo*. Te aseguro que he visto cosas así, por eso te lo cuento. El retorno de una función que solo devuelve un valor (en Java son todas así, mientras que en Python, Ruby y Go, se pueden devolver múltiples valores) está reservado para valores congruentes, no para seguir propagando incoherencias por la aplicación. Más que nada, porque se lo pondríamos extremadamente complicado a quien invoca a la función (¿tendría que buscar si la subcadena *Error* está en la respuesta para tomar decisiones?). Si la función es de tipo *String* (o sea que devuelve un tipo *String*), su contenido será la respuesta a la pregunta que hizo el código llamador y no un mensaje de error. Los errores de los que no podemos recuperarnos se gestionan mediante excepciones en aquellos lenguajes con soporte para excepciones (en lenguaje C no lo hay), o de forma alternativa con tipos específicos. Así, el código llamador no tiene que hacer un escrutinio de la respuesta para saber si puede pintar el texto devuelto en el editor, sino que directamente lo pinta. Cuando no hay excepciones, como sucede en lenguaje Go, se utiliza un segundo valor de retorno para entregar el error y así distinguirlo de la respuesta.

Superado el tema de la consistencia en el retorno, un enfoque puramente defensivo lanzaría una excepción en tiempo de ejecución ante cualquiera de los tres casos de arriba. La actitud defensiva dice, «si no me envías argumentos coherentes, yo no sigo trabajando, renuncio. Búscate la vida para enviarme las cosas bien». Me recuerda a la relación entre compañeros que no se toleran mucho. Ciertamente, hay veces en que no queda más remedio que lanzar excepciones para responder a incongruencias, más que nada para evitar sorprender a la gente. Fíjate, por ejemplo, qué comportamiento tan sorprendente tiene JavaScript en algunas situaciones:

Javascript

```
it('definitely surprises you', () => {
    expect(parseInt('1a3b')).toBe(1);
    expect("a" + 30).toBe("a30");
    expect("a" - 30).toBeNaN();
    expect({a:1} + {b:2}).toBe('[object Object][object Object]');
    expect(3 / 'a').toBeNaN();
    expect((3 / 'a') == (3 / 'a')).toBeFalsy();
});
```

Algunos de estos casos provocarían excepciones en tiempo de ejecución en Java o direc-

tamente no compilarían. Sin embargo, JavaScript las acepta y las interpreta de una manera que ni siquiera es lógicamente consistente, por lo que causa gran sorpresa. La moraleja es que antes de intentar programar para la resiliencia, tragando con valores a los que no podemos dar ningún sentido lógico, debemos programar de manera defensiva, rechazando las entradas mediante excepciones, errores u otros tipos (según el lenguaje y el paradigma).

En este ejemplo, si el parámetro *columnWidth* viene con un valor menor que uno, ¿podemos producir alguna respuesta con sentido?, ¿qué devolvemos ante un valor negativo?, ¿y ante un cero? No se me ocurre ninguna respuesta lógica a estos valores, porque no tienen ningún sentido en esta función. No sería resiliente que me invente como respuesta el mismo texto de entrada, *null* o vacío. Todas esas respuestas serían arbitrarias y sorprendentes. Lo que haré será programar el lanzamiento de una excepción de tipo *InvalidArgumentException* o similar, con un mensaje que diga que no tiene sentido definir un ancho de columna menor que uno. No queda más remedio que detener la ejecución en este caso. La mejor forma de evitar que lleguen valores negativos en un argumento que solo tiene sentido para números mayores que cero, sería mediante un diseño respaldado por el compilador. Esto se consigue definiendo nuestro propio tipo *ColumnWidth*, en lugar de usar el tipo primitivo *int*.

Ahora bien, no siempre hay que escribir código tajante, a veces se agradece que el código sea transigente y resiliente. Asumiendo que *columnWidth* fuese un número mayor que cero, tanto si *text* es nulo, como si es una cadena vacía, ¿causaría sorpresa a quien invoca a la función que retorne una cadena vacía? Es decir:

Java

```
@Test
public void empty_and_null_are_considered_empty_text(){
    asserThat(wordWrap(null, 90)).isEqualTo("");
    asserThat(wordWrap("", 90)).isEqualTo("");
}
```

A mí no me parece sorprendente devolver un texto vacío ante una entrada vacía. Desconozco el motivo por el que ha llegado el valor *null* como texto, y quizá se trate de un error en otra parte del código, pero ¿tiene esta función la responsabilidad de ser tajante sobre lo que pudiera estar mal en otra parte del código?, estando dentro de la función, ¿cómo podemos tener seguridad de que efectivamente hay un *bug* en algún sitio? Me parece mucho suponer. La responsabilidad de la función es devolver texto con líneas, cuyo ancho no sobrepase el especificado. Devolver texto vacío cuando se recibe nulo hace al programa resiliente, porque encauza la ejecución por el camino normal, devolviendo un valor con

sentido; absorbe un valor incongruente, transformándolo en un valor posible. En el caso de que ese valor nulo fuera fruto de un *bug* en otro lugar, el texto vacío podría terminar en la pantalla del editor y sorprender a quién está usándolo, pero ¿cuál es la alternativa? Realmente, la experiencia de usuario no iba a ser mejor si lanzamos una excepción:

- Si la excepción no se gestiona en ningún otro lugar, se termina el programa de forma inesperada (se cierra la *app* súbitamente).
- Si la excepción se captura y se pinta un mensaje en la pantalla que diga «Error 342: ha ocurrido un error inesperado», no hay nada que la usuaria puede hacer, lo cual provoca frustración.

Quizá las tres opciones sean igualmente frustrantes de cara al humano que está usando el *software*, pero de cara al mantenimiento sí hay diferencias. Programar siempre de manera defensiva y tajante puede tener un efecto perjudicial, porque el estilo defensivo se propaga rápidamente por todos los artefactos del código, a causa del miedo a las respuestas tajantes. Sucede que ante la duda sobre el posible lanzamiento de excepciones en un método o función, los programadores envuelven cada llamada en un bloque *try-catch*, al mismo tiempo que optan por lanzar excepciones ante cualquier valor imperfecto. Se genera la desconfianza sistemática de cualquier artefacto que se va a consumir, ensuciando el código con comprobaciones innecesarias por todas partes. La intransigencia genera desconfianza y polución.

Un ejemplo universal de código resiliente lo tenemos en la implementación típica del método *equals*, cuando lo sobreescribimos en nuestras clases. Lo vimos anteriormente en la sección de comparación de objetos, se trata de un método booleano que dice si dos objetos son iguales o no. Lo primero que se hace es comprobar si el objeto que se recibe por parámetro es del mismo tipo, y en caso de que no sea así, se retorna falso. Desde el punto de vista externo a método, ¿tiene sentido comparar tipos diferentes?, ¿*Invoice* y *ShoppingCart*? Desde luego que una comparación de objetos tan disjuntos no tiene sentido, tiene pinta de *bug*, pero la implementación de *equals* no juzga lo que está pasando por fuera. La responsabilidad del método es determinar si dos objetos son iguales, independientemente de que tenga sentido lo que se reciba, porque en cualquier caso incongruente la respuesta sencillamente es *false*. Lo importante es que la salida del método tenga sentido, no que la entrada tenga sentido (y si es un método *void*, es decir, sin salida directa, pues que su comportamiento tenga sentido).

En cierta manera, lo que hacemos al escribir código resiliente es sanear los valores que recibimos para evitar males mayores. No todas las funciones o métodos deben chequear

los valores de sus parámetros al comienzo para defenderse o para sanearlos, sino que dependerá de su nivel de exposición. Me explico; un método público tiene mayor exposición que uno privado, a la par que un método público de una librería de propósito general, tiene mayor exposición que uno del dominio de mi aplicación específica. Dentro del núcleo de negocio de mi aplicación, puedo utilizar mis propios tipos, pero además puedo confiar en que las capas más externas ya han saneado o rechazado valores inadecuados. Dicho núcleo no tiene la responsabilidad de validación de valores, sino que la delega para ser conciso y específico sobre el problema de negocio. No es necesario ni deseable que todos los métodos hagan validaciones de sus entradas y sus salidas. Cuando sí tiene sentido validar concienzudamente es desarrollando *frameworks* y librerías, pues están expuestos a usos de todo tipo (a veces, incorrectos por falta de entendimiento o porque no se lee la documentación). El mensaje de error que proviene de la ejecución de un método de una librería va a ser lo primero que vayamos a ir a buscar en Internet, copiando y pegando el mensaje tal cual. No será nada útil que el error sea una excepción no controlada (por ejemplo, *NullPointerException*), con una traza que muestre las entrañas de la librería. Será mucho más útil un error controlado que diga «*Error: parameter [name] can't be null*». Es la forma más rápida de hacer saber a quien consume la librería que hay un problema en su código y no en la librería. Por esto, las librerías y *frameworks* deben ser más exigentes con la validación, y también más tajantes con la interpretación de los valores, puesto que tienen menos contexto para poderles dar una interpretación con sentido. Como hemos mencionado varias veces a lo largo del libro, escribir código de propósito general requiere unas consideraciones diferentes que las del código específico de una aplicación empresarial.

De este ejemplo *wordWrap*, nos queda una pregunta más por responder: ¿qué sucede si ante un texto nulo la función devuelve nulo? Esto no sería ni programación defensiva ni resiliente, sino desentendida, sería darle una patada al problema para que lo resuelva otro. Devolver *null* hace que cunda el miedo a las excepciones de tipo *NullPointerException*, y, por tanto, nos lleva otra vez a programación defensiva que se expande por la aplicación. Hay que evitar producir o propagar el valor nulo. Para ello, una solución es absorberlo de manera resiliente como en este ejemplo, en que podemos devolver la cadena vacía. En otros casos, existen otras soluciones como el patrón *Null Object* o los tipos *Optional o Maybe*.

Ausencia de valores

Patrón Objeto Nulo

Devolver nulos es un problema, porque obliga al llamador a preguntar si la respuesta es nula para tomar la decisión de qué flujo debe continuar. Una decisión muy inteligente del diseño del lenguaje Kotlin es que por defecto los objetos no pueden ser nulos, el compilador así lo verifica. Es cierto que existe la opción de definir variables como *posibles nulos* de manera explícita, entre otras cosas por compatibilidad con código legado. Que por defecto no haya nulos, evita accidentes. Si utilizas Kotlin en un proyecto nuevo, aprovéchate de que por defecto los tipos que defines no pueden ser nulos y mantenlos así, te ahorrará problemas (no pongas la interrogación al lado del nombre de la clase). Ocurre lo mismo con C#, desde la versión 8.0 en adelante (Kotlin y C# comparten muchas características). Muchos lenguajes añaden azúcar sintáticto para preguntar por los nulos, por ejemplo, mediante operadores como *Elvis operator* o *null coalescing operator*, lo cual no significa que nos quiten de encima las comprobaciones ni los peligros de manejar nulos, sino que simplemente simplifican la sintaxis.

El patrón *null object* o patrón objeto nulo, es un viejo remedio a las preguntas sobre los nulos y a las excepciones causadas por ellos. En lugar de devolver *null*, devolvemos un objeto que hace la función de valor nulo conceptualmente, mediante subclases con implementaciones inocuas. Veamos un caso donde la ausencia de color se representa con una clase concreta:

Java

```java
class Component {
    private Color bgColor;
    /*...*/
    public Color backgroundColor(){
        if (bgColor == null){
            return new NoColor();
        }
        return bgColor;
    }
    /*...*/
}

class Color {
    protected String code;

    public Color(String code){
        this.code = code;
    }
```

```
    public Color blendWith(Color otherColor){
        /*...*/
    }
}

class NoColor extends Color {

    public NoColor(){
        super("NoColor");
    }

    @Override
    public Color blendWith(Color otherColor){
        return new Color(otherColor.code);
    }
    /*...*/
}
```

La implementación del «color nulo» está en la clase *NoColor*. Su método para mezclar colores se limita a devolver el otro color; si mezclo azul con nada, obtengo azul. De esta manera, cuando alguien pida el color de fondo al componente, puede utilizarlo incluso si no hay uno predefinido, sin necesidad de preguntar por el valor nulo. Este patrón es útil cuando nos permite utilizar el objeto obtenido sin distinciones.

Un ejemplo básico del patrón lo podemos encontrar en las colecciones vacías:

Java

```
public List<Task> findAllTasks(){
    if (taskCount == 0){
        return new ArrayList<>();
    }
    /*...*/
}
```

En caso de no haber tareas, devolvemos una lista vacía en lugar de *null*. La API de la lista es la misma tanto si está vacía como si contiene elementos, con lo cual su uso es inocuo:

Java

```
for (var task: findAllTasks()){
    /*...
      Estas  lineas no se ejecutan
      si  la   lista  está  vacia.
      No hay pelibro de excepción.
    */
}
```

Veamos otro ejemplo con subclases:

Java

```java
class TaskScheduler {
    private List<Task> tasks = new ArrayList<>();
    /*...*/
    public Task mostRecentTask(){
        if (tasks.size() == 0){
            return new NullTask();
        }
        return tasks.get(tasks.size() -1);
    }
    /*...*/
}

class TaskRunner {
    private TaskScheduler scheduler;

    public TaskRunner(TaskScheduler scheduler) {
        this.scheduler = scheduler;
    }

    public void executeMostRecentTask(){
        try {
           scheduler.mostRecentTask().execute();
        } catch (TaskExecutionException e){
            /*...*/
        }
    }
    /*...*/
}

class TaskExecutionException extends RuntimeException {
    /*...*/
}

interface Task {
    void execute();
}

class NullTask implements Task {
    @Override
    public void execute() {
        /* this is empty on purpose */
    }
    /*...*/
}
```

Los patrones no encajan en todas partes, este no es adecuado para responder a la pregunta de si un elemento existe en un conjunto. Ejemplo:

Java

```
public User getBy(Id id){
    User user = fetchUserFromUsersTable(id.toString());
    if (user == null){
        return NotAnUser(id);
    }
    /*...*/
}
```

Si el usuario con el identificador buscado no existe y necesito saberlo explíticamente, devolver una instancia de *NotAnUser*, *UnknownUser, NullUser* o similar, será enrevesado, porque obliga al código llamador a preguntar:

Java

```
User user = getBy(someId);
if (user.isValid()){
   /*...*/
}
```

Estaría obligado a añadir un método a la jerarquía con nombre *isValid*, *isNull* o *exists*, para distinguir el caso especial nulo/inexistente, o bien obligando a preguntar por el subtipo (incumpliendo con el principio de sustitución de Liskov). La gracia de este patrón es no tener que hacer preguntas acerca del tipo obtenido, sino usarlo directamente. Le sacamos el máximo partido cuando el código llamador de un método que devuelve objetos «nulos», ignora totalmente este detalle. Si vamos a tener que hacer preguntas, es más simple recurrir a los tipos que vemos a continuación.

El tipo *Optional/Maybe*

Cuando no se pueda o no convenga ocultar el posible objeto nulo en la respuesta de un método, pero aun así queramos eludir problemas con los nulos, el tipo *Optional* es una estupenda solución. Se llama *Maybe* en algunos lenguajes como Haskell o en librerías para C#, mientras que en Java se llama *Optional*, en la librería Arrow para Kotlin se llama *Option,* y en F# se llama *option*.

Java

```
class UserRepository {
    public Optional<User> findBy(Id id){
        User user = findUserFromUsersTable(id.toString());
```

```
        return Optional.ofNullable(user);
    }

    private User findUserFromUsersTable(String id){
        /*...*/
    }

    public void update(User user) {
        /*...*/
    }
}

class UserService {
    private UserRepository repository;

    public UserService(UserRepository repository) {
        this.repository = repository;
    }

    public void changePassword(Id userId, String newPassword, String
        oldPassword){
        Optional<User> optionalUser = repository.findBy(userId);
        optionalUser.ifPresent(user -> {
            user.changePassword(hash(newPassword), hash(oldPassword));
            repository.update(user);
        });
    }

    private String hash(String password){
        /*...*/
    }
}
```

A priori, la solución no parece tan diferente a devolver *null* y a preguntar por él, sin embargo, hay una diferencia importante; el apoyo del compilador. Mediante *Optional*, el código llamador sabe perfectamente a qué atenerse, porque lo tiene en la firma del método, sin sorpresas ¿Debemos entonces usar tipos opcionales para todo? No, ya sabes que no hay receta universal para todo. Es más sencillo manejar funciones que siempre devuelven valores concretos, que opcionales, por eso reservamos los opcionales para los casos en los que no queda más remedio que gestionar la ausencia de valores. Desde luego, cuando hay ausencia es mucho mejor usar *Optional* que *null* en los métodos públicos del núcleo de negocio. Como habrás observado en los ejemplos, los métodos privados o protegidos siguen devolviendo *null*, ¿por qué será? La razón es que dentro de la clase hay suficiente cohesión como para gestionar *null* de forma segura y es lo más simple (salvo que utilice alguna librería que ya me devuelva opcionales desde la base de datos). Ya sabes que buscamos constantemente el equilibrio entre principios.

Optional es un tipo compuesto que además tiene la ventaja de proporcionarnos una API que incluye funciones como *map, flatMap, filter*... lo cual nos permite encadenar llamadas y utilizar un estilo declarativo:

Java

```
public ResponseEntity<UserData> findUser(String id){
    return service.findUser(new Id(id))
            .map(user -> new ResponseEntity<>(
-                toDataTransferObject(user), HttpStatus.OK))
            .orElse(new ResponseEntity<>(
                HttpStatus.NOT_FOUND));
}
```

El método *map* ejecutará la función que recibe como argumento, si dicho opcional contiene un valor (si no está vacío), y envolverá su resultado en otro opcional. Si estuviera vacío, entonces *map* devuelve una instancia de opcional vacío. Resulta extraño leer *map*, cuando solo acostumbras a utilizarlo para colecciones, hay que familiarizarse con este otro uso. *Optional* tiene también el método *orElse*, que se utiliza para recuperar el valor contenido o bien un valor alternativo. Lo que hace es quitar la envoltura, devolviendo el valor contenido por el opcional, o bien el que especifiquemos, en caso de no haber un valor presente. Si la operación que queremos ejecutar para *orElse* fuera muy pesada, podemos utilizar la variante *orElseGet*, que solamente invocará a la función en el caso vacío (porque recibe una función como argumento en lugar de un valor):

Java

```
/*...*/
.orElseGet(this::expensiveQuery);
/*...*/
private ResponseEntity<UserData> expensiveQuery(){
    /*...*/
}
```

Lo que conseguimos con estas construcciones es un estilo declarativo que nos da mucha potencia en pocas líneas:

Java

```
class Group {
    private List<Id> userIds;
    private Id id;
    private String name;

    public Group(List<Id> userIds, Id id, String name) {
```

```
        this.userIds = userIds;
        this.id = id;
        this.name = name;
    }

    public List<Id> userIds(){
        return new ArrayList<>(userIds);
    }
    /*...*/
}

class GroupRepository {
    public Optional<Group> findBy(Id id){
        Group group = findGroupFromTable(id.toString());
        return Optional.ofNullable(group);
    }
    /*...*/
}

class UserService {
    private UserRepository userRepo;
    private GroupRepository groupRepo;

    public UserService(UserRepository userRepo, GroupRepository groupRepo) {
        this.userRepo = userRepo;
        this.groupRepo = groupRe;
    }
    /*...*/
    public List<User> getUsersFromGroup(Id groupId){
        return groupRepo.findBy(groupId)
                .map(group -> group.userIds().stream()
                        .map(userId -> userRepo.findBy(userId))
                        .flatMap(Optional::stream)
                        .collect(Collectors.toList()))
                .orElse(new ArrayList<>());
    }
}
```

La línea que ejecuta *.flatMap(Optional::stream)*, filtra aquellos opcionales que tienen un valor no nulo y aplana la lista de listas. La línea *.collect(Collectors.toList())*, simplemente transforma el flujo (*stream*) en una colección (*list*). Este código es funcionalmente equivalente al siguiente:

Java

```
return groupRepository.findBy(groupId)
    .map(group -> group.userIds().stream()
         .map(userId -> userRepository.findBy(userId))
         .filter(Optional::isPresent)
         .map(Optional::get)
         .collect(Collectors.toList()))
```

```
    .orElse(new ArrayList<>());
```

La versión con *flatMap* es más conveniente porque garantiza a nivel de compilación que se han filtrado aquellos opcionales que contenían un valor, mientras que al hacer *filter + map*, el compilador ya no puede garantizar que hayamos escrito bien la lógica.

Estas construcciones quitan de en medio las sentencias condicionales, a la vez que eliminan la posibilidad de excepciones de tipo *NullPointerException*. Ahora bien, en mi opinión, no sustituyen al patrón objeto nulo (bien implementado), ni tampoco sustituyen la gestión de excepciones (para eso, en todo caso hay otras alternativas), sino que son una alternativa al manejo explícito de valores nulos.

En sistemas que tienen una alta carga de peticiones concurrentes, no sería adecuado acceder a la base de datos de manera síncrona como en estos ejemplos, sino que utilizaríamos otras técnicas para trabajar de forma asíncrona y así aprovechar mejor los recursos del servidor. Sucede lo mismo cuando trabajamos en aplicaciones nativas (escritorio o móviles), porque no deben bloquear el hilo de la GUI mientras se realizan operaciones potencialmente lentas. Cuando se necesita escalar rápidamente, a menudo se recurre al paradigma de la programación reactiva y una forma de hacerlo es mediante *observables*, que se abstraen de la concurrencia mediante flujos y eventos. Los observables pueden ser muy útiles incluso para gestionar peticiones XMLHttp en una aplicación que corre en un navegador web, aunque como toda abstracción que aborda un problema complejo, implica un nivel de complejidad nada despreciable. Para explicar la programación reactiva necesitaría otro libro, es un tema demasiado amplio como para tratar de resumirlo aquí, pero he añadido algunas recomendaciones en el apéndice.

Errores vs. excepciones

Lanzamiento de excepciones

Las excepciones deben utilizarse para manejar situaciones excepcionales, tal como su nombre indica, no para redirigir el flujo de ejecución ordinario. Su papel es muy importante, porque las situaciones excepcionales ponen en jaque al sistema, y una gestión inadecuada puede terminar con la ejecución, o incluso peor, provocar errores en cadena que pasan desapercibidos durante el suficiente tiempo como para que los usuarios cancelen la suscripción a nuestro servicio.

Aunque recurrir a las excepciones parece la solución más rápida para resolver algunos problemas, a menudo hay opciones igual de rápidas y más sencillas. Aquí va un ejemplo de lo que recomiendo *no hacer*, recurrir a la maquinaria de control de excepciones para una situación que no es excepcional:

Java

```java
private boolean isInteger(String expression){
    try {
        Integer.parseInt(expression);
        return true;
    } catch (NumberFormatExcepcion e){
        return false;
    }
}
```

Típicamente, se puede resolver el mismo problema sin excepciones:

Java

```java
private boolean isInteger(String expression){
    return expression.matches("-?[0-9]+");
}
```

La razón para evitar las excepciones en situaciones ordinarias, es que un código con saltos de flujo es más difícil de entender. Realmente, lanzar una excepción (*throw*) supone dar un salto (*goto* o *jmp* en otros lenguajes), mientras que la captura (*try-catch-finally*) supone colocar etiquetas (*label*) para señalizar el destino de ese salto. Comprender bloques de captura y de relanzamiento de excepciones, así como el bloque *finally*, puede llegar a ser un reto. Puede salir muy caro cometer errores manejando excepciones en situaciones no excepcionales, siendo totalmente contraproducente. Además, existe siempre el riesgo de influenciar negativamente al resto del equipo, que ya tendrá suficientes dudas sobre cómo y cuándo usar las excepciones. Es sorprendente lo poco que sabemos sobre trabajar con excepciones, ¿será porque es una herramienta relativamente «nueva» en los lenguajes de programación?

Para distinguir una situación corriente de una excepcional, también necesitamos contexto de negocio. Por ejemplo, ¿debería lanzar una excepción el siguiente método si no encuentra resultados?

