Fundamentos básicos en el desarrollo de software.

Guía básica para comenzar a mejorar tu código y tu carrera

brauliohrdz@gmail.com

V1.0 07/2023

brauliohrdz@gmail.com

Este documento ha sido creado por un desarrollador, para otros desarrolladores, puedes utilizarlo con total libertad para modificarlo o redistribuirlo de la forma que quieras. Cualquier mención al autor original o a este documento será muy bien recibida.

Si te gusta el contenido, tienes alguna sugerencia o quieres contactarme, puedes visitar mi linkedin

https://www.linkedin.com/in/braulio-hernandez-17654560/

NOTA DEL AUTOR

Leemos más código del que escribimos, es algo de lo que no era consciente hasta hace unos pocos años. Durante los primeros años de mi carrera centré mis esfuerzos en aprender cada vez más tecnologías y en intentar dominar conceptos más y más avanzados como procesadores de lenguaje, redes neuronales etc... Llegue a trabajar con bastantes tecnologías diferentes y a conseguir competencias bastante avanzadas en muchas de ellas, pero los proyectos se hacían cada vez más complicados y engorrosos, imposibles de mantener , depurar y a veces incluso de rentabilizar

Investigando para una tarea cualquiera, de un proyecto cualquiera (la verdad es que no recuerdo exactamente cuál fué), comencé a encontrar fragmentos de código claro, limpio , intuitivo. De repente el código más inteligente avanzado y complicado no era el mejor, ni el más apropiado. Fue entonces, cuando descubrí el libro *"Clean Code in Python"* de Mariano Anaya. Éste libro, inició mi curiosidad por hacer las cosas de otra forma, por escribir código para que otras personas pudieran entenderlo. Por intentar que mi código se leyese como un libro y que pudiese comprenderse sin necesidad de dibujar diagramas o realizar complejos cálculos mentales.

Este documento es una recopilación de ideas y de experiencias que he vivido a lo largo de los años y que he recopilado en forma de guía rápida de consulta. Si eres un programador con experiencia, es muy probable que no saques gran cosa de éste documento, pero si no lo eres y quieres mejorar la forma de escribir tu código, ésta guía puede orientarte por el camino que debes seguir.

Ten presente que los conceptos que se presentan se hacen de forma muy básica y superficial y que no se profundiza en ellos. Para hacerlo existe una bibliografía muy buena. En el apartado "bibliografía" podrás encontrar algunos títulos interesantes para adentrarte en el enorme mundo de la ingeniería del software y el código limpio y mantenible.

1 Comunicación sin Bugs

Mejora la forma en la que colaboras con el equipo

Uno de los mayores problemas y los menos detectados que me he encontrado en mi vida profesional, ha sido el de la comunicación. Con los años, he podido observar que muchos de los errores y problemas que surgen en el proceso de desarrollo se deben a una comunicación deficiente, ya sea con el propio equipo, con los usuarios, con los clientes o con cualquier otro agente involucrado en el proceso de desarrollo.

En el libro “El programador pragmático” sus autores, Andy Hunt y Dave Thomas, establecen la idea de que “Tu idioma es otro lenguaje de programación”, para destacar la importancia de comunicarse correctamente con el resto del equipo, como medio para alcanzar el éxito en tu carrera.

Los problemas de comunicación suelen pasar desapercibidos en la mayor parte de los equipos, que comienzan a buscar soluciones esotéricas para resolver contratiempos surgidos de una comunicación deficiente y de la que no son conscientes. Mejorar la comunicación es un proceso de equipo al que, generalmente, no se le da la importancia que merece. A continuación, pasaré a exponer algunas pautas básicas que me han ayudado a entender y trabajar en mis habilidades de comunicación.

Piensa antes de comunicar

Expresar correctamente tus ideas y pensamientos no es siempre una tarea sencilla. Los nervios a ser juzgados, el miedo a equivocarnos o nuestro carácter, son solo algunos de los elementos que establecen barreras en nuestra forma de comunicarnos. Cuando intentas expresar una idea, quieres pedir ayuda o intentas responder a la solicitud de un compañero, piensa. Ten claro lo que quieres decir e intenta utilizar las palabras más precisas posibles para que se entienda aquello que quieres transmitir.

Otro elemento importante a tener en cuenta es el contexto. Cuando vayas a comunicarte con alguien, asegúrate de darle el contexto adecuado sobre aquello de lo que le estás hablando ya que, en muchas ocasiones, cuando trabajamos en un proyecto o una tarea concreta, tendemos a expresar nuestras ideas dando por hecho que la otra parte comprende perfectamente el contexto en el que nos encontramos, sobre todo si lo hemos comentando en algún momento ese mismo día. Expresar las ideas sin proporcionar un contexto adecuado, es desconcertante para la otra persona y

dependiendo de la circunstancia, y puede llegar a percibirse como algo hostil, por ejemplo, si la interrumpimos en su trabajo.

