Universidad Nacional de Córdoba

Facultad de Matemática, Astronomía, Física y Computación

Lineamientos para escribir Código Bonito

por

Francisco Gabriel Zavalla Bresciani

Presentado ante la FACULTAD DE MATEMÁTICA, ASTRONOMÍA, FÍSICA Y COMPUTACIÓN como parte de los requerimientos para la obtención del grado de Licenciado en Ciencias de la Computación de la

UNIVERSIDAD NACIONAL DE CÓRDOBA

Agosto, 2025

Director: Matías David Lee

Agradecimientos

En primer lugar, me gustaría agradecer a mi director, el Dr. Matías D. Lee, por acercarse a mí con esta idea y brindar su acompañamiento, paciencia y confianza a lo largo de estos últimos meses.

Agradezco a mis padres, Alejandra y Hugo, por su constante apoyo e interés en todo lo que hago, sin ellos hoy no estaría donde estoy.

Agradezco al resto de mi familia, principalmente por la comprensión cuando no estaba disponible por estar estudiando.

Agradezco a mis amigos, por estar siempre cerca para escucharme y darme ánimos en la carrera.

Agradezco a los compañeros que tuve la suerte de conocer durante todo el trayecto universitario, por los momentos buenos y los no tan buenos durante la carrera. Y no puedo dejar de agradecerles por las horas de estudio compartidas.

Finalmente, agradezco a la Universidad Nacional de Córdoba y a la Facultad de Matemática, Física, Astronomía y Computación, junto a todos los profesores y ayudantes, por el acompañamiento y el conocimiento ofrecido durante estos años.

I. Abstract

Desde las aulas hasta el ámbito profesional, tanto estudiantes como desarrolladores suelen escribir código que simplemente satisface los requisitos funcionales, dejando de lado aspectos fundamentales como la claridad, la legibilidad y la prolijidad. Esto trae como consecuencia pérdida de tiempo para comprender o refactorizar el código, dificultades en el mantenimiento y una menor confiabilidad del *software*. Este trabajo busca abordar esta problemática mediante la definición de una serie de buenas prácticas, denominadas *lineamientos*. Estos lineamientos, lejos de pretender ser una verdad absoluta, buscan servir a modo de base para escribir código más claro, organizado y sostenible. Además, se espera que estos lineamientos sean de especial utilidad para quienes están comenzando en la programación, y por otro lado, que llegue a la mayor cantidad de desarrolladores posible, para fomentar hábitos que promuevan la escritura de código prolijo y fácil de entender.

From classrooms to professional environments, both students and developers often write code that simply meets functional requirements, leaving aside fundamental aspects such as clarity, legibility and neatness. As a result, more time is wasted trying to understand or refactor code, maintenance becomes harder, and the software is less reliable. This work aims to tackle this problem by defining a set of best practices, called *guidelines*. These guidelines don't claim to be the absolute truth, instead, they're meant to be a starting point for writing clearer, more organized, and more maintainable code. We hope they'll be especially helpful for people who are just starting out in programming, and that this information can reach as many developers as possible to encourage habits that lead to clean, easy-to-understand code.

Índice

II. Introducción

Cuando se estudia una carrera relacionada con la programación, se abordan diversas áreas que conforman la disciplina. Por ejemplo, en la *Licenciatura en Ciencias de la Computación* de la *FAMAF*, se estudian temas como algoritmos, lógica, matemáticas, bases de datos, sistemas operativos, ingeniería del *software*, paradigmas de programación, compiladores, entre otros más específicos. En la mayoría de estas materias, una actividad común es programar. **Programar** consiste en *escribir secuencias de órdenes que una computadora puede ejecutar para realizar una tarea específica*. Al programar obtenemos **programas** que podemos analizar desde múltiples dimensiones: ¿qué hace? ¿cómo lo hace? ¿es claro? ¿se puede probar su funcionamiento? ¿posee una buena modularización? ¿hay acoplamiento entre los módulos? ¿utilizan algún patrón de diseño? ¿es seguro?

Todos estos aspectos son fundamentales para desarrollar un *software* de alta calidad. Sin embargo, durante la formación académica suelen tratarse como elementos complementarios en lugar de objetivos centrales. Como consecuencia, no existe un momento donde los estudiantes puedan aprender estos principios, ni mucho menos una fuente de referencia concreta para escribir código de calidad.

1. Código Bonito

En matemática se habla de *demostraciones elegantes*. Estas demostraciones no solo son correctas, sino también están bien estructuradas y utilizan los elementos adecuados para simplificar la tarea de demostración. Si bien el concepto de *elegancia* no está definido formalmente (algo raro si tenemos en cuenta que en matemática todo parte de una definición precisa), los matemáticos saben reconocer cuando se encuentran con una demostración elegante.

En programación queremos definir un término análogo: *código bonito*. Así como en matemática no existe una definición formal de elegancia, nosotros tampoco daremos una definición formal de *bonito*. Sólo diremos que un código es bonito si es claro, prolijo y está bien estructurado. En otras palabras, el código bonito está en las antípodas del *código espagueti* [1]. Como pasa con la elegancia en la matemática, todo desarrollador con experiencia y conocimiento sabrá identificarlo y apreciarlo.

Lamentablemente, el código bonito no abunda. La experiencia acumulada por profesionales de la industria y docentes evidencian que este tipo de código, en general, brilla por su ausencia. Este trabajo tiene como objetivo recopilar prácticas y recomendaciones ya conocidas, así como aportar nuevas ideas que ayuden a los desarrolladores a escribir mejor código.

[1] `https://es.wikipedia.org/wiki/Código_espagueti`

2. ¿Por qué enseñar/aprender a programar es difícil?

Si tuviéramos que hacer una analogía entre la profesión de desarrollador con otra, probablemente la mayoría de los desarrolladores con experiencia estarían de acuerdo que programar se parece más a ser un albañil que un abogado. Un albañil construye desde cero o trabaja sobre obras ya empezadas. En el segundo caso, tirar todo lo construido no es una opción, hay que adaptarse a lo que se hizo. En el mundo de la programación, muchas tareas son repetitivas, como levantar paredes en la construcción, pero sin estas cosas repetitivas, no habría una obra completa. Sumado a esto, toda obra tiene sus particularidades, que en muchos casos impactan en todo el proyecto, aún en las tareas más estándares.

Esta analogía entre desarrollador-albañil da la clave para entender porque enseñar/aprender a programar es difícil: programar es un **oficio**. La particularidad de los oficios es que se aprenden a través de la experiencia directa, uno puede leer libros y ver videos sobre como levantar una pared/programar, pero no se aprenderá realmente la tarea hasta dedicarle muchas horas a la misma. Es más, haber levantado miles de paredes no te hará necesariamente un buen albañil y, de forma similar, haber programado miles de líneas de código no te hará un buen desarrollador. Esto se debe a que todo oficio tiene sus buenas prácticas, lineamientos que se deben seguir y el hacer por hacer no te garantiza aprenderlos.

En los oficios con más historia (albañil, zapatero, carpintero, etc), este problema se resuelve con la figura del **maestro**. Los maestros son las personas con más experiencia en el oficio y tienen como responsabilidad el traspasar sus conocimientos a los aprendices. Este traspaso de conocimiento sucede en la práctica: mientras el aprendiz realiza alguna tarea, el maestro observa, y en base a lo que observa, brinda consejos, realiza correcciones y agrega explicaciones siempre que la situación lo requiera.

Volviendo al oficio de programar, podemos decir que en el mundo de la programación, principalmente en la parte académica, no existe la figura de **maestro de la programación**. Enumeremos algunas de las razones para pensar esto:

- **Muchos docentes no ejercen el oficio de programar**. Muchos docentes son académicos, entonces en su día a día no programan. Si no programan es difícil que sean **maestros de la programación**. Es más, probablemente tampoco sea para ellos una prioridad el enseñar a programar código bonito, tienen otras cosas importantes que enseñar y eso no está mal.

- **Corregir el código es costoso**. Supongamos ahora que tenemos un grupo de docentes que sí saben programar. Aún así, revisar el código de los alumnos uno por uno sería imposible por el tiempo que eso llevaría. La tarea sería más imposible si le sumamos correcciones escritas y/o devoluciones uno a uno. Para complicar más la situación, sumémosle a esto el hecho de que las carreras informáticas cada vez se vuelven más populares -más alumnos- mientras que el número de docentes disminuye por encontrar ofertas laborales más atractivas.

- **Los tiempos de los proyectos de programación son cortos**. Los proyectos universitarios buscan enseñar conceptos claves de la informática (*deadlocks*, multiprocesamiento, simulaciones, protocolos TCP/UDP, programación de microcontroladores, etc.) en pocos meses. Estos conceptos pueden enseñarse perfectamente realizando proyectos de programación que no siguen buenas prácticas. Forzar a los alumnos a seguirlas durante el proyecto -teniendo en cuenta el tiempo limitado con el que se cuenta- podría atentar con el objetivo principal del mismo.

Entonces, **¿se podría introducir la figura de *maestros de la programación* en las instituciones académicas?** Entendemos que sí, pero esto podría requerir mucho más recursos humanos y financieros que no abundan en el mercado laboral informático/universitario del mundo de hoy. De todas formas, esta discusión está fuera del objetivo de este trabajo.

3. Objetivo y organización de este trabajo

Programar bien es complejo. Hacerlo dentro de una industria lo es aún más, pues esto implica conocer la lógica de los procesos que ahí se desarrollan. Sumado a esto, todo *software* que se vende/utiliza cómo producto debe satisfacer muchos aspectos técnicos para que el mismo sea viable. Por ejemplo, aspectos relacionados al desempeño, seguridad, manejo correcto de los datos, etc.

El objetivo de este trabajo no es abarcar todos estos problemas. Por el contrario, buscamos definir una **línea base** para programar bien. Para eso vamos a introducir una serie de lineamientos que entendemos se aplican casi siempre en todo contexto. También queremos que este trabajo tenga impacto, es decir, que muchas personas lo lean. Por esta razón, trataremos de ser lo más concretos posibles en cada sección que presentemos. **Creemos que este trabajo será sumamente valioso para las personas que están dando sus primeros pasos en la informática y puede servir como material de referencia para materias donde se haga mucho foco en programar**.

A lo largo del trabajo, vamos a presentar diversas secciones:

- Sintaxis y semántica
- Diseño de funciones
- Documentación y comentarios
- Organización de un proyecto de *software*
- Testing

En el capítulo de **sintaxis y semántica** pondremos énfasis en que al momento de escribir código, se tiene que ser lo más evidente posible con respecto al objetivo del sistema que uno está escribiendo. Para esto es fundamental la elección de buenos nombres y el uso de tipos. En **diseño de funciones** daremos lineamientos para escribir funciones prolijas, ya que estas son más fáciles

de entender, utilizar y modificar. El capítulo sobre **documentación y comentarios** será un capítulo muy corto sobre la importancia de documentar el código y lineamientos para hacerlo de la forma correcta.

En **organización de un proyecto de *software***, nos alejaremos un poco del código para hablar de la estructura del mismo. Introduciremos el concepto de *capas* que podemos encontrar en los proyectos y cómo estas nos ayudan a organizar el código. Hablaremos también de cómo las clases y la inyección de dependencias nos ayudan a organizarnos. Además utilizaremos un proyecto real como hilo conductor del capítulo.

Para terminar, nos centraremos en las pruebas que se pueden realizar sobre el código, también conocido como **testing**. Nuevamente, nos apoyaremos en el proyecto de ejemplo presentado en el capítulo previo para hablar sobre *la pirámide del testing* y los diferentes tipos de pruebas que podemos realizar.

Para ejemplificar los lineamientos presentados, a lo largo del trabajo se utilizarán fragmentos de código en **Python** y **JavaScript/TypeScript**. Elegimos estos lenguajes principalmente por ser ampliamente utilizados en la industria y porque consideramos que permiten comprender los ejemplos sin requerir un nivel de conocimiento demasiado avanzado, facilitando así que el contenido sea accesible para un público más amplio.

Además, para complementar este trabajo y facilitar su acceso a más personas, se desarrolló en paralelo una página web donde se recopilan todos los contenidos presentados[2]. Esta página tiene como objetivo ofrecer una vía práctica y sencilla para que cualquier interesado pueda consultar los lineamientos, ejemplos y recomendaciones de forma libre y actualizada. Asimismo, utilizaremos esta plataforma para recopilar métricas y realizar encuestas a los usuarios, lo que nos permitirá generar una conclusión sobre la utilidad y el impacto real de este trabajo.

[2] `https://www.writingprettycode.com/`

III. Sintaxis/Semántica

1. De la sintaxis y semántica a la intención

Todos los lenguajes de programación comparten dos componentes esenciales, la sintaxis y la semántica. La **sintaxis** *es el conjunto de reglas que definen cómo organizar los símbolos y palabras claves de un lenguaje para formar sentencias y expresiones válidas* [12]. Por otro lado, la **semántica** *es cómo se deben interpretar esas expresiones*. Esto se puede formalizar de distintas maneras, una de ellas es la **semántica operacional**, que describe el comportamiento de un programa en términos de cómo se ejecutan paso a paso sus instrucciones. Esta semántica se divide en dos ramas, la semántica *small step* y la semántica *big step*.

La semántica *small step* [11] describe la ejecución de los programas dividiéndolos en pasos pequeños, es decir, evaluando cada instrucción de forma secuencial. Para ello, define una relación binaria que conecta cada estado del programa antes y después de realizar una instrucción. Esta semántica es útil para conocer cuál es el estado del programa en un momento dado.

Por otro lado, la semántica *big step* [11], describe los resultados finales de la computación, sin preocuparse por los estados intermedios. El objetivo de esta semántica es llegar directamente al resultado final sin detenerse en cada paso.

Existen otras semánticas, como la axiomática que describe el significado de los programas mediante pre y post-condiciones, o la semántica denotacional que describe el comportamiento de los programas haciendo uso de objetos matemáticos.

Estas semánticas resultan útiles porque proporcionan diferentes herramientas para analizar y comprender algunos aspectos de los lenguajes de programación, pero no se suelen utilizar directamente al momento de programar. Por esta razón, vamos a introducir una nueva noción de semántica: la **semántica en lenguaje natural**, que *describe el programa según lo que el desarrollador pretende que el código haga*. Esta semántica es imposible de definir formalmente porque la misma es subjetiva al desarrollador que escribió el código, pero es importante darle entidad a su existencia. Pues la misma puede ser más o menos evidente según la calidad del código que se escribe.

Un código que aplica correctamente la semántica en lenguaje natural es un código bonito, y un código bonito sigue buenas prácticas de programación. Entre esas prácticas que vamos a ver en este trabajo, está la elección de buenos nombres de funciones: un buen nombre hace explícito lo que hace la función. Para ilustrar esta idea, consideremos un ejemplo clásico, una función que calcula la secuencia de Fibonacci.

```
1 def fibonacci(n: int) -> int:
2     if n <= 1:
3         return n
4     return fibonacci(n-1) + fibonacci(n-2)
```

Ahora, analicemos las siguientes funciones que hacen uso de esta secuencia.

```python
def rabbit_population_growth(n_months: int) -> int:
    """
    Computes the number of rabbit pairs after a given number of months.
    Params:
        n_months (int): The number of months to calculate.

    Returns:
        int: The total number of rabbit pairs after n_months.
    """
    return fibonacci(n_months)

def count_drone_ancestors(n_generations: int) -> int:
    """
    Computes the number of ancestors of a drone bee after a given
    number of generations.
    Params:
        n_generations(int): The number of generations to trace back.

    Returns:
        int: The total number of ancestors in n_generations.
    """
    return fibonacci(n_generations)
```

Aunque `rabbit_population_growth` y `count_drone_ancestors` compartan implementación y semántica formal, su semántica en lenguaje natural difiere ya que realizan tareas distintas. La primera función comunica una historia sobre la reproducción de los conejos, mientras que la segunda se centra en los ancestros de los zánganos de una colmena. Esta diferencia nos permite entender la intención del desarrollador, porque con un simple vistazo al nombre de la función comprenderemos un poco más sobre el contexto del código en general.

A. El código cuenta una historia

Si un sistema de *software* es lo suficientemente complejo, estará compuesto por una gran cantidad de funciones, módulos y clases que interactúan entre sí (hablaremos simplemente de funciones en pos de mejorar la fluidez del texto). Si el código no es organizado correctamente, no sólo se hará muy difícil de entender, sino que también de extender y mantener. Para evitar estos problemas, es muy importante adoptar un enfoque de trabajo claro y estructurado, como el enfoque ***top-down***.

Cuando hablamos del enfoque *top-down* decimos que debemos ir del nivel más general al más específico, descomponiendo el problema en partes más pequeñas y manejables. En otras palabras, el código que escribe un desarrollador debe contar una **historia**: la función principal o `main` debe actuar a modo de índice o resumen, presentando los 'capítulos' que no son más que las funciones dentro del programa. Cada función detalla una parte específica de la historia, mientras que los componentes como variables y controladores de flujo dentro de las funciones desarrollan el contenido. Observemos el siguiente código:

```
def nuclear_reactor_controller():
    for control in CONTROL_LIST:
        control_result = execute_control(control)
        if control_result.failed():
            trigger_alarm(control, control_result)
            execute_emergency_plan(control, control_result)
```

La mayoría de los lectores probablemente no entienda los detalles técnicos sobre reactores nucleares, pero este fragmento de código cuenta una historia lo suficientemente clara como para comprender a grandes rasgos lo que ocurre. Describe como un reactor realiza una serie de controles rutinarios y, si alguno de ellos falla, se activa una alarma y se ejecuta un plan de emergencia. Si bien hay muchos detalles que no conocemos, como cuáles son los controles o como se activan las alarmas, el diseño facilita explorar las funcionalidades internas y comprender la lógica detrás de estas acciones.

Entonces, para escribir una buena historia en código, primero debemos tener en cuenta conceptos fundamentales, como el uso adecuado de nombres de variables y funciones, escribir comentarios claros, seguir convenciones del lenguaje que se está utilizando y ser consistentes con el idioma a lo largo del código. Estos son los pilares para escribir un código bonito que comunique apropiadamente la intención del desarrollador, logrando que un proyecto complejo tenga una semántica en lenguaje natural evidente.

2. El arte de nombrar

Todo código está compuesto por funciones y variables. Las funciones permiten abstraernos de un bloque de sentencias y reutilizarlo a lo largo de todo el programa. Mientras que las variables nos permiten almacenar y manipular datos. Tanto las funciones, como las variables tienen nombres que nos permiten identificarlas y utilizarlas. A nivel sintáctico, algunos lenguajes imponen restricciones, como exigir que los nombres de las funciones comiencen con minúscula o prohibir comenzar con un número. Pero más allá de esas reglas, el desarrollador es completamente libre de escoger cualquier nombre. El problema es que con frecuencia se utilizan nombres vagos, confusos o ambiguos, lo que dificulta la comprensión del código. Un nombre estará bien elegido si hace inequívoca su semántica en lenguaje natural.

Al nombrar incorrectamente una función, generamos malinterpretaciones, ya que otro desarrollador podría pensar que la función realiza acciones que realmente no ejecuta, o por el contrario, ocultamos funcionalidades que no se reflejan en el nombre. Lo mismo ocurre con las variables, nombres poco claros pueden dificultar comprender el tipo de dato que éstas almacenan o cómo es que ese dato está siendo utilizado en el sistema. Por ello es que debemos elegir cuidadosamente palabras específicas que describan con precisión el propósito de nuestros elementos, evitando términos genéricos o vacíos que puedan causar ambigüedad.

Imaginemos una función llamada `processData()`, ¿qué es lo que pretende el desarrollador qué esta función haga? Comprender esto tan solo mirando el

nombre se vuelve una tarea casi imposible. ¿Suma distintos valores? ¿Filtra elementos según alguna regla específica? En definitiva, no es algo claro. Por otro lado, nombres como `calculateTotalWithTaxes()` o `filterValidatedUsers()` brindan mucha más información sobre la finalidad de la función. Lo mismo ocurre con las variables, un caso recurrente es llamarlas `data` o `value`. Estos nombres no ofrecen nada de información sobre su propósito o contenido. Incluso, en lenguajes sin sistema de tipos como Python o JavaScript, ni siquiera se tiene información sobre el tipo de dato que contiene.

> **Lineamiento:** Las funciones y variables deben tener nombres descriptivos que ayuden a comprender su significado.

¿Cómo podemos elegir un buen nombre para nuestras funciones y variables? La clave está en usar palabras adecuadas para describir claramente lo que pretendemos con ellas. Una regla esencial es utilizar nombres fáciles de localizar y pronunciar [2]. Proyectos grandes suelen contener múltiples archivos y carpetas, que a su vez poseen gran cantidad de variables y funciones, por lo que nombres descriptivos y fáciles de buscar mejoran la legibilidad y ahorran tiempo. Además, los nombres que se pueden decir con naturalidad también son más fáciles de recordar y compartir. En cambio, un nombre críptico que solo entiende su autor complica la comunicación y dificulta el trabajo en equipo.

A. Lineamientos para nombrar funciones

Al momento de nombrar funciones, es fundamental utilizar verbos. Dado que las funciones realizan acciones, qué mejor que utilizar verbos que son perfectos para ello. Elegir el verbo correcto puede marcar una gran diferencia entre un nombre claro y uno ambiguo. Por ejemplo, usar `distribute` en lugar de `send`, o `identify` en lugar de `find` puede dar lugar a nombres mucho más precisos e informativos [2]. Si no somos capaces de encontrar un verbo que describa precisamente la intención de nuestro código, entonces puede ser que la función en cuestión realice más de una acción y deba modularizarse. Asegurar que una función realice una única tarea es muy importante y por eso trataremos este tema en los siguientes capítulos. [9]

Además, como nos enseñan desde los primeros años de escuela, los verbos suelen estar acompañados por otras palabras que brindan más contexto sobre la acción. En las funciones, esto es igual de importante. Necesitamos términos específicos que describan con claridad el alcance de la función. En nuestro ejemplo anterior `calculateTotalWithTaxes`, no solo nos indica que se está calculando un valor total, sino que además, se están considerando impuestos.