Java

```java
public User findBy(Id userId){/*...*/}
```

Generalmente, los métodos de tipo consulta (que devuelven algo) no lanzan excepciones si son capaces de dar una respuesta con sentido. Los comandos o acciones también pueden gestionar los errores esperados sin necesidad de excepciones. Ahora bien, si en el contexto concreto se considera excepcional que el identificador usado como argumento no arroje ningún resultado, porque es algo que no debería pasar, quizá tenga sentido lanzar una excepción. Necesitamos ese contexto para tomar la mejor decisión. A falta de contexto yo diría que, lo mejor ante un método como este, es devolver un tipo opcional.

Los métodos de consulta recurren a excepciones cuando no tienen otra forma de expresar que se les pide algo imposible, como, por ejemplo, intentar acceder a una posición inexistente:

Java

```
List<String> items = new ArrayList<>();
items.get(30); // <-- IndexOutOfBoundsException
```

Si ante esta pregunta el *framework* devolviera *null*, yo no sabría si es porque el elemento en esa posición es nulo o porque me he equivocado con el índice. Devolver un tipo opcional tampoco sería correcto, sino más bien tedioso para quien invoca al método, y también confuso; no sabría si la respuesta significa *no hay* o *te has equivocado en la petición*. Los intérpretes de JavaScript no lanzan excepción ante un acceso fuera de rango y esto da muchos quebraderos de cabeza. Las excepciones son otro mecanismo más para comunicarnos con otros humanos.

Volviendo a nuestro método *findBy*, ¿qué pasaría si el motor de base de datos estuviera fuera de servicio cuando se invoca?, ¿debería devolver un opcional como nulo? Eso causaría sorpresa, sería enmascarar un problema del que no es posible recuperarse, haciendo creer al llamador que no existen datos en la tabla. Por más que queramos ser resilientes, no debemos engañar a nadie tratando de ocultar un problema que el código llamador no puede resolver y que merece ser distinguido de una respuesta vacía de manera explícita. La distinción permitirá a otras capas tomar decisiones, reintentar la petición y/o reportar la incidencia. Que la base de datos no esté disponible es, sin duda, una circunstancia excepcional, y como tal se gestiona con excepciones.

Hay varios escenarios típicos que podrían justificar el uso de excepciones como respuesta a preguntas mal formuladas. Uno de ellos es de la validación de precondiciones. Supongamos que quiero crear una instancia de un tipo que presenta una hora del día:

Java

```java
class Hour {
    private final int hour;

    private Hour(int hour) {
        this.hour = hour;
    }

    public static Hour from(int hour){
        if (hour < 0 || hour > 23){
            throw new IllegalArgumentException(
                "A day has only 24 hours on planet Earth not" + hour);
        }
        return new Hour(hour);
    }
}
```

Para evitar valores sin sentido, el método estático de factoría se cerciora de que solo puede construirse a partir de valores en rango, ya que lo contrario no tiene cabida en el dominio. Si devolviese *null* u *opcional*, de nuevo el llamador no tendría la capacidad de discernir, además de que luego tampoco podría proveer ningún mecanismo de resiliencia, es un obstáculo infranqueable.

Para expresar que un método puede lanzar excepciones, hay ciertos prefijos que están muy extendidos para nombrar a los métodos, como *validate*, *verify*, *ensure*, *check* o *assert*:

Java

```java
void validateWord(word);
void checkDateFormat(date);
void verifyCommand(command);
void assertSignedContract(contract);
```

Gracias a que no devuelven nada (son de tipo *void*) y a su nombre, dejan claro que van a lanzar una excepción si no se cumple la premisa; esto es programar con la intención de dar a entender.

Java

```java
public int getWorkingDays(Date from, Date to){
    validateDateRange(from, to);
    /*...*/
}
```

Quien aprecia la programación funcional nos dirá que la firma de *getWorkingDays* no indica

de ninguna manera que pueda lanzar una excepción, por eso vamos a introducir ahora el tipo *Either*.

El tipo *Either*

No todas las validaciones fallidas deben lanzar excepciones, si son escenarios plausibles para el negocio, situaciones cotidianas, no son circunstancias excepcionales, en cuyo caso sería más conveniente devolver información sobre el error:

Java

```java
class Hour {
    private final int hour;

    private Hour(int hour) {
        this.hour = hour;
    }

    public static Either<ImpossibleHourError, Hour> from(int hour) {
        return (hour < 0 || hour > 23)
                ? Either.left(new ImpossibleHourError(hour))
                : Either.right(new Hour(hour));
    }
}
```

Either es un contenedor que puede alojar uno de los dos tipos, y al igual que *Optional*, es un tipo algebráico. La palabra *either* en inglés significa «una de dos», por eso tiene dos propiedades que son *Left* y *Right*. La palabra *right* en inglés, además de ser la «derecha», significa «correcto», lo cual nos servirá para recordar que si utilizamos *Either*, el valor obtenido por el camino feliz estaría en la derecha, mientras que el error que se hubiera producido estaría en la izquierda. A diferencia de *Optional*, que está integrado en *java.util*, para *Either* estoy recurriendo a la implementación de la biblioteca Vavr. El tipo a la izquierda puede ser todo lo rico que necesitemos para recopilar erorres:

Java

```java
class Address {
    /*...*/
    private Address(String street, String city, String zipCode) {
        /*...*/
    }

    /*...*/
    public static Either<ValidationErrors, Address> from(String street,
        String city, String zipCode) {
```

```
        ValidationErrors errors = new ValidationErrors();
        /*...*/
        if (zipCode == null || zipCode.isEmpty()) {
            errors.add(KnownError.MissingZipCode);
        }
        /*...*/
        return errors.isThereAny()
            ? Either.left(errors)
            : Either.right(new Address(street, city, zipCode));
        }
    }
}
```

Lo que hace el método *from* es devolver una envoltura de una de estas dos opciones: un objeto de tipo *Address* ya correctamente validado, o bien un objeto de tipo *ValidationErrors* que contiene los errores de validación. Un beneficio directo de la firma de este método es que hace explícito el hecho de que podría devolver errores, eliminando la posibilidad de sorpresa para el llamador. En el dominio de este ejemplo, resulta que es habitual que a las direcciones les falte el código postal o algún otro dato obligatorio, es decir, que no es un caso excepcional ni se debe a ninguna anomalía en el sistema. Si decidiéramos lanzar excepciones es muy probable que el código llamador tuviera que envolver la llamada en *try-catch* y gestionar la excepción directamente. Al final, usaríamos las excepciones prácticamente como sentencias condicionales y ya vimos que no es buena idea. Por tanto, este (anti)patrón que consiste en envolver las llamadas en *try-catch* para tomar decisiones condicionales inmediatamente, puede servirnos de pista para ver que probablemente sería mejor devolver un objeto con información (con *Either* o con la envoltura que mejor modele la situación y mejor aproveche las características del lenguaje). Otro ejemplo:

Java

```
class Bucket {
    private final Collection<Item> items;
    private final int capacity;

    public Bucket(int capacity, Collection<Item> items) {
        this.capacity = capacity;
        this.items = items;
    }

    public Bucket(int capacity) {
        this(capacity, new ArrayList<>());
    }
    /*...*/
    public Either<OvercapacityError, Bucket> addItem(Item item){
        /*...*/
        if (items.size() < capacity){
```

```
            var newItems = new ArrayList<>(items);
            newItems.add(item);
            return Either.right(new Bucket(capacity, newItems));
        } else {
            return Either.left(new OvercapacityError(capacity));
        }
    }

    public int capacity() {
        return capacity;
    }
}
/*...*/
@Test
public void either_type_contains_left_or_right_value(){
    var bucket = new Bucket(5);

    var bucketOrError = bucket.addItem(new Item());

    assertThat(bucketOrError.isRight()).isTrue();
    assertThat(bucketOrError.isLeft()).isFalse();
    assertThat(bucketOrError.get().capacity()).isEqualTo(5);
}
```

Patrón Notificación

Una validación no tiene por qué utilizar excepciones si los posibles errores son esperados o plausibles, ni tampoco devolver *Either*, sino que podría rellenar un objeto con el resultado de la validación. Martin Fowler tiene un fantástico artículo donde habla del patrón notificación para recopilar los errores. Ejemplo básico:

Java

```
class Address {
    /*...*/

    public static AddressValidationResult validate(AddressData address){
        var result = new AddressValidationResult();
        if (address == null){
            result.registerError("Address is null");
            return result;
        }
        if (address.street == null || address.street.isEmpty()){
            result.registerError("Street is missing");
        }
        if (address.zipCode == null || address.zipCode.isEmpty() ||
                isInvalidZipCode(address.zipCode)){
            result.registerError("Invalid zip code:" + address.zipCode);
        }
```

```
        /*...*/
        return result;
    }

    private static boolean isInvalidZipCode(String zipCode) {
        /*...*/
    }
}

class AddressValidationResult {
    private final List<ValidationError> errors = new ArrayList<>();

    public void registerError(String error) {
        errors.add(new ValidationError(error));
    }

    public boolean containsErrors(){
        return errors.size() > 0;
    }

    public List<ValidationError> getErrors(){
        return new ArrayList<>(errors);
    }
    /*...*/
}

class ValidationError {
    private final String error;

    ValidationError(String error) {
        this.error = error;
    }
    /*...*/
}
```

El objeto que recopila errores podría ser también enviado a otros métodos como argumento, en caso de haber otras validaciones, de manera que acumulara todos los posibles errores esperados. Personalmente, soy cada vez más partidario de gestionar los posibles errores con técnicas como estas, en lugar de lanzar excepciones. Si puedes limitar el uso de excepciones a situaciones extremas como caídas de servicios, el código será menos sorpresivo, más evidente, más explítico. Dicho esto, me gustaría señalar que hay casos en los que una combinación de validaciones con y sin excepciones sería lo más adecuado: antes que repetir las mismas validaciones en diferentes capas del sistema, podría ser conveniente que la capa del núcleo de negocio asuma que los datos ya llegan validados y que se defienda con excepciones ante precondiciones ilógicas, para no duplicar esfuerzos. Así las capas más externas validan de una manera explícita (sin sorpresas) y las más internas confirman las precondiciones que tienen sentido para el negocio (aunque tampoco es la

receta mágica perfecta para todos los casos).

Vamos a ver un ejemplo avanzado que combina el tipo *Either* con el objeto de notificación, porque puede haber precondiciones que sean más complejas, que vayan más allá de tipos o valores. Consiste en una clase que envuelve una colección para darle sentido de negocio. La instancia se crea mediante un método de factoría que recibe un mapa/diccionario, con la premisa de que todos los objetos de ese mapa compartan un mismo valor, en una de sus propiedades. Lo contrario no tendría sentido para el negocio, por eso se comprueba antes de crear el objeto:

Java

```java
class DailyContracts {
    private final Map<Day, Contract> contracts;

    private DailyContracts(Map<Day, Contract> contracts) {
        this.contracts = contracts;
    }

    /**
     *   Enforces business rules to deal with daily contracts.
     *   @param contracts a dictionary of contracts per day, where
     *                    all the contracts must have the same supply.
     *   @return Either a set of errors or the DailyContracts.
     */
    public static Either<DailyContractErrors, DailyContracts>
        fromContractsWithSameSupply(Map<Day, Contract> contracts) {
        var errors = validate(contracts);
        if (!allContractsHaveTheSameSupply(contracts)) {
            errors.add("All contracts must share the same supply");
        }
        return errors.containsAny()
                ? Either.left(errors)
                : Either.right(new DailyContracts(contracts));
    }

    private static DailyContractErrors validate(Map<Day, Contract> contracts)
         {
        DailyContractErrors errors = new DailyContractErrors();
        if (contracts == null) {
            errors.add("Can't create daily contracts from null");
            return errors;
        }
        if (contracts.size() == 0) {
            errors.add("Can't create daily contracts from empty");
        }
        if (contracts.containsValue(null)){
            errors.add("Can't create daily contracts with null items");
        }
        return errors;
    }
```

```
    private static boolean allContractsHaveTheSameSupply(Map<Day, Contract>
        map) {
        var contracts = map.values().stream();
        return contracts.findFirst().map(contract ->
                contracts.allMatch(
                    c -> c != null &&
                    c.supplyId().equals(contract.supplyId()))
        ).orElse(false);
    }
    /*...*/
}
```

Para que se cumpla la premisa, deben cumplirse también otras más básicas como que el mapa no sea nulo ni vacío, así como que no contenga valores nulos. En este caso, no era posible absorber valores inadecuados, porque cualquier intento de resiliencia hubiera sido incoherente con las reglas de negocio. Tenemos que exponer y trabajar los errores de manera explícita para evitar cálculos que no cuadran, valores raros, etc. No queremos provocar errores en cadena que puedan pasar desapercibidos, ni cuyo rastro se difumine por la aplicación (y menos cuando hablamos de *software* de facturación). Cuando asumimos premisas tenemos que validarlas explícitamente.

Captura de excepciones

Ahora que ya hemos visto cuándo es previsible y conveniente lanzar excepciones, repasemos cómo se gestionan: vamos a estudiar la captura, el registro y el relanzamiento de excepciones.

Todas las excepciones son herederas de una clase base o de una interfaz (según el lenguaje). En el desarrollo de aplicaciones de negocio (código vertical), debemos evitar capturar dicha excepción base:

Java

```
try {
    /*...*/
} catch (Throwable throwable){ // <--- esta no
    /*...*/
}
```

Java

```
try {
```

```
    /*...*/
} catch (Exception exception){ // <--- esta no
    /*...*/
}
```

Si capturamos la excepción genérica, podríamos capturar otros escenarios diferentes al que queremos, y, por tanto, hacer una gestión incorrecta. Siempre pueden fallar muchas más cosas de las que somos capaces de predecir. Si se produce una excepción que no tenemos controlada, es preferible dejarla correr y que la capture el *framework* que usemos. En un *backend*, por ejemplo, lo más común es delegar la fontanería del protocolo HTTP en un *framework* como *Spring Boot o ASP.NET Core MVC*, en cuyo caso, el *framework* devolverá al cliente un error 500, si es alcanzado por una excepción no controlada. El *framework* se encarga de capturar la excepción genérica, por eso cuando desarrollamos una librería o un *framework*, sí tenemos que capturarla. Aquí vemos otra diferencia táctica entre código vertical y código de propósito general. Cuando nos apoyamos en el *framework*, lo que sí debemos hacer es instrumentalizar y monitorizar el sistema, para enterarnos de estos fallos no controlados, y así poderlos arreglar.

Otra mala práctica, que por desgracia se ve mucho, es tragarse las excepciones de manera silenciosa (*swallowing exceptions*); es un horror a la hora de detectar fallos en el *software*, sobre todo si utiliza la excepción base:

Java

```
public void updateUser(User user){
    try {
       /*...*/
    } catch (Exception exception){
       /* ?¨¡¡¡esto se queda vacio!!!? */
    }
}
```

Si ocurre cualquier fallo tratando de guardar el usuario, nadie se entera en el momento, ya que la excepción ha sido capturada, pero no se hace nada con ella. Da la falsa impresión de que la acción se ha ejecutado correctamente. Lógicamente, la gente hace estas cosas con su mejor intención, pensando que si capturan la excepción, la aplicación ya no se termina, ni devuelve errores. Detrás de esta buena intención, suele haber miedo a las excepciones y desconocimiento sobre su comportamiento.

Si lo que se pretende es capturar varias excepciones, es mejor encadenar bloques de captura:

Java

```java
try {
    /*...*/
} catch (ExceptionTypeA ex){
    /*...*/
} catch (ExceptionTypeB ex){
    /*...*/
} catch (ExceptionTypeC ex){
    /*...*/
}
```

Lo cual es equivalente a:

Java

```java
try {
    /*...*/
} catch (ExceptionTypeA | ExceptionTypeB | ExceptionTypeC ex){
    /*...*/
}
```

Encadenar no es lo mismo que anidar; si un bloque *catch* volviese a lanzar una excepción, los otros bloques *catch* que tenga encadenados no la van a capturar. Cada bloque *catch* responde al bloque *try* que tiene al mismo nivel de anidamiento. No hay problema con anidar bloques *try-catch* dentro de *try-catch*, aunque conforme más niveles de indentación hay, más cuesta entender el código. Extraer métodos con buenos nombres facilita la lectura:

Java

```java
public void someAction(SomeCommand command){
    try {
        /*...*/
    } catch (SomeException ex){
       tryOtherApproach(command);
    }
}
private void tryOtherApproach(SomeCommand command){
    try {
        /*...*/
    } catch (SomeOtherException ex){
        /*...*/
    }
}
```

El prefijo *try* para los nombres de método, está bastante extendido para métodos «seguros», que no van a lanzar excepciones, puesto que ya tienen dentro *try-catch*. En general, no

es buena idea que los nombres de los métodos contengan información técnica sobre su implementación, pero puede haber casos donde esto ayude, por ejemplo, en operaciones de bajo nivel o de propósito general. En C# se utiliza esta convención en el *framework*:

C#

```
int number;
bool success = Int32.TryParse(value, out number);
```

El hecho de que se haga a nivel de *framework* no significa que debamos copiar esta táctica en todas las capas de nuestra aplicación.

Volviendo al ejemplo anterior y suponiendo que solo se quiere capturar una excepción, ¿mejoraría la situación con el siguiente código?

Java

```
public void updateUser(User user){
    try {
       /*...*/
    } catch (JDBCConnectionException e){
       logger.log(Level.SEVERE, "Can't update user", e);
    }
}
```

Ya no capturamos la excepción genérica, sino una muy concreta, lo cual está muy bien, pero, ¿deberíamos registrar la excepción en este punto (*log*)?, ¿para qué estamos capturando la excepción? El problema de capturarla es que quien ha llamado a este método, ignora que no se ha podido guardar el usuario, no se entera. Si hemos diseñado algún mecanismo de tolerancia a fallos para reintentar automáticamente la operación más tarde, quizá no haga falta notificar a quién está usando la aplicación, en cuyo caso se le da a entender que todo ha ido bien, y luego en diferido, si no se consigue, se le avisa por email o como mejor sea la experiencia de usuario. Si este mecanismo no está implementado o no tiene sentido implementarlo por la complejidad que conlleva, entonces quien está usando la aplicación debe saberlo en el momento, recibiendo un mensaje de error. En este segundo escenario, la excepción debe seguir su curso hacia las capas exteriores o superiores del sistema. Tenemos tres opciones para darle curso a esta excepción hacia afuera:

1. No capturarla en este método.
2. Capturarla, traducirla a un nivel de abstracción superior y dejar que siga su curso.
3. Capturarla, registrarla y dejar que siga su curso.

La primera opción es la más adecuada si no podemos recuperarnos del fallo y no tenemos claro que las demás alternativas sean mejores. Si no se captura, está asegurado que va a continuar su curso hacia arriba como una burbuja. Si no sabemos qué hacer con ella, no debemos tampoco escribir en el registro (*log*), porque al final habrá información redundante y confusa en él (la responsabilidad de registrarla le corresponde a otro artefacto). Al dejarla pasar, una capa superior (más externa) puede capturar una excepción algo más general como *PersistanceException* (padre de *JDBCConnectonException*) y mostrar el mensaje de error en pantalla o lo que corresponda. Tiene sentido que las capas más externas capturen excepciones más generales.

La segunda opción sirve para evitar que las capas externas tengan conocimiento concreto de problemas de bajo nivel, ya que se traduce la excepción a otra de más alto nivel:

Java

```java
public void updateUser(User user){
    try {
       /*...*/
    } catch (JDBCConnectionException e){
       throw new CantSaveUser(e, user);
    }
}

class CantSaveUser extends RuntimeException {
    private User user;
    public CantSaveUser(Throwable cause, User user){
        super(cause);
        this.user = user;
    }
    public User user(){
        return user;
    }
}
```

El beneficio de definir nuestra propia excepción y que sea esta la que propagamos hacia arriba, es que las otras capas ya no tienen por qué conocer las excepciones del *framework*, sino que trabajan directamente con semántica de negocio. Esto podría ayudar a reducir el acoplamiento entre capas, aunque también puede resultar excesivamente complejo, por lo que debe implementarse con criterio. En caso de que vayamos a elegir esta alternativa, es muy importante tener en cuenta que la excepción original se le está pasando por constructor a la nueva que creamos, para preservar toda la traza de ejecución (la pila de llamadas o *stack trace*). Sin la traza completa, estamos perdidos a la hora de intentar reproducir el error, de modo que hay que conservarla, porque en ella están las pistas para solucionar problemas. Cuando enviamos por constructor la excepción original a la nueva que esta-

mos creando, encadenamos las trazas. Otro beneficio de las excepciones a medida es que podemos guardar datos específicos para facilitar su posterior tratamiento, como en este caso, que nos guardamos el usuario. Así, podemos recuperar los datos que sean necesarios de dicho usuario, sin necesidad de analizar el mensaje de la excepción. Si te ves en la situación de tener que codificar datos concretos dentro del mensaje de una excepción y luego tratando la cadena en otro lugar para extraer información de ella, es posible que sea mejor utilizar una excepción propia.

La tercera opción es útil cuando estamos en el punto con la mejor información para registrar el problema, por ejemplo, porque tenemos acceso a variables locales que aportan mayor contexto al registro:

Java

```java
public void updateUser(User user){
    String key;
    JSONObject jsonObject;
    try (Jedis redis = pool.getResource()) {
        key = getKeyFor(user);
        jsonObject = jsonFrom(user);
        redis.set(key, jsonObject.toJSONString());
        /*...*/
    } catch (JedisException ex) {
        logger.log(Level.SEVERE, "Can't store new user", user, key,
            jsonObject);
        throw ex;
    }
}
```

Fíjate como relanzamos la excepción para que siga su curso al hacer *throw e*. No te olvides de esta línea, porque sino estamos otra vez bloqueando el curso de la excepción.

En C# hay que tener mucho cuidado con la forma de relanzar, puesto que hay una diferencia muy sutil; para preservar la traza hay que escribir *throw* a secas, sin nada detrás.

C#

```csharp
catch(EntitySqlException ex){
    logger.LogWarning("Error saving user", ex);
    throw;    // <-- relanza la excepción preservando la traza.
//  throw ex; // <-- lanzaría la excepción perdiendo la traza.
}
```

A partir de la versión 4.5 de .Net, la mejor forma de relanzar es la siguiente, porque es la que mayor información ofrece:

C#

```
catch(EntitySqlException ex){
    logger.LogWarning("Error saving user", ex);
    ExceptionDispatchInfo.Capture(ex).Throw(); // <-- así
}
```

Cada lenguaje/plataforma tiene sus particularidades en el manejo de excepciones y merece la pena conocerlas en detalle. Ante la duda, es mejor hacer pruebas de concepto con pequeños fragmentos de código y unos test sencillos que nos aclaren bien cómo funciona la herramienta. La inversión de tiempo es muy pequeña y la rentabilidad es muy alta.

¿Para qué sirve el bloque *finally*? Para liberar recursos, tanto si el flujo de ejecución se mantuvo en el bloque *try*, como si saltó al bloque *catch*. El bloque *finally* se ejecuta siempre, incluso si el bloque *catch* a su vez lanzase una excepción. Lo que hace es complicado de entender, por eso no debe utilizarse para devolver valores, ni para relanzar excepciones, sino exclusivamente para liberar recursos:

Java

```
PrintWriter out = null;
try {
    out = printWriterFor(filename);
    out.println(someText);
} catch (FileNotFoundException e) {
    logger.log("File not found:" + filename);
    throw e;
} finally {
    if (out != null) {
        out.close();
    }
}
```

En las versiones modernas de Java y de C# se utiliza menos el bloque *finally*, porque las construcciones del lenguaje llevan implícita la liberación de recursos. Veamos una alternativa al código anterior:

Java

```
try (
    final PrintWriter out = printWriterFor(filename);
) {
    out.println(someText);
} catch (FileNotFoundException e) {
    logger.log("File not found:" + filename);
    throw e;
```

```
}
```

Fíjate que tras la palabra *try* vienen paréntesis y dentro se instancia el recurso. En C# es muy parecido, utilizando la palabra reservada *using*. De hecho, si no necesitamos capturar la excepción, sino simplemente asegurarnos de que se liberan los recursos, no hace falta ni el *try*:

C#

```
using (TextReader reader = File.OpenText("log.txt")) {
    string line;
    while ((line = reader.ReadLine()) != null) {
        Console.WriteLine(line);
    }
}
```

Cuando la ejecución termina el recurso se libera automáticamente, tanto si fue por el camino feliz, como si fue por el excepcional.