Establece un vocabulario común

Ya hemos hablado de la importancia de pensar antes de comunicarnos y de entender nuestro idioma como otro lenguaje de programación más, entrenando nuestras habilidades comunicativas de la misma forma que entrenamos cualquier tecnología.

Aunque todos manejemos el mismo idioma y, por lo tanto, un vocabulario parecido, en muchos casos no lo manejamos de la misma forma, ni le damos el mismo significado. Podría poner muchos ejemplos a este problema, pero el más frecuente y el que más conflictos ha creado en los diferentes equipos en los que he estado es el de "terminado". Definir cuando una tarea está terminada es probablemente a lo que menos atención se presta. Para cada uno por separado está claro, pero en conjunto no suele ser así. Para un desarrollador, la tarea está terminada cuando han implementado los requisitos y se abre la Pull Request, para el Scrum Master cuando se ha integrado con la rama de Release, para el cliente , la tarea está terminada cuando está en producción, etc. Ésto suele crear conflictos, sobre todo, cuando en el momento de revisar el estado de las tareas, salen a la luz las diferentes versiones de "terminado".

Para evitar esto, lo primero que debemos hacer es establecer un vocabulario común y preciso para todo el equipo y los agentes involucrados en el proyecto. Éste vocabulario, por norma general, es el vocabulario del negocio y, en aquellos límites a los que éste no llegue, será el consenso y el compromiso del equipo los que decidan el vocabulario correcto y preciso. La idea no es comunicarse de forma rebuscada o crear un glosario de términos propio para tu equipo, el objetivo es eliminar la ambigüedad y comunicarnos de forma más eficaz y efectiva.

No mientas y no pongas excusas

Esta pauta tiene mucho que ver con el miedo al fracaso y el síndrome del impostor. Yo mismo he vivido momentos en los que, al cometer errores que han llegado a producción, el miedo a la reacción del equipo ha hecho que ocultara el error e intentase solucionarlo por mi cuenta, provocando que la situación fuese a peor y teniendo posteriormente que justificar mi comportamiento.

Tu equipo debe poder confiar en ti, debe saber que puede contar contigo, con tu profesionalidad y con tu integridad. Deben saber que el código que se construye es lo más robusto que vuestro conocimiento y experiencia permita en cada momento. Ser un buen desarrollador no significa no equivocarse, ser un buen desarrollador es tener la confianza suficiente para admitir tus errores y pedir ayuda para solucionarlos, antes de que lleguen al cliente final.

De la misma forma que tu equipo debe poder confiar en ti, tú debes poder confiar en ellos. Tienes que poder confiar en que puedes pedir ayuda y sentir su apoyo y solidaridad para, entre todos, encontrar solución a cualquier problema o dificultad que surja.

Tu opinión es valiosa.

En reuniones de equipo, es frecuente ver como hay personas que prácticamente no participan. Ésto es aún más frecuente en personas con poca experiencia o que llevan poco tiempo en el equipo.

No olvides que eres un profesional y tu opinión es tan valiosa como la del resto, independientemente de la experiencia y conocimiento de cada uno. Si trabajas en un equipo en el que no se respeta tu opinión , tal vez es que no es un buen equipo. Esto no quiere decir que tengas que tener siempre la razón o que lances cualquier opinión al aire y sin criterio (piensa antes de comunicar), quiere decir que el equipo se enriquece de los diferentes puntos de vista y tu criterio como profesional también lo hará si participas en las reuniones de diseño. Piensa , analiza y, si tienes una opinión, apórtala.

No digas lo que crees que los demás quieren oír (aprende a poner límites)

Con cierta frecuencia, nos vemos inmersos en conversaciones donde se nos piden compromisos concretos, ya sea en funcionalidades, plazos de entrega o ambas. Si la persona que nos pide dicho compromiso es alguien que tiene autoridad sobre nosotros o nuestro trabajo, o alguien a quien respetamos o admiramos de alguna forma, tendemos a entrar en un estado de presión autoimpuesta por contentar y agradar a nuestro interlocutor.

Tienes que tener presente que, cada uno de nosotros tiene un conocimiento, una experiencia y una capacidad y esas cualidades son las que establecen nuestras limitaciones y hasta dónde podemos llegar. Incluso cuando el objetivo que tenemos es progresar y mejorar cada día, también tenemos limitaciones en esa capacidad de progreso.

Aceptar funcionalidades, plazos de entrega o cualquier tipo de hito o petición que escapen a esas limitaciones, solo crea estrés, frustración y desconfianza, tanto en nosotros mismos como en la persona con la que nos hemos comprometido. Cada persona solo puede hacer lo que los límites de su capacidad le permiten y nada más. Exigirnos más allá del limite de nuestras capacidades no suele acabar bien.