> Una práctica común al trabajar con clases es nombrar los métodos que acceden o modifican valores internos con prefijos *get* y *set*. Esto indica automáticamente si el método devuelve un valor de una propiedad interna o por el contrario lo modifica.

B. Lineamientos para nombrar variables

Así como podemos dar nombres descriptivos a las funciones, existen algunas buenas prácticas al nombrar variables que hacen que sea más fácil entender el propósito del código. En este caso, el uso de sustantivos es ideal para las variables, ya que representan entidades dentro del programa. No obstante, el tipo de la variable también influye en cómo debería nombrarse. Para variables de tipo `bool`, es recomendado utilizar prefijos como `is`, `has` o `can`. Dado que estas palabras suelen iniciar las preguntas en inglés, nombres como `isVisible` o `hasAccess` resultan intuitivos y ayudan a comprender el significado de su valor en un momento dado. Es importante, sin embargo, evitar nombres con este formato que incluyan una negación, como `isNotOpen`, ya que, aunque se entiende su objetivo, puede generarse confusión al momento de su uso.

En el caso de arreglos, listas o conjuntos de valores, los nombres en plural son una buena práctica, como `adminCommands` o `validUsers` para reflejar la multiplicidad de elementos. Para variables numéricas, prefijos como `max`, `min` o `total` añaden contexto valioso si el valor implica algún tipo de rango o límite. Asimismo, si la variable representa alguna unidad medible (como tiempo, distancia o dinero), incluir una referencia a la unidad en el nombre aporta mucha claridad y reduce posibles errores de conversión innecesarios [2, 9].

Otra buena práctica al nombrar constantes o variables es aprovechar el nombre de la función con la que las inicializamos. Si la función tiene un nombre adecuado, es decir, es descriptivo y no genera confusión, podemos usarlo como referencia para nombrar nuestra variable de manera coherente. Veamos un ejemplo donde esto no se respeta:

```
1 new_product = self._get_product_basic_info(product)
```

El nombre `new_product` sugiere que la variable almacena un objeto de una clase, pero si observamos el nombre de la función, vemos que en realidad devuelve la información básica de un producto. Un nombre más preciso y alineado con su contenido sería:

```
1 product_basic_info = self._get_product_basic_info(product)
```

C. Longitud de los nombres

Muchas veces, al intentar ser específicos con nuestros nombres, surge un nuevo problema: la longitud de estos. Entonces ¿cuál es la longitud perfecta para un nombre? En general, nombres demasiado largos pueden ser difíciles de recordar y ocupan mucho espacio en pantalla, pero por otro lado, nombres cortos no ofrecen tanta información. La clave, como siempre, es encontrar un equilibrio, pero también existen algunas recomendaciones que podemos seguir [2]:

- Si el alcance de la función o variable es pequeño, por ejemplo, una función que sólo se utiliza en el mismo archivo en la cual se define o una variable con vida útil de unas pocas líneas, entonces está bien optar por nombres cortos. Imaginemos que estamos creando un paquete con funciones matemáticas, y tenemos una función auxiliar para calcular la magnitud o norma de un

vector, podríamos nombrar a nuestra función como `norm()` en lugar de `calculateVectorMagnitude()`

- Intentaremos evitar el uso de acrónimos y abreviaciones siempre que sea posible. Los nuevos desarrolladores o aquellos con poco conocimiento del código podrían tener dificultades para comprender su significado. Por ejemplo, en lugar de `calcTtl()`, usar `calculateTotalPrice()`.
- Eliminar palabras que no aporten información relevante. Por ejemplo, usar `toString()` en lugar de `convertToString()`.

Siguiendo estas recomendaciones, lograremos nombres más claros y concisos que aportarán legibilidad y facilitarán la comprensión de las funciones y el código en general.

3. Tipado en el código

A. Tipos de dato, tipos de función y su comportamiento

Un **tipo de dato** (o simplemente tipo) define el conjunto de valores que una variable puede almacenar y las operaciones que se pueden realizar sobre esos valores. De forma similar, las funciones también poseen un tipo, conocido como **tipo de una función**, que describe el tipo de sus parámetros y su valor de retorno.

En la mayoría de los lenguajes de programación, los tipos de datos se pueden clasificar en tres categorías:

- **Primitivos**: Tipos básicos proporcionados por el lenguaje, como `int`, `float`, `char` o `bool`.
- **Compuestos**: Estructuras que agrupan múltiples valores, como `array`, `tuple` o `struct`.
- **Personalizados**: Tipos definidos por el desarrollador a partir de tipos primitivos o compuestos. Estos se utilizan para representar entidades específicas.

Cada tipo de dato requiere distinta cantidad de memoria y permite realizar ciertas operaciones. Por ejemplo, una variable booleana sólo puede almacenar los valores `true` o `false`, lo que generalmente ocupa un solo byte en memoria[3] . Por otro lado, los tipos numéricos pueden representar un rango mucho más amplio de valores, por lo que su tamaño en memoria es mayor.

Un caso reciente que demuestra la importancia de elegir los tipos adecuados se vio con la publicación del modelo de lenguaje de la empresa china **DeepSeek**. A diferencia de sus competidores, los desarrolladores de DeepSeek optaron por utilizar menos bits para sus variables numéricas[4]. Esta decisión permitió que su modelo ocupara significativamente menos memoria, logrando así un sistema más eficiente.

[3] Si bien conceptualmente, un valor booleano debería ocupar 1 bit, muchos lenguajes utilizan un byte al ser esto la unidad mínima direccionable de memoria

[4] `https://www.inferless.com/learn/the-ultimate-guide-to-deepseek-models`

B. Tipado estático vs tipado dinámico

Si bien todos los lenguajes de programación cuentan con algún sistema de tipos, no todos lo manejan de la misma manera. En algunos, el sistema de tipos es explícito y obligatorio, pero en otros casos, existe de forma implícita y sólo se verifica durante la ejecución. Estas diferencias nos llevan tener dos enfoques principales [6]:

Tipado estático

En los lenguajes con **tipado estático** como `C`, `Java` o `Rust`, es necesario especificar el tipo de cada variable declarada. Una vez definido, este tipo no puede cambiar a lo largo del programa. El compilador se encarga de verificar que todas las operaciones y funciones respeten estos tipos, lo que permite detectar errores incluso antes de ejecutar el código.

Tipado dinámico

Lenguajes como `JavaScript` y `Python` utilizan **tipado dinámico**. En ellos, el tipo de una variable se determina durante la ejecución del programa, e incluso puede contener valores de distintos tipos de datos en diferentes momentos. Esta flexibilidad suele agilizar el desarrollo al comienzo, pero también incrementa el riesgo de cometer errores si no se tienen las precauciones suficientes.

C. ¿Por qué queremos tipar?

Algunos desarrolladores consideran que la flexibilidad de tipos en el tipado dinámico es una de las principales virtudes de ciertos lenguajes, pero la verdad es que tipar el código va más allá de una simple formalidad. El tipado es una herramienta clave que mejora la calidad del código. En proyectos pequeños o funciones simples, puede parecer innecesario o incluso una pérdida de tiempo, pero adquirir el hábito de tipar desde el principio es beneficioso. En sistemas más complejos, los tipos permiten comprender rápidamente el propósito de las funciones con un simple vistazo, ya que definen claramente los tipos de entrada y salida. Además reducen errores y facilitan la mantenibilidad. Cuando combinamos un tipado explícito con buenos nombres de variables y funciones, obtenemos un código claro y fácil de entender.

> **Lineamiento:** Tipar siempre las variables y las funciones.

Tipado en JavaScript y Python

Aunque JavaScript y Python utilizan tipado dinámico por defecto, originalmente no contaban con un sistema de tipos formal. Con el tiempo, a medida que los proyectos en estos lenguajes se volvieron más complejos, se hizo evidente la necesidad de incorporar mecanismos de tipado que mejoraran la claridad del código. Esto dio lugar al desarrollo de lenguajes como TypeScript para JavaScript y herramientas como las anotaciones de tipo en Python, que permiten un mayor control sobre los tipos sin renunciar a la flexibilidad que caracteriza a ambos

lenguajes.

Las **anotaciones de tipo** de Python en el módulo `typing`, son *ayudas visuales que se incluyen en las variables, parámetros y funciones*. Estas anotaciones no interfieren de ninguna manera con la ejecución del código pero sirven de guía tanto al desarrollador como a herramientas externas. En el siguiente fragmento de código, podemos observar una función que devuelve un `bool` con un parámetro de tipo `List[int]`.

```
def all_positives(numbers: List[int]) -> bool:
  # code ...
```

Por otro lado, para JavaScript se desarrolló TypeScript, un superconjunto del lenguaje que agrega tipado estático opcional entre otras mejoras. A diferencia de Pyhton, TypeScript si detecta errores de tipos, esto lo hace al momento de *transpilar* el código a JavaScript, ya que TypeScript no se ejecuta directamente, sino que es convertido a un archivo `.js`. En el siguiente fragmento de código podemos observar una implementación de TypeScript.

```
function allPositives(numbers: Array<number>): boolean {
  // code ...
}
```

D. Recomendaciones al tipar

Terminamos esta sección con algunas situaciones a evitar y recomendaciones al momento de trabajar con tipos en los lenguajes Python y TypeScript. Estas recomendaciones deberían adaptarse siempre que sea posible al lenguaje de programación con el que se este trabajando.

Si bien el tipado es una herramienta muy útil con la que podemos contar, existen malas prácticas que muchos desarrolladores suelen cometer.

- **Abusar del tipo** `any`: En TypeScript, el tipo `any` permite omitir la verificación de tipos en las variables donde se utiliza, lo que significa que el transpilador no aplicará comprobaciones de tipo sobre ellas. Entonces, ¿para qué usar un sistema de tipos si se ignora su principal ventaja? Esto no solo complica la lectura del código, sino que también aumenta el riesgo de errores. Si realmente no se conoce el tipo de una variable, es preferible usar `unknown`, que expresa explícitamente que el tipo es desconocido, pero mantiene la seguridad en tiempo de compilación. En Python ocurre algo similar: el uso del tipo `Any` simplemente dificulta la tarea de otros desarrolladores.

- **Evitar el casteo de tipos**: En TypeScript, el casteo de tipos nos permite forzar la interpretación de un dato como otro tipo sin modificar su valor real. A diferencia de Python, donde `int()` o `str()` transforma efectivamente un dato, en TypeScript simplemente se le dice al transpilador que *confíe* en el desarrollador. Esto puede ocultar errores, provocar inconsistencias y hacer que el código sea menos mantenible.

> ℹ El casteo de tipos (*type casting* en inglés) es el proceso de convertir un tipo de dato en otro. En TypeScript, podemos realizar un casteo de un dato en otro tipo haciendo uso de la palabra `as`. Por ejemplo: `'2' as number;` le dirá al transpilador que interprete la cadena de texto que contiene al carácter 2 como un número.

Por otro lado, con el tipado estático evitamos errores y hacemos nuestro código más claro y mantenible. Para aprovechar esta funcionalidad al máximo, es recomendable seguir algunas buenas prácticas:

- **Definir tipos personalizados**: Tanto en TypeScript como en Python, podemos crear nuestros propios tipos mediante interfaces, clases o alias. Esto promueve la reutilización de estructuras de datos bien definidas y mejora la claridad.
- **Validar tipos de fuentes desconocidas**: Al trabajar con APIs, librerías de terceros o datos de origen desconocido, es fundamental validar los tipos para evitar errores. En TypeScript, librerías como `Zod`[5] permiten definir esquemas de validación robustos, mientras que en Python, herramientas como `Pydantic`[6] facilitan la validación de datos en tiempo de ejecución. Si un dato no cumple con el formato esperado, estas herramientas permiten lanzar errores de manera controlada, evitando fallos más graves en el sistema.

4. Otras recomendaciones

A. Seguir las convenciones del lenguaje

Los desarrolladores son libres de escribir el código de la manera que ellos deseen siempre y cuando este funcione correctamente. Sin embargo cada lenguaje de programación cuenta con un conjunto de directrices que recomiendan estilos, prácticas y métodos para distintos aspectos del desarrollo. Estas convenciones buscan estandarizar la jerarquía y la arquitectura de archivos y carpetas, las reglas para comentarios, y el formato de nombres y espaciado, entre otros aspectos.

Seguir estas convenciones ayuda a mantener la uniformidad en los proyectos de *software*. Si el código luce consistente en todos los archivos y módulos, será más fácil comprender su estructura y funcionamiento. Como resultado, el mantenimiento y la colaboración se simplifican. Aunque no es obligatorio seguir estas normas, conocerlas y aplicarlas es esencial para dominar un lenguaje por completo.

A continuación se presentan las convenciones de nombres para funciones, variables, clases y otros elementos en Python y JavaScript.

[5] `https://zod.dev`
[6] `https://docs.pydantic.dev/latest/`

Python

- **Funciones**: en minúsculas, con palabras separadas por guion bajo (*snake_case*). Ejemplo `my_function`.
- **Variables**: siguen la misma convención que las funciones.
- **Clases**: cada palabra inicia con mayúscula y no se usan separadores (*PascalCase*). Ejemplo: `MyClass`.
- **Métodos**: igual que las funciones, en *snake_case*.
- **Constantes**: igual que las funciones, pero completamente en mayúsculas (*SCREAMING_SNAKE_CASE*). Ejemplo `THIS_CONSTANT`.
- **Paquetes**: en minúsculas, sin guiones bajos. Ejemplo `mypackage`.

JavaScript - AirBnB Style Guide [7] -

- **Funciones**: la primer palabra en minúscula, las siguientes con mayúscula inicial y sin separadores (*camelCase*). Ejemplo `myFunction`.
- **Variables**: siguen la misma convención que las funciones.
- **Clases**: igual que en Python, usando *PascalCase*. Ejemplo: `MyClass`.
- **Métodos**: igual que las funciones y las variables, en *camelCase*.
- **Constantes**: se escriben en mayúsculas con guion bajo (*SCREAMING_SNAKE_CASE*), como en Python. Ejemplo `THIS_CONSTANT`.
- **Paquetes**: depende del tipo de proyecto y del archivo, en general no hay una convención definida salvo casos especiales.

B. Ser consistentes en el uso del idioma

Elegir un idioma y mantenerlo a lo largo de un proyecto es fundamental para mantener la coherencia en el código. Si, por ejemplo, en un archivo utilizamos una variable `counter` y luego en otro una variable `contador`, estaremos creando una inconsistencia que puede generar confusión, especialmente en equipos de trabajos con hablantes de diferentes idiomas.

Por lo general, el inglés suele ser el idioma preferido para escribir código, ya que coincide con las palabras claves de la mayoría de los lenguajes de programación y además facilita la comunicación e integración en equipos de trabajo multiculturales. Es importante evitar el uso de caracteres especiales como *ñ*, *á*, *ü* ya que pueden provocar errores de compatibilidad o dificultar la escritura y comprensión del código. Por otro lado, gran parte de la documentación de lenguajes, librerías y APIs están en inglés, por lo que al elegir este idioma también facilitamos el acceso a recursos y buenas prácticas.

[7] `https://github.com/airbnb/javascript`

5. Resumiendo lineamientos

- La semántica en lenguaje natural tiene que ser evidente: Cuando alguien revisa cualquier código, debería ser capaz de explicar qué hace sin problemas.
- Al escribir código con un enfoque top-down: las funciones van desde lo más general a lo más concreto, siempre contando una historia, obteniendo así una semántica en lenguaje natural evidente.
- Es importante elegir buenos nombres de variables y funciones. Ambos tipos de nombres tienen que guardar coherencia.
- Tipar nos ayuda a darle más claridad al código.
- Seguir las convenciones del lenguaje que uses.
- Ser consistente con el idioma.

IV. Diseño de funciones

1. Las funciones como método de organización

Las **funciones** son uno de los elementos esenciales en todo lenguaje de programación. Son *bloques de código que nos permiten agrupar un conjunto de instrucciones o sentencias bajo un mismo nombre para ejecutar tareas específicas*. Usarlas trae múltiples beneficios, siendo uno de los más evidentes la **reutilización de código**. Supongamos una aplicación web donde es necesario validar direcciones de correo electrónico, sería razonable contar con una función llamada `validate_email` que realice esta tarea. Dicha función se utilizaría siempre que sea necesaria la validación.

Cuando nos enseñan a programar, a menudo se nos explica que las funciones sólo existen para evitar la repetición de código. Pero rara vez se menciona que también son una herramienta que puede servir para la organización del mismo. Este propósito suele quedar en segundo plano porque explicarlo requiere de ejemplos más complejos, lo cuál no es siempre viable en un curso introductorio.

Lamentablemente, luego de los primeros cursos de programación, pocas veces se vuelve a estudiar el concepto de función en profundidad. Como resultado, muchos desarrolladores no llegan a comprender su verdadero potencial en la estructuración de código. En realidad, crear funciones incluso cuando no hay código repetido puede ser una estrategia importante para mejorar la legibilidad de un programa.

2. Las funciones deben ser pequeñas

En el cuerpo de las funciones encontramos **sentencias**. Una sentencia en un lenguaje de programación es una *instrucción completa que indica a la computadora qué tarea ejecutar* [3]. Son las unidades fundamentales de ejecución y, en general, están delimitadas por algún símbolo específico, como `;` en lenguajes como `C` o `JavaScript`, o por un salto de linea en `Python`.

Es importante notar la diferencia entre una sentencia y una línea de código. Aunque en muchos casos coincidan, una sentencia puede ocupar múltiples líneas si agregamos saltos de línea para mejorar la legibilidad. Del mismo modo, una línea puede contener múltiples sentencias si las escribimos en secuencia. En resumen: una sentencia es una unidad lógica de ejecución, mientras que una línea de código es sólo un aspecto visual del código de un programa.

> **Lineamiento:** Los cuerpos de las funciones deberían tener idealmente unas 10 sentencias o menos.

El diseño de las funciones es clave para la organización del código. Mantener el cuerpo de las funciones corto puede parecer un límite innecesario para desarrolladores acostumbrados a escribir funciones largas que simplemente cumplen con su propósito. Sin embargo, reducir la cantidad de sentencias en una función trae ventajas importantes [9]:

- **Reduce el número de responsabilidades**, idealmente a una sola.
- **Facilita la asignación de nombres descriptivos** que reflejen claramente su propósito.

Veamos un ejemplo de una función que contradice este principio. En el siguiente código observamos como se realizan varias tareas al mismo tiempo

```
type Matrix = number[][]
function multiplyAndFormatMatrix(matrixA: Matrix, matrixB: Matrix): string {
  // Validate the input
  if (
    matrixA.length === 0 ||
    matrixA[0].length === 0 ||
    matrixB.length === 0 ||
    matrixB[0].length === 0
  )
    throw new Error("Invalid matrix, no elements found");
  if (matrixA[0].length !== matrixB.length)
    throw new Error(
      "Invalid matrix, the number of columns of the first matrix must be equal to the number of rows of the second matrix"
    );

  // Multiply the matrices
  const multipliedMatrix = matrixA.map((row) =>
    matrixB[0].map((_, colIndex) =>
      row.reduce(
        (sum, item, rowIndex) => sum + item * matrixB[rowIndex][colIndex],
        0
      )
    )
  );

  // Pretty print the result
  const formattedRows = multipliedMatrix.map((row) => `| ${row.join("\t")} |`);
  return formattedRows.join("\n");
}
```

Esta función tiene demasiadas responsabilidades:

1. Valida las matrices de entrada.
2. Realiza la multiplicación.
3. Le da un formato específico al resultado en una cadena de texto.

Para mejorarla, veamos como dividirla en funciones más pequeñas, asegurando que cada una tenga un único propósito.

```
function multiplyAndFormatMatrix(matrixA: Matrix, matrixB: Matrix): string {
  validateMatrices(matrixA, matrixB);
  const multipliedMatrix = multiplyMatrices(matrixA, matrixB);
  return formatMatrix(multipliedMatrix);
}

function validateMatrices(matrixA: Matrix, matrixB: Matrix): void {
  validateMatrixHasElements(matrixA);
  validateMatrixHasElements(matrixB);
  validateMatrixDimensions(matrixA, matrixB);
}

function validateMatrixHasElements(matrix: Matrix): void {
  if (matrix.length === 0 || matrix[0].length === 0)
    throw new Error("Invalid matrix, it must have at least one element");
}

function validateMatrixDimensions(matrixA: Matrix, matrixB: Matrix): void {
  if (matrixA[0].length !== matrixB.length)
    throw new Error(
      "Invalid matrix, the number of columns of the first matrix must be equal to the number of rows of the second matrix"
    );
}

function multiplyMatrices(matrixA: Matrix, matrixB: Matrix): Matrix {
  return matrixA.map((row) =>
    matrixB[0].map((_, colIndex) =>
      row.reduce((sum, item, rowIndex) => sum + item * matrixB[rowIndex][colIndex], 0)
    )
  );
}

function formatMatrix(matrix: Matrix): string {
  return matrix.map((row) => `| ${row.join("\t")} |`).join("\n");
}
```

En esta nueva interpretación, la función principal `multiplyAndFormatMatrix` cuenta una historia fácil de seguir: primero se realiza la validación, luego la multiplicación y, finalmente, se da formato. A su vez, dentro de la validación también encontramos una secuencia lógica: primero se verifica cada matriz por separado y, luego, se validan las dimensiones de ambas.

El resto de las funciones no narran una historia, sino que realizan operaciones específicas, cada una reflejada claramente en su nombre. Además, este código no requiere de comentarios adicionales, ya que las funciones son lo suficientemente cortas y sus nombres están bien elegidos.

Seguramente, algunos lectores habrán notado que el segundo código es más extenso. **Esto no importa**. Más líneas de código o sentencias no implican necesariamente una mayor complejidad algorítmica (es decir, el segundo programa no es significativamente más costoso en términos de ejecución). A nivel humano,

siempre es preferible trabajar con un código más extenso pero comprensible, en lugar de uno más compacto pero difícil de entender.