En Java hay un tipo de excepciones, cuyo uso es obligatorio dejar reflejado en la firma de los métodos. El compilador nos obliga a escribir explícitamente el nombre de la excepción junto al nombre del método que la maneja y que, potencialmente, puede lanzarla. Estas son las llamadas *checked exceptions*:

Java

```
void writeToLog(String[] lines) throws FileNotFoundException {
    PrintWriter file = new PrintWriter(logPath);
    /*...*/
}
```

Fíjate en la palabra *throws* de la firma del método. En principio, parece una buena idea, porque así cualquier método que va a invocar a este, ya sabe que puede esperarse una posible excepción de tipo *FileNotFoundException*. Parece que sirve para programar con intención explícita. Me puedo imaginar que cuando lo diseñaron pensaban que así se evitarían sorpresas desagradables y que ayudaría a recuperarse de fallos, por el hecho de hacer explícito que las operaciones pueden fallar. La práctica ha demostrado que no es tan buena idea y yo personalmente prefiero definir mis excepciones como de tipo *unchecked* ¿Qué problemas tienen las *checked exceptions*? Producen más acoplamiento del necesario y como además resultan «molestas», son muy propensas a ser eliminadas de manera silenciosa (para que dejen de «molestar»), lo que nos sitúa en uno de los peores escenarios. Como

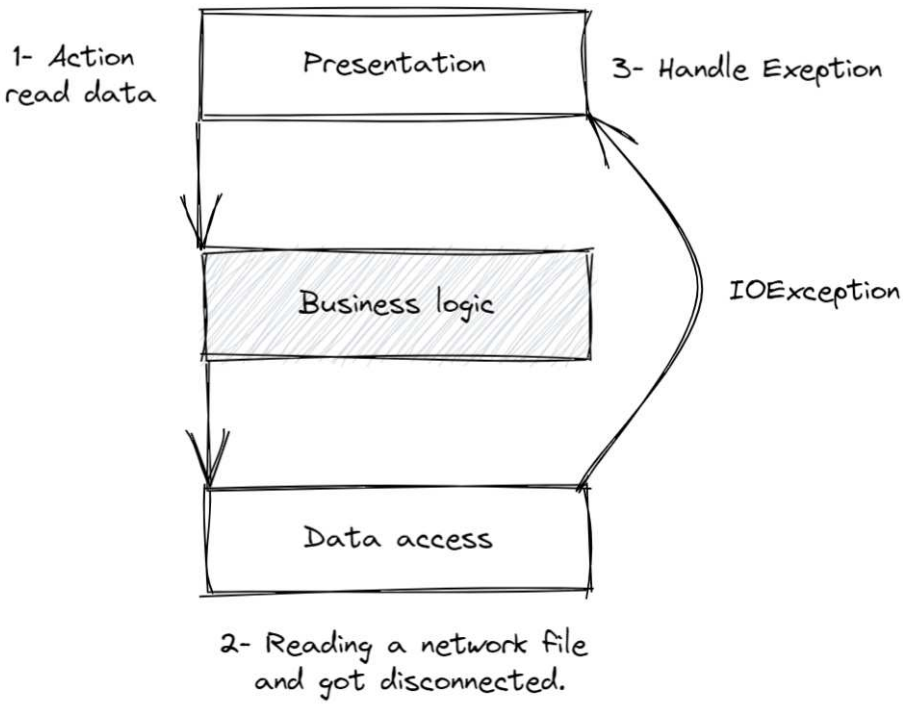
de costumbre, parte del problema no es tanto el mecanismo, sino el uso que se hace del mismo. Típico caso de gestión inadecuada:

Java

```java
void writeToLog(String[] lines) {
    PrintWriter writer;
    try {
        writer = new PrintWriter(logPath);
        /*...*/
    } catch (FileNotFoundException ex){
        /* ?¡¡¡vacio!!!? */
    }
    /*...*/
}
```

El compilador reconoce que la excepción está siendo capturada con un bloque *catch* y por eso ya no exige que se especifique en la firma del método. Esto hace que deje de propagarse por las firmas de los métodos llamadores, con lo cual se termina la «molestia». Ahora bien, ya sabemos que tragarse las excepciones tiene consecuencias muy graves. Hay gente que ha oído que dejar un bloque *catch* vacío es mala práctica, aunque desconoce el motivo, por lo tanto tiende a rellenarlo guardando la excepción en el *log*. Así ya no está vacío, obviamente. Un código que está plagado de llamadas a métodos estáticos de *log*, o que inyecta por todas partes una clase *logger* para invocar a sus métodos en múltiples puntos, tiene una alta probabilidad de haber sido escrito sin entender el manejo de excepciones. Si tienes dificultad para escribir test porque el código a probar abusa del *logger*, podría ser que la gestión de excepciones que se ha implementado sea inadecuada o incluso arbitraria.

¿Qué hay del acoplamiento? Al definir un método con una o más *checked exceptions* en su cabecera, los métodos llamadores se ven obligados a hacer lo mismo, produciéndose un efecto dominó (salvo que capturen esas excepciones). Si pensamos en un sistema de varias capas, siendo la primera la más cercana al usuario y la última la de acceso a datos, sabemos que con frecuencia las capas que tienen la capacidad de gestionar problemas como una caída de la base de datos, son estas dos. La capa de acceso a datos lanza la excepción, las capas de negocio o lógica no pueden hacer nada más que dejarla pasar, y ya, justo antes de que le explote a la usuaria, la capa más cercana la recoge para mostrar un mensaje de error. Como puede observarse en la siguiente figura, con mucha frecuencia las excepciones se gestionan en los extremos, no en el centro (núcleo de la lógica de negocio).



Las *checked exceptions* meten ruido en todas las capas, aunque las centrales no puedan hacer nada. Se produce un acoplamiento innecesario que resulta chocante al leer el código: estoy leyendo un método sobre unas reglas de negocio y lo primero que me encuentro es una amenaza de *IOException… ¿qué?, ¿cómo?, ¿qué se supone que tengo que hacer yo con esto?*

En cambio, si las excepciones son de tipo *unchecked*, solo veremos referencias a ellas en los puntos en que de verdad se puede hacer algo al respecto (acoplamiento inevitable). Este tema de *checked vs. unchecked* lleva siendo polémico décadas.

Para definir nuestras propias excepciones de tipo *unchecked*, heredamos de la clase *RuntimeException*:

Java

```java
public class OutOfOrder extends RuntimeException {
    public OutOfOrder(String message){
        super(message);
    }
    public OutOfOrder(String message, Throwable cause){
```

```
        super(message, cause);
    }
}
```

Como los constructores no se heredan y la clase base tiene un constructor sin argumentos (entre otros), ni siquiera es obligatorio definir constructores. No obstante, es práctico añadir el mensaje y la excepción original, porque un uso típico de nuestras excepciones es traducir otras de bajo nivel. Definir nuestras propias clases de excepciones con semántica de dominio precisa, enriquece la expresividad del código, porque son otro mecanismo más de abstracción. Ya vimos que las abstracciones son poderosas, lo cual conlleva responsabilidad, por eso hay que pensarlas bien y no crear meros alias de excepciones que ya provee el *framework*. Realmente, en los proyectos en los que he trabajado nunca hubo una gran cantidad de excepciones definidas por nuestro equipo, porque el *framework* (*JVM, .Net…*) ya ofrece un conjunto muy versátil, como, por ejemplo, *IllegalArgumentException*, *TimeOutException* o *SecurityException*. Si encuentras que ninguna de las excepciones existentes representa el concepto que está manejando tu aplicación, quizá debas crear una. Ten en cuenta que, habitualmente, hay distintos *frameworks* operando, no añadas acepciones a excepciones de *frameworks*, como, por ejemplo, *ResponseStatusException*, de *Spring Boot*; estas solo debes utilizarlas para el propósito concreto que le da el *framework* (devolver un código de error HTTP en este caso). Quiero decir que no reutilices excepciones del *framework* para expresar otras cosas.

El tipo *Try*

En el paradigma de programación funcional, se tiende a utilizar menos las excepciones, porque no respetan la propiedad de transparencia referencial en las funciones. La gestión de excepciones se combina con los tipos, *Either*, *Try* o similares, porque a menudo las funciones se definen para que devuelvan el resultado del camino feliz o bien un error. *Try* es específico para excepciones, mientras que *Either* puede utilizarse además para representar la unión de tipos. *Try<T>* es isomórfica con respecto a *Either<Throwable, T>*, lo que significa que hay una correspondencia uno a uno entre los valores del primer tipo y los del segundo. Si estamos trabajando con Java o C# y diseñamos con un enfoque funcional, una forma de encarrilar las excepciones consiste en capturarlas y envolverlas en estos tipos. De nuevo nos apoyamos en la biblioteca Vavr para Java:

Java

```
class Logger {
    private Path logPath;

    public Logger(Path logPath) {
        this.logPath = logPath;
    }

    public Try<Void> writeToLog(String... lines) {
        return Try.run(() -> write(lines));
    }

    private void write(String[] lines) throws IOException {
        try (
                PrintWriter writer = getWriter();
        ) {
            for (var line : lines) {
                writer.write(line);
            }
        }
    }

    private PrintWriter getWriter() throws IOException {
        return new PrintWriter(new FileWriter(logPath.toString()));
    }
}
/*...*/
@Test
public void try_may_contain_exceptions(){
    var logger = new Logger(Paths.get("/a_file_that_does_not_exists"));
    Try<Void> result = logger.writeToLog("one line");
    assertThat(result.isFailure()).isTrue();
    assertThat(result.getCause().getClass())
        .isEqualTo(FileNotFoundException.class);
}
```

El tipo de retorno del método público *writeToLog* es un tipo *Try* de *Void*, porque el método no devuelve nada, es una acción. En el test podemos ver que tiene métodos para preguntar cómo ha ido la ejecución y obtener la excepción si es que la hubo, para poder responder a ella. El método privado *write* utiliza *try* para que se libere el recurso, no para capturar la excepción, por eso su cabecera está obligada a especificar la *checked exception*.

Para métodos que devuelven valores por el camino feliz, el código sería así:

Java

```
public Try<Integer> getItem(int index){
    /*...*/
    return Try.of(() -> collection.getItem(index));
}
```

Try también nos ofrece un estilo declarativo para operar con el resultado a través de funciones como *map, filter, getOrElse*…:

Java

```
parser.parse(expression).getOrElse(0);
```

¿Conviene más *Try* que *Either*? Lo que sea más simple suele ser lo mejor, pero a la vez tenemos que balancear principios como el de menor sorpresa y limitar el nivel de acoplamiento, así que cada caso requiere una decisión a medida. Para programar bien se necesita tiempo para pensar y concentración. Abusar de estos tipos podría producir un acoplamiento similar al de las *checked exceptions*, donde unas capas conocen detalles de otras que no les incumben para nada. En programación puramente funcional se siguen usando las excepciones de siempre para gestionar fallos severos del sistema, pero el lanzamiento y captura se realizan en la parte «impura» del programa, fuera de las capas de negocio. Si tengo que elegir entre uno de estos dos tipos para el transporte de las excepciones, prefiero *Try*, porque expone menos información a las capas que atraviesa. *Either* tiene más sentido para devolver información sobre errores predecibles. Personalmente, me encanta que los lenguajes modernos como Java, C# o Kotlin, permitan combinar paradigmas como OOP y funcional, porque así en cada momento podemos elegir lo mejor de cada uno. No tiene mucho sentido decir que OOP es mejor que FP ni viceversa, sino que en cada contexto brinda sus pros y contras; el quid de la cuestión está en saber balancear los principios para facilitar el mantenimiento lo máximo posible.

Depurar problemas

A día de hoy, es inevitable que se cuelen errores en el entorno de producción, a lo más que podemos aspirar es a que llegue la menor cantidad posible, a que su impacto sea mínimo y a que los encontremos y arreglemos rápido, sin introducir nuevos errores al hacer cambios y sin que haya caídas del servicio. No podemos eliminar los errores, pero sí podemos entrenarnos para detectarlos y corregirlos de manera eficaz. Hasta ahora, hemos hablado de cómo evitarlos, así que ahora vamos a ver cómo subsanarlos.

Lo primero es saberlos encontrar y entender. Luego, estudiamos la mejor estrategia para corregir y actualizar el sistema. El siguiente paso consiste en validar la hipótesis aplicando cambios en un entorno de pruebas controlado. Por último, si los resultados son satisfactorios, se procede a aplicar los parches sobre el entorno de producción.

Para encontrar fallos debemos poderlos reproducir en un entorno local en el que podamos experimentar sin afectar al entorno de producción. En mis primeros años como programador y administrador de sistemas, desarrollaba y mantenía yo solo un proyecto web escrito con PHP 3.x, alojado en un servidor Debian que también mantenía yo mismo. Cuando algo fallaba, accedía al servidor vía terminal (*ssh*), abría el editor Vim y me ponía a toquetear los ficheros PHP de producción a cualquier hora, haciendo cambios «en caliente». Ni siquiera hacía primero una copia de respaldo para poder volver atrás, la copia estaba en mi cabeza. Aprendí que no era buena idea hacer cambios en producción cuando los clientes llamaban por teléfono a la empresa, enfadados, porque el sistema no les dejaba trabajar… ¡Qué temeridad!, eso era *cowboy coding* total, al más puro estilo *Juan Palomo*. El entorno de producción no debe alterarse sin seguir un protocolo riguroso, incluyendo pruebas de regresión exhaustivas.

Cuando se trata de operaciones que no alteran el sistema, no hay problema en replicar un *bug* múltiples veces, usando el *software* en producción; si una consulta no trae los resultados correctos para una determinada búsqueda, seguramente no sea un problema repetir la petición. Podría ensuciar el registro del sistema (*logs*), eso sí. Lo que no debemos hacer, de ninguna manera, es alterar el código de producción directamente.

Para poder hacer cambios o pruebas debemos clonar el entorno de producción, cuando sea posible, para disponer de un entorno lo más parecido al real. Si los datos de producción son sensibles, por ejemplo, si son datos sanitarios o bancarios, habrá que anonimizarlos antes de dejarlos en el entorno de pruebas. Si la copia es suficientemente parecida a la real, tendríamos que ser capaces de reproducir el *bug* usando el *software* en el entorno de pruebas. Hay que reducir al máximo los errores en cadena y las falsas alarmas. La finalidad del entorno de pruebas no solo es depurar, sino reproducir el problema antes de solucionarlo, como para luego del arreglo poder verificar que funciona. Depurar un sistema complejo mediante puntos de ruptura o salidas por consola, es una técnica que consume demasiado tiempo. Puede ser interesante hacerlo alguna vez para comprender el flujo, pero poco más valor tiene. Es más productivo aislar al máximo cada uno de los artefactos en los que pensamos que puede haber un error o que intervienen en la operativa. Depurar el sistema entero es mucho más lento y pesado que atacar a las partes por separado. Para saber qué partes interesa estudiar de manera aislada, exploramos la estructura del sistema apoyándonos en diagramas cuando sea necesario, y en las compañeras y compañeros (*pair programming*). En mi experiencia, es mucho más productivo (y seguro) trabajar en pares que por separado (comunicándose mediante *tickets* o mensajes de chat), tanto para reproducir un *bug* como para corregirlo. Ya cuando nos hemos decantado por una clase o por un

método en concreto, lo ejercitamos explícitamente escribiendo test automáticos, y si eso no es posible, al menos un pequeño programa ejecutable con salidas por consola. Puede que lo más rápido sea crear un proyecto nuevo que contenga la menor cantidad posible de código, copiando y pegando del proyecto real a este de pruebas. Cuantas menos partes haya, más fácil será razonar sobre el problema; aumentará nuestra comprensión sobre el comportamiento y el diseño, además de que será más fácil hacer búsquedas en Internet, ya que los problemas serán más específicos o acotados. Para arreglar fallos de manera eficaz, es imprescindible comprender su causa, así como el funcionamiento esperado/correcto. Es peligroso ponerse a hacer cambios aleatoriamente «probando suerte», porque podríamos estar rompiendo otras funcionalidades, causando males mayores. Uno de los mayores focos de *bugs*, es la reparación de otros *bugs*.

Si ejercitar las piezas por separado no da con el problema, el siguiente paso es ir combinándolas para buscar problemas en la integración de las mismas. Estudiaremos las combinaciones, de las más simples a las más complejas. Este análisis exhaustivo de las causas del problema suele conllevar, como efecto secundario, el descubrimiento de otros *bugs* no reportados hasta el momento, y de más deuda técnica. Es un buen momento para tomar nota y gestionar la deuda, planificando sesiones de refactorización en el futuro próximo. Mientras realizamos la búsqueda del fallo, en principio no es aconsejable refactorizar, porque podríamos romper otras funciones. Es aconsejable no mezclar cambios de diferente índole en los *commits*, de manera que la corrección de fallos quede bien distinguida del resto de modificaciones. Todo esto depende de lo urgente y grave que sea la incidencia en producción, que lo normal es que sea urgente, por eso haremos el menor número de cambios para solucionar el problema lo antes posible y con la máxima seguridad.

Una vez que encontramos cuál es el fallo, es fundamental demostrarlo añadiendo un test automático, es decir, escribir un test que falla cuando debiera de estar en verde, si el funcionamiento del *software* fuese correcto. Al corregir el *bug*, el test pasa de estar en rojo a estar en verde, lo cual valida nuestra hipótesis y sobre todo nos permite garantizar que ese mismo fallo jamás volverá a producirse. Para los usuarios es frustrante que un fallo corregido en una versión antigua reaparezca en versiones posteriores. No solo frustra, sino que provoca desconfianza en el equipo de desarrollo por parte de *stakeholders*. Todo el mundo puede entender que los humanos nos equivoquemos y que el *software* falle alguna vez, sin embargo, que el mismo fallo regrese es más difícil de justificar.

Para realizar todo este proceso con éxito, es indispensable apoyarse en una buena herramienta de control de versiones como Git, que nos permita recuperar una copia idéntica del código de producción, realizar las correcciones en el entorno de pruebas, comparar con

precisión y aplicar esos cambios en producción. A menudo, los *bugs* reaparecen por errores en el manejo de las versiones, por eso el proceso debe ser estricto. Es crítico hacer un buen uso de la herramienta de control de versiones y conocer los flujos de trabajo para *hotfixes*. Si no hay control de versiones no hay trazabilidad, de manera que si una o varias personas están haciendo sus cambios directamente en producción (en caliente), al estilo *cowboy*, es imposible garantizar el resultado. Conozco equipos que aún trabajan así, como si el entorno de producción fuese su entorno local, lo cual me parece temerario.

Comprender los informes de fallos

Cuando no se comprende un mensaje de error o no se sabe distinguir cuál es la información relevante en una traza de excepción (pila de llamadas), cuesta muchísimo más encontrar la solución al problema. Al no entender lo que está diciendo el sistema, ni siquiera se puede buscar soluciones en Internet con eficacia. Casi cualquier barrera que te encuentres programando, se la ha encontrado alguien antes, y probablemente la solución esté ya explicada en *StackOverflow*, *Github* o cualquier otro sitio, pero hay que saber buscar para encontrar. Mucha gente se siente incapaz de leer los mensajes de error o se siente intimidada por ellos, como si lo que vieran en pantalla fuesen códigos indescifrables de Matrix; *¡noooo!, ¡hay un error!* Hay gente que ni se plantea detenerse a leer el mensaje de error, como si estuviera escrito en hebreo, directamente lo copian y pegan en el buscador, o en un foro. Todo esto ocurre por desconocimiento o falta de entrenamiento. Para las personas que sí se ocupan de intentar comprender los errores, puede resultar molesto recibir peticiones de ayuda frecuentes para solventar los mismos problemas triviales. Cuando llevas unos meses trabajando con una tecnología, conviene entender los mensajes de error por una cuestión de pura eficacia. Si cada vez que hay un error pides ayuda (a tu compañera de al lado o en un foro), antes de comprender qué dice el error, es posible que los demás se cansen. Es como si camino por la oficina, veo que tengo los cordones de los zapatos sueltos y sucios, y le pido a un compañero que me los amarre.

Pedir ayuda no es malo, al contrario, pedir ayuda fortalece las relaciones, según lo que se pida. Si pido ayuda para que un compañero más experimentado me explique cómo interpretar el error o la traza, seguro que va a acceder con gusto, porque ve que tengo un interés por aprender. En cambio, pedir a los demás que hagan lo que yo no quiero hacer, es una actitud egoísta y causará rechazo.

Cada *stack* tecnológico tiene su propia forma de mostrar los mensajes de error y las pilas de llamadas. En Java, por ejemplo, la información se lee de arriba a abajo. Si es una ex-

cepción, aparece en la primera línea, mientras que la siguiente línea señala el punto en que ha saltado la excepción. De ahí en adelante, se imprime la pila de llamadas hasta llegar al punto de entrada, que suele ser un manejador de un evento como *click* en una interfaz de usuario, un controlador en un *backend*, un método *main* de un ejecutable, un test…

```
java.lang.NullPointerException
    at Node.deepCopy(Node.java:22)
    at java.base/java.util.stream.ReferencePipeline$3$1.accept(
        ReferencePipeline.java:195)
    at java.base/java.util.ArrayList$ArrayListSpliterator.forEachRemaining(
        ArrayList.java:1624)
    at java.base/java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.
        java:484)
    at java.base/java.util.stream.AbstractPipeline.wrapAndCopyInto(
        AbstractPipeline.java:474)
    at java.base/java.util.stream.ReduceOps$ReduceOp.evaluateSequential(
        ReduceOps.java:913)
    at java.base/java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.
        java:234)
    at java.base/java.util.stream.ReferencePipeline.collect(ReferencePipeline
        .java:578)
    at Node.children(Node.java:44)
    at Tests.shallow_copy_is_not_deeply_immutable(Tests.java:878)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native
         Method)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(
        NativeMethodAccessorImpl.java:62)
    at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(
        DelegatingMethodAccessorImpl.java:43)
    at java.base/java.lang.reflect.Method.invoke(Method.java:564)
    at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(
        FrameworkMethod.java:44)
    at org.junit.internal.runners.model.ReflectiveCallable.run(
        ReflectiveCallable.java:15)
    at org.junit.runners.model.FrameworkMethod.invokeExplosively(
        FrameworkMethod.java:41)
    at org.junit.internal.runners.statements.InvokeMethod.evaluate(
        InvokeMethod.java:20)
    at org.junit.runners.BlockJUnit4ClassRunner.runChild(
        BlockJUnit4ClassRunner.java:76)
    at org.junit.runners.BlockJUnit4ClassRunner.runChild(
        BlockJUnit4ClassRunner.java:50)
    at org.junit.runners.ParentRunner$3.run(ParentRunner.java:193)
    at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:52)
    at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:191)
    at org.junit.runners.ParentRunner.access$000(ParentRunner.java:42)
    at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:184)
    at org.junit.runners.ParentRunner.run(ParentRunner.java:236)
    at org.junit.runner.JUnitCore.run(JUnitCore.java:157)
    at com.intellij.junit4.JUnit4IdeaTestRunner.startRunnerWithArgs(
```

```
    JUnit4IdeaTestRunner.java:68)
at com.intellij.rt.junit.IdeaTestRunner$Repeater.startRunnerWithArgs(
    IdeaTestRunner.java:33)
at com.intellij.rt.junit.JUnitStarter.prepareStreamsAndStart(JUnitStarter
    .java:230)
at com.intellij.rt.junit.JUnitStarter.main(JUnitStarter.java:58)
```

En este ejemplo, la excepción es *NullPointerException*, lanzada por la JVM, lo cual ya nos dice que se trata de una excepción no controlada. Luego dice:

```
at Node.deepCopy(Node.java:22)
```

Que significa:

- La excepción ha partido de la línea 22 del fichero Node.java
- Esa línea pertenece al método *deepCopy* de la clase *Node*.