Como profesionales debemos cumplir nuestra palabra y nuestros compromisos. Debemos conocer hasta dónde podemos llegar y hasta qué punto podemos estirar nuestras capacidades y plantear un objetivo realista sobre lo que se nos solicita, aunque pensemos que no es lo que nuestro interlocutor quiere escuchar. Como profesional, tu palabra es tu mejor carta de presentación.

No seas un imbécil

Lo único que te hace insustituible es la confianza y el aprecio que puedas llegar a provocar en tu equipo. El conocimiento es relativamente fácil de reemplazar o adquirir, las buenas personas no. Lo que hará que tus compañeros confíen en ti y que puedas confiar en ellos es la forma en la que los tratas, sobre todo en los momentos en los que más te necesitan.

Tuve un compañero que para defender un código enrevesado, ilegible y muy complicado de entender, llegaba a comparar métricas de tiempos de ejecución para justificar por qué no era mejor una solución mucho más legible y comprensible. Generalmente, la mejora de rendimiento era despreciable con respecto al perjuicio que causaba mantener una aplicación repleta de ese tipo de código, intentando resolver un problema de rendimiento que no existía. Su actitud arrogante, hacía que fuese imposible hacerle cambiar de opinión e incluso pudiendo llegar a puntos de conflicto algo más serios. La verdad es que es una persona con un conocimiento y una curiosidad enormes pero, al menos en aquel momento, era imposible trabajar con él.

Trata a tus compañeros con el mismo respeto que quieres que te traten, mantén la mente abierta, sé crítico contigo mismo y ayuda al equipo siempre que puedas. Comunícate con claridad, elegancia y respeto. Busca siempre la solución que beneficie a todos por encima de la que te beneficie a ti mismo, si quieres ser un miembro valioso e imprescindible, ten una actitud que sea difícil de reemplazar.

El idioma es la herramienta que más utilizas en tu día a día. Dominar la forma en la que lo utilizas para comunicarte mejor con tus compañeros es una habilidad fundamental que fortalecerá la colaboración, evitará los malentendidos y supondrá un aumento en la calidad del código, en la cohesión del equipo y te hará estar más satisfecho con tu trabajo.

2. CONCEPTOS BÁSICOS

Conceptos clave que debes conocer

En esta sección se definirán brevemente algunos conceptos fundamentales que debes tener en cuenta para escribir código de mejor calidad. No se hará una revisión exhaustiva de ellos pero se definirán de forma que te faciliten el seguimiento de este manual y tenerlos en cuenta a la hora de escribir tu código.

Encapsulación (ocultación de la información)

Encapsular es ocultar los detalles de implementación de un componente de forma que la aplicación llamante (o cliente) no necesita conocer sus detalles internos de funcionamiento para obtener un resultado. Mediante la encapsulación, la implementación del componente queda oculta y solo queda visible su interfaz pública que puede ser utilizada por otros componentes.

El tipo *Fecha* de cualquier lenguaje de programación es un ejemplo claro de encapsulación ya que por ejemplo, para obtener el año actual no necesitamos saber si el componente guarda la fecha como string, como int (como el timestamp de unix) o como un objeto. Al utilizar encapsulación, protegemos otras partes del código de posibles cambios en la implementación interna del tipo "Fecha".

Abstracción

En desarrollo de software, la abstracción es el proceso mediante el cual analizamos un elemento y descartamos todo lo superfluo, quedándonos únicamente con lo importante y fundamental. Crear una clase, definir una función o incluso elegir un nombre apropiado para una variable son ejemplos de abstracción. En cada caso, nos enfocamos en capturar la esencia y la funcionalidad necesaria, simplificando y representando de manera clara y concisa el concepto o la tarea que deseamos abordar. La abstracción, utilizada de forma adecuada, nos permite construir software más modular, legible y mantenible.

Cohesión

Un componente de nuestro código es cohesivo, cuando tiene un comportamiento concreto, pequeño y bien definido haciendo lo mínimo posible y teniendo sentido por sí mismo. La cohesión promueve la modularidad , legibilidad y la mantenibilidad del código, lo que mejora la eficiencia en el desarrollo

Acoplamiento

El acoplamiento hace referencia a la forma en la que varios componentes interactúan entre sí y la dependencia que se genera entre ellos. Demasiado acoplamiento entre objetos provoca problemas de reutilización, escalabilidad y mantenimiento.

Ortogonalidad

La ortogonalidad es un concepto muy relacionado con la cohesión. Decimos que un elemento del sistema es ortogonal o independiente, cuando al modificarlo no afectamos al resto del sistema.

Dependencia circular (Inapropiada intimidad)

Dos elementos tienen dependencia circular cuando dependen uno del otro entre sí y ninguno puede funcionar por separado. Por norma general, la dependencia circular es un error de diseño que puede dar lugar a multitud de problemas de reutilización, escalabilidad y evolución del software. Aunque existen casos concretos en los que tener una dependencia circular puede estar justificado, por norma general es un indicio de un diseño deficiente.