A. Los requerimientos evolucionan

En cualquier proyecto, los requerimientos están en constante evolución. Supongamos que el cliente que solicitó la funcionalidad en `multiplyAndFormatMatrix` ahora necesita únicamente validar y multiplicar las matrices, sin dar un formato al resultado. Con el segundo enfoque, implementar este cambio sería tan sencillo como escribir lo siguiente:

```
function multiplyMatrixes(matrixA: Matrix, matrixB: Matrix): string {
  validateMatrixes(matrixA, matrixB);
  return multiplyMatrixes(matrixA, matrixB);
}
```

En cambio, en el primer código, cumplir con este nuevo requerimiento implicaría refactorizar la función lo cual no siempre implica una tarea sencilla.

3. El código crece horizontalmente

Ya hemos visto como el código puede crecer verticalmente y qué debemos hacer para reducir esta extensión. Sin embargo, el código también se expande horizontalmente y esto representa un problema para la legibilidad, y en consecuencia, el mantenimiento.

A. Líneas demasiado largas

El primer culpable que contribuye al crecimiento horizontal son las **líneas demasiado largas**. Estas pueden surgir por diversas razones, como cadenas de texto extensas, nombres de variables o funciones excesivamente largos, expresiones aritméticas o lógicas complejas, y llamadas a funciones con numerosos parámetros.

Históricamente, se estableció un límite de 80 caracteres por línea, una convención que sigue siendo muy apoyada. Con este valor, ningún desarrollador debería tener problemas de visualización en su pantalla. De todos maneras, con la evolución de las mismas, de los editores y los lenguajes, algunos desarrolladores han ampliado este límite hasta 120 caracteres. Más allá del número exacto, lo importante es evitar líneas excesivamente largas que dificulten la lectura, y más importante aún, **prevenir el desplazamiento horizontal**, ya que esto afecta gravemente la navegabilidad en el código.

> **Lineamiento:** Una línea de código jamás debe provocar desplazamiento horizontal.

Para solucionar este problema, podemos aplicar varias estrategias. Por ejemplo [9]:

Dividir expresiones en múltiples líneas

En lugar de una expresión larga en una única línea

```
total_price = base_price + (base_price * tax_rate) - (base_price *
    discount) + shipping_fee
```

Podemos dividirla en varias líneas que mejoran la lectura

```
total_price = (
  base_price
  + (base_price * tax_rate)
  - (base_price * discount)
  + shipping_fee
)
```

Notar que esta solución si bien añade más lineas a nuestro código, no añade más sentencias. Por otro lado, todo editor moderno tiene la opción de colapsar sentencias, luego, usando esta opción uno vería más o menos lo siguiente:

```
> total_price = (...
```

y podría expandir la sentencia cuando sea necesario.

Utilizar variables intermedias

Si tenemos una línea con múltiples operaciones

```
final_value = (quantity * price_per_item) + (quantity * price_per_item
    * tax) - discount
```

Podemos descomponerla en variables intermedias

```
subtotal = quantity * price_per_item
tax_amount = subtotal * tax
final_value = subtotal + tax_amount - discount
```

En este caso sí estamos añadiendo más sentencias a nuestra función, pero no está suponiendo ninguna complejidad visual al código.

Reestructurar funciones con muchos parámetros

La siguiente función posee muchos parámetros en una sola línea:

```
def send_email(receiver: str, subject: str, message: str, is_html: bool
    , attach_signature: bool, template: str) -> bool:
  # code ...
```

Podemos reescribirla de la siguiente forma:

```
def send_email(
  receiver: str,
  subject: str,
  message: str,
  is_html: bool,
  attach_signature: bool,
  template: str
) -> bool:
  # code ...
```

Esta estrategia también es aplicable en las llamadas a función, veamos el siguiente código donde tenemos muchos parámetros:

```
send_email(user.email, "Welcome!", "Hello, we are happy to have you.",
    True, False, "footer.html")
```

Al escribirlo en múltiples líneas, hacemos que sea más legible:

```
send_email(
  user.email,
  "Welcome!",
  "Hello, we are happy to have you.",
  True,
  False,
  "footer.html"
)
```

Si bien no es recomendable que una función tenga demasiados parámetros, en algunos casos las librerías externas nos imponen esta estructura. Más adelante en este capítulo abordaremos esta problemática en detalle.

Reestructurar diccionarios u objetos

Un problema similar ocurre con los diccionarios de Python y los objetos de JavaScript. Estos se vuelven muy largos para definirlos en una única línea. La solución anteriormente presentada también se aplica a estos casos:

```
login_error = {"name": "Login error", "http_status": 400, "context": "
    ...", "message": "The username or password is incorrect"}
```

El código anterior, puede ser modificado de la siguiente manera:

```
login_error = {
  "name": "Login error",
  "http_status": 400,
  "context": "...",
  "message": "The username or the password in incorrect"
}
```

B. Muchos niveles de indentación

El exceso de niveles de indentación es otro factor que contribuye al crecimiento horizontal del código. La indentación, que consiste en agregar espacios al inicio de las líneas, se utiliza para reflejar la estructura jerárquica del programa y facilitar la lectura del flujo de ejecución. En lenguajes como Python, es parte obligatoria de la sintaxis, mientras que en otros cumple principalmente una función visual.

Si bien una buena indentación ayuda a entender la organización del código, cuando se acumulan demasiados niveles suele ser indicio de una lógica innecesariamente compleja. En estos casos, conviene reorganizar el código usando funciones auxiliares o instrucciones como `return`, `break` o `continue` para evitar bloques anidados y mejorar la claridad.

> **Lineamiento:** Una función no debe tener más de 3 niveles de indentación.

A continuación serán presentadas algunas estrategias simples para reducir la indentación en los programas.

Abstraer niveles de indentación en nuevas funciones

Observemos la siguiente función `process_nested_json()`, que se encarga de procesar una lista de objetos anidados:

```
def process_nested_json(data: List) -> List:
  results = []

  for user in data.get("users", []):
    for order in user.get("orders", []):
      if order.get("status") == "completed":
        for item in order.get("items", []):
          if item.get("type") == "special":
            results.append({
              "user_id": user.get("id"),
              "order_id": order.get("id"),
              "item_id": item.get("id"),
            })

  return results
```

Claramente la función no sigue el lineamiento definido sobre 3 niveles máximos de indentación. Por esto mismo, comprender qué realiza el cuerpo de la función no es una tarea fácil. Comparemos esta implementación con una que modulariza mejor la tarea introduciendo funciones auxiliares:

```
def process_nested_json(data):
  special_items = []
  for user in data.get("users", []):
    special_items += get_special_items_from_completed_orders(user)

  return special_items


def get_special_items_from_completed_orders(user):
  special_items = []
  for order in user.get("orders", []):
    if order.get("status") == "completed":
      special_items += get_special_items_in_order(order)

  return special_items


def get_special_items_in_order(order):
  special_items = []
  for item in order.get("items", []):
    if item.get("type") == "special":
      special_items.append({
        "user_id": user.get("id"),
        "order_id": order.get("id"),
        "item_id": item.get("id"),
      })

  return special_items
```

En este caso, la función principal `process_nested_json` se encarga exclusivamente de iterar sobre los usuarios y delegar tareas a otras funciones. Este

enfoque mejora mucho la lectura del código, ya que no es necesario leer por completo toda la implementación. Basta con observar el ciclo `for` y la llamada a la función correspondiente para entender a grandes rasgos qué está ocurriendo: *la función devuelve todos los ítems especiales de las órdenes completadas para todos los usuarios*. Luego, en caso de querer comprender más a fondo, siempre se puede revisar las implementaciones de las funciones auxiliares. De todas maneras, nos gustaría destacar que consideramos que la lógica en esta función no es ideal y se podría intentar realizar una refactorización del código.

Retornar valores de manera temprana

La discusión sobre si las funciones deben tener más de un punto de retorno no tiene una respuesta universalmente correcta, depende en gran medida de cómo el desarrollador implemente la lógica. Sin embargo, los retornos múltiples pueden ser útiles para simplificar la lógica, especialmente cuando queremos prevenir niveles de indentación excesivos [2]. Veamos el siguiente ejemplo:

```
type User = {
  isEmailVerified: boolean;
  age: number;
};

function isValidUser(user: User) {
  let isValid = false;
  if (user) {
    if (user.isEmailVerified) {
      if (user.age >= 18) {
        console.log("Valid user");
        isValid = true;
      } else {
        console.log("Underage user, not valid");
      }
    } else {
      console.log("User email not verified, not valid");
    }
  } else {
    console.log("No user provided, not valid");
  }
  return isValid;
}
```

Si bien es un ejemplo simple, en códigos más complejos podría dificultarse su lectura, principalmente debido a la cantidad de condiciones que hay que tener en mente. Ahora, comparemos con una versión mejorada de esta función que hace uso de retornos tempranos para reducir la anidación y mejorar la lectura:

```
function isValidUser(user: User) {
  if (!user) {
    console.log("No user provided, not valid");
    return false;
  }

  if (!user.isEmailVerified) {
    console.log("User email not verified, not valid");
    return false;
```

```
  }

  if (user.age < 18) {
    console.log("Underage user, not valid");
    return false;
  }

  console.log("Valid user");
  return true;
}
```

En esta segunda versión, las condiciones que invalidan al usuario se manejan inmediatamente, dejando un flujo más claro y eliminando indentación innecesaria.

Hacer uso de `continue` en los ciclos

La estrategia de retornar valores tempranamente no siempre es posible, como en el caso de un ciclo. Su análogo para este caso es hacer uso de la sentencia `continue` para ejecutar la siguiente iteración y evitar anidar más lógica en un ciclo [2].

```
def calculate_foo(value: int) -> int:
  # code ...

def process_values(values_to_compute: List[Optional[int]]) -> List[int
    ]:
  computed_values = []
  for i in values_to_compute:
    if i is not None:
      print("Possible candidate: ", i)
      if i >= 0:
        computed_values.append(calculate_foo(i))
  return computed_values

values = [2, None, -16, 1, -1, None, 5]
process_values(values)
```

Para este simple ejemplo, vemos que el bucle tiene dos condiciones `if` anidadas. Cada una de ellas incluye una indentación nueva y una condición a tener en mente para el lector. Ahora observemos esta nueva versión:

```
def process_values(values_to_compute: List[Optional[int]]) -> List[int
    ]:
  computed_values = []
  for i in values_to_compute:
    if i is None: continue

    print("Possible candidate: ", i)
    if i < 0: continue

    computed_values.append(calculate_foo(i))
  return computed_values
```

En este código podemos observar que, al no cumplirse las condiciones del `if`, automáticamente se realiza un `continue`. Con esto no solo simplificamos la lógica y la lectura, sino que en casos extremos podríamos evitar cálculos costosos a nivel computacional.

4. Espacios en blanco

Los espacios en blanco son cualquier tabulación, salto de línea o simplemente separaciones entre palabras claves, operadores o bloques de código. Si bien no aportan a la funcionalidad real del programa, los espacios en blanco son esenciales para que los programas o clases sean más legibles. Por lo que son un factor muy importante a la hora de reorganizar el código.

Así como en un texto literario el escritor utiliza signos de puntuación para que el lector comprenda el flujo del texto, los desarrolladores deben utilizar los espacios en blanco para permitir que el código respire. Es posible eliminar el desorden visual simplemente separando funcionalidades o acciones similares dentro de una función, o agregando espacios entre operadores. Consideremos el siguiente fragmento de código:

```
interface Information {
  userId:number;
  message:string;
  codification:"hex"|"utf8";
}
function hexToString(toConvert:string) {
  return Buffer.from(toConvert,"hex").toString('utf8');
}
async function getUserById(id:number) {
  const user=db.select().from(db.users).where(eq(db.users.id,id));
  return user.name;
}
async function parseUserInformation(info:Information) {
  const userName=await getUserById(info.userId);
  let message=info.message;
  if (info.codification==="hex") {
    message=hexToString(info.message);
  }
  return `User ${userName} sent the message: ${message}`;
}
```

En este ejemplo, la falta de espacios en blanco hace que el código sea difícil de leer. No hay líneas en blanco entre funciones ni espacios entre operadores, lo que dificulta identificar las distintas secciones del código. Si este estilo desordenado se extiende a un archivo completo, el código se vuelve inmanejable.

> **Lineamiento:** Utilizar espacios en blanco entre las diferentes partes del código.

Veamos ahora la versión corregida:

```
interface Information {
  userId: number;
  message: string;
  codification: "hex" | "utf8";
}

function hexToString(toConvert: string) {
  return Buffer.from(toConvert, "hex").toString('utf8');
```

```
}

async function getUserById(id: number) {
  const user = db.select()
    .from(db.users)
    .where(eq
      (db.users.id, id)
    );

  return user.name;
}

async function parseUserInformation(info: Information) {
  const userName = await getUserById(info.userId);
  let message = info.message;

  if (info.codification === "hex") {
    message = hexToString(info.message);
  }

  return `User ${userName} sent the message: ${message}`;
}
```

Este código es más legible, respira y permite que el desarrollador que lo lee pueda diferenciar más fácilmente cada una de las partes.

A. ¿Cuándo incluir líneas en blanco?

Si observamos el ejemplo anterior como parte de un archivo más grande, podemos notar que existen diferentes *momentos* en el código:

- **Definición de interfaces**: `Information`
- **Funciones auxiliares**: `hexToString` y `getUserById`
- **Función principal**: `parseUserInformation`

Dentro de esta función principal, también existen distintos *momentos*:

- **Inicialización de variables**
- **Controladores de flujo**: `if`
- **Retorno del resultado**

Todos estos *momentos* son las partes de nuestro código, saber diferenciarlas es fundamental para hacer uso del espaciado entre ellas y mejorar la comprensión del código.

Reglas básicas para el uso de espacios en blanco

1. **Separar funciones, clases, interfaces o tipos**

- Esto facilita la identificación rápida de los principales componentes del código.

2. **Agrupar lógicamente los bloques de código dentro de las funciones**
   - Separar secciones dentro de una función con líneas en blanco para distinguir:
     - Definición de variables
     - Llamadas a funciones
     - Bloques de control de flujo (`if`, `while`, `for`, ...)
     - Retorno del resultado
3. **Agregar espacios alrededor de operadores y condiciones** [2]
   - Agregar espacios entre operadores binarios o condiciones complejas dentro de estructuras de control ayuda tanto a quién escribe como a quién lee el código. Esto facilita distinguir los elementos y comprender la precedencia de las operaciones.
   - No es lo mismo leer `a + b` que `a+b`, y esta diferencia se vuelve aun más evidente a medida que las expresiones se vuelven más complejas o se añaden paréntesis. Veamos un ejemplo:

En una condición muy compleja, la falta de espacios dificulta la comprensión de la precedencia de operadores

```
while((isEven||(isOdd&&n%5!==0)&&errorStr===null)){
  // code ...
}
```

Al agregar espacios, la condición se vuelve un poco más clara

```
while ((isEven || ( isOdd && n % 5 !== 0)) && errorStr === null) {
  // code ...
}
```

Aún así, esta condición no deja de ser imperfecta, es complicado leer y comprender que se esta verificando. Revisar y repensar este código debería ser una primera aproximación de cualquier desarrollador.

Uso de herramientas de formateo

Es posible automatizar el manejo de los espacios en blanco mediante *herramientas de formateo de código*, como **Prettier**[8] en JavaScript o **Black**[9] en Python. Estas herramientas aplican reglas para que el código se mantenga con un estilo uniforme. Estas reglas pueden ser adaptadas al estilo que prefiera el desarrollador o requiera el proyecto mediante un archivo de configuración.

Algunos editores de código permiten configurar estas reglas de modo que se ejecuten automáticamente cada vez que se guarda un archivo. Lo que garantiza que todo el código del proyecto mantenga un estilo consistente y sea fácil de leer.

[8] https://prettier.io/
[9] https://black.readthedocs.io/en/stable/

B. Alineación vertical

Un último tipo de espaciado importante es la **alineación vertical** [9] del código mediante tabulaciones o espacios. En esta estrategia, líneas contiguas son organizadas de modo que queden visualmente alineadas, lo que facilita la comprensión del código.

Comúnmente, utilizamos esta técnica en torno al símbolo de igualdad `=`, o al estructurar los elementos de un arreglo de manera que mejore la legibilidad. Aunque no es estrictamente necesario, la alineación vertical añade orden y claridad en fragmentos repetitivos, lo que ayuda a detectar errores de escritura u otros problemas en el código. Sin embargo, esta práctica puede entrar en conflicto con ciertas herramientas de formateo automático, que no siempre preservan la alineación y fuerzan un estilo diferente.

Consideremos el siguiente ejemplo:

```
function configureEndpoints() {
  const userEp          = getEndpointUrl("user",    "v1", true);
  const paymentEndpoint = getEndpointUrl("payment", "v1", false);
  const orderEndpoint   = getEndpoitUrl("order",    "v1", true);
  // ...
}
```

Al alinear las asignaciones en torno al `=`, se facilita la detección de errores. En este caso, podemos notar rápidamente que en la tercera línea hay un error tipográfico: `getEndpoitUrl` en lugar de `getEndpointUrl`.

5. Resumiendo lineamientos

Para mejorar la legibilidad de nuestras funciones debemos tener muy en cuenta los siguientes aspectos:

- El cuerpo de las funciones debe mantenerse corto, 10 sentencias sería ideal. Funciones largas suelen realizar muchas acciones y esto no es deseable.
- Es importante que las líneas de código no produzcan desplazamiento horizontal, ya que esto sólo entorpece el desarrollo y la lectura. Podemos hacer uso de diversas opciones para evitar esto
  - Dividir expresiones en múltiples líneas
  - Utilizar variables intermedias
  - Reestructurar funciones con muchos parámetros
  - Reestructurar diccionarios u objetos
- La indentación excesiva es un problema, dificulta seguir el flujo del código. Nuevamente existen diferentes soluciones:
  - Abstraer niveles de indentación en nuevas funciones
  - Retornar valores de manera temprana

    - Hacer uso de `continue` en los ciclos

- Saber aprovechar los espacios en blanco para que el código respire. Cuando añadimos espacios entre bloques lógicamente similares, logramos que los lectores diferencien *momentos* en el código. Con esto es más simple seguir la idea principal.

V. Documentación y comentarios

1. El valor de los comentarios en el código

El código evoluciona constantemente: se modifica, se borra, se reescribe. Durante este proceso, los desarrolladores deben considerar múltiples factores, desde precondiciones y casos límite hasta suposiciones que no siempre pueden expresarse directamente en el código. Como resultado, cuando un nuevo desarrollador se une al equipo o revisa el código, surgen preguntas inevitables: *¿Por qué no se realizó este chequeo?* o *¿Cuál es la razón detrás de esta decisión?* Para evitar estas situaciones, es fundamental que el código exprese claramente esas consideraciones no triviales. En este capítulo trataremos este tema a través de la **documentación**.

Es importante aclarar que, aunque hablaremos de documentación, no nos referimos a la documentación externa de un proyecto, como planes de desarrollo o descripciones de una API. Documentar externamente puede ser costoso, ya que requiere un mantenimiento constante para mantenerse alineado con el código. Muchas veces, la realidad del sistema está en el código mismo, y la documentación externa tiende a quedarse atrás, lo que genera inconsistencias. Dado esto, los desarrolladores más experimentados siempre terminan revisando el código antes que la documentación. Es por todo esto que nos enfocaremos en la documentación interna que acompaña al código y lo enriquece, ayudando a desarrolladores a comprender más rápidamente la **semántica en lenguaje natural** del código.

Para hacer explícitas las consideraciones que influyen en el código, se emplean dos herramientas principales: los comentarios informativos y los comentarios de documentación interna. Los **comentarios informativos** son *anotaciones dentro del código, y explican decisiones, suposiciones o señalan momentáneamente aspectos a revisar*. Los **comentarios de documentación interna**, por otro lado, se refieren a los *docstrings* de Python o *JSDoc* de JavaScript, los cuales proporcionan descripciones a las funciones, clases y módulos.

Los comentarios también están presentes en la **semántica en lenguaje natural**, es decir, la *descripción del programa según lo que el desarrollador pretende que el código haga*. Un código bien escrito y legible no es suficiente si contiene suposiciones que sólo el desarrollador original conoce. Agregar un comentario preciso puede sumar mucho valor, ya que contextualiza decisiones y explica la *historia* del código. Del mismo modo que un narrador describe las motivaciones de los personajes en una novela, un buen comentario puede aclarar una línea de código que, a simple vista, podría parecer confusa.

En este capítulo analizaremos más a fondo los comentarios y documentación interna. Además daremos recomendaciones que diferencian a un comentario cualquiera de uno que realmente es útil.

2. Tipos de documentación en el código

Como ya vimos, existen distintos tipos de comentarios, cada uno con un propósito específico dentro del código. Comprender sus diferencias es clave para utilizarlos de manera efectiva y evitar comentarios redundantes o innecesarios.

A. Comentarios informativos

Los comentarios informativos explican aspectos del código que no son evidentes a simple vista. Su propósito es aclarar decisiones de diseño, suposiciones o detalles importantes que podrían no ser obvios para otros desarrolladores. Estos comentarios no siguen un formato rígido y pueden encontrarse tanto en una única línea como en un conjunto de estas (comentario en bloque). Al usarlos nos estamos anticipando a las dudas del lector, respondiendo preguntas que aún no se han formulado. Veamos un ejemplo:

```
type Coordinates = {
  latitude: number;
  longitude: number;
};

function calculateDistanceBetweenSatellites(
  satellitePosition1: Coordinates,
  satellitePosition2: Coordinates
): number {
  // Code...

  // Calculate the distance between the satellites using the Haversine formula
  const partialHaversine =
    Math.sin(latitudeDifference / 2) * Math.sin(latitudeDifference / 2) +
    Math.cos(lat1Rad) *
      Math.cos(lat2Rad) *
      Math.sin(longitudeDifference / 2) *
      Math.sin(longitudeDifference / 2);

  const centralAngleRadians = 2 * Math.atan2(
    Math.sqrt(partialHaversine), Math.sqrt(1 - partialHaversine)
  );

  // Code...
}
```

Sin el comentario, la operación matemática principal podría parecer un cálculo arbitrario. Sin embargo, busca responder una pregunta clave: *¿De donde proviene este cálculo?* Notar que este comentario podría no ser necesario si en su lugar escribimos una función con un nombre descriptivo que realice el cálculo de la distancia, dejando los comentarios sólo para aclaraciones que el código por sí solo no pueda transmitir.