A continuación, aparece la pila de llamadas, ocupada en su mayoría por funciones del *framework* sobre las que no tenemos ningún control. Aquí hay poco que podamos hacer, por lo que vamos a rastrear la traza hacia abajo hasta encontrar alguna función que pertenezca a nuestro código. Encontramos lo siguiente:

```
at Node.children(Node.java:44)
at Tests.shallow_copy_is_not_deeply_immutable(Tests.java:878)
```

Que significa:

- La ejecución viene de la línea 44 del fichero Node.java
- Dentro del método *children* de la clase *Node*
- A su vez viene del fichero Tests.java, en la línea 878
- Dentro del método *shallow_copy_is_not_deeply_immutable* de la clase *Tests*

La mayor parte de la traza es información irrelevante para el caso que nos ocupa, ya que es código que no podemos cambiar. Sin embargo, es información que debe estar, porque a veces sí es importante conocer la pila de llamada entera (cuando trabajamos con una librería de terceros poco documentada, cuando estamos optimizando rendimiento…). Los

IDEs suelen plegar parte de la traza, en un intento por mostrarnos las líneas que más nos pueden interesar, porque la mayoría de las veces la filtramos visualmente en busca de nuestras líneas de código. Así que cuando trabajes con Java, no te abrumes por una larga traza de excepción, navega hacia la parte superior y verás que todo tiene sentido.

Cuando la línea que lanza la excepción no es código nuestro, sino de una librería/*framework* de terceros, cuesta más entender el problema, porque al fin y al cabo es código que no conocemos. La excepción puede aparecer por varios motivos:

- Porque estamos haciendo un uso incorrecto/inesperado de la librería/*framework*.
- Porque tenemos una configuración de versiones conflictiva.
- Porque hay un *bug* en dicha herramienta.

Puede ser lanzada a propósito por la librería para dejar claro que está siendo usada incorrectamente, o puede ser lanzada por el *framework* subyacente si escapa del control de la librería. Supongamos que utilizo una librería que trabaja con colecciones, y paso como argumento un valor nulo, o quizá un elemento de la colección es nulo. Si la librería no se asegura explícitamente de rechazar los nulos, lanzando una excepción que diga «no se admiten nulos», lo que nos encontraremos será una excepción no controlada de tipo *NullPointerException*, señalando a una línea interna de dicha librería. Aquí sí que nos interesa estudiar bien la traza completa. Si la librería es de código abierto, podemos abrirlo y revisar las líneas que aparecen en la traza, empezando por la que lanza la excepción, para entender cuál es el uso esperado para el que fue diseñada. Mirar código *open source* de terceros es una forma muy buena de aprender. Suele ser práctico depurar nuestro código en el punto justo anterior a la invocación de la librería, para saber qué argumentos le estamos enviando y comparar con lo que dice su documentación, por si algo no concuerda. Si pensamos que se trata de un *bug*, podemos escribir un test que lo demuestra y adjuntarlo al informe de incidencia que podríamos enviar al equipo de contribuyentes de la librería. Reportar una incidencia con un test que la evidencia, facilita mucho el trabajo y si además enviamos una propuesta de parche que corrige el *bug*, todavía mejor.

Vamos a ver un ejemplo en C#, provocado por el mal uso de una librería de terceros, que termina de forma no controlada. Al igual que en Java, las excepciones se leen de arriba a abajo:

```
System.InvalidOperationException : The binary operator Multiply is not
    defined for the types 'Stacktraces.Node' and 'Stacktraces.Node'.
```

```
   at System.Linq.Expressions.Expression.GetUserDefinedBinaryOperatorOrThrow(
       ExpressionType binaryType, String name, Expression left, Expression
       right, Boolean liftToNull)
   at System.Linq.Expressions.Expression.Multiply(Expression left, Expression
        right, MethodInfo method)
   at System.Linq.Expressions.Expression.Multiply(Expression left, Expression
        right)
   at Towel.Statics.MultiplicationImplementation`3.<>c.<.cctor>b__1_0(TA a,
       TB b)
   at Towel.Statics.Multiplication[TA,TB,TC](TA a, TB b)
   at Towel.Statics.Multiplication[T](T a, T b)
   at Towel.Mathematics.Vector`1.Multiply(Vector`1 a, T b, Vector`1& c)
   at Towel.Mathematics.Vector`1.Multiply(Vector`1 a, T b)
   at Towel.Mathematics.Vector`1.op_Multiply(Vector`1 a, T b)
   at Towel.Mathematics.Vector`1.Multiply(T b)
   at Stacktraces.Example.Multiply() in /home/carlosble/CodigoSostenible/
       Example.cs:line 10
   at Stacktraces.Tests.CantMultiplyVectorWithNodes() in /home/carlosble/
       CodigoSostenible/Tests.cs:line 10
```

Lo primero que podemos ver es que la excepción es lanzada por el sistema (*System.Linq.Expressions.Expression.Multiply*), y que es de tipo *InvalidOperationException*. Si continuamos la traza, vemos que el flujo viene de la libreria *Towel*, y, por último, que el punto de mi código donde termina mi control es el fichero *Example.cs*, en la línea 10.

Si como usuario no entiendo esta traza, y además me creo que uso la librería de forma correcta, es posible que hasta reporte un *bug* (cuando no es *bug*, sino una malinterpretación mía). Si la forma en la que uso una librería es muy extravagante, copiar y pegar la traza del error en Internet no conseguirá ningún resultado, puesto que es un fallo muy particular mío. Por eso es que cuando utilizo una librería que muchas personas usan y me encuentro con un mensaje de error, que nadie más parece tener, me planteo que posiblemente la estoy usando mal. También puede ser que mi sistema esté mal configurado, con versiones incompatibles o alguna otra situación anormal.

Por otro lado, si programo una librería, procuraré anticiparme a usos incorrectos validando parámetros y lanzando excepciones, para que el mensaje quede claro y así evitar que me abran incidencias falsas. Veamos un ejemplo de uso incorrecto de otra librería, pero que sí está controlado por la misma:

```
Newtonsoft.Json.JsonReaderException : Error reading JObject from JsonReader.
    Current JsonReader item is not an object: StartArray. Path '', line 1,
    position 1.
   at Newtonsoft.Json.Linq.JObject.Load(JsonReader reader, JsonLoadSettings
       settings)
```

```
   at Newtonsoft.Json.Linq.JObject.Parse(String json, JsonLoadSettings
       settings)
   at Newtonsoft.Json.Linq.JObject.Parse(String json)
   at Stacktraces.Example.GetJson(String json) in /home/carlosble/
       CodigoSostenible/Stacktraces/Example.cs:line 17
   at Stacktraces.Tests.CantParseJsonWithWrongFormat() in /home/carlosble/
       CodigoSostenible/Stacktraces/Tests.cs:line 16
```

Aquí vemos que se trata de una excepción propia de la librería (*Newtonsoft.Json.JsonReaderException*), que me dice que la cadena enviada no es un objeto JSON. A diferencia de la excepción no controlada, ahora es mucho más probable que cuando alguien ponga esta traza en Internet, encuentre respuestas de otras personas que dirán, «fíjate que estás usando mal la librería o que tienes un *bug* en tu código, porque no le estás enviando un formato JSON válido». Las personas que mantienen las librerías, los compiladores o los intérpretes, no siempre pueden diseñar mensajes certeros, porque no siempre está clara la intención de quien las usa. Por ejemplo, esta misma librería dice cosas diferentes si la cadena contiene otros caracteres:

```
Newtonsoft.Json.JsonReaderException : Error parsing boolean value. Path '',
    line 1, position 1.
```

No estoy enviando ningún *boolean*, pero se ve que para este caso es lo mejor que han podido hacer.

No todas las trazas son iguales, en Python, a diferencia de Java y C#, se les llama *tracebacks* y se leen al revés, de abajo hacia arriba:

```
Traceback (most recent call last):
  File "quicksort.py", line 36, in <module>
    quickSort(arr, 0, n-1)
  File "quicksort.py", line 30, in quickSort
    quickSort(arr, pi+1, high)
  File "quicksort.py", line 28, in quickSort
    pi = partition(arr, low, high)
  File "quicksort.py", line 20, in partition
    arr[i+1], arr[high] = arr[high], arr[i+5]
IndexError: list index out of range
```

La excepción es de tipo *IndexError*, también procedente del sistema, es una excepción no controlada. Las líneas penúltima y antepenúltima, nos dicen dónde ha surgido la excepción

y además nos muestran la línea en cuestión:

```
File "quicksort.py", line 20, in partition
    arr[i+1], arr[high] = arr[high], arr[i+5]
```

De aquí hacia arriba se nos muestra la pila de llamadas. En este ejemplo es todo código que controlamos, no hay llamadas a librerías del sistema, se trata de un *script* lanzado desde la terminal. Sería mucho más verboso si el error lo lanzase un controlador web de algún *framework* como Django, porque habría una larga pila de llamadas entre nuestro código y el punto de entrada.

Veamos una traza de un proyecto complejo:

```
Traceback (most recent call last):
  File "/usr/local/lib/python3.8/runpy.py", line 193, in _run_module_as_main
    return _run_code(code, main_globals, None,
  File "/usr/local/lib/python3.8/runpy.py", line 86, in _run_code
    exec(code, run_globals)
  File "/home/carlosble/projectX/_main_.py", line 39, in <module>
    clean_data(config)
  File "/home/carlosble/projectX/lib/python3.8/site-packages/data_components/
      config/logging.py", line 309, in wrapped_method
    raise err
  File "/home/carlosble/projectX/lib/python3.8/site-packages/data_components/
      config/logging.py", line 306, in wrapped_method
    return method(*args, **kwargs)
  File "/home/carlosble/projectX/lib/python3.8/site-packages/data_components/
      config/logging.py", line 221, in wrapped_method
    return method(*args, **kwargs)
  File "/home/carlosble/projectX/lib/python3.8/site-packages/data_components/
      config/logging.py", line 269, in wrapped_method
    ans = method(*args, **kwargs)
  File "/home/carlosble/projectX/_main_.py", line 31, in clean_data
    return Transform(config, os.getenv("CLEAN_DATA_CONFIG_KEY", f"
        transform_ras_trc.clean")).run()
  File "/home/carlosble/projectX/transform.py", line 80, in run
    df_ras = generate_data_type_column(df_converted, df_ras)
  File "/home/carlosble/projectX/transform_utils.py", line 152, in
      generate_data_type_column
    return fix_data_type_column(df_ras)
  File "/home/carlosble/projectX/transform_utils.py", line 155, in
      fix_data_type_column
    df["data_type"] = np.where(df['data_type'] == 'float64', 'float', df['
        data_type'])
  File "/home/carlosble/projectX/lib/python3.8/site-packages/pandas/core/
      frame.py", line 3024, in _getitem_
```

```
    indexer = self.columns.get_loc(key)
  File "/home/carlosble/projectX/lib/python3.8/site-packages/pandas/core/
      indexes/base.py", line 3082, in get_loc
    raise KeyError(key) from err
KeyError: 'data_type'
```

La traza es complicada de leer, porque la excepción está siendo lanzada por la librería *pandas* (fichero *pandas/core/indexes/base.py* en línea 3082), que no parece esmerarse en arrojar un mensaje de error digestivo (cuando miré su código por curiosidad, no entendí para qué relanza la excepción y me dio mala impresión que el fichero tuviera 6687 líneas). Por en medio de la traza, están las llamadas a mi código (*projectX*), mientras que el inicio muestra el punto de entrada, que también es código de terceros (*runpy.py*). Esta «estructura de sándwich», hace que tengamos que leer con detenimiento toda la traza para entenderla.

El cuidado en la gestión de casos y mensajes de error, suma puntos para mí cuando tengo que elegir una librería entre varias. También suma puntos que tengan una buena cobertura de test, de hecho, si no tiene test evito usarla.

Cuando empieces a trabajar con una tecnología nueva para ti, tómate un tiempo para documentarte y comprender la forma en la que se expresan los errores, porque es una inversión de tiempo altamente rentable.

Los errores de compilación también son complicados de interpretar al principio; cada compilador tiene sus particularidades. Por ejemplo, en C y C++, los errores se amontonan de manera que los que aparecen más abajo, en el texto de salida, suelen ser consecuencia de los que aparecen por encima. Por eso, en estos lenguajes tenemos que irnos a la parte superior del texto de salida del compilador y ocuparnos de arreglar el primer error, ya que, a menudo, los demás se arreglan también. Veamos un ejemplo (main.c):

```c
#include <stdio.h>

int main() {
    printf(Hello, world!");
    return 0;
}
```

Cuando compilo, me encuentro la siguiente salida:

```
main.c:4:12: error: use of undeclared identifier 'Hello'; did you mean
      'ftello'?
    printf(Hello, world!");
           ^~~~~
           ftello
/usr/include/stdio.h:712:16: note: 'ftello' declared here
extern __off_t ftello (FILE *__stream) __wur;
               ^
main.c:4:19: error: use of undeclared identifier 'world'
    printf(Hello, world!");
                  ^
main.c:4:25: warning: missing terminating '"' character
      [-Winvalid-pp-token]
    printf(Hello, world!");
                        ^
1 warning and 2 errors generated.
exit status 1
```

En realidad solo hay un error: falta la comilla doble de comienzo de cadena en *Hello, world!*. Lo que hace el compilador es seguir adelante con su análisis, no solo se queja de que no entiende la palabra *Hello*, sino que también dice que no entiende *world*. Además, se encuentra con una comilla que interpreta como de apertura y echa en falta la de cierre. Los compiladores e intérpretes buscan encontrarle un sentido a lo que escribimos, de manera que cuando cometemos errores no saben lo que pretendemos hacer y por eso su salida resulta tan confusa. Todo tiene un sentido, los mensajes de error no son textos místicos aleatorios.

La verdad es que requiere entrenamiento comprender ciertos mensajes de error, sobre todo porque a veces son tan cortos que no resultan explicativos. Veamos un ejemplo de mensaje de error en JavaScript:

Javascript

```
TypeError: Cannot read property ‘’length of undefined
```

¿Qué significa? Para que el intérprete asuma que una palabra es una *property*, tiene que ser que ha encontrado dicha palabra después del símbolo punto, puesto que el punto denota acceso a propiedades (igual que en Java y C#). Así sabemos que se refiere a la propiedad *length* de algo, pero dice que ese algo es indefinido. Entonces está diciendo que la variable que hay a la izquierda del punto, en lugar de ser el objeto que pensamos que sería, en realidad tiene el valor especial *undefined*. Saber inglés siempre nos va a ayudar, porque *property of* se traduce como *propiedad de*. El mensaje sería más fácil de entender si dijera… *«Cannot read property ‘length’ of object ‘x’ because the value of ‘x’ is undefined»*. Los compiladores y

los intérpretes no siempre aciertan con el mensaje, porque el error podría tener múltiples causas. Sobre los mensajes de error típicos de los distintos intérpretes de JavaScript, hay más información en un artículo de mi blog.

Registro y monitorización de errores

Para que podamos corregir errores tenemos que detectar que se están produciendo. Si no monitorizamos el entorno de producción, estaremos ignorando los problemas hasta que alguien reporte una incidencia, es decir, cuando los usuarios ya estén sufriendo sus consecuencias y tengan la generosidad de avisarnos, será cuando nos enteremos de que algo va mal. Eso es demasiado tarde, llegados a ese punto ya habremos perdido usuarios, reputación y/o dinero. He visto a empresas perder ingentes cantidades de dinero por no saber que su sistema tenía fallos. Las primeras personas que deben enterarse de cualquier anomalía o disfunción del sistema, son las integrantes del equipo que lo mantiene y lo desarrolla.

A la hora de monitorizar, tenemos que incluir servicios de terceros sobre los que se apoyan nuestras aplicaciones (servidor web, sistema de colas, envío de emails…), así como el propio sistema operativo, para evitar, por ejemplo, que el espacio de almacenamiento se agote y colapse el sistema. Así que hay una parte del trabajo de monitorización que es externa a las aplicaciones y otra que es interna, por tanto, se requiere la colaboración de todo el equipo. No es una responsabilidad que recaiga en el departamento de sistemas, sino que todos somos responsables de cuidar de la calidad del servicio.

Para monitorizar tus aplicaciones es imprescindible que registres (*log*) todos los errores/excepciones que ocurran. Si utilizas algún *framework*, simplemente tienes que configurarlo para que cualquier excepción no controlada se refleje en un registro, para que se pueda monitorizar. Si no utilizas *framework* (por ejemplo, si se trata de aplicación ejecutable vía terminal), asegúrate de capturar cualquier excepción no controlada y guardarla en el registro. Si el código se está ejecutando sobre una plataforma que no puedes controlar (una aplicación de escritorio nativa, una aplicación móvil o un *frontend web* ejecutándose en el navegador), tendrás que dotar a la aplicación de la capacidad de enviar los errores a través de la red, para que puedan monitorizarse. Por ejemplo, en el caso del *frontend web*, hay librerías que capturan los errores que se producen en el código JavaScript ejecutado por el navegador y los envían a nuestro *backend* mediante peticiones HTTP.

Además de monitorizar errores inesperados, debemos monitorizar atributos de calidad

transversales, como el consumo de recursos o la experiencia de usuario. Si no controlamos el gasto que hacemos en servidores en la nube, las facturas nos darán más de un disgusto. Si no monitorizamos el comportamiento de los humanos usando nuestras aplicaciones, nunca sabremos si comprenden la interfaz visual. El tema de la monitorización nos daría para unos cuantos capítulos, pero no es mi intención profundizar en ello, solo quería terminar remarcando su importancia. Si el equipo escribe código sostenible con una fabulosa cobertura de test, pero luego el servicio se cae durante horas y nadie se da cuenta, el balance será muy negativo. Lo malo no es que el sistema caiga, sino que tarde demasiado tiempo en levantarse (o que no se recupere) y que se dé cuenta primero la persona que lo ha contratado.

10. Tipos específicos del dominio

Independientemente del sistema de tipado del lenguaje, podemos dividir los tipos en dos categorías, los que vienen integrados en el lenguaje (*built-in types*) y los que creamos como usuarios del lenguaje (*user-defined types*). Entre los tipos integrados están los primitivos (tales como *int, float, boolean o char*), las cadenas de caracteres y los tipos compuestos como las tuplas, *arrays*, listas, etc. Los integrados son tipos de propósito general, bloques de construcción básicos que nos abstraen de la codificación binaria y que resuelven problemas universales. Gracias a los tipos integrados podemos interconectar subsistemas heterogéneos, como, por ejemplo, distintos motores de bases de datos, al igual que podemos utilizar distintos mecanismos de entrega para una misma aplicación (web, móvil, *desktop*…).

A diferencia de los tipos integrados, los tipos que definimos al programar tienen un propósito específico, resuelven un problema concreto. Hablamos de tipos específicos del dominio cuando diseñamos una clase o una interfaz (en lenguajes como Java o C#), para representar conocimiento de un dominio concreto. En lenguajes como TypeScript o F# se usa también la palabra reservada *type* para definir estos tipos específicos del dominio.

La potencia expresiva de un tipo específico varía según el lenguaje. En lenguajes como Haskell, F# o Idris, la potencia de los tipos permite expresar programas completos en sí mismos, es decir, el propio tipo define las reglas de negocio. Conseguir que compile un programa escrito en estos lenguajes es mucho más laborioso que hacerlo compilar en Java o en C#, porque los tipos son mucho más ricos; el compilador hace un trabajo mucho más profundo. Lenguajes como C++, Rust, Scala o TypeScript, están a medio camino, ofrecen una gran riqueza expresiva en sus tipos. En el resto de lenguajes comercialmente más extendidos, como Java, C#, JavaScript, Python o Ruby, la potencia de los tipos es más limitada. La ventaja de la simplicidad en los tipos, es que resulta más fácil razonar sobre ellos, entenderlos. La desventaja es que se requiere más código para conseguir el mismo resultado y que el compilador realiza menos comprobaciones, dejando más espacio para los errores en tiempo de ejecución. Los test automáticos son indispensables para confirmar nuestras hipótesis sobre el comportamiento esperado de los tipos que programamos.

Ventajas de los tipos específicos

¿Cuáles son las ventajas de definir nuestros propios tipos específicos del dominio con el que estamos trabajando? Como cualquier otra abstracción, dotan a nuestros programas de expresividad, haciendo explícitos los conceptos del dominio del problema y del dominio de la solución. Las abstracciones adecuadas contribuyen a que el programa se comprenda más rápido y a que sea más intuitivo realizar cambios. Los tipos específicos nos permiten cohesionar datos y operaciones en una misma unidad conceptual. Atraen comportamiento hacia ellos, así como buenos nombres de variables: si existe una clase *Invoice*, es posible que las instancias de la misma se nombren como *invoice*, al mismo tiempo que es posible que el cálculo del monto neto resida en dicha clase. Cuando existen tipos concretos para representar conceptos, resulta intuitivo acomodar las nuevas funciones que van siendo requeridas. De hecho, la dificultad para decidir dónde colocar un método o función, puede señalar que tenemos un déficit de tipos específicos en nuestro código (o de módulos específicos). Cada vez que expresemos un concepto mediante valores primitivos o cadenas, tenemos la oportunidad de crear un tipo específico: en vez de utilizar los números mágicos, por ejemplo, cuando resulta que el *valor -1* representa un concepto, el *valor 0* otro, el *valor 1* otro, el *777* otro… un primer paso puede ser construir un tipo enumerado, aunque a menudo podemos ir mucho más allá.

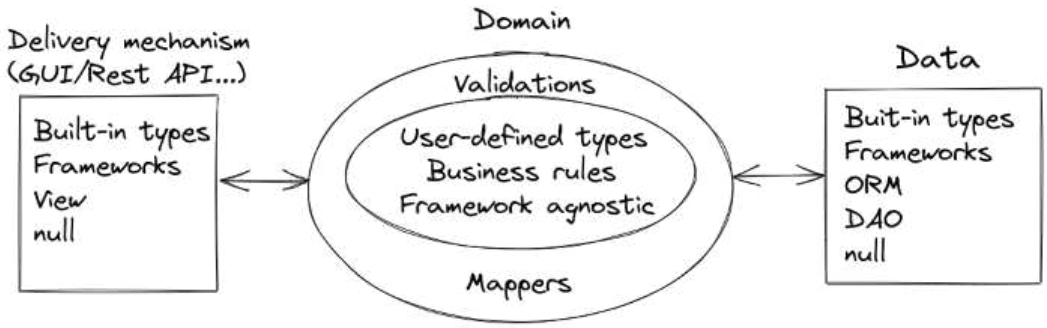
Los tipos específicos del dominio son fundamentales para escribir código que habla el idioma del negocio, con expresividad y con intencionalidad explícita; código semántico y autodocumentado. Después de ver muchos proyectos, la experiencia me dice que abusamos de los tipos integrados codificando conceptos mediante valores, ofuscando gravemente el código. De hecho, este problema es bien conocido y tiene un nombre en el catálogo de olores de código: *primitive obsession*. La gracia de los lenguajes de alto nivel frente a los de bajo, como ensamblador, es que podemos crear abstracciones poderosas. Con demasiada frecuencia infrautilizamos la potencia que nos ofrecen los lenguajes (o abusamos de ella), complicando las soluciones mucho más de lo necesario. Allá donde hay un valor mágico, yace escondido un tipo específico que espera ser descubierto.

¿Existe el abuso de los tipos específicos del dominio?, ¿puede ser que tengamos problemas de mantenimiento si creamos demasiados tipos? Yo jamás he visto un proyecto que sufriera por una cantidad excesiva de tipos definidos por el equipo de desarrollo, sino que siempre me encontré problemas por abusar de los tipos integrados. Lógicamente, si las abstracciones son marcianas, le complicaremos la vida a quien venga luego a mantener nuestro código, así que como cualquier otra abstracción no es gratis. Los principios que

hemos visto a lo largo del libro, nos van a dar una orientación sobre si los tipos que creamos son adecuados o no:

- ¿Podemos nombrar al tipo con un buen nombre?, ¿es adecuada la abstracción?
- ¿Utiliza palabras del dominio de negocio o son invenciones marcianas?
- ¿Cómo afecta a la cohesión?, ¿y al acoplamiento?
- ¿Qué tan útil resulta?, ¿cuánto poder de atracción de comportamiento posee?
- ¿Cómo de sorprendente termina siendo el código?, ¿la intención queda clara?
- ¿Cuánto cuesta comprender la solución?, ¿encaja con el nivel de conocimiento que tenemos en el equipo?
- ¿Cómo estamos balanceando el resto de pros y contras de la solución?, ¿merece la pena la complejidad?