Deuda técnica

La deuda técnica hace referencia a la acumulación de sobrecoste de trabajo, que se suma a un proyecto debido a decisiones de diseño tomadas sin un análisis adecuado o que son llevadas a cabo de forma descuidada. Cuanta más deuda técnica tiene un proyecto, más difícil se hace su evolución y mantenimiento a lo largo del tiempo. Algunos ejemplos de elementos que conducen a la deuda técnica son:

- Mala nomenclatura de variables funciones o clases.
- Duplicación de conocimiento
- Inconsistencia en el código
- Componentes demasiado acoplados
- Abstracciones prematuras
- Desarrollar funcionalidades no solicitadas por el cliente (YAGNI)
- Soluciones demasiado complejas.

DRY (Don't Repeat Yourself)

También conocido como OAOO (Once and only once), este principio busca hacer incapié en evitar la duplicidad del código a cualquier coste. En muchas ocasiones el objetivo del principio se confunde ya que lo que se pretende es no duplicar el conocimiento que el código representa, no el código en sí. Dos componentes pueden tener el mismo código si representa conocimientos diferentes y por lo tanto, pueden evolucionar de forma diferente.

YAGNI - (You ain't gonna need it)

Este acrónimo quiere decir que hay que evitar a toda costa la *sobre ingeniería del código*, o lo que es lo mismo determinar la importancia de concentrarse única y exclusivamente en los requerimientos existentes y no intentar anticipar necesidades futuras, por muy atractivas o inminentes que puedan parecer.

KISS (Keep it simple and stupid)

Este principio se refiere a la importancia de desarrollar la funcionalidad mínima necesaria que resuelva el problema, utilizando los componentes y elementos más simples que nos sea posible, para de esta forma mantener un diseño sencillo que permita mantener y evolucionar el código con mayor facilidad.

Diseño por contrato (Design by Contract)

Esta idea se aplica a la interacción entre componentes de software, por ejemplo una api y un cliente o entre diferentes componentes del mismo sistema. La idea es establecer unas condiciones de obligado cumplimiento y en caso de que no se cumplan, disparar una excepción que especifique el motivo del error. Por ejemplo, podemos establecer que una función que recupera los datos de un usuario, debe recibir el identificador de dicho usuario y de no ser así, lanzar una excepción. El diseño por contrato busca evitar la entrada de datos inesperados.

Programación defensiva

La programación defensiva trata de proteger las partes del sistema, contra las entradas de datos inesperadas (a diferencia del diseño por contrato que no las permite). La idea

es responder de forma elegante a aquellos errores esperados intentando decidir si el código puede continuar su ejecución o no y tratar aquellos errores que no se deberían producir nunca. Un ejemplo de programación defensiva sería el de comprobar que un archivo existe antes de intentar abrirlo y en caso de error, devolver un mensaje amigable al usuario.

Mejor pedir perdón que permiso (Easier to Ask Forgiveness than permission)

La propuesta de este principio es la de escribir nuestro código dando por hecho que va a funcionar y posteriormente controlar las consecuencias en caso de que no sea así (generalmente mediante el uso de excepciones). Este principio establece el comportamiento opuesto a la programación defensiva. Un ejemplo de este principio sería el , intentar leer el contenido de un archivo sin comprobar si el archivo existe, si no existe capturaremos la excepción, asumiendo que, en la mayor parte de los casos, el archivo existirá.

Profiling

El profiling , “perfilaje” o “análisis de rendimiento” es el proceso por el cual intentamos cuantificar la cantidad de recursos que consume una determinada sección del código que estamos escribiendo.

Dominio

Definimos como dominio, a todos los detalles relacionados con el problema del negocio que estamos intentando resolver. Este dominio incluye única y exclusivamente los detalles relacionados con los requisitos, los casos de uso y el problema que intentamos modelar y resolver, dejando a un lado cualquier detalle técnico lenguajes de programación, frameworks o cualquier otra tecnología.

Modelo del Dominio

El modelo del dominio es la representación de las partes relevantes y fundamentales del problema que intentamos resolver y la interconexión que se produce entre los componentes. En el modelo de dominio tampoco se incluyen detalles técnicos ni de implementación (Lenguajes, Frameworks, arquitecturas...)

Funciones de orden superior

Las funciones de orden superior (o en inglés, first-class functions) son una característica de algunos lenguajes de programación, que permiten que algunas funciones tomen otras funciones como argumentos y devuelvan una función como resultado.Esta capacidad de manipular funciones de manera dinámica permite aplicar parcialmente funciones y simular el almacenamiento del estado del sistema en lenguajes funcionales. Las funciones de orden superior son una poderosa herramienta para la construcción de software expresivo, modular y flexible.