Si un comentario hace referencia a varias sentencias, puede ser una señal de que esas líneas deberían ser encapsuladas en una función. Si a esa función le

sumamos un nombre descriptivo, tenemos una función que puede explicarse por si misma, evitando así el uso de comentarios. Idealmente, **el código no debería depender de comentarios para su comprensión**. Aunque a veces es necesario aclarar aspectos que no se pueden expresar con el código, estos casos deberían ser la excepción y no la regla.

> **Lineamiento:** Los comentarios informativos deben utilizarse excepcionalmente.

Existen situaciones donde los comentarios informativos pueden aportar un valor real al código y debemos asegurarnos de que esto realmente ocurra. Una sentencia y un comentario ocupan el mismo espacio en pantalla, por eso es que debemos saber cuando utilizarlos. Ahora bien, ¿qué debería incluir un comentario útil? [2]

- **El por qué antes que el qué**: Un comentario debe aclarar la intención detrás de una sentencia, en lugar de describir lo que hace.
- **Contexto adicional que el código no pueda expresar por si mismo**: Por ejemplo, si hay una limitación técnica o una convención específica que seguir.
- **Decisiones técnicas importantes**: Explicar por qué se eligió una estructura de datos sobre otra o por qué se implementó un algoritmo en particular.
- **Explicación de soluciones no triviales**: Si se resolvió un problema de manera poco convencional, es útil documentarlo para futuros desarrolladores.

No sólo es importante saber cuándo y qué comentar, sino también cómo hacerlo. Un buen comentario debe ser claro y fácil de entender sin omitir detalles esenciales. Además, debe ser breve y directo, evitando cualquier aclaración innecesaria.

Comentarios de marca

Dentro de los comentarios informativos, podemos encontrar una subcategoría: los **comentarios de marca** [9]. A diferencia de los comentarios que explican el código, estos buscan comunicar información a los desarrolladores señalando posibles problemas, tareas pendientes o errores conocidos. Se distinguen porque comienzan con una *palabra de marca* escrita en mayúsculas lo que facilita su identificación en el código. Algunas de las marcas más comunes son:

- **TODO**: Indica una tarea pendiente o alguna funcionalidad que necesita ser implementada.
- **FIXME**: Indica un problema que necesita ser revisado.
- **BUG**: Señala un error conocido que debe ser solucionado.
- **HACK**: Marca una solución temporal o poco ideal que podría mejorarse.

Grandes equipos de trabajo o empresas suelen definir convenciones sobre cuándo y cómo utilizar estas marcas. En algunos casos, incluso crean sus propias *palabras de marca* para reflejar necesidades específicas dentro del proyecto.

No debemos olvidar que estos comentarios deben ser temporales y no permanecer indefinidamente en el código. Idealmente, si se está trabajando en código aledaño y es posible resolver el comentario es bueno hacerlo. Otra opción es, de manera periódica, realizar una búsqueda global en el proyecto para identificar estas anotaciones.

B. Documentación interna

En la **semántica en lenguaje natural**, a veces un buen nombre de función o variable simplemente no alcanza para comunicar completamente la intención del desarrollador. Es por ello que es útil acompañar el código con *docstrings*.

Un ***docstring*** no es más que *un comentario especial ubicado al inicio de una función, cuyo propósito es documentar brevemente su uso y servir como guía para los desarrolladores*. Generalmente se compone de tres partes:

1. **Descripción de la función**: Explica su propósito y contexto de uso.
2. **Descripción de los parámetros**: Detalla los argumentos de entrada, pudiendo incluir pre y post condiciones, así como información extra como los tipos de datos esperados.
3. **Valor de retorno**: Indica qué devuelve la función, con una descripción opcional del resultado y su tipo.

Notar que los *docstrings* no sólo pueden aplicarse a funciones o clases, sino también a variables y otros elementos del código que requieran documentación estructurada.

> ⚠ *docstring* es el término utilizado en Python y otros lenguajes para este tipo de comentarios, pero la mayoría de los lenguajes modernos cuentan con formatos similares. Por ejemplo, JavaScript y TypeScript utilizan **JSDoc**, mientras que Java emplea **Javadoc**, entre otros estándares de documentación

Muchos editores de código permiten visualizar los *docstrings* al colocar el cursor sobre el nombre de una función. Esto resulta especialmente útil al trabajar con librerías externas, ya que permite comprender mejor su uso sin necesidad de revisar la implementación o la documentación externa.

Al escribir el *docstring* de una función, debemos siempre comparar el nombre de la función con lo escrito. Si un *docstring* resulta redundante con respecto al nombre de la función, entonces el nombre está bien elegido. Por otro lado, si el *docstring* utiliza verbos o sustantivos que no surgen en el nombre de la función, esto puede ser indicio de que el nombre está mal elegido. Por esta razón introducimos el siguiente lineamiento:

> **Lineamiento:** Siempre escribir *docstring* y compararlos con el nombre de la función.

El docstring debe aportar información que el nombre de la función no puede expresar por sí solo, como las excepciones que maneja, las unidades de medida de las variables o el formato del valor de retorno. Veamos nuevamente el ejemplo de la función anterior y analicemos su descripción mediante la documentación interna de TypeScript:

```
/**
 * Latitude and longitude in degrees.
 */
type Coordinates = {
  latitude: number;
  longitude: number;
};

/**
 * Calculate the distance between the satellites using the Haversine formula
 * @param {Coordinates} satellitePosition1 - Latitude and longitude of the first satellite in degrees.
 * @param {Coordinates} satellitePosition2 - Latitude and longitude of the second satellite in degrees.
 * @throws {ValueError} - If the latitude or longitude is a non valid number
 *    (i.e. abs(latitude) > 90 or abs(longitude) > 180, NaN, Infinity, etc.)
 * @return {number} - The distance between the two satellites in kilometers
 */
function calculateDistanceBetweenSatellites(
  satellitePosition1: Coordinates,
  satellitePosition2: Coordinates
): number {
  // Code...

  const partialHaversine =
    Math.sin(latitudeDifference / 2) * Math.sin(latitudeDifference / 2) +
    Math.cos(lat1Rad) *
      Math.cos(lat2Rad) *
      Math.sin(longitudeDifference / 2) *
      Math.sin(longitudeDifference / 2);

  const centralAngleRadians = 2 * Math.atan2(
    Math.sqrt(partialHaversine), Math.sqrt(1 - partialHaversine)
  );

  // Code...
}
```

En primer lugar, podemos observar que hay un comentario en la definición del tipo `Coordinates`, que nos indica que si utilizamos este tipo, estamos tratan-

do con valores de latitud y longitud en grados. Esto es crucial, porque previene errores inesperados relacionados con las unidades de medida.

Por otro lado, el comentario principal se encuentra en la función que calcula la distancia entre satélites. En este caso, se especifica que se utiliza la fórmula de *Haversine*, y se proporciona información sobre los tipos. Aunque en TypeScript la definición de tipos hace que esta parte sea redundante, en JavaScript puede ser muy útil. Por último, el comentario nos brinda información extra que no sabríamos sin leer la implementación, como que el tipo del valor de retorno es un número que expresa la distancia entre los dos satélites en **kilómetros**, o que la función lanzará una excepción en caso de valores inválidos para la latitud y la longitud.

A continuación se presenta el código traducido a Python con su correspondiente *docstring*:

```python
# Latitude and longitude in degrees
Coordinates = Tuple[float, float]

def calculate_distance_between_satellites(
    satellite_position1: Coordinates, satellite_position2: Coordinates
) -> float:
    """
    Calculate the distance between the satellites using the Haversine formula.

    Parameters:
    satellite_position1 (Coordinates): Latitude and longitude of the first satellite in degrees.
    satellite_position2 (Coordinates): Latitude and longitude of the second satellite in degrees.

    Raises:
    ValueError: If the latitude or longitude is an invalid number
                (i.e., abs(latitude) > 90 or abs(longitude) > 180, etc.)

    Returns:
    float: The distance between the two satellites in kilometers.
    """
    # Code...

    partial_haversine = (
        math.sin(latitude_difference / 2) ** 2
        + math.cos(lat1_rad)
        * math.cos(lat2_rad)
        * math.sin(longitude_difference / 2) ** 2
    )
    central_angle_radians = 2 * math.atan2(
        math.sqrt(partial_haversine), math.sqrt(1 - partial_haversine)
    )

    # Code...
```

En este código podemos notar algunas diferencias y similitudes entre **JSDoc**[10] y **docstring** que veremos en la siguiente tabla:

	JSDoc - JavaScript/TypeScript	**Docstring - Python**
Posición	Antes de la declaración de la función.	Al comienzo de la función.
Definición o información extra	Primeras líneas.	Primeras líneas.
Parámetros	Cada uno precedido por `@param`, con el tipo entre llaves, seguido por el nombre y una breve descripción.	Precedidos por el título `Parameters`, luego el nombre del parámetro, su tipo entre paréntesis y una pequeña descripción.
Errores o excepciones	Cada una precedida por `@throws`, seguido del tipo de excepción y una breve descripción.	Precedidos por el título `Raises`, luego el nombre de la excepción y una pequeña descripción.
Valor de retorno	Precedido por `@return`, seguido del tipo entre llaves y una pequeña descripción.	Precedidos por el título `Returns`, luego el tipo de retorno y una pequeña descripción.

Estas son solo las características principales de **JSDoc** y **docstring** y existen más detalles que pueden ser incluidos dependiendo del lenguaje y la implementación. Además, podemos encontrarnos con diversos formatos adicionales, como el utilizado en la librería `numpy` de Python, que tiene su propia convención para los *docstrings*. En general para Python es recomendable utilizar el formato propuesto por `PEP 257`[11] o con modificaciones similares.

3. Resumiendo lineamientos

- Es importante que si vamos a realizar comentarios informativos, lo hagamos de manera inteligente para que realmente aporten valor. Usaremos este tipo de comentarios para:
    - explicar el por qué detrás del código
    - añadir contexto que el código no sea capaz de explicar
    - explicar decisiones técnicas importantes

[10] `https://jsdoc.app/`
[11] `https://peps.python.org/pep-0257/`

    - explicar situaciones no triviales

- Los comentarios de marca son muy útiles al momento de trabajar en grandes equipos de desarrollo.
- Los comentarios de documentación interna, como los *docstrings* nos ayudan a agregar contexto a bloques de código como funciones o clases.
- Los *docstrings* deben ser redundantes con respecto al nombre de la función.

VI. Organización de un proyecto de software

1. La importancia de una estructura correcta

Hasta el momento hemos hablado de los componentes esenciales del código: funciones y variables, su tipado y cómo la documentación mediante comentarios puede ayudar a aportar claridad. También estudiamos cómo crear una buena estructura interna en el código, organizando las funciones de modo que sean bloques legibles, reutilizables y fáciles de mantener.

Pero, por más ordenadas que estén nuestras funciones, existe algo más grande que ellas y que también requiere atención: **la arquitectura del *software***. Entenderemos a la arquitectura del *software como la organización del sistema en partes lógicas que pueden ser comprendidas de forma independiente, junto con los elementos de software que las componen y las relaciones entre ellos. También abarca las propiedades externamente visibles de esos componentes y las relaciones entre ellos* [7, 1]. Esta organización no se limita simplemente a la disposición de archivos y carpetas, sino que implica decisiones sobre cómo estructurar y conectar el sistema para que sea comprensible, escalable y mantenible. Una estructura mal definida puede convertirse en un obstáculo a largo plazo. Por eso, en este capítulo, pondremos el foco en los aspectos claves para construir una estructura de proyecto sólida.

Es importante dejar en claro que no existe una única arquitectura válida. Cada tipo de proyecto tiene características que influyen en cómo deben organizarse. No es lo mismo una aplicación *backend* que una de análisis de datos, y aún dentro de la misma categoría, dos desarrolladores distintos pueden elegir utilizar estructuras diferentes que resulten igualmente efectivas. Es por ello que no propondremos una única arquitectura universal, sino que trabajaremos sobre conceptos y aspectos generales que pueden aplicarse a múltiples contextos.

Para acompañar el capítulo y hacerlo más didáctico, utilizaremos un proyecto real como hilo conductor. Estudiaremos su estructura y las decisiones detrás de ella. El objetivo no es tomarlo como modelo perfecto, sino como una oportunidad para comprender cómo organizar un proyecto.

A. El proyecto

Nuestro ejemplo práctico busca mostrar, a pequeña escala, un proyecto real y funcional, a la vez que sencillo para no perdernos en detalles innecesarios. Se trata de un *backend* escrito en Python que permite administrar productos y sus precios.

El sistema nos provee de las siguientes funcionalidades:

- **Añadir** nuevos productos.
- **Actualizar** los precios mediante un factor.
- **Listar** los productos con su valor en pesos argentinos.

- **Listar** los productos con su valor en dólares, a través de una interacción mediante una API externa.

El objetivo de este proyecto no es ser complejo, sino lo suficientemente completo para enseñar los conceptos de una arquitectura real.

Las tecnologías elegidas fueron las siguientes:

- **Poetry**[12]: herramienta para la gestión de dependencias y empaquetado.
- **FastAPI**[13]: *framework* web moderno y rápido para construir APIs.
- **Pydantic**[14]: biblioteca de validación y serialización de datos.
- **PonyORM**[15]: ORM que permite escribir consultas a la base de datos usando expresiones Python en lugar de SQL.
- **SQLAlchemy**[16]: ORM robusto y flexible que proporciona herramientas para mapear clases de Python a tablas de bases de datos relacionales.
- **SQLite**[17]: motor de base de datos ligero y embebido que guarda la información en un solo archivo, ideal para prototipos y aplicaciones pequeñas.

> ⚠ Un **ORM** es un *framework* que busca abstraer el uso de SQL. En ellos, uno escribe código siguiendo las reglas propias del mismo, este código luego es traducido a SQL y finalmente es ejecutado.

Notar que la aplicación utiliza dos ORMs. Por ahora no entraremos en detalles al respecto, ya que esto responde a motivos didácticos que se aclararán más adelante.

El código completo se encuentra disponible en el siguiente repositorio de GitHub (`https://github.com/FranZavalla/codigo-bonito-api-rest`). Allí se incluye un archivo `README.md` con todas las instrucciones necesarias para ejecutar la aplicación. De todos modos, a lo largo del capítulo se incluirán fragmentos representativos del código para guiar la lectura.

2. Una arquitectura simple basada en capas

Diseñar la arquitectura de un *software* es un tema recurrente y ampliamente estudiado. Entre algunas arquitecturas famosas nos podemos encontrar con **Domain-Driven Design (DDD)** por Eric Evans [4], **Onion Architecture** por Jeffrey Palermo [10] y **Clean Architecture** de Robert C. Martin [9]. Si bien existen diferencias entre ellas, todas estas propuestas comparten un denominador común:

[12] `https://python-poetry.org/`
[13] `https://fastapi.tiangolo.com/`
[14] `https://docs.pydantic.dev/`
[15] `https://ponyorm.org/`
[16] `https://www.sqlalchemy.org/`
[17] `https://www.sqlite.org/`

dividen al sistema en capas bien definidas, cada una con una responsabilidad y reglas claras.

Sin embargo, en la práctica, estas estructuras rara vez se implementan de forma estricta. Los proyectos reales suelen requerir adaptaciones o simplificaciones según el contexto. En ocasiones, el problema a resolver no es claro o evoluciona en el tiempo, por lo que se terminan mezclando distintos enfoques dentro de una misma arquitectura, a veces erróneos, dando como resultado una arquitectura vaga, la cual es difícil de mantener.

Con el objetivo de entender los beneficios de una arquitectura por capas sin caer en una complejidad excesiva, en esta sección proponemos una arquitectura simplificada basada en cuatro capas. La propuesta tiene como objetivo que el lector entienda la responsabilidad asignada a cada capa y los beneficios de estructurar el código de esta forma, para luego poder profundizar en arquitecturas más complejas que compartan los mismos fundamentos.

Las capas que componen a nuestra arquitectura simplificada son las siguientes:

- **Capa 0 - Definición de datos**: Esta capa define e implementa los datos con los que el sistema trabajará. Por ejemplo, si trabajamos con SQL crudo, esta capa contendrá los archivos SQL que defininen las tablas. Si nuestra aplicación no tiene datos persistentes, esta capa estará vacía.
- **Capa 1 - Acceso de datos**: Es la capa encargada de contener la lógica necesaria para acceder a los datos que utiliza la aplicación. En ella se pueden acceder tanto a datos propios (los definidos en la capa 0) como a datos provenientes de servicios o fuentes externas.
- **Capa 2 - Lógica de la aplicación**: Contiene el código que implementa las funcionalidades propias del sistema.
- **Capa 3 - Interfaz de usuario**: Funciona como conexión entre el sistema y el mundo exterior, ya sean otros sistemas o usuarios que lo utilizan.

Para reforzar estas ideas y favorecer el aspecto didáctico, en el código de nuestro proyecto encontraremos explícitamente las 4 capas representadas por carpetas. Cada carpeta estará nombrada con el número y el nombre de la capa. Por ejemplo, la carpeta asociada a la primer capa será `layer_0_db_definition`.

Numerar las capas nos permite expresar de forma sencilla un lineamiento que debería respetarse en cualquier arquitectura por capas:

> **Lineamiento:** En una **arquitectura por capas**, un elemento de la capa **n** solo puede interactuar con elementos de la capa **n** o inferiores, pero jamás con capas superiores.

Es gracias a este lineamiento que las arquitecturas por capas promueven aspectos como el desacople de componentes. Además, si la implementación está bien realizada, se obtiene una propiedad muy valiosa y deseable: la posibilidad

de tener **sistemas parciales funcionales**: es decir, si tomamos el código de la capa **n** junto con todas sus capas inferiores, deberíamos tener un sistema completamente funcional:

- Con las **capas 0** y **1**, podemos acceder y manipular datos.
- Al agregar la **capa 2**, obtenemos la implementación de la lógica específica de nuestro sistema sobre los datos. Con esto, deberíamos poder ejecutar cualquier funcionalidad del sistema en forma programática. Por ejemplo, en Python, debería ser posible iniciar un intérprete y ejecutar cualquier funcionalidad del sistema.
- Finalmente, al incluir la **capa 3**, contemplamos todo el sistema, habilitando la posibilidad de que interaccione con el usuario final.

Es importante remarcar que en nuestro modelo, la capa 2 es muy general y por lo tanto con pocos detalles, restricciones y/o lineamientos. Pero en proyectos grandes, esta capa es realmente compleja dado que contiene código con distintas particularidades:

- **Código que implementa lógica específica de negocio**. Por ejemplo, en nuestro proyecto es el único tipo de código que existe en la capa 2 y será el encargado de implementar la funcionalidad de mostrar los precios de los productos en dólares. Otro ejemplo de este tipo de código podría ser procesar datos para generar un reporte específico.
- **Código que implementa procesos más generales o auxiliares**. Por ejemplo, funciones que puedan recibir datos, subirlos a un servicio en la nube y enviar un correo para poder acceder a esos datos. Este mismo código se podría usar para guardar el resultado de generar cualquier reporte. Este tipo de módulos se los suele denominar servicios.
- Algunos de estos procesos no necesitan una respuesta inmediata, entonces suelen ejecutarse en segundo plano. Para ello, es común implementar *jobs*, colas de mensajes [18] y procesos encargados de su ejecución (*workers*).
- Si fuera necesario realizar estas tareas de forma periódica, también podríamos incluir un **planificador de tareas**.

⚠ Toda la organización e implementación de estas funcionalidades escapan de nuestra arquitectura simplificada y no están presentes en nuestro proyecto guía.

Por último, vale mencionar la existencia de una capa **transversal**, la cual contiene funcionalidades que no pertenecen a una capa específica, sino que pueden

[18] `https://aws.amazon.com/es/message-queue/`

ser utilizadas por todas ellas. Como su nombre lo indica, esta capa no se ubica junto a una capa en particular, sino que ofrece servicios auxiliares a todo el sistema. Sus módulos suelen ser genéricos y reutilizables, facilitando su traslado hacia otros proyectos sin mucha modificación.

Un ejemplo común en esta capa es la implementación de un componente de *logging*, que permite registrar eventos como errores, advertencias o información relevante para el monitoreo del sistema. En nuestro proyecto de ejemplo, buscamos mantener la estructura lo más simple posible, por lo que no incluiremos código perteneciente a esta capa.

3. Organizando el código dentro de cada capa

Existen diferentes formas de implementar el código en una arquitectura por capas. Lo más importante no es el estilo exacto de la implementación, sino **respetar los límites de responsabilidad y alcance de cada capa**. Es decir que mientras cada capa se mantenga enfocada en su función dentro del sistema, el diseño será válido.

Una primera buena aproximación puede basarse en el uso de **funciones**. Las funciones son herramientas claras y concisas para resolver problemas bien delimitados. Lenguajes como `C`, que sólo conocen de funciones y procedimientos, han sido utilizados hasta la actualidad para crear sistemas complejos y completamente funcionales.

Sin embargo, a medida que un sistema crece y con él, el número de funciones involucradas, surgen algunas limitaciones. Cuando las funciones están dispersas, se dificulta saber que es lo ya está implementado y que no, lo que puede derivar en la duplicación de lógica por simple desconocimiento. Esto hace que el código se vuelva propenso a errores y reduce la reutilización del mismo.