Si pensamos en un sistema de varias capas, ya sea una arquitectura hexagonal o cualquier otra, hay unas capas externas que interactúan con *software* de terceros y otras internas sobre las que tenemos control total (núcleo de negocio). Mi recomendación es limitar el uso de los tipos integrados del lenguaje a esas capas externas que trabajan con la *GUI*, la base de datos u otros servicios de terceros. Las capas de lógica de negocio no deberían trabajar con tipos integrados sino con tipos específicos del dominio.



Es cierto que los tipos integrados no pueden eliminarse y que estarán dentro de los tipos específicos, lo que sucede es que serán invisibles para las operaciones de negocio, las cuales solo conocerán la interfaz de los tipos específicos. La forma típica de envolver a los tipos integrados es mediante clases que los reciben por constructor:

Java

```java
public class EmailAddress {
    private final String email;
```

```
    EmailAddress(String email) {
        this.email = email;
    }
}
```

La primera capa tomará la cadena con el email, creará la instancia de *EmailAddress* usando su constructor y, a continuación, pasará dicha instancia a la capa de negocio. Los tipos que representan conceptos de un dominio vertical concreto pueden dividirse en entidades y valores. Un objeto valor (*Value Object*) se considera idéntico a otra instancia si el valor que contiene es el mismo. *EmailAddress* es un objeto valor si equivale a otro que contiene exactamente el mismo valor. Por eso, cuando construimos objetos valor, es típico implementar los métodos *equals* y *hashcode*:

Java

```java
public class EmailAddress {
    private final String email;

    EmailAddress(String email) {
        this.email = email;
    }

    @Override
    public boolean equals(Object other) {
        if (this == other){
           return true;
        }
        if (!(other instanceof EmailAddress)){
           return false;
        }
        EmailAddress that = (EmailAddress) other;
        return Objects.equals(email, that.email);
    }

    @Override
    public int hashCode() {
        return Objects.hash(email);
    }
}
```

Los objetos valor son un gran remedio para evitar los valores sin sentido, porque pueden restringir el rango mediante validaciones:

Java

```java
public class ColumnWidth {
    private final int width;
```

```
    private ColumnWidth(int width) {
        this.width = width;
    }

    public static ColumnWidth from(int width){
        if (width < 0){
            throw new IllegalArgumentException(
                "Width must be a positive number");
        }
        return new ColumnWidth(width);
    }
    /*...*/
}
```

Si un objeto de tipo *ColumnWidth* se ha podido crear, significa que tengo la garantía de que contiene un valor con sentido en el dominio. Aquellas validaciones que contemplan situaciones de error típicas (no excepcionales), se pueden gestionar también con objetos valor:

Java

```
class VisaCreditCard {
    private final CardNumber cardNumber;
    private final ExpirationDate date;

    private VisaCreditCard(CardNumber number, ExpirationDate date) {
        this.number = number;
        this.date = date;
    }

    public static Either<CardValidationError, VisaCreditCard> from(String
        number, String month, String year){
        if (isInvalidVisaNumber(number)){
            return Either.left(
                    new CardValidationError(
                      "Invalid card number for Visa"));
        }
        if (isInvalidMonth(month) || isInvalidYear(year)){
            return Either.left(
                    new CardValidationError(
                      "Invalid expiration date"));
        }
        return Either.right(
            new VisaCreditCard(
                new VisaNumber(number), new ExpirationDate(month, year)));
    }
    /*...*/
}
```

Una importante propiedad de los objetos valor que les diferencia de las entidades, es que son inmutables. Cualquier operación sobre el objeto devolverá una nueva instancia sin modificar el estado de la clase. Los objetos de tipo *string* se comportan así, todos sus métodos devuelven nuevas cadenas sin modificar la original (hay lenguajes donde sí se puede modificar una cadena, pero son los menos). Supongamos que la clase del ejemplo requiere un método para modificar el dominio de la dirección de correo ¿Cómo podemos hacerlo sin modificar la dirección original?

Java

```
class EmailAddress {
    /*...*/

    public EmailAddress changeDomain(String newDomain){
        return new EmailAddress(mailboxName() + "@" + newDomain);
    }

    private string mailboxName() {
        return email.split("@")[0];
    }
    /*...*/
}
/*...*/
EmailAddress email = new EmailAddress("carlos@carlosble.com");
/*...*/
EmailAddress otherEmail = email.changeDomain("leanmind.es");
```

Cuando surja la necesidad de que otra persona en el futuro amplíe la funcionalidad de una dirección de email, es probable que al encontrarla aquí, decida añadirla también aquí. La probabilidad de que la cohesión aumente es mucho mayor que si lo metemos todo en una clase o módulo *Utils*. Los módulos o paquetes como *Utils* terminan como cajón desastre, incrementan el acoplamiento y limitan la cohesión.

A diferencia de los objetos valor, las entidades son objetos compuestos por atributos que sí pueden mutar con el paso del tiempo y que se distinguen de otros objetos por un identificador único. En programación funcional las entidades también se suelen programar como objetos valor, creando nuevos objetos cada vez que se requiere un cambio, para evitar mutaciones.

Los tipos específicos añaden flexibilidad al código, porque nos aislan de cambios de bajo nivel que puedan producirse en artefactos de terceros. Por ejemplo, supongamos que los identificadores de las entidades los genera el motor de base de datos y que utiliza números para ellos. Entonces, supongamos que en código utilizamos un tipo de dato numérico

como *long*, para manejar estos identificados:

Java

```
long id;
```

¿Qué sucede si mañana cambiamos el motor de base de datos o la forma de los identificadores y pasan a ser cadenas? Pues que vamos a tener que realizar cambios en todas las capas del sistema. Si hubiéramos abstraído el identificador con un tipo, el cambio sería mínimo:

Java

```
public class Id {
    private final string id;
    public Id(long id){
        this.id = String.valueOf(id);
    }
    public Id(string id){
        this.id = id;
    }
    /*...*/
}

public class User {
    private final Id id;
    /*...*/
}
```

Para sacarle partido a la orientación a objetos, tanto en los objetos valor como en las entidades, nuestras clases específicas de dominio contendrán datos y operaciones juntos. De lo contrario serían modelos anémicos. Dentro de nuestra capa de lógica de dominio, debemos evitar los modelos anémicos igual que evitamos los tipos integrados. Para trabajar en esa línea, eliminamos los *setters* y limitamos al máximo los *getters*, añadiendo métodos con lógica a las clases. Ciertas herramientas como los ORM (*Hibernate*, *Entity Framework*…) utilizan modelos anémicos para la persistencia, por eso recomiendo que dichos objetos no penetren en la capa de dominio, sino que haya una conversión a objetos de dominio. Siguiendo las ideas de *Domain-driven Design*, sobre todo cuando trabajamos con arquitectura hexagonal, es bastante típico trabajar de esta forma.

Lo que resta de capítulo puede resultar especialmente duro de comprender si estás empezando a programar, soy consciente de ello, no te preocupes si no lo entiendes. Mi objetivo es que puedas conocer que hay mucho que estudiar sobre los tipos, que sepas que existen,

por si alguna vez quieres tirar del hilo y seguir ampliando conocimiento.

Sistemas de tipos

Cada lenguaje opera con un sistema de tipos, que es el sistema de reglas que especifican cómo puede asignarse variables, expresiones, funciones u objetos a las propiedades que llamamos tipos. El sistema de tipos tiene por objetivo reducir la probabilidad de cometer errores programando, pero también es un objetivo ser flexible, tanto para programar algoritmos más eficientes como para hacer la experiencia de programar más cercana o amigable. Ambos aspectos están reñidos, dado que al aumentar la flexibilidad, aumenta típicamente la posibilidad de cometer errores. Se dice que un sistema de tipos puede ser fuerte o débil (*strongly typed* o *weakly/loosely typed*). No parece haber consenso sobre lo que significa que un lenguaje sea débilmente tipado, además de que hay diferentes grados de fortaleza o debilidad. Los lenguajes más conocidos que se asocian con tipado débil son C, JavaScript, Perl, Pascal y PHP. Suele decirse que son débiles porque un mismo valor puede asociarse a tipos diferentes y porque estos pueden mezclarse, por ejemplo, mediante la coerción (cuando en JavaScript sumas un *array* con un *string* y no rechista). Esta definición que acabo de dar es muy imprecisa desgraciadamente, confieso que no soy ningún experto en la materia. La mayoría de lenguajes son de tipado fuerte: Java, C#, Kotlin, Python, Ruby… lo que a efectos prácticos significa que los tipos son más restrictivos. Otra cualidad de los sistemas de tipos es que pueden ser dinámicos o estáticos. Esto es más fácil de definir, ya que el sistema estático es aquel que realiza las comprobaciones de las reglas en tiempo de compilación, mientras que el dinámico las hace durante la ejecución. Java, C#, C, C++, Kotlin, TypeScript y Scala son estáticos, mientras que Ruby, Python y JavaScript son dinámicos. En algunos lenguajes compilados se puede inferir los tipos sin especificarlos explícitamente, de manera que el tipo de una variable se sabe por el valor que se le asigna. Sucede así cuando en Java o en C# utilizamos la palabra reservada *var*. La inferencia no resta seguridad, porque el compilador valida las reglas perfectamente. Los lenguajes de tipado estático suelen tener una sintaxis más verbosa, porque hay que especificar más detalles para que el compilador pueda hacer bien su trabajo. A mucha gente le encanta Python o Ruby, porque pueden escribir programas con menos palabras y menos símbolos. La contrapartida es que cuesta más detectar errores de tipos. Hoy en día, los entornos de desarrollo integrado hacen un trabajo espectacular autocompletando y autogenerando código. Ya no resulta tedioso la verbosidad del lenguaje si sabes pedirle al IDE que haga el trabajo mecánico por ti. Lo que más me gusta de los lenguajes de tipado fuerte y estático,

cuando el IDE tiene buen soporte para ellos, es que hacer refactorizaciones resulta sencillo y seguro, parece magia. Además, las herramientas de análisis de código estático pueden hacer un valioso trabajo diagnosticando posibles problemas en el código. Para muchos, los lenguajes con tipos fuertes y estáticos, son el futuro. Incluso hay quien aboga por el desarrollo guiado por tipos. Conocer el sistema de tipos de los lenguajes es importante para ponderar las decisiones sobre cuál elegir en cada momento.

Genéricos

Más allá de entidades y objetos valor, existen tipos específicos que envuelven a otros tipos compuestos, como, por ejemplo, las colecciones de objetos, los decoradores, las promesas, los observables, etc. En Java y C# existe el concepto de genéricos, para el que se usa una sintaxis similar, con los símbolos de *mayor y menor que*:

C#

```
class Maybe<T>    // <-- indica que T es un tipo generico
{
    /*...*/
}
/*...*/
Maybe<User> maybeUser = repository.findUserBy(id);
```

A lo largo del libro, hemos visto varios ejemplos de uso de los genéricos. La pregunta que me surgía a mí cuando quise empezar a diseñar mis propios genéricos era, ¿en qué momento diseño mis clases como genéricas? La genericidad permite parametrizar métodos y clases sin llegar a concretar el tipo de cada parámetro. Una forma de darte cuenta de su potencial es cuando quieres evitar las conversiones de tipos (*typecast*). Imagina que queremos diseñar un pequeño mecanismo de caché en memoria, o quizá es una abstracción de un sistema de caché como *Redis*. Si no recurro a los genéricos y quiero poder guardar/recuperar cualquier tipo de objeto, voy a tener que recurrir al *typecasting* para que la solución compile. Veamos un ejemplo en C#:

C#

```
public class Cache
{
    private readonly IDictionary store = new Hashtable();

    public void StoreItem(object item, string key)
    {
```

```
        store.Add(key, item);
    }

    public object RetrieveItem(string key)
    {
        return store[key];
    }
}
/*...*/
var cache = new Cache();
cache.StoreItem("someString", "key1");
string retrievedItem = (string) cache.RetrieveItem("key1");
```

Al extraer un elemento de la caché, si lo quiero recuperar como cadena (porque sé que debe ser una cadena), no me queda más remedio que hacer la conversión de tipos. Para evitarla debemos sustituir el tipo concreto *object*, por un tipo genérico (indeterminado). En un primer paso, podemos mover la conversión de tipos adentro de la clase:

C#

```
    public T RetrieveItem<T>(string key)
    {
        return (T) store[key];
    }
}
/*...*/
string retrievedItem = cache.RetrieveItem<string>("key1");
```

Si queremos eliminar por completo la necesidad de la conversión de tipos, el siguiente paso es fijar en la clase el tipo genérico, elevándolo desde el método hacia la clase y apoyándonos en el hecho de que la clase *Dictionary* del *framework* admite genéricos:

C#

```
public class Cache<T>
{
    private readonly Dictionary<string, T> store = new Dictionary<string, T
        >();

    public void StoreItem(T item, string key)
    {
        store.Add(key, item);
    }

    public T RetrieveItem(string key)
    {
        return store[key];
    }
}
```

```
/*...*/
var cache = new Cache<string>();
cache.StoreItem("someString", "key1");
string retrievedItem = cache.RetrieveItem("key1");
```

En la definición de la clase, el tipo *T* es genérico. Es en el momento en que se instancia la clase cuando se resuelve el tipo concreto, que en este caso ha sido *string*. Tanto Java como C# permiten añadir restricciones al tipo T, como que pertenezca a un subtipo concreto, que sea un tipo con constructor,…

C#

```
class Cache<T>
    where T: SomeInterface
{
    /*...*/
}
```

La ventaja de los genéricos es que las comprobaciones se hacen en tiempo de compilación y que, además, los IDEs infieren perfectamente los tipos en cada uso, lo cual nos da mayor seguridad (y más rapidez gracias al autocompletado). Su uso es más común en *frameworks* y librerías que en el código específico de un dominio de negocio. Los genéricos tal cual los hemos visto aquí no utilizan herencia, sino una sustitución de tipos durante la compilación (*Cache<string>* no hereda de *Cache<T>*). En cierto modo, los genéricos son como plantillas donde el tipo se determina cuando se utiliza la plantilla. La relación que los genéricos tienen con el polimorfismo de subtipos, reside en la capacidad de usar tipos más o menos específicos que el especificado en la plantilla, a lo cual se llama covarianza y contravarianza.

Una «plantilla» puede estar compuesta de varios tipos genéricos:

C#

```
public abstract class Presenter<V, M>
    where V: View
    where M: Model
{
    public V View { get; }
    public M Model { get; }

    public Presenter(V view, M model)
    {
        View = view;
        Model = model;
    }
```

```
    public void ReadForm()
    {
        Model.Update(MapToModel(View));
    }

    protected abstract M MapToModel(V view);
}

/*...*/

public class LoginPresenter : Presenter<LoginView, LoginModel>
{
    public LoginPresenter(LoginView view, LoginModel model) : base(view,
        model) {}

    protected override LoginModel MapToModel(LoginView view)
    {
        /* ... mapping implementation here ... */
    }
}
```

Este ejemplo combina genéricos con herencia para que se pueda intuir hasta donde puede llegar la potencia de los genéricos. La primera clase es abstracta, porque no tiene suficiente conocimiento para implementar el último de sus métodos, ya que está trabajando con tipos genéricos, cuyos detalles específicos no puede conocer en ese punto. La segunda clase hereda de ella, y además ya fija los tipos con los que trabaja, de manera que se realiza la sustitución. Esta segunda clase podría dejar abiertos los tipos si hiciera falta, para seguir siendo genérica:

C#

```
public class LoginPresenter<V> : Presenter<V, LoginModel>
    where V : View
{
    public LoginPresenter(LoginViewType view, LoginModel model) : base(view,
        model) {}

    /*...*/
}
/*...*/
var loginPresenter = new LoginPresenter<QuickLoginView, LoginModel>();
/*...*/
```

La mejor forma de comprender cómo funcionan los genéricos es instanciando las clases, para que se produzca la sustitución y les encontremos el sentido. A la hora de nombrar, he decidido utilizar una sola letra para indicar que es genérico, lo cual parece romper con todo

lo que hablamos de buenos nombres en los primeros capítulos. En el caso de los genéricos puede tener sentido usar letras, porque el tipo es desconocido, inespecífico, realmente es una abstracción de un tipo. En este ejemplo, que se trata de una clase pequeña, las variables de instancia *View* y *Model* contribuyen a que se entiendan mejor tipos genéricos *V* y *M*, respectivamente. Además, el compilador de C# no deja que el tipo genérico y la variable de instancia tengan el mismo nombre (no compila). Está bastante extendido recurrir a una especie de «notación húngara» para los genéricos, con nombres como *TView* o *TModel*. Personalmente, para los genéricos prefiero nombres pronunciables como *ViewType* o *ModelType* o directamente una sola letra, cuando el contexto está bien acotado. Repito que es poco común recurrir a genéricos en código de dominio vertical, en lenguajes como Java o C#, porque los conceptos de negocio son bien concretos. El uso de genéricos en la capa de lógica de negocio podría ser un mal olor. En otros lenguajes donde los tipos son más ricos es posible que los genéricos sí jueguen un papel interesante en el dominio, esto es algo que tengo pendiente investigar y experimentar.

Cuando descubres y comprendes los genéricos, te dan ganas de usarlos por todas partes; esto nos pasa cada vez que aprendemos alguna característica potente del lenguaje. Aquí es cuando solemos cometer el error de introducir mucha más complejidad de la que es necesaria en el sistema. Si te emocionas mucho con lo que acabas de aprender sobre una determinada característica del lenguaje, ten especial cuidado de no introducirla en cualquier sitio, porque quizá la gente que venga luego a mantener el código no tenga el mismo entusiasmo. Se trata de encontrar un balance entre simplicidad y otros principios, como la reducción de la posibilidad de errores. Si finalmente se opta por una solución compleja, recuerda documentarlo todo pensando en quienes vendrán luego. Una buena batería de test complementa muy bien a esa documentación.

Tipos estructurales

En las primeras versiones de Java y de C#, los tipos específicos se limitaban a interfaces, clases y *enums*. Posteriormente, aparecieron tipos que representaban funciones (mediante delegados en C# e interfaces funcionales en Java). En ambos lenguajes, los tipos son nominales, es decir, un tipo es distinto de otro por el nombre que tenga. En cambio, en otros lenguajes como TypeScript o Go, los tipos son estructurales, es decir, un tipo es igual a otro si su estructura es la misma. Ejemplo en TypeScript:

Typescript

```
interface Point {
    x: number,
    y: number
}

interface AnotherPoint {
    x: number,
    y: number
}

function printPoint(p: Point){
    console.log(p);
}

let p1: Point = {x:1, y:2};
let p2: AnotherPoint = {x:3,y:4};

printPoint(p1);
printPoint(p2);   // <-- compila y funciona perfectamente
```

De hecho, un tipo puede pasar por el otro si su estructura es un superconjunto de la otra, por ejemplo, si *AnotherPoint* tuviera otra propiedad más, también funcionaría.

La capacidad expresiva de los tipos en el lenguaje TypeScript es mayor que en Java o C#, no solo por los tipos estructurales, sino por la cantidad de operadores restrictivos y funcionales que se aplican sobre los tipos:

Typescript

```
interface Car {
  manufacturer: string;
  model: string;
  year: number;
}

let taxi: Car = {
  manufacturer: "Tesla",
  model: "Model 3",
  year: 2017,
};

function getProperty<T, K extends keyof T>(anObject: T, propertyName: K): T[K
    ] {
  return anObject[propertyName]; // o[propertyName] is of type T[K]
}

let manufacturer: string = getProperty(taxi, "manufacturer");
console.log(manufacturer);                    // <- "Tesla"
let year: number = getProperty(taxi, "year");
console.log(year);                            // <- 2017
```

```
let unknown = getProperty(taxi, "unknown"); // <- no compila
// Argument of type '"unknown"' is not assignable to parameter of type 'keyof
     Car'.
```

La función *getProperty* usa tipos genéricos que, al igual que en Java y C#, se expresan con los símbolos de mayor y menor. El tipo *T* es el del primer parámetro de la función, mientras que *K* es el tipo del segundo parámetro. Las palabras reservadas *extends keyof*, restringen la cantidad de valores que se admiten para un argumento de tipo *K*, especificando que deben ser propiedades que pertenecen a *T*. La función genérica no solo comprueba los tipos en tiempo de compilación, sino también su rango de valores posibles. Además, se restringe el tipo y el valor devuelto, forzando a que solo puedan ser valores existentes dentro del objeto (*T[K]*). El tipado está haciendo buena parte de las comprobaciones que en JavaScript tendríamos que hacer escribiendo test automáticos. El beneficio es una rotunda seguridad sobre los tipos de entrada y de salida. El coste de esta potencia del tipado, es un mayor esfuerzo para comprenderlo y para razonar sobre lo que implica. Este ejemplo es sencillo, pero los tipos se pueden complicar mucho más, por ejemplo, si combinamos tipos condicionales con tipos mapeados (que realizan operaciones de mapeo cuando se usan).

Tipos algebraicos

Los tipos como *Optional*, *Try* o *Either* son tipos suma. La suma, el producto y el exponente, son tipos compuestos, tipos algebraicos concretamente. En el producto se requieren todos los campos que componen al tipo (es como una operación lógica *and*), mientras que en la suma solo hace falta uno de ellos (como un operación lógica *or*). Una tupla sería un tipo producto por ejemplo. El tipo exponente son las funciones. Hay lenguajes cuyo sistema de tipos está diseñado para poder definir fácil, rápida y explícitamente tipos algebraicos. El origen de estos tipos está en lenguajes puramente funcionales como Haskell, así que están muy bien soportados en Scala y F#, pero también en lenguajes multiparadigma modernos como TypeScript, Kotlin o Swift.