Arquitectura Hexagonal

La arquitectura hexagonal (también conocida como arquitectura de puertos y adaptadores) establece una estructura que permite ordenar las dependencias de un sistema según las diferentes capas en las que se compone. Este tipo de arquitectura establece el orden de prioridad de “fuera hacia adentro” en 3 capas

- **Infraestructura**: Es la capa externa de la arquitectura y hace referencia a todo lo relacionado con la tecnología utilizada (lenguaje de programación, framework, librerías, sistemas de caché...
- **Aplicación**: Es la capa intermedia en la que se encuentran los casos de uso del sistema
- **Dominio**: Es el núcleo de la arquitectura que contiene toda la lógica de negocio, esta capa es totalmente independiente de las dos anteriores y no conoce ningún detalle de su implementación ni depende de ellas.

En esta arquitectura la capa de infraestructura depende de la capa de aplicación y la capa de aplicación depende de la capa de dominio lo que establece la dependencia en el sentido **Infraestructura ⟶ Aplicación ⟶ Dominio**

3. Principios Fundamentales

Pilares para el desarrollo de aplicaciones de calidad.

En esta sección te presento una serie de pautas que me ayudan a escribir mejor código en mi día a día. Como todo lo de este manual, son pautas básicas que pueden servirte como guía para que establezcas tu propia filosofía y metodología.

Asegurate de repartir bien las responsabilidades (Cohesión)

Idealmente cada componente debe tener un única responsabilidad o, lo que es lo mismo ,“una funcionalidad bien definida y delimitada”.

Determinar con exactitud lo que es una responsabilidad no es una tarea sencilla, de hecho es de las cosas más complicadas en el diseño de software. Para poder dividir responsabilidades debes conocer a la perfección el negocio que intentas modelar y de esta forma podrás determinar dónde se encuentran los límites de una responsabilidad.

Cada responsabilidad debe separarse en diferentes componentes, de forma que cuando debamos realizar modificaciones éstas se encuentren bien definidas y acotadas.

Como se ha dicho anteriormente, reconocer una responsabilidad no es una tarea sencilla por lo que, si tienes dudas de si debes dejar un componente como está o partirlo en dos… no lo dividas. Si en algún momento detectas que, al realizar diferentes cambios debes modificar el mismo componente, entonces será el momento de refactorizar y repartir la responsabilidad. Recuerda que el objetivo es que el software sea lo más sencillo posible, dividir elementos en exceso puede llevarnos a una complejidad innecesaria y a conseguir lo contrario de lo que pretendemos. Mediante el reparto efectivo de responsabilidades buscamos conseguir que:

- El código es más robusto ya que evitamos la propagación de errores entre diferentes componentes (Ripple Effect)
- El código se vuelve más fácil de modificar
- Los cambios en una funcionalidad no afectan al resto de componentes (siempre que no se modifique la interfaz pública)

Una buena nomenclatura puede ayudarnos a distinguir cuando un elemento tiene más de una responsabilidad. Por ejemplo si cuando damos nombre a una clase o una función debemos utilizar conjunciones en su nombre (‘y’, ‘o’), esto nos da una pista clara de que dicho elemento probablemente deba ser separado en múltiples elementos.

No hay buen código si no hay tests

Los test automatizados nos ayudan a construir código robusto y de calidad y además nos dan la seguridad de que los cambios que realizamos no afectan inesperadamente a

nuestro sistema. Esto , entre otras cosas, facilita la refactorización y la mejora contínua en la evolución del software en el que estamos trabajando. Un código sin una buena cobertura de test, es débil y propenso a la introducción de errores accidentales. Muchos desarrolladores creen que los test automáticos hacen que el desarrollo sea más lento y que les perjudica en los tiempos de entrega y que pueden sustituirse por pruebas manuales. La realidad es que los test proporcionan una red de seguridad que, en muy corto plazo, hacen que tu proceso de desarrollo sea más seguro, ágil y eficiente.

Por otro lado, hay que tener presente que el código de los test no es código de segunda y debe cuidarse en la misma medida que el resto de la aplicación aunque siguiendo unos principios ligeramente diferentes, que veremos más adelante en este mismo documento.

El código debe ser legible

Pasamos mucho más tiempo leyendo código que escribiéndolo, por eso, debemos intentar escribir código que se lea fácilmente, sin necesidad de comentarios o aclaraciones complejas. Si nos vemos en la necesidad constante de comentar nuestro código, es probable que el problema sea que no estemos eligiendo los nombres o las abstracciones apropiadas o que nuestro diseño sea demasiado rebuscado. Cuando escribas código intenta hacerlo como si supieras que otra persona va a leerlo y a juzgarlo, aunque esa persona seas tú mismo en un futuro no muy lejano.

El creador de Python Guido Van Rossum, fue entrevistado a su salida de Dropbox por cómo había sido su experiencia en el proyecto y con respecto a la legibilidad del código dijo lo siguiente .