Para estos escenarios, es que podemos recurrir a la **programación orientada a objetos**, que nos ofrece una solución más robusta. Las clases organizan el código de forma más concreta. Dentro de este paradigma, una herramienta útil son las **clases abstractas** las cuales permiten definir interfaces claras que favorecen al desacople de las implementaciones. En otras palabras, se explicita el *qué* hace cada clase y no el *cómo* lo hace. De esta forma se puede reemplazar una implementación por otra sin afectar al resto del sistema. Esta práctica es conocida como **programar contra interfaces** [8], y promueve la mantenibilidad y escalabilidad del sistema.

A. Tipos de clases

Al implementar un sistema con objetos, es importante entender que existen distintos tipos de clases, las cuales definen objetos con distintas particularidades. En nuestro proyecto vamos a encontrar tres tipos de clases:

1. **Clase de datos**: son clases que contienen datos específicos, sin lógica asociada. Estas clases se usarán para definir la información que espera y devuelve un servicio o módulo. Utilizando este tipo de clases se desacopla la interacción entre los mismos.

En nuestro proyecto, ejemplo de este tipo de clases serán `CreateProductData` y `ProductData`. La primera tendrá los datos necesarios para crear un producto: el nombre y el precio. El segundo tendrá la información de un producto en nuestra base de datos: id, nombre y precio. Notemos que id es un valor único que se define a nivel base de datos, por lo tanto no es un dato que se necesite al momento de crear un producto.

Python es un lenguaje de tipado dinámico, entonces no es directo definir una clase a 'datos específicos', por esta razón es que utilizamos el estándar de facto para esta tarea: **Pydantic**. Pydantic es un paquete que ejecuta la validación de tipos en tiempo de ejecución, además de proveer otras funcionalidades extras para el manejo de datos. Por otro lado, no queremos olvidarnos de que no todos los lenguajes necesitan de este tipo de clases de datos, lenguajes como TypeScript ya poseen constructores predefinidos para esta tarea, como `type` e `interface`, cada uno con sus particularidades.

2. **Tipos abstractos de datos (TAD)** [19]: son clases que además de contener datos específicos poseen un conjunto de operaciones que se pueden realizar sobre los datos o a partir de los mismos. En general son abstracciones de entidades del mundo real y, en contraposición a las clases de datos antes mencionadas, este tipo de clases son para uso interno de un servicio o módulo. El conjunto de operaciones que un TAD realiza está fuertemente ligado al uso interno que se le da.

   En nuestro proyecto, una clase de este tipo es `Product(db.Entity)` en el archivo `models_ponyorm.py`. Esta clase se crea dentro del *framework* PonyORM. La abstracción de los productos en la base de datos contiene información similar a la que encontrábamos en `CreateProductData`, pero además contiene datos internos que pertenecen a PonyORM y provee métodos para manipular tanto la tabla que contiene los datos, así como un dato específico (crear entradas nuevas, traer un dato particular, modificarlo y guardarlo, etc). Observemos aquí la importancia de tener distintas estructuras. Las capas 0 y 1 (definición y acceso a datos) entenderán de `Product(db.Entity)` pero se comunicarán con la capa 2 (lógica de aplicación) usando `CreateProductData` y `ProductData`, de esta forma, la capa 2 nunca sabrá detalles sobre cómo se implementa la persistencia de datos ni cómo se manipulan internamente. En consecuencia, la capa 2 estará totalmente desacoplada de esta implementación.

3. **Clases de tipo funcionalidad**: son clases que encapsulan operaciones o precedimientos útiles para el sistema. Estas operaciones suelen construirse a partir de otras operaciones 'más simples' provistas por otras clases del sistema. A menudo, estas clases hacen uso de otras de su mismo tipo para cumplir su propósito. En estos casos, una buena práctica es utilizar

[19] `https://es.wikipedia.org/wiki/Tipo_de_dato_abstracto`

el patrón de diseño conocido como **inyección de dependencias** [20]. Este patrón se basa en pasar instancias de clases auxiliares como argumento al momento de instanciar la clase principal. Cuando hacemos esto, estamos promoviendo el desacople de componentes

En nuestro proyecto encontramos varios ejemplos de clases de tipo funcionalidad: `ProductRepository` es una clase que se implementa en la capa 1 y se utiliza para interactuar con la base de datos. En esta misma capa también encontramos la clase `DollarConnector`, la cual interactúa con la API externa que nos provee del precio del dólar en tiempo real. Como último ejemplo, mencionaremos la clase `ProductWithDollarBluePrices`. Esta clase implementa una funcionalidad que informa el valor de los productos de la base de datos con su valor en dólares. Para ello, hace uso de la inyección de dependencias: en su inicialización se recibirán instancias de las clases previamente nombradas. La instancia de `ProductRepository` será utilizada para acceder a los datos de los productos, mientras que la instancia de `DollarConnector` será utilizada para obtener los precios del dólar.

Comprender y aprovechar correctamente la programación orientada a objetos es una tarea compleja, ya que requiere tiempo y práctica. Pero una vez internalizada, la estructura del código mejora significativamente, afectando principalmente a la mantenibilidad y escalabilidad.

4. Capas del sistema

A. Capa 0: Definición de datos

La **definición de datos** corresponde al primer eslabón en la arquitectura de cualquier sistema de *software*. Su propósito es definir los elementos fundamentales con los que trabajará el sistema: los **datos persistentes**. Esta tarea no es trivial, ya que implica decisiones importantes. Distintos objetivos, introducen distintos desafíos y requerimientos. No es lo mismo diseñar un sistema que debe manejar:

- datos asociados a entidades relacionadas (usuarios, amigos, publicaciones),
- series temporales (precios de activos actualizados cada segundo),
- grandes volúmenes de imágenes,
- videos,
- una combinación de todos estos tipos de datos.

[20] `https://es.wikipedia.org/wiki/Inyección_de_dependencias`

En esta capa no se realiza lógica específica del sistema ni procesamiento de datos. Su función es definir las estructuras, tipos y restricciones de los datos para que las demás capas puedan trabajar con ellas de forma consistente y confiable. Aquí también se suelen especificar los **componentes físicos** encargados de almacenar los datos.

Este último punto no es menor. Supongamos que estamos implementando una red social que permite subir imágenes. En los inicios, la cantidad de usuario será poca, entonces podría bastar con guardar las imágenes dentro del mismo servidor que ejecuta la aplicación. Sin embargo, si el sistema crece y comienza a recibir millones de usuario que suben imágenes constantemente, un único disco con capacidad física limitada no será suficiente.

¿Qué podemos encontrar en esta capa?

- **Modelos de almacenamiento**: Tablas (SQL), colecciones (MongoDB), estructuras jerárquicas (XML/JSON), datos en archivos planos, etc.
- **Inicialización de estructuras persistentes**: Código para crear archivos, bases de datos, carpetas, etc.
- ***Scripts* de migración o carga inicial**: Código que modifica la base de datos, inserta información de prueba o estados iniciales del sistema.
- **Definiciones de tipos o interfaces**

Ejemplos en nuestro proyecto

En nuestro caso, la capa 0 está contenida en la carpeta `/layer_0_db_definition`.

```
backend-products/
└── layer_0_db_definition/
    ├── database_sqlalchemy.py
    ├── models_sqlalchemy.py
    ├── database_ponyorm.py
    └── models_ponyorm.py
```

Analicemos los siguientes archivos:

- `database_sqlalchemy.py` contiene la función que inicializa la base de datos con SQLAlchemy, `init_sqlalchemy()` y la función que devuelve sesiones para trabajar con ella, `get_database()`. En nuestro proyecto, configuramos a SQLAlchemy para usar una instancia local de SQLite. Es decir, nuestro componente físico será nuestro propio disco duro y los datos se guardarán usando un único archivo binario. Podemos hacer estas elecciones dado que estamos desarrollando un proyecto de ejemplo, pero ambas decisiones son malas si tenemos en cuenta el desempeño y escalabilidad.

```python
def init_sqlalchemy():
  Base.metadata.create_all(bind=engine)

# Versión simplificada
def get_database():
  return SessionLocal()
```

- En `models_sqlalchemy.py` definimos la única tabla que va a utilizar nuestro sistema (`product`) con sus columnas y restricciones. Cuando lee este archivo, SQLAlchemy se conecta a la base de datos. Luego, si no encuentra la tabla, la crea con las restricciones definidas.

```python
class Product(Base):
  __tablename__ = "product"

  id: Mapped[int] = mapped_column(primary_key=True, autoincrement=True)
  name: Mapped[str] = mapped_column(nullable=False)
  price: Mapped[float] = mapped_column(nullable=False)
```

Además de estos archivos, también se incluyen `database_ponyorm.py` y `models_ponyorm.py`. Estos archivos son análogos a los que acabamos de presentar, pero implementados en PonyORM. La idea es mostrar más adelante como la definición de los datos puede cambiar sin que esto afecte a la lógica de la aplicación (capa 2) gracias a las abstracciones provistas en la capa de acceso de datos (capa 1).

B. Capa 1: Acceso de datos

El propósito de la capa de **acceso a datos** es **abstraer** las acciones de obtener, almacenar, modificar y/o eliminar información, ya sea accediendo directamente a la capa inferior, o bien interactuando con fuentes externas, como por ejemplo APIs de terceros.

Esta capa depende totalmente de la capa 0. Por lo tanto, cualquier cambio en la forma en que se definen los datos implicará ajustes en esta capa para mantener la coherencia.

¿Qué podemos encontrar en esta capa?

Solemos encontrar en esta capa componentes como:

- **Repositorios**: Abstraen el acceso a base de datos, permitiendo a capas superiores obtener o modificar información sin escribir consultas ni código SQL.
- **Conectores con APIs**: Encapsulan lógica de conexión con APIs, ya sea de terceros o propias.
- **Abstracciones de almacenamiento**: Se encargan de proveer funciones que escriben/leen archivos, manejan caché, entre otras.

Ejemplo en nuestro proyecto

La capa 1 está contenida en la carpeta `/layer_1_data_access`. Allí distinguimos dos componentes principales: los repositorios (`repositories`) que gestionan el acceso a la tabla `products` en la base de datos y los conectores (`connectors`), encargados de interactuar con la API externa del dólar.

```
backend-products/
└── layer_1_data_access/
    ├── connectors/
    │   ├── dollar_connector.py
    │   └── bluelytics_connector.py
    └── repositories/
        ├── product_abstract.py
        ├── product_pony.py
        └── product_sqlalchemy.py
```

Repositorios

Dentro de `/repositories`, encontramos el archivo `product_abstract.py`, el cual contiene dos clases de datos `CreateProductData` y `ProductData` y la clase abstracta `AbstractProductRepository`. Las clases de datos, como ya dijimos antes, definen los datos con los cuales uno puede comunicarse con el repositorio para acceder a los productos. Por otro lado, `AbstractProductRepository` define los métodos que debe proveer un repositorio de productos 'válido' para nuestro sistema sin especificar nada con respecto a la implementación de los mismos.

```python
class ProductData(BaseModel):
  id: int
  name: str
  price: float

  model_config = {"from_attributes": True}

class CreateProductData(BaseModel):
  name: str
  price: float

class AbstractProductRepository(ABC):
  @abstractmethod
  def get_all(self) -> List[ProductData]:
    pass

  @abstractmethod
  def get_by_id(self, product_id: int) -> ProductData:
    pass

  @abstractmethod
  def create(self, product: CreateProductData) -> ProductData:
    pass
```

Los archivos `product_pony.py` y `product_sqlalchemy.py` proveen implementaciones de `AbstractProductRepository`. Cabe destacar que no tenemos

nada conceptualmente relevante que decir de estos archivos, en ellos solo encontramos implementaciones específicas. Lo importante ya ha sido definido por la clase abstracta.

Conectores

En la carpeta `/connectors`, dentro del archivo `dollar_connector.py` definimos la clase abstracta `DollarConnector`:

```
class DollarConnector(ABC):
  @abstractmethod
  def get_price(self) -> float:
    """
    Retrieves the current price of the dollar.

    Returns:
      float: The current price of the dollar.
    """
    pass
```

Esta clase establece que toda implementación concreta debe incluir un método `get_price` que retorna el precio del dólar en el momento actual.

En el archivo `bluelytics_connector.py` tenemos una implementación de esta clase: `BluelyticsConncector`.

```
class ExchangeRate(BaseModel):
  value_avg: float
  value_sell: float
  value_buy: float

class BluelyticsResponse(BaseModel):
  oficial: ExchangeRate
  blue: ExchangeRate
  oficial_euro: ExchangeRate
  blue_euro: ExchangeRate
  last_update: datetime


class BluelyticsConnector(DollarConnector):
  def __init__(self, endpoint=BLUELYTICS_API_URL):
    self.endpoint = endpoint

  def get_price(self) -> float:
    price_response = requests.get(self.endpoint)
    price_response.raise_for_status()

    json_data = price_response.json()
    try:
      bluelytics_parsed = BluelyticsResponse.model_validate(json_data)
    except Exception as e:
      raise ValueError(f"Error parsing Bluelytics response: {e}")

    return bluelytics_parsed.blue.value_avg
```

`BluelyticsResponse` corresponde a una clase de datos que utilizamos para validar la respuesta recibida desde la API externa. Esto es muy importante porque

al tratarse de un servicio de terceros, sus respuestas podían cambiar sin previo aviso.

Por otro lado, notemos que la implementación actual define el precio del dólar como el promedio entre el valor de compra y el de venta del *dólar blue*. Si en el futuro se requiriese cambiar esto, por ejemplo, usar únicamente el valor de compra o de venta, o incluso cambiar del dólar blue al dólar oficial, bastaría con modificar la implementación en esta clase para que ese cambio impacte en todo el sistema.

C. Capa 2: Lógica de aplicación

Esta capa representa el **núcleo** de nuestra aplicación. Aquí encontramos lógica que define a nuestro sistema. Como ya mencionamos antes, no iremos en profundidad sobre los lineamientos de esa capa, porque la misma puede ser muy compleja y sólo nos limitaremos a contar qué es lo que encontramos en nuestro ejemplo.

¿Qué podemos encontrar en esta capa?

No contamos con una receta fija para esta capa. La lógica de la aplicación varía fuertemente de un proyecto a otro. Sin embargo podemos nombrar algunos elementos comunes que suelen aparecer en esta capa:

- **Procesadores o transformadores de datos**: convierten datos en estructuras útiles para el usuario o la misma aplicación.
- **Manejadores de *endpoints***: encargados de recibir datos y solicitudes externas. Realizan una serie de operaciones y entregan una respuesta acorde.
- **Validaciones**: que no pertenecen a la definición de datos, más bien surgen de reglas específicas de esta capa.
- **Cálculos específicos**: algoritmos que responden a las necesidades de la aplicación.
- **Clases de funcionalidad**: cómo las mencionadas previamente.

Ejemplo en nuestro proyecto

Esta capa la encontramos en la carpeta `/layer_2_logic` de nuestro proyecto:

```
backend-products/
└── layer_2_logic/
    ├── product_with_dollar_blue.py
    └── factory.py
```

Dentro de `product_with_dollar_blue.py` se encuentran, por un lado, la clase de datos `ProductDataWithUSDPrice` y por otro, la clase de tipo funcionalidad `ProductWithDollarBluePrices`, que se encarga de recuperar productos desde la base de datos y agregarles un nuevo atributo: su precio en dólares.

```python
class ProductDataWithUSDPrice(ProductData):
  usd_price: float

class ProductWithDollarBluePrices:
  def __init__(
    self,
    product_repository: AbstractProductRepository,
    dollar_blue_connector: DollarConnector,
  ):
    self.product_repository = product_repository
    self.dollar_blue_connector = dollar_blue_connector

  def get_product(self, product_id: int) -> ProductResponseWithUSDPrice
    :
    # code ...
    return ProductResponseWithUSDPrice(
      # code ...
    )

  def get_products(self) -> List[ProductResponseWithUSDPrice]:
    # code ...
    return [
      # code ...
    ]
```

Las instancias de la clase `ProductWithDollarBluePrices` se construyen a partir de dos dependencias: un **repositorio de productos** y un **conector para obtener los precios del dólar**. Ambos provienen de la capa de acceso a datos y son provistos externamente como argumentos del constructor. De esta forma `ProductWithDollarBluePrices` accede a los productos y al precio del dólar sin tener noción de las implementaciones subyacentes.

El otro archivo en esta capa es `factory.py`. Este archivo es una **fábrica** pues implementa funcionalidades que *crean instancias de clases utilizadas en el proyecto*, en función de la configuración o del contexto:

```python
def select_product_repository(
  db: Optional[Session] = None,
) -> AbstractProductRepository:
  """
  Returns the appropriate product repository based on the configuration
    settings.

  Args:
    db (Session, optional): The database session to use. Defaults to
    None.

  Returns:
    Union[SQLARepo, PonyRepo]: An instance of the appropriate product
    repository.
  """
  ...

def get_product_repository() -> AbstractProductRepository:
  with get_database() as db:
    return select_product_repository(db)
```

```
18
19 def get_dollar_blue_repository() -> ProductWithDollarBluePrices:
20   product_repository = get_product_repository()
21   dollar_blue_connector = BluelyticsConnector()
22   return ProductWithDollarBluePrices(product_repository,
      dollar_blue_connector)
```

La función `get_product_repository` utiliza `select_product_repository` para devolver, dependiendo la configuración del proyecto, una instancia de un repositorio de productos implementado en SQLAlchemy o en PonyORM. Este ejemplo es muy simple, pero muestra el poder de trabajar con abstracciones para acceder a los datos: dado que ambos repositorios implementan la misma interfaz, podemos hacer uso de ellos indistintamente.

En este caso, ambos ORMs son tecnologías similares, pero podríamos estar utilizando tecnologías diferentes para almacenar los datos, y aún así abstraer esas diferencias mediante una interfaz común como `AbstractProductRepository`.

El criterio de selección también es muy simple: una variable de configuración externa. Sin embargo, en proyectos reales, podríamos basarnos en criterios mucho más complejos, como por ejemplo elegir una tecnología con alto rendimiento para usuarios *premium*, y otra más económica para el resto de los usuarios.

D. Capa 3: Interfaz de la aplicación

La última capa de nuestra arquitectura corresponde a la **interfaz de aplicación**: esta capa implementa la interfaz accesible desde el exterior para comunicarse con nuestro sistema. Por lo tanto, la función de esta capa es **recibir solicitudes externas** y **devolver resultados** generados por la lógica de la aplicación.

Esta capa incluye la lógica necesaria para transformar las solicitudes externas al formato utilizado por la lógica de la aplicación. De forma análoga, todo resultado generado por la lógica de la aplicación debe ser transformado a un formato adecuado para que sea recibido por el usuario final. Dependiendo el tipo de aplicación, en esta capa se pueden implementar otras funcionalidades, por ejemplo la autenticación de usuarios, el chequeo de permisos y/o el manejo de errores.

¿Qué podemos encontrar en esta capa?

Algunas interfaces frecuentes que encontramos en esta capa son:

- **API Web (REST, GraphQL, ...)**: comunes en *backends*, permiten que usuarios u otras aplicaciones interactúen con nuestro sistema a través de solicitudes HTTP.
- **Páginas web**: aplicaciones mostradas al usuario mediante un navegador web.
- **Interfaces gráficas (GUI)**: presentes en aplicaciones de escritorio o móviles.
- **Líneas de comandos (CLI)**: utilizadas en herramientas o *scripts* de automatización

- **Gráficos**: comunes en análisis de datos, donde los resultados se presentan de forma visual.

Todos estos mecanismos comparten una característica: **hacen visible o utilizable la funcionalidad principal del sistema**.

Ejemplo en nuestro proyecto

Esta última capa la encontramos en la carpeta `/layer_3_api`, que contiene los archivos encargados de definir los *endpoints* HTTP que expone la funcionalidad del sistema. La implementación de esta capa se construye con el *framework* `FastAPI`, el cual nos simplifica tareas que en otros entornos serían repetitivas al momento de crear nuestra API.

```
backend-products/
├── layer_3_interface/
│   ├── products.py
│   └── products_with_usd_prices.py
└── main.py
```

Dentro de la carpeta `/layer_3_interface` tenemos dos archivos, `products.py`, donde definiremos los *endpoints* asociados a los productos con los precios en pesos, y `products_with_usd_prices.py` donde se definen los *endpoints* asociados a los productos con los precios en dólares. En estos archivos se usan funciones para definir los puntos de acceso a la aplicación. Por ejemplo, en `products.py` encontramos la función `get_product`:

```python
@router.get("/product/{product_id}")
def get_product(
  product_id: int,
  product_repository: AbstractProductRepository = Depends(
    get_product_repository),
):
  try:
    product = product_repository.get_by_id(product_id)
    json_product = product.model_dump()
    return JSONResponse(status_code=200, content=json_product)
  except ValueError:
    return JSONResponse(status_code=404, content={"detail": "Product
     not found"})
  except Exception:
    return JSONResponse(
      status_code=500, content={"detail": "Internal server error"}
    )
```

En este fragmento de código utilizamos el decorador `@router.get(...)` para definir un *endpoint* GET en la ruta `/product/product_id`. Al colocar el decorador junto a la función `get_product`, estamos asociando su funcionalidad a dicha ruta. El segmento `product_id` dentro de la ruta representa un *path parameter* [21], es decir, un valor proporcionado por el usuario en la URL. En la signatura

[21] https://fastapi.tiangolo.com/tutorial/path-params/

de la función, este parámetro se declara como un entero (`product_id: int`), indicando que se espera un valor numérico que será utilizado para buscar un producto en la base de datos.

Por otro lado, FastAPI permite definir dependencias del *endpoint* directamente en la definición de la función. En este caso, `product_repository` es una instancia inyectada mediante `Depends(get_product_repository)`. Esta abstracción permite desacoplar la obtención del repositorio de la lógica del núcleo del *endpoint*, manteniéndola simple y enfocada en su propósito: recuperar un producto por su id.