Un simple registro es un tipo producto en F#:

F#

```
type WiredConnection = {
//  nombre del campo: su tipo
```

```
    Name: ConnectionName
    Mac: MacAddress
    Ip: IpAddress
}
```

El compilador obliga a que absolutamente todos los campos estén rellenos para dar el registro por válido y es capaz de inferir el tipo:

F#

```
let wiredConnection = {
// campo = llamada al constructor del tipo
   Name = ConnectionName "eth0";
   Mac = MacAddress "F8:A9:63:DD:E2:39";
   Ip = IpAddress "192.168.1.190";
}
```

Aunque también puede especificarse el tipo explícitamente:

F#

```
let wiredConnection : WiredConnection = {
   Name = ConnectionName "eth0";
   //...
}
```

Los tipos como *ConnectionName* son abstracciones de tipos integrados que he creado para ocultarlos del dominio y se pueden definir como uniones o como registros:

F#

```
// single case union
type ConnectionName = ConnectionName of string
// o bien, record type:
type ConnnectionName = { ConnectionName: string }
```

A diferencia del producto, la suma se define como una serie de opciones excluyentes (unión discriminada):

F#

```
type NetworkConnection =
    | Wired of WiredConnection
    | Wifi of WirelessConnection
```

Una variable de tipo *NetworkConnection*, podría tener rellena la propiedad *Wired* o bien la propiedad *Wifi* (pero no las dos), y cada una tiene a su vez su propio tipo. Los lenguajes que integran una sintaxis explícita para definir tipos suma, suelen ofrecer también la capacidad de reconocer patrones (*pattern matching*) para el manejo de los mismos. Veamos un ejemplo de función que imprime información según el tipo. El símbolo flecha se utiliza para las funciones anónimas, igual que en JavaScript o TypeScript:

F#

```
//  nombre de la funcion (nombre argumento: tipo argumento)
let printConnectionType (connection: NetworkConnection) =
    match connection with
    | Wired (wired) -> printfn "Wired: %A" wired.Name
      // ^ si connection es tipo Wired
    | Wifi (wifi) -> printfn "Wifi: %A" wifi.SSID
      // ^ si es de tipo Wifi
```

Uso de la función:

F#

```
// definición de variable:
let networkConnection : NetworkConnection = Wired {
    Name = ConnectionName "eth0";
    Mac = MacAddress "F8:A9:63:DD:E2:39";
    Ip = IpAddress "192.168.1.190";
}
// también sería inferido el tipo definiendo la variable así:
// let networkConnection = Wired {

// llamada a la función:
printConnectionType networkConnection
// Imprime: "Wired: ConnectionName eth0"
```

La unión de tipos tiene mucha utilidad, desde atajos como la sobrecarga de métodos en TypeScript, hasta el modelado de un dominio de negocio rico. Por compatibilidad con JavaScript, TypeScript no tiene una sobrecarga al estilo de Java o de C#, pero se puede simular con uniones:

Typescript

```
type Id = {
    id: string;
}
function createId(id: number | string) : Id {
    if (typeof id === "string") {
       return {id : id.toLowerCase()};
```

```
    }
    else {
       return {id : String(id)}
    }
}
```

La potencia de los tipos algebraicos para el modelado es envidiable, Scott Wlaschin lo explica de maravilla en su fantástico libro *Domain Modeling Made Functional*. Vamos a poner como ejemplo el modelado de un billete de avión para un vuelo dentro de España. El gobierno subvenciona parte del coste de los vuelos nacionales para familias numerosas y residentes de regiones periféricas. Los descuentos aplicables son los siguientes:

- *Large Family* (10 %)
- *Resident of Islands or Ceuta* (75 %)
- *Special Resident/Large Family* (85 %)

Si lo modelamos como una clase en Java, tenemos múltiples opciones. La peor de todas es usar primitivos:

Java

```java
class FlightOrder {
    private final boolean isLargeFamily;
    private final float largeFamilyDiscountPercentage = 10;
    private final boolean isResidentOfIslandsOrCeuta;
    private final float residentsDiscountPercentage = 75;
    private final boolean isSpecialResident;
    private final float specialResidentDiscountPercentege = 85;
    /*...*/

    FlightOrder(boolean isLargeFamily, boolean isResidentOfIslandsOrCeuta,
        boolean isSpecialResident /*,...*/) {
        this.isLargeFamily = isLargeFamily;
        this.isResidentOfIslandsOrCeuta = isResidentOfIslandsOrCeuta;
        this.isSpecialResident = isSpecialResident;
        /*...*/
    }
    /*...*/
}
```

No es buena idea por la cantidad de inconsistencias (*bugs*) que se pueden dar, ya que este diseño admite que varios de los campos booleanos pudieran ser verdaderos a la vez y eso en el dominio no tiene sentido. Un buen diseño debe respetar las propiedades invariantes del dominio. Una mejor alternativa sería definir un enumerado:

Java

```
enum SpanishFlightTicketDiscount {
    LargeFamily(10),
    ResidentofIslandsorCeuta(75),
    SpecialResidentLargeFamily(85),
    None(0);

    private final float percentage;

    SpanishFlightTicketDiscount(float percentage) {
        this.percentage = percentage;
    }

    public float percentage(){
        return percentage;
    }
}

class FlightOrder {
    private final SpanishFlightTicketDiscount discount;
    /*...*/
    FlightOrder(SpanishFlightTicketDiscount discount /*, ...*/) {
        this.discount = discount;
        /*...*/
    }
    /*...*/
}
```

Con este diseño ya no es posible seleccionar dos descuentos a la vez, el compilador no lo permite. Ahora añadimos un nuevo requisito, porque resulta que los pasajeros del *Club Bussines* también tienen un descuento y que es combinable con la subvención del gobierno en el caso de residentes. Con los productos y las sumas, podríamos modelar el descuento de un vuelo con F#, tal que así:

F#

```
type SpanishSubsidy =
   | SpecialResident of SpecialResidentDiscount
   | Resident of ResidentDiscount
   | Family of FamilyDiscount

type FlightDiscount =
   | Subsidy of SpanishSubsidy
   | Bussines of BussinesClub
   | SubsidyPlusBussines of ResidentDiscount * BussinesClub
   | NoDiscount
```

La opción *SubsidyPlusBussines* es el producto de *ResidentDiscount* y de *BussinesClub*, lo

que quiere decir que ambos están a la vez. Con esta definición de tipos, el compilador se asegura de que sea imposible que una variable de tipo *FlightDiscount* pueda contener una combinación incorrecta de descuentos. La regla de negocio queda implícita en la definición del tipo, validada por el compilador[1].

El código que calcula el porcentaje, podría ser algo como esto:

F#

```
// tipo funcion que recibe FlightDiscount y devuelve Percentage:
type CalculateDiscountPercentage = FlightDiscount -> Percentage
// implementación del tipo:
let  calculatePercentage : CalculateDiscountPercentage =
    fun discount ->
        match discount with
        | Subsidy (s) -> SpanishSubsidy.percentage s
        | Bussines (b) -> BussinesClub.percentage b
        | SubsidyPlusBussines (s, b) -> BussinesClub.cumulativePercentage b s
        | NoDiscount -> Percentage.zero
```

Cuando utilizamos *match*, el programa no compila si olvidamos poner alguna de las opciones posibles. Cuanto más potencia expresiva tienen los tipos, más comprobaciones hace el compilador y más se reduce la probabilidad de errores.

Si pensamos en implementar lo mismo con un lenguaje como Java o C#, nos damos cuenta enseguida de que se necesitaría bastante más código y quizá test unitarios para asegurarnos de que la regla de negocio de los descuentos se cumpliera. Los tipos algebraicos tampoco son infalibles, podríamos cometer un error rellenando una instancia de este tipo a partir de los datos que nos llegan:

F#

```
// tipo de los datos de entrada:
type FlightOrderRequest = {
    SpanishSubsidy: SpanishSubsidy option  // valor opcional
    BussinesClub: BussinesClub option      // valor opcional
    // passenger info over here...
}
// tipo de la función que crea el descuento:
type DiscountFromOrderRequest = FlightOrderRequest -> FlightDiscount
// posible implementación:
```

[1] En un grupo de estudio hicimos el ejercicio de intentar transportar este código a TypeScript, tratando de comprender qué tan potente es el soporte para uniones discriminadas en este lenguaje, y encontramos una posible solución usando *type* (https://bit.ly/3LpAkbR) y otra usando *class* (https://bit.ly/3GU1Vih). Descubrimos que el soporte del compilador no es tan fuerte como lo es en F#, pero aún así puede llegar a ser bastante útil para modelar.

```
let discountFromOrderRequest : DiscountFromOrderRequest =
    fun request ->
        match request.BussinesClub with
        | Some(bussinesClub) ->
            match request.SpanishSubsidy with
            | Some(subsidy) ->
                match subsidy with
                | Resident (r) -> SubsidyPlusBussines (r, request.
                    BussinesClub.Value)
                | _ -> Subsidy (subsidy)
            | None -> Bussines (request.BussinesClub.Value)
        | None ->
            match request.SpanishSubsidy with
            | Some(subsidy) -> Subsidy (subsidy)
            | None -> NoDiscount
```

¿Quién dice que este código no tiene *bugs*? Podría equivocarme si elijo el descuento incorrecto en función del dato que recibo, y aunque el tipo seguiría siendo consistente, habría un *bug* en el sistema, por lo que se requieren test unitarios aquí.

La contrapartida de los tipos avanzados es que conforme crece su complejidad, lo hace también la carga cognitiva que supone trabajar con ellos. En otros lenguajes funcionales como Haskell, comprender los tipos puede llegar a ser todo un reto.

Espero que este capítulo te anime a crear tus propios tipos específicos para modelar el negocio y a relegar los tipos integrados al manejo de *software* de terceros y a la composición de tus tipos.

11. Principios malinterpretados

El auge del *software* durante la primera mitad del siglo XX, estuvo marcado por la Segunda Guerra Mundial y la aparición de las primeras máquinas digitales, usadas para defensa y para la industria. La segunda mitad estuvo marcada por la carrera espacial, la Guerra Fría, la evolución de las máquinas industriales y, finalmente, la aparición de las computadoras domésticas. Prácticamente hasta la década de los 80, no hubo computadoras personales en los hogares, ni aplicaciones para ellas. Durante la mayor parte de la historia de la computación, el contexto en el que se desarrolló el *software* fue radicalmente diferente al de la actualidad.

Los programas consistían básicamente en cálculos matemáticos para cifrado, fabricación industrial, cálculo de rutas… y las computadoras eran calculadoras gigantes. ENIAC fue probablemente la segunda máquina electrónica programable de propósito general de la historia, inicialmente diseñada para calcular tablas de tiro de artillería. Es del año 1946, pesaba 27 toneladas y curiosamente, todas las personas que la programaban eran mujeres. Los recursos de memoria y de CPU eran muy limitados. El mecanismo de interacción con las máquinas fue el de las tarjetas perforadas, prácticamente hasta la década de los 70 (no había pantallas, sino tarjetas de cartulina con agujeritos). Esta era la herencia de las máquinas industriales mecánicas «programables». Se usaban diferentes codificaciones según las instrucciones soportadas por cada máquina específica. Con la aparición de máquinas digitales, la codificación estaba fuertemente influenciada por el sistema binario, perforando las tarjetas con código BCD (*Binary-coded decimal*) o EBCDIC, entre otros. El lenguaje ensamblador fue concebido en 1947. A finales de 1949, apareció la primera máquina con ensamblador inspirada en el trabajo de John von Neumann (EDSAC) y con ella el que fue considerado como primer libro de programación de computadoras digitales (1951). Más tarde, aparecieron los lenguajes de alto nivel; FORTRAN, ALGOL, Lisp y COBOL (en torno a 1957). Las primeras ideas de programación orientada a objetos implementadas en un lenguaje de programación, se plasmaron en Simula 67 (en 1967), influenciado por ALGOL. Tanto FORTRAN como ALGOL, influenciaron después al lenguaje C, que nació en 1973, se popularizó durante la década de los 80 y para muchos hoy sigue siendo el

lenguaje de programación por excelencia[1]. C++ es de 1982. Los lenguajes con gestión automática de la memoria no aparecieron hasta finales de los 90, cuando las computadoras domésticas alcanzaron una capacidad de cálculo y de memoria, impensables pocos años antes. La llegada de Internet a los hogares de los países más desarrollados también fue a finales de la década de los 90 (yo di mi primer paseo por Internet en 1999, desde el centro de cálculo de la Facultad de Informática de la ULL, cuando tenía 18 años).

La historia[2] es muy reciente, ha habido muchos cambios en poco tiempo, y no todo el mundo ha sabido adaptarse a la evolución del contexto. El estilo de programación predominante durante más años ha sido el científico que, como es natural, está influenciado por las matemáticas y la física. Por eso, era común usar variables con una sola letra, abreviaturas y comentarios en el código. Además, era típico mezclar lógica de cálculo con lógica de presentación en pantalla. Veamos un ejemplo escrito en FORTRAN:

Fortran

```
C Heron formula to calculate the area
C of a triangle based on the length of
C its three sides a, b & c.
C Area = √s(s-a)(s-b)(s-c)
C s is the semi-perimeter:
C s = (a+b+c) / 2

subroutine heron(a,b,c)
  implicit none
  real, intent(in) :: a, b, c
  real :: s
  real :: r
  logical :: t1, t2

  t1 = ((a .gt. 0.0) .AND. (b .gt. 0.0) .AND. (c .gt. 0.0))
  t2 = ((a+b .gt. c) .AND. (a+c .gt. b) .AND. (b+c .gt. a))
  if (t1 .AND. t2) then
     s = (a + b + c) / 2.0
     r = sqrt(s*(s-a)*(s-b)*(s-c))
     write(*,*) 'Triangle area is ', r
  else
     write(*,*) 'Not a triangle'
  end if

end subroutine heron
```

[1] En mi opinión, conocer un poco el lenguaje C es muy enriquecedor para cualquiera que programe, si bien es cierto que no lo recomiendo como primer lenguaje. También me gustó mucho conocer un poco de ensamblador para entender mejor la arquitectura de las computadoras.

[2] El libro (https://bit.ly/3gG9yOx) de Charles Petzold sobre historia de la computación, es una lectura muy recomendable para ampliar con rigor la información que he resumido en esta introducción.

Al fin y al cabo, en matemáticas es típico expresar las fórmulas con una sola letra, típicamente con símbolos del alfabeto griego[3]. Ciertamente, para una fórmula matemática bien conocida, resultaría más farragoso utilizar nombres expresivos y pronunciables incluso en lenguajes modernos:

Javascript

```
function heronFormula(sideA, sideB, sideC){
    let semiperimeter = (sideA + sideB + sideC) / 2;
    let area = Math.sqrt(semiperimeter *
       (semiperimeter - sideA) * (semiperimeter - sideB) * (semiperimeter -
           sideC));
    return area;
}
```

Ahora bien, en toda mi carrera profesional desarrollando *software* empresarial, no recuerdo haber programado nunca fórmulas matemáticas, lo más parecido que hice fue utilizar alguna librería que ya las implementaba.

Hoy los programas se desarrollan para personas conectadas a Internet, no para máquinas industriales; el *software* para fábricas ocupa un nicho minúsculo. El *software* está por todas partes y su cometido es resolver problemas a los humanos, nada que ver con el contexto previo a la década de los 90. Aunque todo ha cambiado, se enseñan formas de programar arcaicas sin saber que ya no tienen sentido, porque son cultos del cargo, es decir, se aprendieron una vez como un dogma y se transmiten como tal. No solo son principios o técnicas innecesarias, sino que pueden llegar a ser dañinas. Conocer el contexto histórico nos permite razonar sobre algunos de los consejos «clásicos» que se han enseñado a muchas promociones de programadoras y programadores, para entender cuándo son contraproducentes. A continuación, revisamos algunos de estos clásicos consejos pasados de moda, estudiando de dónde vienen y por qué ya no sirven. De nuevo, hablaré de *función* y de *método* como sinónimos.

Un solo *return* por función

Cuando se programaba en ensamblador, se utilizaban los saltos a etiquetas para simular rutinas y bucles, dado que el lenguaje no las soportaba. Ejemplo en ensamblador x86:

[3] A mí, personalmente, me cuesta mucho comprender teoremas matemáticos y de ciencias de la computación cuando hay más de tres o cuatro símbolos del alfabeto griego juntos.

Assembly

```
mov    ax, 5          ; set ax to 5.
mov    bx, 2          ; set bx to 2.
jmp    calc           ; go to 'calc'.
back:  jmp stop       ; go to 'stop'.
calc:
add    ax, bx         ; add bx to ax.
jmp    back           ; go 'back'.
stop:
ret
```

La utilización de saltos como mecanismo de control del flujo de ejecución, se heredó a la siguiente generación de lenguajes, siendo muy popular en BASIC. Leer un programa donde el flujo se gestiona con saltos, es un rompecabezas. Cuando apareció la programación estructurada, se hizo mucho énfasis en dejar de utilizar los saltos, considerando que el retorno de una subrutina (función/procedimiento/método) era un salto, y que, por tanto, debía de ser único.

Otro motivo importante para tener un único punto de retorno tiene que ver con la gestión de la memoria en lenguaje C, ya que se programa reservando memoria (*malloc*) y liberándola (*free*) de manera explícita. Los errores en la gestión de la memoria son un quebradero de cabeza, cualquiera que haya programado en C teme al error de *Segmentation fault* y a las *memory leaks*. La disciplina para liberar la memoria antes de salir de una función era crítica y la mejor forma de acordarse era liberando justo antes de retornar, para lo cual un solo punto de retorno era muy recomendable.

En lenguajes modernos con gestión automática de la memoria ya no se utilizan los saltos para programar y el recolector de basura se encarga del trabajo de liberación de memoria. Las fugas de memoria son un *bug* muy poco frecuente comparado con la mutación indebida del estado o con los errores en condiciones lógicas. Si cuando encontramos uno de los posibles resultados de una función, esperamos a devolverlo en la última línea de código, estamos arriesgándonos a que esa variable mute accidentalmente por el camino y se devuelva un valor incorrecto. Además, para conseguir un solo punto de retorno a menudo tenemos que recurrir a complicadas condicionales o a bloques anidados:

Java

```
public List<String> listChannels(){
    List<String> channels = new ArrayList<>();
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O){
      NotificationManager manager = notificationManager();
      if (manager != null){
```

```
            List<NotificationChannel> listChannels = manager.
                getNotificationChannels();
            for(NotificationChannel channel : listChannels) {
               channels.add(channel.getId());
            }
        }
    }
    return channels;
}
```

Veamos una alternativa con la misma funcionalidad, usando varios puntos de salida:

Java

```
public List<String> listChannels() {
    if (Build.VERSION.SDK_INT < Build.VERSION_CODES.O){
        return Collections.emptyList();
    }
    if (notificationManager() == null){
        return Collections.emptyList();
    }
    List<String> channels = new ArrayList<>();
    for(var channel : notificationManager().getNotificationChannels()) {
        channels.add(channel.getId());
    }
    return channels;
}
```

No tiene sentido seguir recomendando que las funciones tengan un único punto de retorno, es un dogma, un culto del cargo. En lenguajes de alto nivel modernos, priorizamos la simplicidad y la reducción de la probabilidad de defectos.

Las constantes siempre van definidas arriba del todo del fichero

Una constante no es diferente de cualquier otra abstracción salvo por el hecho de que su valor es asignado solamente una vez. Ser constante no es suficiente motivo para hacerla accesible/visible a todo el fichero, cuando solo se usa en un lugar puntual. Hay varios problemas al programar así. El primero es que quien lee el fichero se encuentra enseguida con una constante que no sabe para qué es, porque está lejos de su contexto. Esto es como contar partes de una historia que todavía ni ha comenzado. En el cine es muy interesante el *flashforward*, pero en el código no. El segundo es que la exposición excesiva puede aco-

plar partes del fichero que deberían de ser independientes. La definición de la constante en el ámbito más accesible es una peligrosa invitación a usarla donde no corresponde. Si además su visibilidad es pública para el resto de ficheros, las probabilidades de aumentar el acoplamiento accidental se disparan.

¿Por qué se ha recomendado tanto que las constantes vayan arriba? No sé el motivo, pero me lo puedo imaginar. He visto sistemas de gestión empresarial (ERP) escritos en COBOL hace 30 años (y mantenidos en la actualidad) que resultan infernales por la cantidad de códigos que tienen repetidos por todo el fichero. Me refiero a literales mágicas, tanto de tipo cadena como de tipo numérico. El problema no es COBOL, sino el estilo de programación que han usado (el problema no suele estar en el lenguaje, sino en la forma de usarlo), en el cual es muy probable equivocarse repitiendo esos códigos en ficheros con miles de líneas (no era raro encontrar ficheros con 50k+ líneas). Cuando hay tanta repetición de valores *hardcoded*, veo todo el sentido del mundo a definir las constantes arriba del todo, para que por lo menos no haya errores escribiendo esos códigos. En ausencia de código modular, con gigantescos y caóticos *scripts*, poner las constantes arriba era mejor que no hacerlo. He aquí un fragmento de código (real), extraído de un fichero de 30k líneas, donde ayudarían las constantes globales:

Cobol

```
IF SWPAN = 3 IF TAB2N(K)(198:1) NOT = "R"
DISPLAY TAB2N(K)(1:17) LINE J COL 1
DISPLAY TAB2N(K)(134:64) LINE J COL 17 END-IF
IF TAB2N(K)(198:1) = "R"
DISPLAY TAB2N(K)(1:17) LINE J COL 1 LOW REVERSE
DISPLAY TAB2N(K)(134:64) LINE J COL 17
LOW REVERSE END-IF
END-IF
```

Salvo casos extremos como este, cuanto más local sea la definición de una constante, mejor se entiende el código, se gana cohesión y se reduce acoplamiento.

Como curiosidad, el compilador de Kotlin quita la definición de la constante cuando está lejos de donde se usa, reubicándola allí (*constant inlining*).

Las variables se definen todas al comienzo de cada función

Si has leído todo el libro con atención hasta llegar aquí, ya puedes explicar por qué definir las variables al comienzo de la función no suele ser buena idea. Por un lado, está el efecto *flashforward* de las constantes globales que, además, en el caso de las variables, se puede convertir en *flashback* cuando leemos código y tenemos que regresar, pantalla arriba, para buscar una inicialización o una definición.

Hace décadas, definir las variables al comienzo de un procedimiento pretendía poner orden en los programas. Lenguajes como Pascal (heredero de ALGOL) están diseñados de manera que las variables se definen al principio, para estructurar el código de forma homogénea (programación estructurada). Como hemos visto, la programación con saltos era tan difícil de mantener que se necesitaron medidas disciplinarias drásticas para acabar con ella. Si además tenemos en cuenta que el código se leía en papel, en vez de un IDE, es comprensible que se programase así (estas medidas son de la década de los 70). Incluso hoy en día, al programar en C, también puede tener sentido obligarnos a reservar memoria al principio de la función y liberarla antes de salir (no siempre).

En lenguajes modernos es al contrario, lo más conveniente es definir las variables en el lugar más cercano a su utilización, reduciendo lo máximo posible su ámbito (*scope*). Esto es así por varios motivos:

1. Porque las consecuencias serán lo más locales posibles, reduciendo la probabilidad de mutaciones accidentales: cuanto menos tiempo viva la variable, menor riesgo hay de que se use para lo que no se debe.
2. Porque es más rápido comprender un bloque o una función de un solo vistazo, que haciendo movimientos oculares arriba y abajo, o haciendo *scroll* por el fichero.
3. Porque al reducir el contexto en el que vive la variable, nos va a resultar más sencillo encontrar nombres precisos para ella, puesto que dicho contexto aporta valiosa información complementaria. Incluso, es posible que nos demos cuenta de que ciertas variables auxiliares son innecesarias y las podamos eliminar aplicando el *refactor inline*.
4. Porque al estar la definición y el uso en el mismo ámbito, el compilador es capaz de alertar de problemas, como la falta de inicialización de la variable.
5. Porque es más intuitivo, rápido y seguro aplicar refactorizaciones como extraer método, cuando todos los elementos que van a ir al nuevo método están juntos. Conseguimos código que predispone a la refactorización, aumentando la probabilidad de

practicarla a diario.

Asignar a todas las variables un valor por defecto en su definición

Cuando se programa en C y se define un puntero sin inicializar (que no apunta a una dirección de memoria explícita), el acceso al mismo provoca efectos inesperados difíciles de detectar, a menudo concluyendo con la interrupción del programa (*segmentation fault*), incluso en puntos del programa aparentemente no relacionados. Los punteros han sembrado el pánico durante muchas generaciones, siendo origen de mitos y leyendas. No faltan razones para ello. Si bien son muy eficientes (por eso, se siguen usando en C y C++ sobre todo), son duros de entender y de manejar.

A diferencia de C, en lenguajes con la memoria gestionada automáticamente, las variables están ya inicializadas a su valor por defecto. Los tipos integrados tienen todos un valor:

- números: *0 (cero)*
- *boolean*: *false*
- *char*: *"\u0000"*
- *string*: *null*

Por tanto, es redundante especificar que un número esté inicializado en cero. En cuanto a los objetos, el valor por defecto es nulo, siempre. Es cierto que existe temor a no inicializar un objeto, por ejemplo, una cadena, y que el programa se interrumpa con una excepción de tipo *NullPointerException*, *NullReferenceException* o similar. Por eso, hay quien tiene la vieja costumbre de asignar siempre la cadena vacía en la definición de la variable, para no olvidarse. Hoy en día es preferible dejar que el IDE y el compilador nos lo recuerden. Cuando intentamos usar un objeto no inicializado, cuya definición está en el mismo ámbito, tanto el IDE como el compilador nos avisan de que hay un error. Si inicializamos con valores irrelevantes o incorrectos perdemos esta asistencia de las herramientas. Es muchísimo más barato descubrir los errores en tiempo de edición que en tiempo de ejecución. No hay necesidad de inicializar las variables con valores que no se van a usar. Otro beneficio de no asignar valores irrelevantes al definir variables, es que nos obligamos a razonar sobre el uso de cada variable; *¿de verdad necesito una variable para esto?* Cada variable supone, al final, una elección de nombre y nombrar es difícil. Debemos conocer cómo se comporta nuestro

lenguaje de programación del día a día, alejándonos de la idea de que hace magia y de que los comportamientos son misteriosos. No conseguiremos escribir programas sólidos con la mentalidad de que los lenguajes son mágicos. Queremos tener el control total sobre el comportamiento del código en todo momento. Si no tienes claro el comportamiento exacto del código en cada punto, mejor haz las pruebas de concepto que necesites para comprenderlo, en vez de «probar a tener suerte».