“Había un reducido grupo de programadores muy listos y muy jóvenes que generaban una gran cantidad de código muy ingenioso que solo ellos podían comprender. [...] Cuando me preguntaban, solía dar mi opinión de que el código mantenible es más importante que el código ingenioso. [...] Si me encontraba con código ingenioso pero particularmente críptico y tenía que hacerle algún mantenimiento, optaba por reescribirlo. De esa manera, daba ejemplo tanto a través de mi trabajo como mediante conversaciones con otras personas.” [1]

El código debe leerse como un libro y su intención debe quedar clara sin necesidad de comentarios o de computar resultados mentalmente.

[1] Puedes leer todo el artículo en: https://blog.dropbox.com/topics/company/thank-you--guido

No hay buen código si lo escribe una sola persona.

Este punto no habla de la calidad del código que pueda escribir una persona en orden de eficacia, eficiencia o incluso de cobertura de tests. Este punto habla de la legibilidad y mantenibilidad del mismo ya que realizar prácticas como las revisiones de código o el *pair programing* pueden ayudarnos a detectar aquellas partes cuya legibilidad o diseño sean más complicados y de esta forma ayudarnos a mejorarlo. Si escribimos código que no enseñamos a nadie, es probable que sea código que solo entenderemos nosotros mismos. Las palabras del creador de python citadas en el punto anterior pueden aplicarse también en este principio.

No te quedes con la primera solución

Generalmente la primera solución que se nos ocurre no es la mejor. No hace falta decir que lo más importante al escribir código es que sea eficaz, es decir, que resuelva el problema en un tiempo razonable y con una cantidad de recursos aceptable y proporcional a la complejidad del problema. Sin embargo, una vez llegamos a la primera solución viable, es una buena práctica el analizarla e intentar refactorizar para encontrar soluciones más óptimas o mejor diseñadas. Normalmente no suelo preocuparme por la estructura del código en la primera solución que genero, solo quiero que funcione. Una vez tengo esta primera versión (que suele ser tosca, poco elegante y de legibilidad cuestionable), empiezo a refactorizar y a mejorar el código respaldado por los tests. Las iteraciones sobre el código son las que me van ayudando a crear abstracciones y separar responsabilidades de forma que, en algún momento ya no parece necesario modificarlo más.

Por otro lado, hay que pensar bien las refactorizaciones y las nuevas abstracciones y asegurarnos de que no estamos “partiendo de más” nuestro código ya que esto le añadiría complejidad

Controla el acoplamiento

Como se ha mencionado en el capítulo anterior. El acoplamiento es el nivel de dependencia que tienen dos componentes entre sí. El acoplamiento en sí mismo no es algo malo, los sistemas complejos se construyen acoplando elementos sencillos unos a otros. El problema viene cuando los componentes dependen en exceso unos de otros y no somos capaces de delimitar la forma en que interactúan para poder controlar sus relaciones.
Mantener el acoplamiento al mínimo es un arte que requiere de práctica y

refactorización. Pero como primero paso para controlar tu acoplamiento puedes tener en cuenta algunas características de tu código

1. Vigila las dependencias circulares: Si llegas a un punto en el que dos elementos dependen uno del otro, lo más probable es que haya un error de diseño y un acoplamiento excesivo
2. "Cirugía de escopeta"[2]. Este antipatrón o smell code puedes verlo cuando, al realizar un pequeño cambio en un componente, te ves en la necesidad de modificar otros muchos componentes para que funcionen correctamente.
3. Llamadas encadenadas. Salvo en casos excepcionales como cuando utilizamos un ORM, cuando tenemos muchas invocaciones encadenadas (más de un punto en la misma sentencia ej : *objeto.metodo().metodo()*), esto significa que el componente cliente, o llamante está acoplado a la implementación concreta del componente servidor. Para evitar esto puedes intentar seguir la Ley de Demeter[3].

No escribas tu código con la intención de reutilizarlo.

La intención principal al escribir código debe ser garantizar su correcto funcionamiento y su legibilidad. Si escribimos código con la intención de reutilizarlo, aumentamos la probabilidad de utilizar nombres genéricos y agregar lógica innecesaria, lo cual dificulta el mantenimiento del sistema. La reutilización del código debe surgir de la refactorización y la correcta aplicación del principio DRY (Don't Repeat Yourself), evitando duplicar conocimiento innecesariamente. Existe una pauta que indica que no debemos crear una abstracción hasta haber reutilizado el mismo código al menos 3 veces aunque no lo utilizo como norma estricta si que me sirve como criterio (entre otros) para abstraer una pieza de código.

No copies y pegues código.

Cuando copiamos y pegamos código, ya sea dentro del mismo proyecto o entre proyectos diferentes, estamos tomando una parte del conocimiento y moviendola de sitio (tal vez duplicándolo). Si no hemos leído y comprendido perfectamente lo que necesitamos conseguir y por qué estamos copiando y pegando ese conocimiento, es bastante probable que estemos introduciendo complejidad (y acoplamiento), duplicando ese conocimiento y haciendo que el código sea menos legible.