En la lógica de la función, se intenta obtener el producto llamando a `product_repository.get_by_id(product_id)`. Si la búsqueda es exitosa, el resultado se convierte a un diccionario mediante el método `model_dump()` y luego se completa la respuesta en formato JSON con código HTTP 200, indicando éxito.

En el caso de que algo no ocurriese como lo esperamos, el *endpoint* maneja explícitamente dos tipos de errores. En primer lugar, si el producto no existe, se lanza una excepción `ValueError` en el repositorio y se devuelve una respuesta JSON con código de error 404 indicando lo sucedido. Por otro lado, si se produce cualquier otra excepción durante la ejecución (por ejemplo, una base de datos no disponible o mal configurada), se devuelve una respuesta con código 500. Este manejo genérico evita exponer detalles internos del sistema que podrían brindar información de utilidad para un atacante malicioso.

Cabe destacar que FastAPI incluye validaciones automáticas de los parámetros definidos en la ruta. Si bien esto no se refleja directamente en el cuerpo de la función, cuando el servidor recibe una solicitud con un valor no numérico en la URL (por ejemplo, una solicitud a la ruta `/product/no_soy_un_numero`), FastAPI responderá automáticamente con un error informando que el valor proporcionado no es válido, dado que se esperaba un número entero.

Todo lo desarrollado hasta ahora está fuertemente ligado al *framework* FastAPI. Esto fue intencional, ya que nos permitió ejemplificar concretamente los siguientes cuatro momentos a la hora de implementar un acceso a nuestra aplicación:

- **Validación de la solicitud**. En esta primera etapa, se verifica que quién realiza la solicitud envié datos válidos. En nuestro ejemplo, la validación esta a cargo del propio *framework* que se asegura de que el `product_id` sea un entero.

- **Instancias e importaciones**. Aquí se preparan los recursos necesarios para manejar la solicitud. En nuestro caso, corresponde a la instanciación automática del repositorio mediante `get_product_repository`.

- **Ejecución**. Esta es la etapa central, donde se lleva a cabo la lógica adecuada para cumplir con la solicitud. En el ejemplo, simplemente encontramos la llamada al método `get_by_id` del repositorio de productos.

- **Retorno del resultado**. Finalmente, se devuelve una respuesta al cliente en el formato adecuado. Si la solicitud fue exitosa, entonces los datos del

producto son devueltos en el formato JSON. Si ocurrió un error, se informa mediante un mensaje y un código HTTP adecuado. En nuestro caso, se manejan explícitamente errores esperables, como un producto no existente y errores genéricos.

Notemos que estos mismos cuatro momentos están replicados en todo *endpoint* de nuestra aplicación. En particular observamos que ocurre con la ruta que se encarga de devolver todos los productos de la base de datos con los precios en dólares. Esta función es `get_products_with_usd_price` y la podemos encontrar en el archivo `products_with_usd_prices.py`:

```
@router.get("/products_with_usd_prices/products_with_usd_prices/")
def get_products_with_usd_price(
  dollar_blue_repository: ProductWithDollarBluePrices = Depends(
    get_dollar_blue_repository
  ),
):
  try:
    products = dollar_blue_repository.get_products()
    json_products = [product.model_dump() for product in products]
    return JSONResponse(status_code=200, content=json_products)
  except Exception:
    return JSONResponse(
      status_code=500, content={"detail": "Internal server error"}
    )
```

Veamos los cuatro momentos:

- **Validación de la solicitud**. En este caso no hay nada que validar, la solicitud no depende de ningún dato externo, siempre se devuelven todos los productos.
- **Instanciaciones e importaciones**. Se instancia `dollar_blue_repository` mediante `get_dollar_blue_repository`.
- **Ejecución**. Utilizamos el método `get_products` de `dollar_blue_repository` para obtener todos los productos con los precios en dólares.
- **Retorno del resultado**. En caso de éxito, se devuelve la lista de los productos y si ocurre un error inesperado, un error genérico.

Por último, nos encontramos con el archivo `main.py` que si bien no se encuentra dentro de la carpeta `/layer_3_interface`, también forma parte de esta capa. Allí se inicializa la instancia principal de FastAPI y la conexión a la base de datos. Actúa como punto de entrada real de la aplicación y por lo tanto forma parte de la **interacción con el usuario**.

```
def init_db():
  print("Initializing database...")
  if settings.ORM == "sqlalchemy":
    init_sqlalchemy()
  else:
    init_pony()
```

```
7
8  @asynccontextmanager
9  async def lifespan(app: FastAPI):
10   init_db()
11   yield
12
13 app = FastAPI(lifespan=lifespan)
```

5. El desafío de una buena abstracción

Esta capítulo fue orientado para enseñar a organizar el código mediante abstracciones y encapsulamiento de tareas. Creemos que esta es la forma correcta de escribir código y estructurar un sistema. Sin embargo, nos toca reconocer que este enfoque no es perfecto y mucho menos está libre de problemas.

Uno de los primeros desafíos es que crear abstracciones correctas no es fácil. Aún con mucha experiencia, es común que algunas partes del sistema no sean óptimas o estén mal organizadas. Además, alcanzar una organización perfecta puede requerir un nivel de abstracción tan alto que los beneficios obtenidos no justifican el esfuerzo de implementación.

Otro punto a tener en cuenta es que una organización excesivamente modularizada puede afectar la compresión del código. Cuando una funcionalidad está dividida en múltiples archivos, clases y capas, el flujo de ejecución se vuelve difícil de seguir, especialmente para aquellos desarrolladores no familiarizados con el sistema. Entonces, un código sobremodularizado puede llevar a un código 'correcto' pero ilegible.

Algo peor que no encapsular tareas, es intentar hacerlo y hacerlo mal. En este capítulo mostramos un ejemplo sencillo con buenas propiedades, pero no profundizamos en cómo llegar a ella. Esta es una tarea compleja que requiere experiencia, iteración y comprensión del sistema.

Es importante aceptar que en las primeras etapas de un proyecto es normal refactorizar el mismo. Por lo tanto no hay que desanimarse si, meses después de haber implementado una funcionalidad, sentimos que su estructura puede mejorar. Esto es parte del proceso de desarrollar, principalmente en las funciones núcleo de nuestro sistema.

También es importante hacer una mención de los tiempos de ejecución. Por ejemplo, Python no es un lenguaje de programación que brille por su desempeño, en sistemas grandes implementar tantas capas lógicas puede afectar al rendimiento del sistema. Un ejemplo interesante de este dilema, se desarrolla en el artículo Beyond Clean Code [15], donde se analiza en profundidad cómo la búsqueda de una organización modular y orientada a objetos puede, en ciertos contextos, perjudicar significativamente el rendimiento. El mensaje central es que una organización basada en capas y abstracciones no es siempre la mejor opción: depende mucho del dominio del problema y de las operaciones que se realizan.

VII. Testing

1. Haciendo pruebas sobre nuestro código

Cuando escribimos código, una parte importante del trabajo es asegurarnos de que funcione tal como esperamos. Una forma completa de abordar este problema es a través de la **verificación de programas**. La verificación de programas busca *comprobar matemáticamente la corrección de un programa con respecto a su especificación*. Existen herramientas diseñadas específicamente para esto, pero no es el enfoque que tomaremos en este capítulo. Lo que buscamos, es algo mucho más accesible y práctico: **realizar pruebas sobre nuestro código**.

La diferencia entre verificar y probar puede ser sutil en un comienzo, pero en la práctica están distanciados. Como ya dijimos, la verificación corresponde a un enfoque mucho más formal. Mientras que las **pruebas** buscan *confirmar con algún grado de confianza que el código se comporte como esperaríamos*. Es importante que durante la etapa de desarrollo de un sistema de *software* dediquemos parte del tiempo a crear estas pruebas a las que llamamos tests.

Un **test** no es más que un fragmento de código que ejecuta de forma automática una función, módulo o flujo completo de nuestro sistema con el objetivo de comprobar que el resultado sea el esperado. Estas comprobaciones pueden ir desde algo tan simple como comprobar que un cálculo matemático devuelve el valor correcto, hasta situaciones más complejas como simular el comportamiento de un usuario en una aplicación completa. Al proceso de escribir y ejecutar estas pruebas lo llamamos **testing**.

Es importante comprender que el testing no nos garantiza que el programa esté completamente libre de errores. Que un conjunto de pruebas pase exitosamente sólo garantiza que en esos casos específicos el sistema funciona como se esperaba. Pero siempre puede existir la posibilidad de casos no contemplados, como los que ocurren con ciertas combinaciones de datos o condiciones específicas que no fueron cubiertas. Es fundamental que, como desarrolladores, contemplemos esta posibilidad e intentemos cubrir la mayor cantidad de casos posibles, pensando que esos casos excepcionales siempre pueden ocurrir.

A. Beneficios del testing

Realizar pruebas nos permite detectar errores de forma temprana y en entornos controlados. Gracias a esto, no solo reducimos la cantidad de fallos en producción, sino que además mejoramos la calidad del código. En muchos casos, es posible diseñar y escribir los tests sin mirar directamente la implementación en el cuerpo del código, simplemente utilizamos su interfaz o especificación, este enfoque es conocido como *caja negra* [22] y es muy utilizado por equipos dedicados exclusivamente al testing. Esta etapa también es una oportunidad para revisar el código ya escrito, y muchas veces nos lleva a notar funciones demasiado ex-

[22] En inglés: `https://en.wikipedia.org/wiki/Black-box_testing`

tensas, nombres poco claros o flujos muy complejos. Cuando escribimos tests, también repensamos el código.

En lenguajes interpretados, como Python o JavaScript, el testing cumple una función adicional: nos ayuda a identificar errores de sintaxis o de tipado que, de otro modo, podrían permanecer ocultos hasta el momento de su ejecución en producción. Esto se debe a que, a diferencia de los lenguajes compilados (que nos permiten detectar errores antes de ejecutar el código), en los lenguajes interpretados el código sólo es analizado cuando está corriendo. Por eso, los tests, incluso los más simples, fuerzan la interpretación del código y permiten que se lancen los errores adecuados en caso de que estos existan.

En definitiva, además de ayudar a escribir mejor y detectar errores rápidamente, el testing aporta beneficios concretos:

- **Facilitar los cambios en el código**. Cuando tenemos un conjunto de pruebas confiables, podemos modificar el sistema con tranquilidad. Si alguna parte del código se rompe, los tests deberían hacernos notar estos errores.
- **Documentar el comportamiento esperado**. Los tests son una forma de documentar el código de manera no oficial, al menos para un conjunto finito de casos. Los desarrolladores deberían ser capaces de entender partes del sistema observando simplemente los tests.
- **Aumentar la confianza**. Si los tests implementados son exitosos, la confianza en el sistema crece y la probabilidad de que ocurran errores disminuye. De todas maneras, como explicamos previamente, no hay que tener fe ciega sobre las pruebas, siempre es posible que existan caminos no cubiertos o situaciones no contempladas.

Además, el testing nos ofrece una retroalimentación inmediata sobre lo que estamos construyendo. Saber que una parte del sistema funciona como se espera, y tener esa confirmación instantánea genera cierta satisfacción en el desarrollador, lo que refuerza su motivación en continuar el desarrollo.

B. *Testing bonito*

El código de *testing* no debe pensarse como algo externo al sistema, ambos trabajan juntos para construir un *software* confiable. Por eso, todos los lineamientos y buenas prácticas nombrados en capítulos anteriores deben ser respetados durante esta etapa.

> **Lineamiento:** El código de *testing* debe seguir las buenas prácticas de programación.

Los tests se representan como funciones, y por ello deben tener nombres descriptivos que haga explícito lo que se está probando. Por ejemplo: `test_product_endpoint_raises_error_on_bad_request`, si bien este nombre podría parecer

excesivamente largo, en este contexto no hay problema, lo importante es que sean precisos.

Otros lineamientos a tener en cuenta:

- Los tests deben enfocarse en un único comportamiento y tener una longitud adecuada. Muchas sentencias en una prueba es un síntoma de estar realizando múltiples acciones.
- La indentación debe mantenerse baja.
- Aprovechar los espacios en blanco para mejorar la legibilidad y separar bloques lógicos.
- Los comentarios deben utilizarse únicamente cuando la intención del código no sea suficiente.

Cuando el código de un sistema es *feo*, también lo serán sus pruebas. Y cuando las pruebas son *feas*, se pierde uno de sus propósitos fundamentales: aumentar la confiabilidad del sistema. Los lineamientos y buenas prácticas nos ayudan a que las pruebas, al igual que el resto del código, sean claras, útiles y sostenibles.

2. La pirámide del testing

Cuando queremos empezar a realizar las pruebas sobre nuestro código, es útil contar con una guía que nos ayude a organizarnos, del mismo modo que lo hicimos al estructurar nuestro proyecto mediante capas en el capítulo anterior. La **pirámide del testing** [13] es una de las referencias que utilizaremos para este propósito. Esta idea propone una estructura clara para clasificar las pruebas y decidir cuántas escribir en cada nivel.

La pirámide se compone de tres niveles:

- En la base se encuentran los **tests unitarios**, que verifican funciones pequeñas del código;
- En el medio están los **tests de integración**, que prueban cómo interactúan distintos módulos o componentes del sistema entre sí;
- Finalmente, en la cima están los tests **end-to-end** (E2E), que simulan el comportamiento completo del sistema.

La clave de esta pirámide está en la proporción: deberíamos tener muchos tests unitarios, menos tests de integración y pocos tests E2E. Esto se debe a que los tests unitarios son más rápidos, aislados y fáciles de mantener, mientras que los test end-to-end son costosos (en tiempo y a veces en recursos), frágiles y más difíciles de depurar.

Si bien en la práctica, estas proporciones no siempre se respetan al pie de la letra, la pirámide sigue representando una muy buena referencia para los desarrolladores. Nos recuerda que existen distintos niveles de granularidad en las

pruebas y que todos ellos son igual de importantes para mantener un código libre de errores.

Cada tipo de test posee sus propias estrategias de implementación, herramientas y objetivos que veremos a lo largo de este capítulo. Sin embargo, todos comparten una estructura común al momento de implementarlos: el patrón *Arrange, Act Assert* [14]. Este patrón funciona como una mnemotecnia, que nos ayuda a organizar la lógica del test:

1. En primer lugar, se prepara el escenario (*Arrange*), normalmente mediante funciones que se ejecutan antes de las pruebas y configuran los datos y el entorno necesario para simular una situación real.

2. Luego, se ejecuta la acción que queremos probar (*Act*), se llama a la función con parámetros específicos. Este es el cuerpo de nuestra prueba.

3. Finalmente, se verifica el valor esperado (*Assert*). Por lo general, esta es la última línea de la prueba, donde comparamos el resultado obtenido con el valor esperado. Si coinciden, la prueba finaliza correctamente indicando éxito, si no, el sistema indica un fallo enseñando el valor que no cumplió con la condición

3. Tipos de prueba

Al igual que en el capítulo anterior, estaremos utilizando un código de ejemplo para guiar la lectura. En este caso realizaremos pruebas sobre la aplicación *backend* de productos y precios del capítulo anterior.

Todo el código correspondiente lo encontramos en la carpeta `/testing` en la raíz del proyecto. Además contamos con un archivo `Makefile` para ejecutar más rápidamente las pruebas. En el archivo `README.md`, nuevamente en la raíz del proyecto encontramos las instrucciones para ejecutar las pruebas desde el archivo `Makefile`.

Para esta sección incluimos dos nuevas tecnologías:

- pytest [23]: *framework* que nos ayuda a escribir y ejecutar tests.

- unittest [24]: módulo de la biblioteca estándar de Python.

Si bien ambas tecnologías son útiles para realizar testing, utilizamos `pytest` como base para nuestras pruebas, y `unittest` como soporte con algunas herramientas que presentaremos más adelante.

A continuación revisaremos los tres tipos de tests que fueron nombrados con anterioridad. En cada uno de ellos explicaremos su alcance, algunas herramientas que se utilizan, revisaremos una implementación real y enseñaremos su ejecución y lectura de los resultados.

[23] `https://docs.pytest.org/en/stable/`
[24] `https://docs.python.org/3/library/unittest.html`

A. Tests unitarios

Los tests unitarios corresponden al primer nivel de la pirámide del *testing* y deberían abundar en cualquier proyecto de *software*. Su objetivo es verificar el comportamiento de unidades pequeñas del código de forma aislada, generalmente son funciones o métodos de clases. Es importante que estas pruebas sean rápidas y simples, ya que se ejecutan en gran cantidad. Además, no deben depender de bases de datos o servicios de terceros reales.

Ahora bien, esto no significa que no podemos probar funciones que interactúan con servicios externos o bases de datos. Lo que hacemos en estos casos es reemplazar temporalmente esas dependencias por versiones simuladas controladas. Para ello existen los *mocks* y los *stubs*, conocidos como *dobles de tests*. Ambos permiten reemplazar funciones reales por versiones falsas, cuyo comportamiento es conocido. La diferencia principal, es que un *mock*, además de simular comportamientos, pueden registrar mucha más información: cuantas veces se invocaron las funciones, con que argumentos, entre otros [5].

Si bien es posible crear los dobles a mano, la mayoría de las librerías modernas de testing nos facilitan estas tareas. En Python, el módulo `unittest.mock` ofrece utilidades como `MagicMock`, que permite crear objetos simulados configurando qué deben devolver o cómo deben comportarse. Luego, la función `patch` durante el test, nos permite reemplazar temporalmente los objetos del sistema por estos mocks.

Ejemplo en nuestro proyecto

En nuestro proyecto tenemos dos instancias de tests unitarios, la primera para la clase `ProductWithDollarBluePrices` y la segunda para `BluelyticsConncetor`, ambos dentro de la carpeta `/unit`. En este ejemplo, estudiaremos la segunda implementación.

Además, dentro de la carpeta `/mocks` encontraremos múltiples dobles que simulan esta clase de nuestro sistema. A continuación se presentan dos funciones que generan *mocks* para simular el comportamiento de una API que devuelve la cotización del dólar. Uno de ellos representa un escenario exitoso y el otro una respuesta con error.

```
def get_happy_mock_response(value_avg=1):
  mock_response = MagicMock()
  mock_response.raise_for_status.return_value = None
  mock_response.json.return_value = {
    "oficial": {"value_avg": 1, "value_sell": 1, "value_buy": 1},
    "blue": {"value_avg": value_avg, "value_sell": 1, "value_buy": 1},
    "oficial_euro": {"value_avg": 1, "value_sell": 1, "value_buy": 1},
    "blue_euro": {"value_avg": 1, "value_sell": 1, "value_buy": 1},
    "last_update": datetime.now(),
  }

  return mock_response

def get_bad_status_mock_response():
  mock_response = MagicMock()
  mock_response.raise_for_status.side_effect = HTTPError(
```

```
    "Bad status", response=mock_response
  )

  return mock_response
```

Ambos *mocks* son instancias de `MagicMock`, lo que nos permite configurar el comportamiento. En el caso de `get_happy_mock_response`, se define explícitamente que el método `raise_for_status` no haga nada (no produce ningún tipo de error), y, por otro lado que el método `json` devuelva un diccionario con los datos esperados por el sistema.

Si observamos `get_bad_status_mock_response`, veremos un escenario fallido. Al llamar a `raise_for_status`, se lanza una excepción `HTTPError`. Esto nos permite probar como reaccionaría el sistema ante situaciones inesperadas, sin depender de que el servicio externo falle realmente en ese momento.

Para complementar, hay que realizar efectivamente el test. Es por ello que definimos las siguientes funciones que hacen uso de los *mocks*:

```
def test_get_prices_return_avg_value_on_success():
  mock_response = get_happy_mock_response()
  with patch("requests.get", return_value=mock_response):
    connector = BluelyticsConnector()
    price = connector.get_price()
    assert price == 1

def test_get_prices_raises_http_error_on_bad_status():
  mock_response = get_bad_status_mock_response()
  with patch("requests.get", return_value=mock_response):
    connector = BluelyticsConnector()
    with pytest.raises(HTTPError):
      connector.get_price()
```

En el primer test, utilizamos `get_happy_mock_response()` para simular una respuesta válida de la API. Luego, con la función `patch`, reemplazamos temporalmente `request.get` por nuestra versión modificada. De este modo, cuando el método `get_price` de `BluelyticsConnector` intente hacer una llamada HTTP, en realidad estará recibiendo la respuesta simulada. Finalmente, usamos `assert` para verificar que el valor devuelto sea el esperado.

En el segundo test, usamos el *mock* `get_bad_status_mock_response()` para simular una respuesta fallida que lanza una excepción. Nuevamente empleamos `patch` para reemplazar a `requests.get` dentro del método `get_prices`. En este caso, la línea `with pytest.raises(HTTPError)` cumple el rol del `assert`, asegurando que efectivamente se lance una excepción `HTTPError`.

Es importante destacar que los tests unitarios no sólo deben validar los valores de retorno correctos, sino también cubrir otros aspectos como el comportamiento de una función: excepciones, efectos secundarios, e incluso detalles como la cantidad de veces que se llamó a una función interna.

Ejecución y salida

Como ya mencionamos anteriormente, gracias al archivo `Makefile` podemos ejecutar las pruebas rápidamente. En este caso, al correr el comando `make`

`run_unit_tests`, se ejecutarán todas las pruebas ubicadas dentro de la carpeta `/unit`.

Este comando, internamente ejecuta:

```
poetry run pytest testing/unit/
```

A continuación se muestra un ejemplo de su salida en la terminal:

```
poetry run pytest testing/unit/
==================== test session starts ====================
platform linux -- Python 3.12.3, pytest-8.3.5, pluggy-1.6.0
rootdir: /codigo-bonito-api-rest
configfile: pyproject.toml
plugins: cov-6.1.1, anyio-4.9.0
collected 13 items

testing/unit/test_bluelytics_connector.py ......... [ 61%]
testing/unit/test_product_with_dollar_blue.py ..... [100%]

==================== 13 passed in 0.19s =====================
```

En esta salida se destacan varios elementos. Primero, la cabecera, que indica información sobre la plataforma de ejecución, la versión de Python, los *plugins* activos y la cantidad de pruebas encontradas (`collected 13 items`). Luego, se listan los archivos de test junto con una serie de puntos (`.`) que representan tests que se ejecutaron con éxito y al final de la línea, un porcentaje que indica cuántas pruebas representa cada archivo sobre el total. Finalmente, se resume la ejecución con el total de pruebas pasadas y el tiempo tomó completarlas.