No sobrescribir los parámetros de una función

Este es otro consejo que probablemente venga del respeto a los punteros. Cuando una función en C recibe como argumento el puntero al primer caracter de una cadena (*char**), y dicha función accede a esa posición de memoria actualizando los caracteres en ella, produce una mutación permanente. O sea, que la variable definida por fuera de la función queda alterada tras su ejecución. Tal efecto secundario puede resultar sorprendente para quien invoca a la función. Ocurre lo mismo si definimos parámetros en C# con la palabra reservada *ref*, puesto que entonces no se produce una copia de la referencia (cuidado con *ref*). Por eso, se desaconseja que los valores de los parámetros se sobreescriban. Sin embargo, en lenguajes modernos no hay ningún tipo de problema con reasignar valores a los parámetros:

Java

```java
public String removeSpecialSymbols(String expression){
    expression = expression.replace("*", "");
    /*...*/
    return expression;
}
```

Prohibir la reasignación de los parámetros en lenguajes donde no hay punteros, tiene varios inconvenientes. Por un lado, produce una sensación de inmutabilidad que es falsa cuando trabajamos con objetos, ya que las mutaciones en los campos de un objeto son permanentes, aunque no se sobrescriba el valor de la variable que apunta al objeto (la referencia):

Java

```java
public Item normalize(Item item){
    item.name = item.name.toLowerCase(); // mutación permanente
    /*...*/
}
```

Por otro lado, obliga a introducir variables redundantes en el código que abren la puerta a nombres sin sentido, introduciendo ruido en el código:

Java

```java
public String removeSpecialSymbols(String expression){
    String aux = expression;   // redundante: aux, tmp...
    aux = aux.replace("*", "");
    /*...*/
    return aux;
}
```

Incluso introduciendo nombres con sentido, puede llegar a ser demasiado verboso o confuso:

Java

```java
public String removeSpecialSymbols(String expression){
    String expressionWithoutAsterisk = expression.replace("*", "");
    String expressionWithoutSpaces = expressionWithoutAsterisk.replace(" ", "
        ");
    /*...*/
    return expressionWithoutSpaces; // <- parece que solo quitó los espacios
}
```

No hay necesidad de complicarse tanto. El paso de argumentos produce copias de los valores en el caso de los primitivos y copias de las referencias en el caso de los demás objetos. Ocurre lo mismo con la asignación de variables. En realidad, el comportamiento es igual en C, de hecho, cuando se utilizan punteros como argumentos, se pasa una copia del puntero y por eso en aquellos casos en los que se necesita modificar punteros, se utiliza el doble puntero (doble asterisco). Ejemplos de test en lenguaje C con cmocka:

C

```c
static void pointers_assignment_produces_copies() {
    int i = 10;
    int j = 25;
    int *x = &i;
    int *y = x;
    y = &j;
    assert_ptr_not_equal(x, y);
    assert_int_equal(*x, 10);
    assert_int_equal(*y, 25);
}
```

C

```
static void double_pointers_are_used_to_change_pointers() {
    int i = 10;
    int j = 25;
    int *x = &i;
    int *y = &j;
    int **z = &x;
    *z = y;
    assert_int_equal(*x, 25);
}
```

Para las personas con menos experiencia programando, es necesario recalcar que cuando reasignamos el valor de un parámetro o de cualquier otra variable, debemos mantener su tipo, aunque el lenguaje nos permita cambiarlo. Por ejemplo, con Python tendríamos la capacidad de recibir una cadena, trabajar con ella y en algún momento asignarle a la misma variable un número:

Python

```
def sum_numbers_from_expression(expression):
    total = 0
    expression = expression.strip()
    #...
    expression = total       # ???
    #...
```

La consistencia de tipos es fundamental, tanto en las variables como en las funciones, queremos preservarla incluso si el lenguaje nos da permiso para no hacerlo. Cuando se diseña una función para que devuelva cadenas, siempre devolverá cadenas y así con todos los tipos. La consistencia evita sorpresas a quien lee y hace más clara la intención de quien programa.

Comprender el comportamiento de cada lenguaje de programación es fundamental, tanto para escribir programas robustos, como para encontrar y corregir *bugs* de forma ágil y segura. Entender es un camino más productivo que temer (a maldiciones o a magia negra). Otra razón para escribir test unitarios es que ganamos confianza en lo que hacemos y podemos poner el foco en escribir código legible, conciso y con buenos nombres.

Las interfaces desacoplan

Las comunidades de Java y de C#, han atravesado épocas de abundante sobreingeniería, sobre todo a principios de los 2000, cuando aún era tecnología muy reciente. Algunas ideas eran buenas, pero las sacamos de contexto y las usamos para lo que no estaban pensadas, como las JavaBeans, que eran para GUI, pero se usaron para todo. Otras ideas han sido descartadas por su complejidad, como sucede con los EJB (*Enterprise Java Beans*) y reemplazadas por tecnología más simple y menos invasiva. Un principio de esa época que está muy arraigado y que todavía se ve mucho es el de que cada clase debe implementar una interfaz:

Java

```
interface ShoppingService {/*...*/}
class ShoppingServiceImpl implements ShoppingService {/*...*/}
```

Las interfaces son un mecanismo para definir un contrato que varias clases van a cumplir. En realidad, una clase también expone un contrato formado por sus métodos públicos. Definir interfaces que solamente van a tener una implementación y que además se van a desplegar siempre en el mismo paquete, no mejora el código. Es tedioso navegar por proyectos en el que cada clase implementa una interfaz, porque cuando quieres ver la implementación, el IDE salta a la interfaz, así que hace falta otro salto más para llegar. Terminas con el doble de ficheros abiertos, la mitad de los cuales no aportan nada.

Crear interfaces de antemano puede servir para que diferentes equipos avancen en paralelo trabajando en subsistemas diferentes, estableciendo las interfaces como fronteras. La separación de equipos tiene sentido cuando cada uno puede desplegar sus artefactos con independencia del resto, para lo cual han de poder generar los entregables con independencia. En este caso, es práctico diseñar una interfaz, incluso aunque al final termine con una sola implementación (el paquete que contiene la interfaz se distribuye en una librería compartida por los equipos, mientras que las implementaciones van en paquetes separados). De todas formas, yo haría un esfuerzo por escoger nombres pronunciables y sin información técnica, descartando el comodín del sufijo *Impl* o del prefijo *I*, como vimos en el capítulo sobre los nombres, para tener más pistas sobre el diseño.

Históricamente, en *stacks* tecnológicos como J2EE, se han usado las interfaces en *frameworks* para que diferentes servidores de aplicaciones las implementasen a su manera. En el contexto de librerías y *frameworks*, se tienen necesidades diferentes a las que surgen en

código vertical, por lo que las técnicas de diseño también son diferentes. Las buenas prácticas de ingeniería de los fabricantes de código horizontal, de propósito general, resultan sobreingeniería cuando las copiamos en el desarrollo de aplicaciones verticales.

Las interfaces siempre se pueden extraer mediante refactorización automática cuando llega el momento en que son necesarias. Los IDE que había a principios de los 2000, no tienen nada que ver con los de hoy. Ahora cualquier IDE genera de forma automática una interfaz a partir de una clase y cambia todo el código que la consume para hacer referencia a la interfaz, en lugar de a la clase. Como estas herramientas no existían antes, se creaban las interfaces «por si acaso». Estas mismas herramientas modernas de refactorización sirven para eliminar las interfaces redundantes (*refactor inline*), así que si vuestro equipo sufre demasiada indirección al navegar, lo podéis solucionar con poco esfuerzo (sobre todo en lenguajes como Java y C#, pues las herramientas no son tan potentes en lenguajes interpretados).

A lo largo del libro, hemos tenido ocasión de estudiar sobre cohesión y acoplamiento, lo suficiente como para entender que una interfaz no minimiza el acoplamiento ni maximiza la cohesión por sí misma. De hecho, el exceso de interfaces con implementación única reduce la cohesión: lo que debiera ser una única pieza, está partido en dos. Cuestionarnos por qué y para qué hacemos las cosas, así como la historia, nos ayuda a tomar mejores decisiones de diseño.

Todos los atributos deben ser accesibles mediante *getters* y *setters*

Si dividimos el sistema al menos en tres capas, dos de ellas son exteriores (GUI y datos) y otra es central (lógica de negocio). Las dos capas externas se interconectan con artefactos de terceros, intercambiando información mediante tipos integrados (universales e interoperables) y mediante convenciones. Una de esas convenciones consiste en dotar a las clases de métodos para leer y escribir en las propiedades de su estado (*getters* y *setters*):

Java

```
public class Profile {
    private String name;
    private String surname;
```

```
    public String getName(){
        return name;
    }
    public void setName(String name){
        this.name = name;
    }
    public String getSurname(){
        return surname;
    }
    public void setSurname(String surname){
        this.surname = surname;
    }
}
```

En C# incluso hay que escribir menos para conseguir lo mismo:

C#

```
public class Profile
{
    public String Name {get; set;}
    public String Surname {get; set;}
}
```

Esta es una forma de utilizar las clases como *structs* (estructuras de datos básicas) para el intercambio de datos agrupados. Gracias a la convención, se pueden ligar a la interfaz gráfica (mediante *data binding*, por ejemplo) o mapear con las tablas de una bases de datos (mediante *ORM*, por ejemplo), de manera automática. La convención JavaBean, está basada en *setters* y *getters* para el intercambio de datos con la GUI.

Hasta aquí todo bien, los problemas vienen cuando aplicamos la misma convención a la capa central, la de negocio. En el momento en que una clase deja sus atributos a merced de cualquiera para que los manipule a su antojo, pierde el control de su estado. La encapsulación que propone el paradigma de la orientación a objetos, va mucho más allá de envolver los atributos privados en *getters* o *setters*, en realidad va de preservar los secretos y la soberanía del estado de una clase. Los *getters* y *setters* añaden indirección, pero no suponen una encapsulación sustanciosa, sino más bien engañosa. Un diseño orientado a objetos no necesita *setters* y minimiza la exposición de *getters*, al igual que minimiza los demás métodos públicos para cumplir con el principio de responsabilidad única. Cuantos menos *getters* haya, mejor guardados estarán los secretos de una clase. Una guía básica para conseguir un diseño orientado a objetos en nuestra capa de negocio incluye:

- No utilizar *setters*, solo constructores y métodos de factoría.

- Reducir al máximo el uso de *getters*.
- No utilizar tipos de datos integrados (envolverlos en nuestros propios tipos).
- Cumplir la ley de Demeter.
- Reducir al máximo el número de clases que solo tienen datos o solo tienen métodos, siendo predominantes las clases que combinan datos y métodos.

El abuso de los *getters* y los *setters* termina por hacer que programemos en Java (o en C#, Python, Ruby, TypScript…) como si programásemos en COBOL, es decir, programación estructurada en lugar de orientada a objetos. Un amigo mío dice en tono de broma que eso es programar en JaBOL. Mi recomendación es estudiar los libros de Sandi Metz, que aun utilizando Ruby son entendibles para cualquiera que programe en cualquier lenguaje moderno.

El código debe ser óptimo en el consumo de CPU y de memoria

Antes de la llegada de computadoras domésticas, las científicas y científicos de la computación tenían que pedir cita para correr sus programas; con suerte había una computadora compartida en la institución donde trabajaban. El tiempo de ejecución era oro, ahorrar en ciclos de CPU era cuanto menos una cuestión de compañerismo entre colegas.

Desgraciadamente, el código óptimo en cuanto a consumo de recursos es enemigo del código legible. No se puede maximizar para legibilidad y a la vez para rendimiento, sino buscar el mejor equilibrio en cada circunstancia. Si sé que existen unos requisitos cuantificables y precisos referentes al tiempo de ejecución, al número de peticiones concurrentes, al ancho de banda o al consumo de memoria, entonces busco un diseño y una infraestructura que los cumpla. No puede ser que un sitio web público, con el objetivo de atender a la ciudadanía en una emergencia global, se caiga a los pocos minutos de ponerse en producción por exceso de peticiones. No se puede culpar a los usuarios ni pedirles que se coordinen para acceder. Salvo en estos casos, donde la exigencia es bien conocida, es mejor diseñar para maximizar la comprensibilidad y para reducir la carga cognitiva de la lectura de código.

Los requisitos no funcionales (*quality attributes*) suponen un coste y un compromiso, no se deben tomar a la ligera, sino estratégicamente, ya que incluso pueden llegar a entrar en conflicto entre ellos. Una vez, trabajando en una librería con un experto en criptografía y seguridad, aprendí que la siguiente forma de validar una contraseña podría no ser tan segura como yo pensaba:

Java

```java
public boolean doesPasswordMatch(String plainPassword, String
    correctHashedPassword){
    return hashPassword(plainPassword).equals(correctHashedPassword);
}
```

La comparación de igualdad entre cadenas comparará cada caracter y terminará en el momento en que encuentre la primera diferencia. Mi compañero me explicó que si esa función se ejecutaba en local (sin latencia de red), un atacante podría medir el tiempo de ejecución de cada llamada, y así por fuerza bruta descubrir letra por letra si estaba acertando o no, en función de los tiempos de respuesta (si tarda más es que ha avanzado uno o más caracteres en la comprobación). Muy ingenioso, ¿verdad? Me propuso escribir código que recorriese siempre toda la cadena para tardar lo mismo independientemente de los aciertos, ¡aunque esto no fuese lo más eficiente! Espero que el ejemplo sirva para explicar posibles conflictos entre requisitos no funcionales (seguridad vs. rendimiento en este caso). Esto no significa que ahora todas las comprobaciones de contraseñas se implementen así, puesto que si la función se consume a través de una API por la red, la propia latencia invalidará la información de tiempos de respuesta de la función (por ser varios órdenes de magnitud mayor).

El contexto marca la diferencia en las decisiones de implementación, especialmente en cuanto a los requisitos no funcionales, no podemos alegar categóricamente que el código más rápido siempre sea mejor.

En lenguajes de bajo nivel, las optimizaciones al programar tienen un gran impacto en el rendimiento y en el consumo de recursos, nada despreciable cuando hablamos de programar en C, C++ o ensamblador para entornos industriales. Una vez trabajé con una empresa que fabricaba máquinas de diagnóstico, las cuales estaban dotadas de varios brazos robóticos terminados en agujas que se introducían en líquidos reactivos; la sincronización de los brazos exigía que la ejecución del *software* no aumentase ni en 100 milisegundos, porque sino los brazos colisionarían.

Los compiladores modernos juegan en otra liga, los de Java, C#, Kotlin, etc., van a optimizar el código mejor que cualquier humano la gran mayoría de las veces (son mucho más que un compilador, toda una máquina virtual). La optimización es un trabajo que las máquinas saben hacer muy bien. Ante código como el siguiente, el compilador sabe perfectamente cómo optimizar:

Java

```
class A {
   B b;
   public int calculate() {
      y = b.get();
      /*...*/
      z = b.get();
      sum = y + z;
      /*...*/
   }
}
class B {
    int value;
    final int get() {
        return value;
    }
}
```

El código que finalmente se ejecute se parecerá más bien a este:

Java

```
class A {
   B b;
   public int calculate() {
       y = b.value;
       /*...*/
       sum = y + y;
       /*...*/
}
```

La conclusión es que no merece la pena ofuscar el código para ganar unos cuantos ciclos de CPU, es más, incluso puede que nuestros intentos de optimización impidan a la máquina virtual aplicar las debidas optimizaciones, terminando con un código más ineficiente. Si en algún momento decido comprometer la legibilidad del código en favor del rendimiento, procuro justificar la decisión con unas rigurosas pruebas comparativas (*benchmark*), en lugar de adivinar, no sea que salga perdiendo.

Las optimizaciones importantes son las que tienen que ver con la complejidad algorítmica, por lo cual recomiendo adquirir la capacidad de distinguir si un algoritmo tiene complejidad lineal o exponencial; esto sí que marca una diferencia (a veces crítica) que las máquinas no saben todavía resolver.

Todo lo que hace el código debe estar explicado con comentarios

En el ámbito científico es típico utilizar letras del alfabeto para las fórmulas. Hace décadas era natural que los científicos de la computación recurriesen a las letras para programar. Cuantos menos caracteres tenía el programa, menos tamaño ocupaba en la memoria y en el disco. El espacio era un factor limitante cuando la memoria RAM era muy escasa (el legendario Commodore 64, de 1982, tenía 64 Kb de RAM). También había escasez de almacenamiento para los ejecutables, los disquetes flexibles de 8 pulgadas llegaron a tener una capacidad máxima de 1,2 Mb (a finales de los 70). Imagínate el cambio de contexto comparado con los discos de terabytes de hoy y las unidades de estado sólido. Las limitaciones de antaño incidían marcadamente en la forma de codificar, los programas necesitaban comentarios de todo tipo por todos lados, ya que el código se parecía más a un jeroglífico que a un libro de texto.

Hoy en día, los comentarios ya no son necesarios para explicar lo que hace el código, sino para contar aquellas cosas que el propio código no puede contar por sí mismo. Los lenguajes han ganado en capacidad expresiva y las máquinas en recursos (¡estamos a años luz!). Tenemos todas las herramientas necesarias para escribir código autoexplicativo, que se pueda entender como un manual. Cada vez que sientes la necesidad de poner un comentario sobre un bloque de líneas de código, tienes la oportunidad de pensar si se puede extraer una función cuyo nombre aporte la misma información que el comentario. La fuente de la verdad está en el código, no en los comentarios (porque no se ejecutan y siempre compilan). Es más, la fuente de la verdad tampoco está en los test, porque el primer lugar al que acudimos para comprender la solución es el código de producción, si bien es cierto que unos test cuidados son un excelente complemento.

Con el paso de los años, navegando por códigos de todo tipo y por el hecho de que los IDE colorean los comentarios con tonos apagados, mi cerebro ha desarrollado la capacidad de no verlos; son como invisibles para mí y los ignoro sin darme cuenta. Hay mucha gente como yo en este sentido. Los años me han ido enseñando que los comentarios que encuentro suelen estar obsoletos o ser imprecisos/inútiles, haciéndome perder tiempo o confundiéndome más todavía. El principal problema de comentar lo que hace el código es que la gente se toma la libertad de escribirlo tipo jeroglífico y se excusa porque va acompañado de comentarios. Es la excusa perfecta para nombrar las abstracciones de cualquier manera. En la práctica, comentar ni siquiera garantiza mayor claridad, porque los comentarios

no suelen estar bien redactados (se hacen con prisa o no se tiene habilidad redactando). La ventaja de expresarnos en código es que elimina la ambigüedad en lo que decimos. La comunicación es fundamental en nuestro trabajo y al igual que es importante aprender a escribir un email, es importante aprender a expresarse con el lenguaje de programación. Mi consejo es aprender a utilizar el lenguaje formal para comunicarse intuitivamente con las personas, en lugar de esforzarse por escribir prosa en lenguaje natural (comentarios).

Los comentarios se quedan obsoletos muy pronto; con el paso del tiempo, el código cambia, pero nadie se toma la molestia de actualizar el comentario. Llega un punto en que el código es imposible de entender, y por encima, por debajo o a un lado, hay un bloque de comentario que explica malamente otra cosa distinta (del pasado). Estas son las vivencias de las personas que desaconsejamos los comentarios para explicar el comportamiento del código. Estos problemas no se suelen experimentar en el mundo académico, porque no mantienen el *software* a lo largo del tiempo.

Dicho todo esto, quiero señalar que ningún extremo es bueno y sería extremo no escribir absolutamente ningún comentario. Desarrollar grandes proyectos con cero comentarios también es un problema. Los he echado de menos, incluso con un código comunicativo y bien testado, porque el código no puede explicar lo que es invisible. Los motivos que llevaron a tomar una decisión o a no tomarla, no están en el código.

Por tanto, la pregunta no es si estamos a favor o en contra de los comentarios, sino ¿para qué utilizar los comentarios? Aquí va una lista de posibles motivos por los que añadir comentarios para documentar:

- Por qué, para qué, por qué no, para qué no:
    - Por qué se decidió implementar concretamente la actual solución.
    - Por qué se descartó implementarlo de otra manera que quizás pareciera más obvia o natural.
    - Para qué se usa ese código, dónde encaja dentro de la foto global, qué interdependencias invisibles tiene. Si existe alguna situación en la que pudiera ser prescindible y borrarse.
    - Por qué no se deben realizar ciertos cambios, avisando de sus peligros.
- Lo que se probó y no funcionó, para que no se vuelva a invertir el tiempo en algo que se descartó.
- Lo que se estudió y se descartó enseguida. Cuáles fueron las conclusiones de la investigación o de la prueba de concepto.

- Los efectos colaterales que provocarían cambios aparentemente inocuos, por ejemplo, condiciones de carrera o interbloqueos al cambiar el orden de ejecución de varias líneas de código.
- Si existen parámetros globales de configuración u otros factores externos que podrían afectar a este código, o que haya que tener en cuenta para su correcto funcionamiento.
- Si en caso de fallo se puede ir a mirar algún tipo de *log* o cualquier otra forma de depuración. Sobre todo en test de integración.
- Si hay algún problema que impida lanzar X conjuntos de test en paralelo, o si se requiere de cierta configuración previa al lanzamiento. Cuando fallan múltiples test a la vez se agradece mucho tener una mínima documentación con los problemas comunes (*troubleshooting*).
- Si el sistema puede crecer hasta un punto en que una solución dejará de servir. Es decir, si hemos elegido una solución de corto recorrido, como, por ejemplo, la inclusión de una librería que ya no tiene soporte o una infraestructura que solo aguantará hasta N usuarios. Dicho de otro modo, la deuda técnica.
- Ayudar a entender cómo se comporta una librería o *framework* cuando hacemos un uso particular o su documentación es escasa o incomprensible.
- Sorpresas que nos ha dado un *software* de terceros, para que solo nos las llevemos una vez.
- Describir una API pública de propósito general.

No es que haya que documentar todos los elementos de esta lista siempre, dependerá del caso. Es evidente que no todos los bloques de código llevan comentarios, ni muchísimo menos, aunque todavía me encuentro con estudiantes que me dicen que les bajan la calificación si no ponen comentarios a todas las funciones.

¿Qué hay de los comentarios de cabecera para documentación de API, tipo *javadoc*? Cuando una API es buena, no necesitamos leer su documentación, se cae de maduro el uso. Sinceramente, estamos deseando conectar con APIs que no nos obliguen a leer documentación. En lenguajes compilados donde podemos hacer uso de nuestros propios tipos, podemos ahorrarnos muchos de esos comentarios de cabecera. Cuando se trata del núcleo de lógica de negocio, si usamos tipos con un nombre expresivo en lugar de abusar de los integrados, no hace falta describir cada parámetro. Personalmente, solo documento funciones de APIs de propósito general, con un uso muy horizontal, por ejemplo, cuando son librerías. Además, solo documento los métodos que son públicos, porque si tienes acceso a los privados o protegidos puedes leer un código que está escrito con afán de ser autoex-

plicativo. La misma lista de motivos para comentar que vimos antes, puede aplicarse a los comentarios para la documentación de APIs; seguramente la virtud no está ni en añadirlos a todas las funciones, ni en su total ausencia.

Con lenguajes interpretados como PHP, poner comentarios de cabecera en las funciones ayuda a que el IDE autocomplete e indexe mejor. Si el comentario aporta un contexto que el código en sí no puede aportar y de verdad no es redundante, es buena idea describir los parámetros de la función y el valor de retorno.