---

[2] https://refactoring.guru/es/smells/shotgun-surgery
[3] https://www.adictosaltrabajo.com/2015/07/24/ley-de-demeter/

Si necesitas reutilizar código en la misma aplicación, asegúrate de que no estás duplicando el conocimiento que representa ya que, tal vez, lo que necesitas es refactorizar. Si este no es el caso, no pegues el código, reescríbelo y asegúrate que los nombres de variables y abstracciones utilizados, tienen sentido en el dominio en el que estás trabajando.

Código limpio vs código eficiente

Por norma general, el llamado código limpio, suele ser menos eficiente que un código centrado simplemente en la eficacia y la eficiencia. En aplicaciones empresariales y de gestión en las que la mayoría de nosotros trabajamos, la pérdida de eficiencia de un código más organizado y con más abstracciones a un código más directo suele ser despreciable en la mayoría de los casos y de los proyectos. Sinceramente , nunca me he encontrado con un problema de rendimiento debido a que he utilizado algunas clases o abstracciones de más. Los problemas de rendimiento generalmente están en otro sitio, como por ejemplo, en las consultas a bases de datos, accesos a disco etc..

En la red y la bibliografía, puedes encontrar toneladas de información sobre si un código es mejor que otro. Nunca he entendido ni me han gustado estas “guerras” y este caso no es una excepción. Para mi la elección está clara

- Prioriza la legibilidad del código en todas aquellas partes de la aplicación que se modifiquen con mayor frecuencia y en donde el rendimiento no es , a priori, un problema. Por norma general este será el mayor porcentaje de tu código. Por ejemplo en las vistas, los controladores o los servicios.

- Prioriza la eficiencia del código en todos aquellos lugares donde la eficiencia es algo crítico y el código debe funcionar de la forma más eficiente, por ejemplo en el acceso a base de datos, en los cálculos de DataSets complejos...

Prioriza no quiere decir ignorar la otra parte, quiere decir que si tienes que elegir entre una opción u otra, elijas la más adecuada según la capa del software en la que estés trabajando.

4. Cómo escribir código mantenible

Introducción al arte de programar con claridad

En esta sección por fin llegamos a lo que probablemente estabas esperando, los ejemplos de código. Escribir código claro, legible y elegante es todo un arte, requiere entrenamiento y conocimiento a parte de autocrítica y ganas de mejorar y evolucionar. Los siguientes consejos y ejemplos no son reglas inamovibles que debas seguir a rajatabla (al menos no todas) Son pautas que buscan hacerte pensar sobre el código que escribes y que no seas un simple autómata que teclea sentencias.

Variables y Constantes

Utiliza nombres descriptivos

Ya sea para declarar variables, constantes o funciones, utiliza nombres que describan, de la forma más precisa posible, el elemento que estás definiendo. Elegir nombres claros y descriptivos no es una tarea sencilla, pero es un elemento muy importante al escribir código de calidad. Evita elegir nombres genéricos que no dicen nada, siempre que puedas utiliza el vocabulario del negocio para nombrar tus variables y constantes. Veamos el siguiente ejemplo.

```
for item in element.items :
       total += item.calculate_total()
```

Al leer el código anterior, no podemos conocer el dominio del problema que estamos modelando. Podemos entender que es lo que hace el código de forma general (recorrer un iterable y sumar sus valores en un acumulador), pero en caso de error necesitaremos conocer todo el código en cuyo contexto se ejecutan estas líneas, para poder comprender el problema que intenta resolver. ¿ Y si te dijera que este código suma el total de los elementos de un carro de la compra de un e-commerce ? ¿Lo habrías averiguado con el código anterior ? Probemos a reescribirlo de forma más legible

```
for cart_product_line in cart.product_lines:
      cart_total_amount += cart_product_line.calculate_total_amount()
```

El código anterior se lee prácticamente como el lenguaje natural, podemos ver claramente qué problema resuelve sin necesidad de leer el contexto en el que se ejecuta.

A continuación listaremos algunas pautas para elegir mejores nombres:

- **Escribe los nombres en inglés:** Ya que la mayor parte de los lenguajes de programación están escritos en el idioma de Shakespeare, es una buena práctica que lo utilices para nombrar tus abstracciones**.**
- **Utiliza nombres concretos del dominio:** Evita palabras genéricas como “object”, “item”, “element”, etc ...
- **Utiliza nombres fáciles de pronunciar** : no acortes los nombres comiéndote letras, si esto hace que sean menos legibles.
- **No utilices acrónimos:** Los acrónimos hacen que el código sea mucho más difícil de leer para aquellos que no los conozcan, o para nuevos miembros del equipo.
- **Los nombres pueden ser tan largos como sea necesario**: Es mejor un nombre largo y descriptivo que un nombre corto y confuso. Tampoco es bueno que el nombre de demasiados detalles de implementación por ejemplo “user_password” será un nombre adecuado mientras que “user_password_encrypted_sha256” es un nombre que proporciona demasiados detalles.
- **Nombra los booleans como si les fueras a preguntas :** utiliza los prefijos; “is”, “has”, “does”, “are”, “contains”,”should”, etc... (por ejemplo is_active, has_perms ...)
- **Utiliza sustantivos para nombrar clases módulos y paquetes**
- **Utiliza verbos para nombrar métodos o funciones**