En el caso de que alguna prueba falle, el resumen cambia para incluir detalles del error. Por ejemplo:

```
testing/unit/test_bluelytics_connector.py F.......              [ 61%]
testing/unit/test_product_with_dollar_blue.py .....             [100%]

============================== FAILURES ==============================
_ test_get_prices_return_avg_value_on_success _

>           assert price == 2
E           assert 1.0 == 2

testing/unit/test_bluelytics_connector.py:20: AssertionError
==================== short test summary info ====================
FAILED testing/unit/test_bluelytics_connector.py::
    test_get_prices_return_avg_value_on_success - assert 1.0 == 2
=================== 1 failed, 12 passed in 0.22s ====================
```

Aquí podemos observar que una prueba falló (`F.......`), y el sistema muestra el detalle del error:

- En primer lugar, se nos informa cuál fue el caso de test que falló, `test_get_prices_return_avg_value_on_success`.
- Luego, la línea que produjo el error `assert price == 2`, y a continuación, el valor obtenido contra el esperado, `assert 1.0 == 2`. Finalmente, se menciona el archivo y la línea específica del fallo, junto a la excepción ocurrida, `AssertError`.

- Por último se muestra un resumen de las pruebas que fallaron junto con las exitosas, y el tiempo empleado.

B. Tests de integración

El segundo nivel de la pirámide corresponde a los tests de integración. A diferencia del nivel anterior, donde se validaban simplemente piezas de código aisladas, los tests de integración se enfocan en verificar cómo se relacionan e interactúan diferentes componentes del sistema. Su objetivo es asegurarse de que las partes del sistema colaboran correctamente respetando el flujo de datos.

Una herramienta comúnmente utilizada en este tipos de pruebas son los *fixtures*, proporcionados en Python por librerías como `pytest`. Los *fixtures* permiten definir un entorno de pruebas que se prepara antes (y opcionalmente después) de ejecutar cada prueba. Esto los hace ideales para inicializar datos, establecer conexiones o limpiar recursos, asegurando que cada prueba se ejecute en un contexto controlado y repetible.

Ejemplo en nuestro proyecto

En este caso, en nuestro proyecto realizamos testing de integración para comprobar cómo se relacionan componentes de las capas 0 y 1, es decir, la definición de datos y el acceso a ellos mediante repositorios. Como se explicó en el capítulo anterior, contamos con dos implementaciones de repositorios (una basada en SQLAlchemy y otra en PonyORM), ambas respetando una misma interfaz.

Las pruebas de nuestro proyecto se encargan de verificar que ambas implementaciones satisfacen correctamente la interfaz. En este ejemplo nos enfocaremos en las pruebas del repositorio de PonyORM, que se encuentran en el archivo `test_ponyorm_product_repository.py` dentro de la carpeta `/integration`.

En la siguiente prueba podemos ver la implementación de un *fixture* para estos casos de test:

```
@pytest.fixture()
def db_with_products():
  db.bind(provider="sqlite", filename=":memory:", create_db=True)
  db.generate_mapping(create_tables=True)

  with db_session:
    Product(name="Pretty shirt", price=7500.0)
    Product(name="Cool mug", price=4000.0)
    Product(name="TV 4K", price=1500000.0)
    commit()

  yield

  db.provider = None
  db.schema = None
  db.disconnect()
```

Este código representa un *fixture* que configura una base de datos en memoria utilizando SQLite. El decorador `@pytest.fixture()` sobre la definición de la función `db_with_products` indica a `pytest` que esta función puede ejecutarse antes de cada prueba. Dentro del cuerpo del *fixture*, se crea una base de datos

limpia, se generan las tablas correspondientes y se insertan tres productos de ejemplo.

El uso de la palabra clave `yield` permite suspender temporalmente la ejecución para correr un test. Una vez finalizado el mismo, se continúa con la desconexión y la limpieza de la base de datos. Este patrón nos asegura que cada test se ejecute sobre una base de datos limpia, sin verse afectado por efectos secundarios de la ejecución de pruebas anteriores.

Veamos ahora cómo se utiliza este *fixture* en casos concretos de testing:

```
def test_get_by_id_returns_product(db_with_products):
  with db_session:
    repo = PonyProductRepository()

    product = repo.get_by_id(1)
    assert product.name == Product.get(id=1).name

def test_create_product(db_with_products):
  with db_session:
    repo = PonyProductRepository()
    product_count = count(p for p in Product)

    repo.create(CreateProductData(name="Candy bar", price=100.0))
    assert count(p for p in Product) == product_count + 1
```

En estas dos pruebas, el *fixture* `db_with_products` se incluye como parámetro en la definición de cada función. Esto le indica a `pytest` que debe ejecutar el *fixture* antes de correr la prueba. Así, en el caso de tener múltiples *fixtures* en un mismo archivo, podríamos indicar precisamente cual utilizar en cada caso.

La primer prueba verifica que, al buscar el producto con id 1 (insertado previamente por el *fixture*), el repositorio devuelve un objeto válido. Para valida el resultado, se compara el nombre del producto del repositorio con el obtenido directamente desde la base de datos.

En la segunda prueba, se comprueba que la creación de un nuevo producto funcione correctamente. Para ello, se cuenta la cantidad de productos existentes antes de la operación, luego se crea un nuevo producto mediante el repositorio, y finalmente se verifica que la cantidad de productos haya aumentado en uno.

Es importante destacar que, si bien estamos introduciendo el concepto de *fixtures* en los tests de integración, esta herramienta es completamente funcional en cualquier nivel de testing, esto debido a que el concepto de 'nivel de testing' es puramente teórico y no existe ni para `pytest` ni para cualquier otra librería.

Ejecución y salida

En este caso, la ejecución de las pruebas se realiza con el comando `make run_integrarion_tests` que internamente realiza `make run pytest testing/integration`. Nuevamente, en la salida observamos los archivos y las pruebas ejecutadas, ya sean exitosas o fallidas.

```
poetry run pytest testing/integration/
======================= test session starts =======================
platform linux -- Python 3.12.3, pytest-8.3.5, pluggy-1.6.0
rootdir: /codigo-bonito-api-rest
configfile: pyproject.toml
```

```
plugins: cov-6.1.1, anyio-4.9.0
collected 18 items

testing/integration/test_ponyorm_product_repository.py ......... [ 50%]
testing/integration/test_sqlalchemy_product_repository.py ...... [100%]

======================== 18 passed in 0.38s ========================
```

C. Tests end-to-end

Finalmente, en la cima de la pirámide, encontramos las pruebas *end-to-end*. Este tipo de pruebas busca validar el funcionamiento de todo el sistema, desde los componentes pertenecientes a las capas inferiores hasta las interfaces accesibles por los usuarios. En este nivel, es fundamental que el entorno de pruebas se asemeja lo máximo posible al entorno de producción. Por ejemplo, si bien en los niveles anteriores utilizamos una base de datos en memoria, eso no es aceptable en E2E, ya que nuestro sistema real utiliza una base de datos persistente en archivo.

El objetivo de este nivel es responder a una pregunta clave: ¿el sistema completo se comporta correctamente de principio a fin?

Ejemplo en nuestro proyecto

Este nivel lo encontramos dentro de la carpeta `/e2e`, y en este caso contamos con un único archivo, `test_endpoints`, que realizará las pruebas sobre los *endpoints* de nuestro *backend*.

Estas pruebas tienen una particularidad: como requieren que la aplicación esté en ejecución, es necesario preparar el entorno antes de lanzarlas. Para eso, definimos un *script* en el archivo `Makefile`. Este *script* establece variables de entorno para que el sistema utilice una base de datos de prueba y el ORM SQLAlchemy, y luego se encarga de iniciar y detener automáticamente la aplicación antes y después de correr las pruebas.

Observemos el *fixture* que utilizan estos tests:

```python
@pytest.fixture(autouse=True)
def clear_db():
  database_path = os.getenv("DATABASE_PATH", "./test_db.sqlite")
  database_url = f"sqlite:///{database_path}"

  engine = create_engine(database_url)
  Session = sessionmaker(bind=engine)
  session = Session()

  try:
    session.query(Product).delete()
    session.commit()

    products = [
      Product(name="Pretty shirt", price=7500.0),
      Product(name="Cool mug", price=4000.0),
      Product(name="TV 4K", price=1500000.0),
    ]
```

```
    session.add_all(products)
    session.commit()
  finally:
    session.close()

  yield

  session = Session()
  try:
    session.query(Product).delete()
    session.commit()
  finally:
    session.close()
```

Este *fixture* comparte muchas similitudes con el utilizado en la capa anterior, aunque con algunas diferencias clave. Por un lado, aquí utilizamos SQLAlchemy en lugar de PonyORM, y por el otro, estamos trabajando con una base de datos persistente, no en memoria, lo cual requiere que eliminemos los datos manualmente antes y después de cada prueba.

También es importante destacar el uso del parámetro `autouse=True` en el decorador del *fixture*. Esto le indica a `pytest` que debe ejecutar automáticamente la función antes de cada test, sin necesidad de pasarla como parámetro.

El único test que revisaremos en este nivel es el siguiente:

```
def test_update_products_price_returns_422_if_the_factor_is_invalid():
  response = requests.put("http://localhost:8000/products?factor=
    NOTANUMBER")
  assert response.status_code == 422
```

Aquí podemos observar que se está realizando una llamada HTTP real a la aplicación mediante `requests.put`. En este caso, se llama al *endpoint* encargado de actualizar los precios de los productos, pero con la particularidad de usar `NOTANUMBER` como factor multiplicativo. Ante esta situación, la aplicación debería lanzar una excepción y responder con un código HTTP `422 Unprocessable Entity`, indicando un error en el parámetro ingresado.

Ejecución y salida

Para el caso de las pruebas *end-to-end*, la ejecución es algo más compleja. Para correrlas, utilizamos el comando `make run_e2e_tests`, que ejecuta una serie de pasos adicionales de forma secuencial:

```
DATABASE_PATH=./test_db.sqlite ORM=sqlalchemy \
poetry run uvicorn app.main:app > uvicorn.log 2>&1 & \
echo $! > uvicorn.pid; \
for i in $(seq 1 10); do curl -s http://localhost:8000; if [ $? -eq 0
   ]; then break; fi; echo "Esperando que el backend inicie..."; sleep
    1; done;  \
poetry run pytest testing/e2e/test_endpoints.py; \
TEST_EXIT_CODE=$?; \
kill `cat uvicorn.pid`; rm uvicorn.pid; \
unset DATABASE_PATH; unset ORM; \
exit $TEST_EXIT_CODE
Esperando que el backend inicie...
Esperando que el backend inicie...
```

No nos detendremos en explicar en detalle cada una de estas líneas, pero su propósito es el siguiente: arrancar el *backend* en segundo plano, esperar a que esté disponible, ejecutar las pruebas y luego apagar el servidor. Este proceso asegura que el sistema esté corriendo al momento de realizar las pruebas, y al mismo tiempo permite controlar el entorno con precisión mediante variables como la ruta de la base de datos y el ORM a utilizar.

La salida generada por estas pruebas mantiene el mismo formato que vimos anteriormente: primero se imprime un resumen del entorno de ejecución, y luego el resultado de los casos de prueba.

```
========================= test session starts ==========================
platform linux -- Python 3.12.3, pytest-8.3.5, pluggy-1.6.0
rootdir: /codigo-bonito-api-rest
configfile: pyproject.toml
plugins: cov-6.1.1, anyio-4.9.0
collected 13 items

testing/e2e/test_endpoints.py .............                      [100%]

========================== 13 passed in 2.19s ==========================
```

Observemos que en este caso, la ejecución tomó poco más de 2 segundos. Aunque esto sigue siendo rápido, se nota una diferencia notable en comparación a las pruebas unitarias y de integración que apenas sumaban un segundo entre las dos. Es por este motivo que debemos mantener una cantidad razonable de pruebas *end-to-end* y evitar probar casos triviales en este nivel, ya que podrían ralentizar aún más el proceso.

D. Errores en nuestra aplicación

Durante el desarrollo de las pruebas para este capítulo, encontramos un error real en nuestra aplicación *backend*. Al intentar crear un nuevo producto con un precio negativo, esperábamos que se arrojara un error. Sin embargo el sistema aceptó el valor. Este comportamiento quedó en evidencia a través del siguiente test, que en un sistema correcto debería haber pasado sin problemas:

```
def test_create_product_with_negative_price_raises_error(session):
  repo = SQLAlchemyProductRepository(session)

  data = CreateProductData(name="Invalid Product", price=-100.0)
  with pytest.raises(ValueError):
    repo.create(data)
```

Podríamos haber corregido el repositorio agregando una validación sobre el precio del producto, pero decidimos mantener el error y la prueba fallida para reforzar la importancia del testing. Este tipo de errores son claves para construir un sistema confiable, a priori nunca conocemos a los usuarios de nuestra aplicación, y en consecuencia, no sabemos como pueden llegar a hacer uso de ella. Un conjunto de pruebas exhaustivas nos permite anticiparnos a estos escenarios inesperados y lograr una aplicación robusta frente a errores.

4. Unificando código y testing

Ahora cambiemos la mentalidad del testing: en lugar de realizarlo en una etapa posterior a la programación, lo pensemos como algo complementario al momento de escribir el código. Una de las estrategias más conocidas es *Test-Driven Development* (TDD) [25].

En TDD, el desarrollador primero escribe una prueba para una función específica. Luego implementa el código mínimo necesario para que esa prueba pase correctamente. Este proceso se repite hasta que la funcionalidad quede completa, y finalmente se refactoriza el código si es necesario. Siempre procurando que la prueba siga corriendo exitosamente. Las ventajas de este patrón son evidentes, todo el sistema queda probado desde el inicio, y sólo se escribe el código estrictamente necesario, ni más ni menos.

Parecería todo ventajas, pero TDD tiene sus dificultades. El desarrollador necesita una visión amplia y clara del sistema antes de construirlo, es decir que requiere conocer todo el sistema, sus responsabilidades, flujos principales, secundarios y casos excepcionales. Esto no siempre ocurre, especialmente en etapas tempranas del desarrollo. Así, forzar un test previo antes de la etapa de programación del sistema, se convierte en un obstáculo más que en una guía.

Aun así, podemos realizar pruebas mientras escribimos el código sin utilizar TDD. La mayoría de los lenguajes ofrecen herramientas externas de *debugging* (o depuración [26]) que nos permiten inspeccionar y experimentar con el código en tiempo de ejecución. En Python existe `ipdb` [27], mientras que en JavaScript tenemos el depurador de `Node.js` [28], el cual suele estar integrado en editores como VS Code. Esta herramienta permite detener la ejecución del programa en un punto específico y realizar diversas acciones como:

- Examinar y modificar variables.
- Recorrer el código instrucción por instrucción.
- Inspeccionar la pila de llamadas (*stack*).
- Entre otras acciones útiles para entender el estado interno del sistema.

Consideramos que saber depurar código es muy importante, ya que nos ayuda a enfrentar errores difíciles de rastrear o simplemente a observar el comportamiento de nuestro programa mientras lo desarrollamos. Sin embargo, profundizar en estas herramientas se escapa del alcance de este capítulo y el trabajo.

[25] https://es.wikipedia.org/wiki/Desarrollo_guiado_por_pruebas
[26] https://es.wikipedia.org/wiki/Depurador
[27] https://github.com/gotcha/ipdb
[28] https://nodejs.org/en/learn/getting-started/debugging

5. La importancia del buen testing

Escribir buenas pruebas no es una tarea trivial. Como cualquier otra habilidad en el desarrollo, requiere de práctica y buen criterio para detectar problemas relevantes. Al comienzo, es normal caer en pruebas demasiado simples que no verifican correctamente el comportamiento del sistema, o en contraparte, pruebas demasiado estrictas, que se rompen ante el mínimo cambio. El verdadero desafío es lograr escribir pruebas que actúen como un mecanismo de seguridad efectivo, es decir, que detecten errores sutiles, pero que también logren probar los comportamientos importantes, los casos bordes e incluso las situaciones inesperadas.

Un testing mal aplicado puede, de hecho, jugar en contra del sistema. Ralentizando el desarrollo y generando una falsa sensación de seguridad. Es por eso que métricas como cobertura en los tests no siempre aportan un valor real al testing. Podemos tener conjuntos de tests que verifiquen cada una de las líneas y flujos en nuestro código, pero que sean pobres cuando hablamos de calidad y confiabilidad.

En definitiva, el testing es una de las herramientas más poderosas del desarrollo. Y cuando se tiene mucha práctica y atención a los lineamientos correctos, se convierte en una pieza clave para construir sistemas con una buena base de código y a prueba de errores.

VIII. Conclusiones

1. ¿Cómo nació este trabajo?

Este trabajo comenzó con algo que mi director notó desde su rol como docente de la FaMAF y su experiencia en la industria: muchos compañeros, colegas y alumnos escriben código complicado, poco claro y difícil de seguir o mantener. Por mi parte, siempre busqué mejorar mi forma de programar, poniendo atención a los nombres que utilizo, al estilo de cada lenguaje y la forma de estructurar las funciones. Y del mismo modo, muchas veces me encontré con códigos que no respetaban esto o que simplemente eran innecesariamente complejos.

De ahí surgió la necesidad de establecer algunas reglas claras y, a la vez, lo suficientemente simples como para no tener que sobrepensar cada línea al programar. Muchas de las bibliografías que ya abordan estos temas suelen volverse demasiado complejas, por lo que no son ideales para quienes recién empiezan, perdiendo así su propósito.

En un principio, este trabajo también buscó servir de base para una materia optativa o un curso breve que enseñe estos principios de una forma más práctica. Como ya se mencionó en la introducción, enseñar a escribir un buen código no es una tarea fácil: faltan recursos y el tiempo disponible muchas veces no alcanza para dar a cada alumno la atención necesaria. Por ahora, esta idea de asignatura está pausada, pero no descartamos ofrecer en el futuro este contenido como una herramienta adicional a quienes están dando sus primeros pasos.

Hoy este trabajo se presenta como un punto de partida, resumiendo criterios y proponiendo formas de escribir y compartir un código bonito. A continuación, retomamos esta idea con algunos aspectos clave que pueden servirnos para validar si estamos frente a un **código bonito**.

2. Aspectos para validar un código bonito

Durante los primeros parrafos de este trabajo introdujimos por primera vez el concepto de código bonito: *un código es bonito si es claro, prolijo y está bien estructurado*, idea que nos acompañó a lo largo de todos los capítulos. Dada que esta definición no es para nada formal y puede tener muchas interpretaciones, nos dedicamos a explicar cómo mejorar en la escritura de este tipo de código mediante los *lineamientos*.

Entonces, con todos estos lineamientos explicados, nos gustaría complementar respondiendo a la siguiente pregunta:

A. ¿Cómo detecto un código bonito?

El código bonito debería ser identificable a simple vista; sin embargo, como todo, esto se logra con mucha práctica, escribiendo y leyendo una gran cantidad de códigos diferentes. Al final, esta no es una habilidad que se aprenda de la noche a la mañana.

Entonces, nos parece necesario conocer la siguiente lista para no sólo escribir, sino también reconocer código bonito:

1. Al leer funciones, clases y variables, deberíamos ser capaces de entender su propósito con sólo leer su nombre. El tipado y la documentación interna del código deben reforzar esta claridad.

2. Las funciones y métodos deben estar bien organizados, ser cortos y concisos. Sin flujos excesivamente complejos que dificulten la lectura. Además, las sentencias deben **respirar**, cada *momento* del código debe estar claramente diferenciado.

3. Debemos encontrar consistencia en el código, no sólo en el idioma, sino también en la forma de escribir y estructurar las sentencias. Un código que sigue las convenciones de su lenguaje en general será bien conciso.

4. No deberíamos encontrarnos con comentarios por todo el código. Estos deben ser pocos y realmente aportar un valor cuando el código no puede ser lo suficientemente expresivo por si mismo.

5. La arquitectura del sistema tiene que mostrar una organización clara, todas las partes deben poseer una responsabilidad bien definida y con límites en su alcance.

6. Por último, debemos encontrar tests en el código de nuestro sistema, organizados por nivel de complejidad e interacción entre componentes. El cuidado en la escritura de las pruebas debe ser el mismo que posee el resto del código.

Con todo esto y junto a los lineamientos desarrollados a lo largo del trabajo, culmina nuestra forma de explicar qué es un *código bonito*. Es posible que algún desarrollador no esté de acuerdo con algunos de los puntos mencionados -y esto está bien-, al no existir una definición formal, no se puede pretender que todos encontremos el código claro de la misma manera. Pero más allá de la opinión personal, un código que cumple con estas ideas rara vez será difícil de leer y mantener.

3. ¿Qué aprendí y cómo cambió mi forma de escribir código?

Con el pasar de los días mientras realizaba este trabajo, fui notando cómo mi forma de escribir código fue mejorando. Como mencioné al principio, siempre me importó la prolijidad, pero durante el desarrollo de este proyecto pude realmente reforzar hábitos y descubrir prácticas que quizás antes pasaba por algo o no valoraba lo suficiente.

En primer lugar, este proyecto me abrió las puertas a conocer aún más Python, un lenguaje que había usado muy poco porque suelo trabajar con TypeScript.

En cuanto a los lineamientos también aprendí algunas cosas, una de las principales fue la importancia de tener funciones cortas. Con el paso de las semanas comencé a modularizar y repensar más la forma en la que implemento mis funciones. También cambió mi manera de escribir comentarios: hoy escribo muchos menos, sólo los necesarios, sobre todo cuando trabajo en equipo y hay que dejar en claro algunos aspectos importantes. La cantidad bajó en gran manera y la calidad subió.

