Código Sostenible

Cómo programar código fácil de mantener

Carlos Blé Jurado

Prólogo de Javier Ferrer

savvily

Este pdf es una muestra de prueba gratuita que contiene el índice, el prefacio y el primer capítulo, solamente.

El libro completo se encuentra en

https://savvily.es/libros/codigo-sostenible/

Código Sostenible

Código Sostenible

Cómo programar código fácil de mantener

Carlos Blé Jurado

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.

Índice general

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.