La verdad es que no puedo ser dogmático y decir que absolutamente todos los comentarios que describen el comportamiento del código son despreciables, porque el trabajo continuado con código legado me ha dado muchas lecciones. Me he encontrado con la necesidad de comentar código que era muy difícil de entender, escrito por personas que ya no estaban en la empresa (nadie a quien pedir ayuda). En alguna brega con código sin test no me atreví a refactorizar por miedo a romperlo y comenté lo que buenamente pude descifrar depurando con prueba y error. En el momento del hallazgo, me aportó dejar comentarios aunque fuese para seguir los días posteriores. En estos casos, conviene explicar los motivos que han llevado a tener que añadir comentarios al código.

Como conclusión, los comentarios son un buen aliado cuando se usan sabiamente acorde a los tiempos que corren y al contexto, cuando el resto de herramientas ya no alcanzan a expresar hasta donde necesitamos. Este apartado se comprende mejor si has leído el resto del libro.

Apéndice

Conclusiones

Escribir código sostenible es una labor cuyo aprendizaje no tiene fin. Todos somos aprendices, por ejemplo, cuando cambiamos de lenguaje, de paradigma, de *stack* tecnológico o simplemente de nicho para el que desarrollamos *software*. Hace falta humildad para reconocer lo que nos falta por aprender. Se necesita una mentalidad abierta para crecer, porque se evoluciona saliendo de la zona de confort y practicando deliberadamente. Se progresa con los años y se aprende con los fracasos. Es imprescindible tener ganas de hacer el mejor trabajo posible con el conocimiento y las herramientas que tenemos en cada momento. Al mismo tiempo, se necesita cultivar el desapego para innovar y para descubrir mejores técnicas. No hay una meta final, sino un camino del que disfrutar. No existe el cinturón negro de programadora o programador de código sostenible, o como dice mi amigo Néstor, igual existe, pero luego quedan muchos *dan* que subir.

Leyendo tantas recomendaciones sobre cómo escribir el código a lo largo de este libro, quizá hayas sentido que te abruman y que tienes un largo camino por recorrer. Yo también siento que tengo un largo camino por recorrer y por descubrir, sobre todo, porque esta profesión todavía está en pañales. No dejamos de aprender en todo momento, porque la única constante es el cambio. Cambia la técnica, la tecnología, los problemas, el contexto, los equipos, las empresas, las escuelas, las ideas…

Cuando dudo entre principios y heurísticas o no consigo encontrar un diseño que me contente, me agarro a las reglas básicas:

- Cubrir bien el código con test.
- Escribir los test con código sostenible.
- Definir las abstracciones con sentido.
- Dejar clara mi intención en el código que escribo.

Cualquier persona puede empezar a escribir código más sostenible aplicando estas reglas, independientemente de su experiencia. Cuida los nombres y cuida los test para construir una base sólida. Casi todo lo demás se puede mejorar con el tiempo mediante *refactoring*,

estudio, investigación y experimentación. Saber diseñar una arquitectura que soporte los requisitos no funcionales también es muy importante, aunque eso nos daría para varios libros.

Procuro hacer *pair programming* con frecuencia para exponer mis puntos de vista (haciendo el ejercicio de verbalizarlos), recibir *feedback* rápido sobre ellos, y como no, para escuchar los otros puntos de vista. A menudo se nos ocurren mejores ideas cuando construimos sobre las de otras personas. La sinergia de varias personas trabajando juntas delante de un mismo código es mucho mayor que la suma de las aportaciones individuales. Si no es posible trabajar en pares lo suficiente, procuro que tengamos reuniones para revisar código juntos.

Recurro a mis principios y valores a la vez que respeto los del resto del equipo para tomar decisiones. Por eso, priorizo lo que tenga mayor impacto para el negocio en el corto, medio y largo plazo, así como lo que facilite más el trabajo de las futuras mentes que se encargarán de mantener el *software*. No quiero *pan para hoy y hambre para mañana*, sino entrega de valor sostenible.

Hay un antes y un después de escribir código sostenible y estoy seguro de que te va a encantar. Estoy convencido de que vas a poner aún más énfasis en escribir tu código para que otras personas lo entiendan sin romperse la cabeza. También vas a escribir buenos test para que lo puedan cambiar sin miedo. Te felicito por ello.

Fuentes de inspiración y conocimiento

Durante los trece años anteriores al momento en que estoy escribiendo estas líneas, mi trabajo ha consistido en ayudar a profesionales y organizaciones a cumplir con sus objetivos. He tenido la gran suerte de programar con multitud de personas en equipos de toda España y de varios países de Europa. Lo que he podido aprender de todas ellas ha acelerado mi crecimiento de manera inusual. Conocerles y trabajar con ellas me ha permitido descubrir técnicas, conceptos y recursos que difícilmente hubiera encontrado por mí mismo.

Ya desde antes de adentrarme en el mundo laboral, tuve la gran suerte de conocer comunidades de práctica que abrieron para mí nuevas dimensiones en la forma de entender la técnica y las demás habilidades de la profesión. El intercambio de experiencias y conocimientos en eventos como las conferencias o congresos, ha supuesto para todos los que participamos en ellos un hito sobresaliente.

Buena parte de lo que está escrito en este libro, ha sido aprendido en estos intercambios en las empresas y en las comunidades de práctica. Me gustaría mencionar especialmente a las comunidades *Agile Spain* y *Software Crafters*, porque tengo verdaderos amigos allí, que además de enseñarme mucho, me han ayudado profesionalmente siempre que lo he necesitado. No creo que haya ningún consejo o técnica original mía en este texto, no se me cayó ninguna manzana en la cabeza. Con el paso de los años, he olvidado de quién aprendí cada técnica particular, simplemente las he ido interiorizando y haciendo mías día tras día. Esto hace que no pueda citar a las personas de las que aprendí tantos de estos principios, patrones, etc. Me hubiera gustado ser más consistente con mi blog para tener ahora un registro de ello.

Los libros son fuentes de conocimiento que, entre otras cualidades, me ayudan a recordar a sus autoras y autores. Por eso, a ellos sí puedo citarles aquí. Este texto se apoya claramente en las experiencias de profesionales con magníficos libros, como, por ejemplo, Sandi Metz, Kent Beck, Katrina Owen, Martin Fowler, Emily Bache, Robert C. Martin, Barbara Liskov, Edsger Dijkstra, Ward Cunningham, Douglas Crockford, J.B Rainsberger, Corey Haines, Steve McConnell, Bertrand Meyer o Sandro Mancuso, entro otros muchos. Si este libro te anima a leer también a estos otros autores, estaré muy satisfecho.

¿Cuál es el siguiente paso?

Dependiendo de tu bagaje y de tus objetivos de futuro, ahora tienes múltiples caminos por los que continuar aprendiendo. Mi recomendación es priorizar los temas que perduran en el tiempo, mejor que estudiar tecnologías concretas y efímeras. Conocer los fundamentos hace más fácil el aprendizaje de nuevas tecnologías, a la vez que facilita la innovación. Si no has estudiado ciencias de la computación, porque vienes de un *bootcamp* o de un *ciclo de formación profesional*, pienso que te resultará útil e interesante aprender un poco sobre algoritmia y sobre teoría de lenguajes de programación. No tiene que ser lo siguiente que hagas, puede ser algo que hagas de vez en cuando, de manera intercalada, porque es denso y a veces duro. Aquí tienes algunos libros sobre temas clásicos de ciencias de la computación:

- A. Drozdek. *Data Structures and Algorithms*. 2ª ed. Boston: Course Technology Inc, 2004
- A. V. Aho, M. S. Lam, R. Sethi y J. D. Ullman. *Compilers. Principles, Techniques, and Tools*. 2ª ed. Massachusetts: Addison-Wesley, 2006

- D. E. Knuth. *Art of computer programming*. Vol. 1-4A. Massachusetts: Addison-Wesley, 2011
- D. Jungnickel. *Graphs, Networks and Algorithms*. 4ª ed. Suiza: Springer, 2012
- M. D. Kelley. *Teoría de autómatas y lenguajes formales*. Madrid: Pearson Educación, 1995
- R. J. Trudeau. *Introduction to Graph Theory*. Nueva York: Parker Publishing Company, 2017
- T. H. Cormen, C. E. Leiserson, R. L. Rivest y C. Stein. *Introduction to Algorithms*. 3ª ed. Massachusetts: The MIT Press, 2009

Si ya tuviste bastante en la universidad, lo que quizá te interese más es ver otros paradigmas aplicados a lenguajes modernos. Sería muy interesante que aprendieses cómo funcionan lenguajes como *F#, Scala, Elixir y Clojure*. Si quieres luego ir un paso más allá, puedes estudiar lenguajes como *F**, *Idris* o *Haskell*.

- C. Allen y J. Moronuki. *Haskell Programming from first principles*. 2ª ed. San Francisco: Gumroad, 2017

En cuanto a técnica, metodología y actitud, te recomiendo los siguientes libros:

- C. Buenosvinos, C. Soronellas y K. Akbary. *Domain-Driven Design in PHP*. Reino Unido: Packt Publishing, 2017
- J. Bloch. *Effective Java*. 3ª ed. Reino Unido: Addison-Wesley Professional, 2017
- J. Humble y D. Farley. *Continous Delivery*. Reino Unido: Addison-Wesley Professional, 2010
- J. Kerievsky. *Refactoring to Patterns*. Reino Unido: Addison-Wesley Signature Series (Fowler), 2004
- K. Beck. *Extreme Programming Explained*. 2ª ed. Reino Unido: Addison-Wesley, 2004
- L. Koskela. *Effective Unit Testing*. Nueva York: Manning Publications, 2013
- M. A. Gómez. *Clean JavaScript. Aprende a aplicar código limpio, SOLID y Testing*. Pamplona: Software Crafters, 2020
- S. Freeman y N. Pryce. *Growing Object-Oriented Software Guided by Tests*. Reino Unido: Addison-Wesley Signature Series (Fowler), 2009
- S. Mancuso. *The Software Craftsman*. Nueva Jersey: Prentice Hall, 2014
- S. Metz, K. Owen y TJ Stankus (2016). *99 Bottles of OOP* (2ª ed) [Online]

La arquitectura de *software* y los diferentes paradigmas de programación son otros pilares

importantes para cualquiera que desarrolla *software*. Aquí tienes algunas recomendaciones al respecto:

- M. Fowler. *Patterns of Enterprise Application Architecture*. Reino Unido: Addison Wesley Signature Series (Fowler), 2002
- N. Ford, R. Parsons y P. Kua. *Building Evolutionary Architectures*. Delaware: O'Reilly Media, 2017
- N. Rozanski y E. Woods. *Software Systems Architecture*. 2ª ed. Reino Unido: Addison-Wesley Professional, 2011
- P. Chiusano y R. Bjarnason. *Functional Programming in Scala*. Nueva York: Manning Publications, 2014
- R. C. Martin. *Clean Architecture*. Madrid: Pearson, 2017
- R. Ford. *Fundamentals of Software Architecture*. California: O'Reilly Media, 2020
- S. Brown. *Software Architecture for Developers*. Atlanta: Leanpub, 2018
- T. Nurkiewicz y B. Christensen. *Reactive Programming with RxJava*. California: O'Reilly Media, 2016

Las técnicas de análisis de requisitos son también un punto muy importante a estudiar. Algunos libros claves son los siguientes:

- G. Adzic. *Specification by Example*. Nueva York: Manning Publications, 2011
- G. Nagy y S. Rose. *The BDD books - Discovery*. Atlanta: Leanpub, 2018
- J. Patton. *User Story Mapping*. Delaware: O'Reilly Media, 2014
- M. Cohn. *User Stories Applied*. Reino Unido: Addison Wesley, 2004
- M. Wynne, A. Hellesoy. *The Cucumber Book*. 2ª ed. Carolina del Norte: Pragmatic Bookshelf, 2012

Por otra parte, te vendrá muy bien conocer metodologías de gestión de proyectos tales como Scrum o Kanban, además de nociones generales de gestión (*management*), de liderazgo y de *coaching*. Aquí el abanico de libros y cursos que puedes hacer es gigante. Los libros de Roberto Canales te resultarán muy prácticos.

Las habilidades no técnicas, las llamadas *soft skills* o habilidades esenciales, no pueden faltar en la caja de herramientas de una persona que programa, aunque parezca mentira. Aprender a comunicarse con eficacia, así como a desarrollar la inteligencia emocional, es tan importante o más que saber programar. Los proyectos fracasan mayormente por los

problemas de comunicación entre personas. Un libro que siempre recomiendo es *Comunicación no violenta*, de Marshall B. Rosenberg. Uno de mis autores favoritos en castellano es Daniel Gabarró, con libros como *Liderazgo Consciente*. Los cursos y libros sobre comunicación eficaz a la hora de hacer exposiciones, también son muy recomendables.

La lectura de este libro no implica la total asimilación de su contenido, igual te viene bien volver a leer algunos capítulos o consultarlo de vez en cuando. No hay nadie mejor que tú para decidir cuál es el siguiente paso en tu carrera profesional, solo quería darte algunas ideas. Te deseo todo lo mejor.

Agradecimientos

Es injusto que solamente aparezca mi nombre y el de Javier en portada, porque este libro ha sido un verdadero trabajo de equipo, ¡de equipazo! Gracias a las inestimables y generosas aportaciones de grandes profesionales, puedo sentirme orgulloso del resultado final y muy afortunado de haber contado con un asesoramiento de auténtico lujo.

Estoy muy agradecido a mi amigo Miguel A. Gómez (autor del libro *Clean JavaScript*), porque ha revisado a conciencia las mil y una versiones del libro en profundidad, detectando fallos y aportando valiosas sugerencias que han marcado profundamente el resultado final. Además, me ha animado a dar lo mejor de mí mismo durante el año y medio que he estado escribiendo y editando. Gracias por esos cafés de terraza discutiendo sobre el libro.

Los meticulosos análisis de mi amiga Raquel M. Carmena han tenido como resultado que la organización de los capítulos sea coherente y que no haya errores conceptuales de bulto que supo detectar, además de los numerosos defectos que encontró en los ejemplos de código. La definición de código sostenible que aparece al principio del libro, la escribimos entre los dos ¡Muchísimas gracias Raquel!

Este libro no hubiera salido a la luz sin la motivación y el apoyo que me brinda el fabuloso equipo de *Lean Mind*, la empresa en la que trabajo. Quiero dar las gracias explícitamente a Kevin Hierro, Manuel Pérez, María Soria, Juan Antonio Quintana, Francisco Mesa, Mireia Scholz, Cristian Suárez, Airán Sánchez, Sara Revilla, Iván Santos, Noe Delgado, Manuel Moranchel, Jose Luis Rodríguez, Nazaret Miranda, Jorge Aguiar y Eric Driussi, por todas las sugerencias y comentarios que me han ido haciendo mientras escribía el libro.

Lean Mind no existiría sin sus maravillosos clientes, por eso me gustaría que sepan que el impacto de habernos elegido es muy grande para muchas personas y que una de sus consecuencias es la creación de este libro. Sin su confianza no hubiera podido permitirme el lujo de cumplir con este compromiso que tenía conmigo mismo desde hace años.

Me siento tremendamente afortunado de haber contado con el apoyo de todos los amigos que han leído borradores en fases beta y que han propuesto mejoras brillantes y necesarias, desde *bugfixes* hasta mejoras idiomáticas y matices de la redacción. Muchísimas gracias a Raúl Ávila, Alex Fernández, Eladio López, Néstor Bethencourt, Juan Manuel Serrano, Luis Rodríguez, Concha Asensio, Luis Ruiz Pavón, Darío Beiró, Jessica Aguado, Jorge Jardines,

Manuel Hermán y Juan M. Gómez.

El toque de calidad en la redacción se lo debo a mi editora, Liset Gómez, que ha sabido entender perfectamente lo que quería contar, me ha ayudado a mejorar la redacción y se ha asegurado de que no haya incorrecciones.

Es todo un honor para mí que un profesional como Javier Ferrer haya escrito el prólogo de este libro, estoy muy agradecido y emocionado por su regalo. Soy un admirador de *Co-dely.tv*.

El trabajo que ha hecho Vanesa Gónzalez con su diseño de portada me parece toda una obra de arte, estoy muy agradecido por su trabajo. También estoy muy agradecido a Inés Arroyo por su trabajo en la dirección de arte.

Y por supuesto estaré eternamente agradecido a mis seres queridos, porque son el motor de mi vida; sin ellos, este libro nunca hubiera existido. Ellos y ellas saben quiénes son, aunque algunos no sepan leer.

Acerca del autor

Las fascinación de Carlos Blé por la programación empezó cuando era niño, tenía unos 9 o 10 años cuando su padre compró unos libros de programación en BASIC y se puso a aprender por su cuenta, practicando con el primer PC que hubo en casa (Intel 8086). Desde 1999 se ha mantenido programando a diario, si bien es cierto que en la actualidad dedica más tiempo a dirigir su empresa y a divulgar. Desde 2009 ha colaborado con decenas de equipos en todo el país y en el extranjero, ayudándoles a conseguir sus objetivos y a mejorar la técnica. Publicó el primer libro sobre TDD en castellano en 2010. No contento con el libro con el paso del tiempo, escribió otro totalmente nuevo con el mismo título, en 2019. Tras terminar *Código sostenible* está ya pensando en escribir otros libros sobre técnicas como *refactoring* o *pair programming*.

Carlos es ingeniero técnico en informática de sistemas por la Universidad de La Laguna, miembro de diversas comunidades de práctica como *Agile Spain* o *Software Crafters*, conferenciante, docente, consultor, *podcaster*, *blogger* y empresario. En la actualidad no participa en redes sociales, con lo que el mejor sitio para conectar es su web (carlosble.com), así como los congresos a los que asiste.

Bibliografía

La literatura en la que me he apoyado para escribir este libro es la siguiente:

- A. Davies. *Async en C # 5.0*. Delaware: O'Reilly Media, 2012
- B. Liskov y J. Guttag. *Program Development in Java*. Reino Unido: Addison-Wesley Professional, 2000
- B. Meyer. *Object-Oriented Software Construction*. Reino Unido: Pearson College Div, 2000
- C. Blé, *Diseño Ágil con TDD*. Atlanta: Leanpub, 2019
- C. Haines. *Understanding the Four Rules of Simple Design*. Atlanta: Leanpub, 2014
- C. Petzold. *Código: El lenguaje oculto del hardware y software informático*. Estados Unidos: Microsoft Press, 1999
- D. Crockford. *JavaScript, The Good Parts*. Sunnyvale: Yahoo Press, 2008
- D. Gabarró. *Liderazgo consciente*. Lleida: Boira editorial, 2015
- D. Thomas y A. Hunt. *The Pragmatic Programmer*. Reino Unido: Addison-Wesley, 2019
- E. Brady. *Type-Driven Development with Idris*. Nueva York: Manning Publications, 2017
- F. Saussure. *Curso de lingüística general*. Buenos Aires: Editorial Losada, 1945
- K. Beck. *Implementation Patterns*. Reino Unido: Addison-Wesley Professional, 2007
- K. Lieberherr, I. Holland y A. Riel, «Object-Oriented Programming: An Objective Sense of Style», OOPSLA '88: Actas de congresos sobre sistemas, lenguajes y aplicaciones de programación orientada a objetos, 1988. [En línea]
- M. B. Rosenberg. *Comunicación no violenta*. Argentina: Gran Aldea Editores, 2011
- M. Fowler. *Refactoring*. 2ª ed. Reino Unido: Addison-Wesley Signature Series (Fowler), 2018
- M. Page-Jones, «Comparación de técnicas mediante encapsulación y connascencia», ACM. Digital Library, 1992. [En línea]
- R. Braithwaite. *JavaScript Allongé*. 6ª ed. Atlanta: Leanpub, 2013
- R. C. Martin. *Clean Code*. Londres: Pearson, 2018
- S. McConnell. *Code Complete*. 2ª ed. Estados Unidos: Microsoft Press, 2004
- S. Metz. *Practical Object-Oriented Design*. 2ª ed. Reino Unido: Addison-Wesley Professional, 2018
- S. Wlaschin. *Domain Modeling Made Functional*. Carolina del Norte: Pragmatic Bookshelf, 2018

Las autoras y autores de estos libros se han apoyado en el trabajo de las generaciones anteriores, desde Ada Lovelace hasta Erich Gamma, pasando por Edsger Dijkstra, Margaret Hamilton, Alan Kay o Ole-Johan Dahl.

En cuanto a los vídeos de apoyo utilizados, pueden observarse los siguientes:

- Confreaks, «Acts as Conference 2009 The Grand Unified Theory by Jim Weirich», 22-feb-2014. [Vídeo]
- InfoQ, «Douglas Crockford: Programming Style & Your Brain», 24-ago-2012. [Vídeo]
- InfoQ, «The Powe Of Abstraction», 3-may-2013. [Vídeo]
- Jbrains762, «Understanding Coupling and Cohesion», 2-jul-2013. [Vídeo]

Por último, pero no menos importante, podemos encontrar diversas entradas a numerosos blogs. Se trata de una fuente de información esencial, ya que permite acceder al contenido que se desee de una forma selectiva e inmediata. En este libro se han consultado los siguientes blogs:

- C. Blé, «Comprensión de los errores de Javascript», Carlos Blé. Artesano del software, 2014. [En línea]
- J. Ferrer, «Introducción Arquitectura Hexagonal -DDD», Codely.tv, 2016. [En línea]
- J. Spolsky, «La ley de las abstracciones con fuga», Joel en el software, 2002. [En línea]
- M. Fowler, «AnemicDomainModel», martinFowler.com, 2003. [En línea]
- M. Fowler, «CommandQuerySeparation», martinFowler.com, 2005. [En línea]
- M. Fowler, «DosCosasDuras», martinFowler.com, 2009. [En línea]
- M. Fowler, «Reemplazo de excepciones de lanzamiento con notificación en validaciones», martinFowler.com, 2014. [En línea]
- M. Fowler, «TellDontAsk», martinFowler.com, 2013. [En línea]
- R. Ávila, «Revisa tu código autogenerado», Raúl Ávila, 2016. [En línea]
- R. Carmena, «Cobertura del código del 99 %: ¿tenemos una buena red de seguridad para cambiar este código heredado», Rachel M. Carmena, 2017. [En línea]
- R. Carmena, «Refactorización», Rachel M. Carmena, 2019. [En línea]

Savvily

¡Enhorabuena por leer hasta aquí! Agradecemos el tiempo que te has tomado estudiando este libro y que lo recomiendes a cualquier persona que pienses que le puede ser de ayuda.

Si has comprado el libro en la tienda Savvily (*savvily.es*) tienes acceso a foros, actualizaciones gratuitas y otros recursos exclusivos para lectores y autores. Si lo compraste en otra tienda puedes registrar tu copia enviándonos un email, adjuntando un justificante de compra, para que podamos darte de alta en la comunidad y que disfrutes de todos los beneficios. En cualquier caso, en la web encontrarás un pdf de descarga gratuita que contiene enlaces a los recursos citados en el apéndice y la bibliografía.

Estaría genial que envíes una valoración del libro por email, una reseña de lo que te ha parecido. Si has encontrado erratas, por favor envíalas también por email y con gusto las arreglaremos en futuras revisiones y ediciones del libro.

Para publicar libros como este, se necesita todo un equipo multidisciplinar, desde las autoras y autores, al equipo de revisión, pasando por diseñadoras y especialistas en marketing digital. Por todo ello, te pedimos que si el libro te ha gustado y no lo habías comprado, te animes a comprarlo. Es el mejor reconocimiento a la labor de todas las personas que lo han hecho posible.

Si eres docente en una institución académica pública, y quieres utilizar este libro en tus clases, Savvily ofrece acuerdos para que docentes y estudiantes puedan tener acceso a los libros de forma gratuita. Si trabajas para algún centro de formación privado, podemos ofrecerte descuentos. Por favor escríbenos, cuéntanos tu caso, preséntanos a tu organización y te enviaremos más información.

¿Te has planteado alguna vez escribir tu propio libro? Seguro que tienes mucho que contar, ¿te animas? Hacen falta buenos libros en castellano. Visita nuestra web y descubre cómo puedes convertirte en autora o autor, estaremos encantados de acompañarte.

Muchas gracias de parte de todo el equipo de Savvily.

Contacto: *contacto@savvily.es*

ypramos1986@hotmail.com