Si pensamos en las clases, ya conocía el concepto de **inyección de dependencias**, pero nunca lo había usado de verdad ni tampoco le veía una funcionalidad práctica. Recién cuando me tocó implementarla en un proyecto real entendí su utilidad y el potencial que posee.

Por último, con el testing, sentí algo similar. Nunca realicé muchas pruebas en mi código más allá de lo básico sobre funciones núcleo. Pero ahora tengo más claro cómo debo organizar las pruebas y comprendo en profundidad cómo utilizar los diferentes tipos de tests que existen. Si bien todavía no es parte del día a día en mi trabajo, me siento preparado para hacerlo bien cuando llegue el momento adecuado.

Si alguien me preguntara **¿qué fue lo más difícil de entender?**, respondería que las capas en el capítulo de **organización de un proyecto de *software***. Si bien entendía la idea de dividir y ordenar el sistema en diferentes partes con sus responsabilidades claras, escribir esto fue un desafío. Principalmente porque es un tema muy estudiado por muchas personas y no quería 'reinventar la rueda' ni mucho menos ir en contra de prácticas ya consolidadas.

A. ¿Cómo cambié a mi entorno?

En cuanto a los cambios que noté a mi alrededor, principalmente en el ámbito laboral, puedo destacar dos cosas. Primero, mis compañeros más cercanos empezaron a mostrar mayor interés en escribir mejor cuando me toca revisar su código. Y segundo, incluso quienes no trabajan directamente conmigo, cada vez que surge algo relacionado con escribir buen código, hacen referencia a este proyecto. Eso me hace creer que realmente se logró dejar una huella.

4. Recepción del trabajo

Uno de los objetivos al iniciar este trabajo era hacer pública toda la información, de forma accesible y práctica, con la idea en mente de que llegue al mayor número de personas posibles. Para lograrlo, desarrollamos una página web [29], donde cada ciertas semanas se publicaba un nuevo capítulo. Aprovechando esto, decidimos incorporar herramientas para medir y analizar el impacto real en los visitantes. A continuación se presentan algunas de las métricas obtenidas:

Usuarios totales

[29] `https://www.writingprettycode.com/`

Figura 1: Histórico de usuarios activos

En la figura 1 se observan los usuarios conectados por día, desde el primero de abril de 2025 hasta el 7 de julio de 2025, casi 100 días. En la cabecera de la figura, podemos ver que el total de usuarios activos en el período fue de 731 (los usuarios nuevos corresponden a usuarios que abrieron la página por primera vez, lo ignoraremos en este caso ya que parece hubo algún error de conteo).

Podemos notar tres picos de visitas que sobresalen sobre el resto. El primer pico corresponde a la primera quincena de abril, durante estos días hicimos la primera publicación en redes, principalmente LinkedIn, de la página. Luego, los picos de junio y julio corresponden a las publicaciones de los capítulos de **organización de un proyecto de *software*** y **testing**, ambos también fueron difundidos por redes sociales.

En la cabecera también podemos observar el tiempo medio de interacción de los usuarios activos, pero esta métrica también la ignoraremos ya que considera los días donde no hubo mucho tráfico.

A. Encuestas por capítulo

Además de la cantidad de usuarios, nos interesamos por la recepción, opinión y perfil de los mismos. Es por ello que cada capítulo incluía una breve encuesta de dos preguntas y un campo de texto opcional para comentarios adicionales. Las preguntas eran:

- ¿Qué opinas sobre el contenido del capítulo X?
- ¿Cuál es tu perfil?

A continuación observaremos los resultados obtenidos:

Introducción

En la figura 2 observamos 9 respuestas: 5 lectores afirmaron que *aprendieron cosas nuevas*, mientras que los 4 restantes indicaron que *reforzaron los conocimientos*. Cabe destcar que, salvo una excepción, quienes eligieron la primera opción son entusiastas autodidactas o estudiantes nuevos en carreras relacionadas a la programación, lo que muestra cómo los lectores con menos experiencia

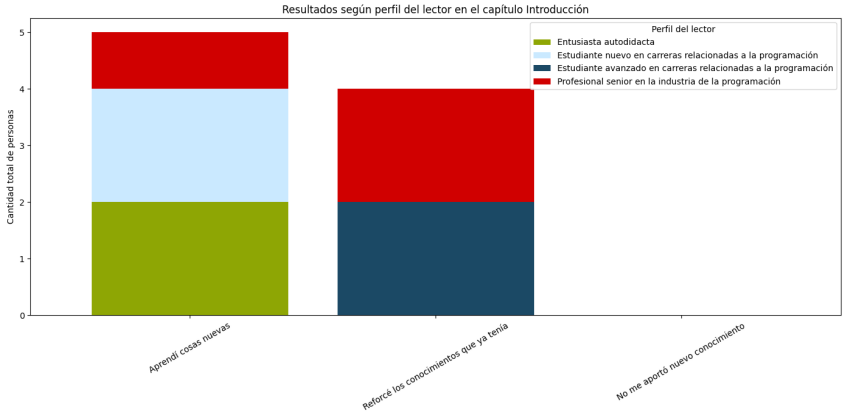

Figura 2: Opinión del capítulo/cantidad de perfiles en la opinión

pudieron aprovechar mejor el contenido del capítulo. En contraste con esto, quienes indicaron que *reforzaron sus conocimientos* son estudiantes avanzados y desarrolladores *senior*.

Queremos destacar un comentario de un usuario: *«Entiendo que esté narrado diferente a Clean Code [...], pero no terminé de entender la propuesta de valor [...] ¿Qué conocimiento nuevo aporta para alguien que ya leyó el libro? [...]»*

A esto respondemos que *Lineamientos para escribir código bonito* no es ni una crítica ni una reversión o reinvención de la bibliografía ya existente. Por el contrario, este trabajo se nutre de libros que ya exploran esta problemática. Nuestro objetivo es simplificar y hacer accesible este conocimiento a todos, con ejemplos claros y directos. Muchas veces la bibliografía sobre el tema se aleja de su propósito principal y profundiza en temas más complejos que muchos lectores quizás no buscan en un primer acercamiento.

Sintaxis y semántica

Este es el capítulo donde encontramos mayor cantidad de respuestas, con 14. En la figura 3 observamos que más de la mitad de los lectores *reforzaron sus conocimientos*, mientras que 5, *aprendieron cosas nuevas*. En este caso, la distribución es más variada y no sigue ningún patrón destacable, simplemente podemos decir que de algún modo el contenido del capítulo fue útil para todos.

Lo que si nos parece importante destacar, es la presencia de un lector estudiante avanzado de Licenciatura en Física. Esto es de suma importancia para nosotros, ya que nuestros lectores no son sólo estudiantes de programación. Llegar a otras áreas siempre es algo destacable.

Diseño de funciones

Al igual que en el capítulo anterior, en la figura 4, no encontramos ningún patrón sobre los perfiles de las respuestas. Solamente, al igual que en el capítulo de introducción, hubo 5 lectores que *aprendieron cosas nuevas*, contra 4 que *reforzaron sus conocimientos*. Además, nuevamente, contamos un estudiante de

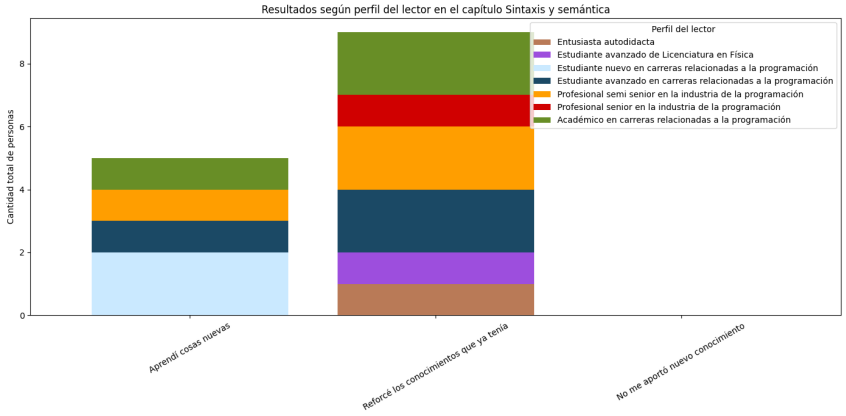

Figura 3: Opinión del capítulo/cantidad de perfiles en la opinión

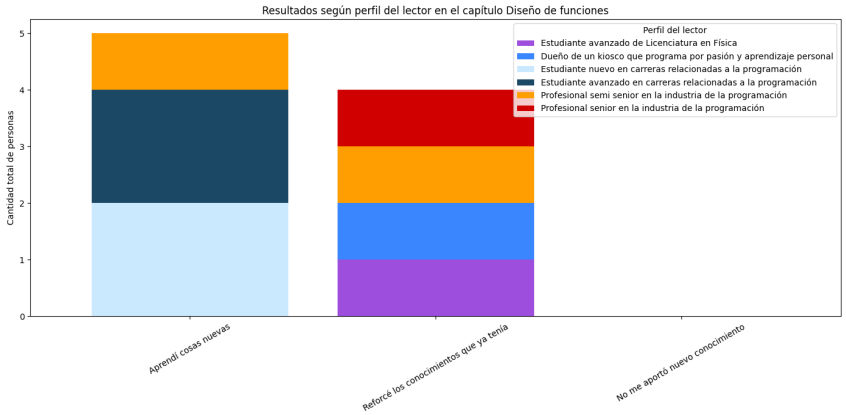

Figura 4: Opinión del capítulo/cantidad de perfiles en la opinión

Física y un *dueño de kiosco que programa por pasión*.

En este caso un lector dejó un comentario mostrando su aprobación sobre el capítulo: *«[...] Me sentí identificado con la importancia de escribir funciones pequeñas y con una sola responsabilidad, eso lo aplico cada vez que puedo. Me pareció útil también todo lo relacionado con el manejo de indentación y espacios en blanco, que muchas veces se subestima.»*.

Otro comentario, habló de la función `isValidUser`: *«[...] recomendaría analizar si tantos condicionales son necesarios sólo para un log y no convendría simplificarla a un si/no [...]»*.

Estamos totalmente de acuerdo con esta observación y queremos aclarar que la función, al igual que muchos otros fragmentos de código presentes en el trabajo tienen un propósito exclusivamente **didáctico**. No pretenden ser una implementación lista para producción o del día a día.

Documentación y comentarios

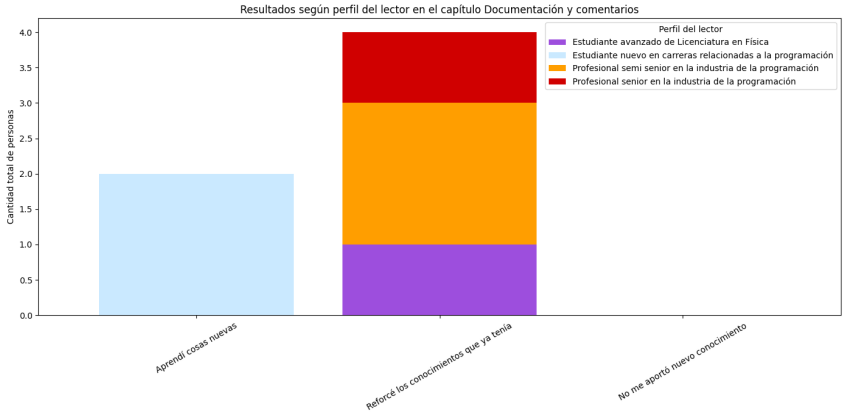

Figura 5: Opinión del capítulo/cantidad de perfiles en la opinión

En este capítulo nos encontramos con 6 respuestas. En la figura 5 podemos observar que dos estudiantes nuevos en carreras relacionadas a la programación señalaron que *aprendieron cosas nuevas*, mientras que el resto de lectores, profesionales de la industria y estudiante de Física, *reforzaron sus conocimientos*. Al igual que en el primer capítulo, el contenido le fue útil a los lectores con menos experiencia. Creemos que esta estadística es importante, dado que en las carreras de programación no se suele enseñar a documentar/comentar, estos temas simplemente se explican como una cualidad que poseen los lenguajes de programación, y que luego el estudiante desarrollará en la práctica.

Organización de un proyecto de *software*

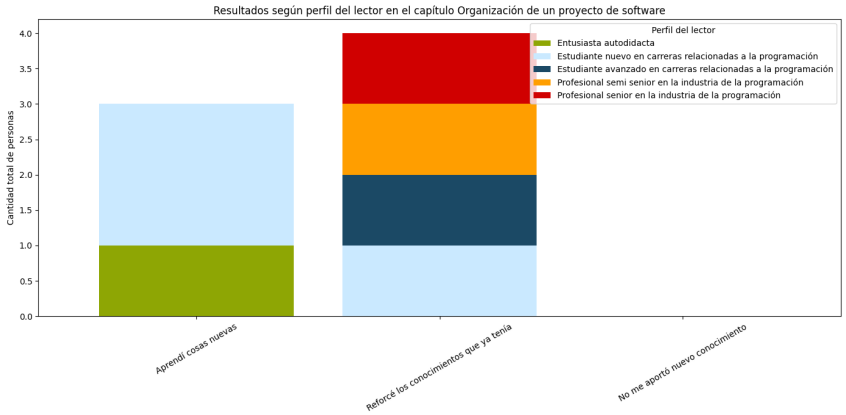

Figura 6: Opinión del capítulo/cantidad de perfiles en la opinión

En *organización de un proyecto de software* participaron 7 lectores en la encuesta. En la figura 6 se observa que 3 de ellos, estudiantes nuevos o entusiastas, indicaron que *aprendieron cosas nuevas*. Por otro lado, resulta interesante

que un estudiante principiante afirmara que *reforzó conocimientos que ya tenía*. Sin embargo, no es extraño pensar que un desarrollador experimentado ingresó en alguna carrera relacionada a la programación. Para complementar el análisis, los demás perfiles que *reforzaron sus conocimientos* corresponden a estudiantes avanzados y profesionales con algunos años de experiencia.

Este capítulo tiene mucho contenido y hay mucho de lo que no se habló, ya sea por prioridades o alcance del trabajo. De todas maneras queremos remarcar el siguiente comentario de un lector:

«[...] este capítulo debería mencionar las siguientes particularidades sobre organización de código: [...] el lenguaje, frameworks y/o herramientas utilizadas tiene un impacto muy grande sobre cómo se estructura el código [...] . Se debería aclarar que la utilización de `layer_X_Y` *en el código [...] -a mi entender- no es algo que debería ser copiado»*.

En primer lugar, estamos de acuerdo con que la estructura de un proyecto depende en gran medida del lenguaje y *framework* utilizado, es por ello que procuramos hacer el capítulo lo más genérico posible. Por ejemplo, en la capa 0 nos encontramos con la **definición de datos**, un concepto que puede tener muchas interpretaciones: una base de datos para un *backend*, un archivo con los tipos de datos de las respuestas que recibe un *frontend*, entre otros casos.

Por otro lado, consideramos que hacer uso de `layer_X_Y`, es algo totalmente subjetivo a cada desarrollador. Algunos encontraran útil hacer una diferenciación de carpetas y capas, mientras que otros preferirán no hacer esto tan explícito. Al final, lo que buscamos con este enfoque de organización por carpetas es hacer que el código sea lo más evidente y comprensible posible.

Testing

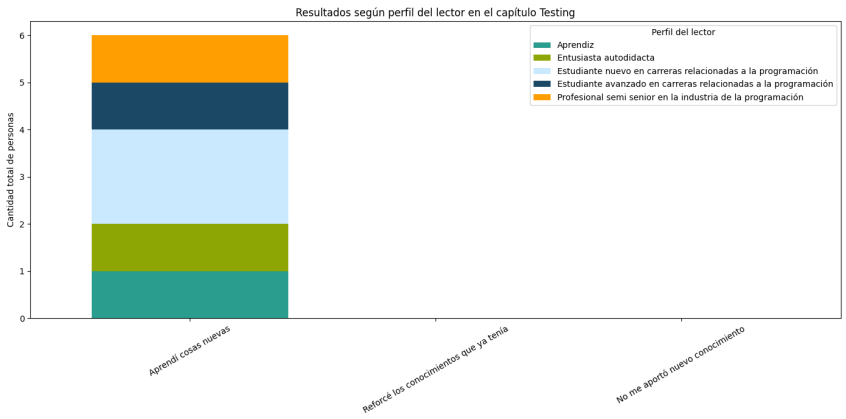

Figura 7: Opinión del capítulo/cantidad de perfiles en la opinión

Por último, tenemos el capítulo sobre testing, los tiempos de este capítulo fueron muy inferior al resto, de todos modos consiguió juntar 6 respuestas. En la figura 7 observamos que el 100 % de los lectores afirmo que *aprendió cosas nuevas*, dato que no es menor, pues nos hace pensar que no todos los desa-

rrolladores realizan pruebas a su código. En este caso, los perfiles son diversos: estudiantes, profesionales y autodidactas o aprendices.

En este capítulo no encontramos ningún comentario sobre el cual podamos hacer algún tipo de devolución.

Como último dato significativo, nos gustaría remarcar que en ninguna de las 6 encuestas, nos encontramos con una respuesta que contenga la tercera opción: *No me aportó nuevo conocimiento*, en la pregunta *¿Qué opinas sobre el contenido del capítulo X?*. Con esto, podemos asegurar que el público que pudimos medir, se llevó algún aprendizaje en mayor o menor medida.

Para cerrar esta sección, si bien no se trata de una métrica directa de la página o de una respuesta cuantificable, podemos afirmar que el proyecto recibió una gran aceptación por parte de grupos muy diversos en el mundo de la computación: docentes, estudiantes, profesionales con y sin formación universitaria, entre otros.

Con todos estos datos, puedo concluir que se logró el objetivo de poner esta información a disposición de todos los interesados. De todos modos esto no concluye aquí, la página seguirá abierta y las encuestas también. Por lo que seguiremos recopilando información.

5. Próximos pasos

Antes de dar el cierre, me gustaría comentar que este trabajo será presentado en un evento llamado **BeerJS** [30], un espacio donde desarrolladores de JavaScript y TypeScript se reúnen mensualmente a escuchar charlas y pasar un buen momento en un entorno relajado. Esta presentación se realizará el 31 de julio de 2025 e incluirá una charla breve, donde resumiré los principales lineamientos y temas del proyecto. El principal objetivo del evento es que más personas conozcan el concepto de código bonito y se animen a profundizar en él a través de la página.

Además, seguimos en la búsqueda de que algunas materias de nuestra facultad, y por que no de otras, puedan utilizar este material como herramienta complementaria al momento de desarrollar proyectos.

6. Reflexión final

A simple vista, escribir código bonito puede parecer una tarea sencilla: basta con estar atento a los momentos en que el código se vuelve confuso y buscar alternativas más claras, como muchas de las que se presentaron en este trabajo. Sin embargo, dominarlo de verdad y sostenerlo en el tiempo es la parte difícil. Nadie empieza a escribir código prolijo y de forma automática de un día para otro, esto requiere mucho tiempo, práctica, atención, interiorizar conceptos y, sobre todo, ganas de mejorar.

Aún así, creo que el simple hecho de conocer las ideas y tenerlas en mente ya

[30] `https://beerjscba.com/`

nos ayuda a programar con conciencia. Es muy probable que, tarde o temprano, alguien -quizás nosotros mismos- deba leer, entender y mantener el código que escribimos. Darle (o darnos) una mano escribiendo de forma prolija es algo que siempre se agradece, y en parte es una forma de empatizar con el resto de desarrolladores. Si retomamos algunas de las analogías mencionadas en los primeros capítulos, donde comparábamos albañilería y programación o el código con una novela literaria, nadie agradecería trabajar con un albañil que no coloca bien los ladrillos, así como nadie disfrutaría leer una novela llena de errores ortográficos.

Espero que este trabajo sirva como una herramienta útil, no sólo para aquellos que están comenzando a programar, sino también para los que ya tienen algunos años de experiencia y aún así eligen seguir mejorando día a día. Y, sobre todo, que sirva de recordatorio que aunque escribir código bonito sea una tarea difícil de dominar, siempre es posible avanzar de a poco, una línea a la vez.

Referencias

[1] Len Bass, Paul Clements y Rick Kazman. *Software Architecture in Practice*. 4th. Addison-Wesley, 2021.

[2] Dustin Boswell y Trevor Foucher. *The Art of Readable Code: Simple and Practical Techniques for Writing Better Code*. O'Reilly Media, 2011.

[3] MDN Web Docs. *Glossary - Statement*. Último acceso: 01-07-2025. URL: https://developer.mozilla.org/en-US/docs/Glossary/Statement.

[4] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.

[5] Martin Fowler. *Test Double*. Último acceso: 01-07-2025. 2006. URL: https://martinfowler.com/bliki/TestDouble.html.

[6] Geeks For Geeks. *Type Systems:Dynamic Typing, Static Typing & Duck Typing*. Último acceso: 01-07-2025. 2019. URL: https://www.geeksforgeeks.org/python/type-systemsdynamic-typing-static-typing-duck-typing/.

[7] Pankaj Jalote. *An Integrated Approach to Software Engineering*. Springer, 2005.

[8] Juval Löwy. *Programming .NET Components*. O'Reilly Media, 2005.

[9] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson, 2008.

[10] Jeffrey Palermo. *Onion Architecture*. Último acceso: 01-07-2025. 2013. URL: https://jeffreypalermo.com/tag/onion-architecture/.

[11] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[12] Robert W. Sebesta. *Concepts of Programming Languages*. 12th. Pearson, 2023.

[13] Ham Vocke. *The Practical Test Pyramid*. Último acceso: 01-07-2025. 2018. URL: https://martinfowler.com/articles/practical-test-pyramid.html.

[14] Bill Wake. *3A - Arrange, Act, Assert*. Último acceso: 01-07-2025. 2011. URL: https://xp123.com/3a-arrange-act-assert/.

[15] Philip Winston. *Beyond Clean Code*. Último acceso: 01-07-2025. 2024. URL: https://tobeva.com/articles/beyond/.