No utilices nombres distintos para definir el mismo concepto

Si ya has definido un concepto con un determinado nombre, utilízalo en todos los contextos donde se represente el mismo concepto. Por ejemplo, si estamos en el contexto de un e-commerce y hemos decidido utilizar el término “client” para referirnos a los clientes, utiliza este nombre todo el tiempo y no lo sustituyas por “customer” , “buyer” , “user”, etc... , No hagas lo siguiente

```
def create_client(client_data):
        # lógica para crear el cliente   ...

def remove_customer(customer_data):
        # lógica para eliminar al cliente…
```

Una vez asociado un concepto del dominio a un nombre, utiliza siempre dicho nombre. Utilizar diferentes nombres para el mismo concepto, hace nuestro código más confuso, difícil de modificar y propenso a la introducción de errores accidentales.

```
def create_client(client_data):
        # lógica para crear el cliente   ...

def remove_client(client_data):
        # lógica para eliminar al cliente...
```

Da nombre a los valores literales, evita los “magic values”

Cualquier valor o cadena literal, debe tener un nombre que explique su significado para hacer que el código sea más legible, veamos un ejemplo

```
def has_expired(creation_date_in_seconds):
      now = int(time.time())
      return (now - creation_date_in_seconds ) > 10080
```

Aunque podemos entender la funcionalidad del código anterior ¿ Cuánto tiempo debe pasar para que “*has_expired*” devuelva True ? Tendríamos que sacar la calculadora para verlo. Nombrando este “*magic value*” la función se vuelve mucho más legible y clara.

```
def has_expired(creation_date_in_seconds):
      ONE_WEEK_SECONDS = 10080
      now = int(time().time())
      return (now - creation_date_in_seconds ) > ONE_WEEK_SECONDS
```

Evita los nombres redundantes y la información innecesaria

No añadas información innecesaria a los nombres de las variables o parámetros ya que ensucian el código y no facilitan su lectura.

```
def login_user(user_username, user_password):
    # ...
```

Añadir el prefijo *user_* a los parámetros de la función no aportan información y son redundantes

```
def login_user(username, password):
    # ...
```

No declares todas tus constantes al principio del fichero o clase

Por norma general, al declarar constantes, tendemos a situarlas al principio del fichero en el que estamos trabajando. Esto es buena idea si la constante se utiliza a lo largo de todo el fichero pero ... ¿qué pasa si el ámbito de uso es más reducido... por ejemplo a dos métodos de una clase? Cada vez que queramos leer el valor de la constante deberíamos hacer scroll lo que hace el código más difícil de leer y además, en lenguajes como python, tendríamos declarada esa constante en algún lugar de la memoria cuando tal vez no accedemos a ella en la ejecución actual (Recuerda que priorizar legibilidad frente a eficiencia no significa que la ignoremos).

En el momento de declarar una constante, hazlo siempre desde el ámbito más reducido al más general según su uso. Por ejemplo, si la constante se utiliza solamente en una función, declarala en esa función, en el lugar más próximo a la primera línea en la que se utiliza. Posteriormente podremos ampliar su alcance al de la clase, módulo o paquete, según sea necesario.

Funciones

Cada función debe tener una única responsabilidad

Sé que ya he mencionado esto varias veces y lo volveré a mencionar en este documento. Entender correctamente el reparto de responsabilidades ha cambiado radicalmente mi manera de desarrollar software y la comodidad para leer y modificar mi código. Es un elemento básico que debemos interiorizar e intentar aplicar en nuestro día a día pero vamos a algo más concreto ¿Qué significa que una función solo debe tener una responsabilidad ?

Este principio está contenido dentro de los principios SOLID, y tanto en manuales como en tutoriales normalmente se aclara diciendo que *"debe tener un único motivo para cambiar"*, a mi, siendo sinceros, esto no me aclara nada. Pongamos un ejemplo en el que tenemos una función "*signup_user*" que se encarga del registro de un usuario. ¿Es el registro de un usuario una responsabilidad única? En principio parece que si, no está registrando un usuario y encargando una pizza, lo "único" que hace es registrar al usuario pero ... ¿Qué significa registrar al usuario? Veamos un ejemplo en código:

```
def signup_user(email:str, first_name:str, last_name:str)-> None:

    try:
        user = User.create(
            email=email
            first_name=first_name,
            last_name=last_name,

            )

        letters = string.ascii_lowercase
        password = ''.join(random.choice(letters) for i in range(10))
        user.password(sha256(password))
        user.save()

        send_email(
            subject="Tu registro en example.com",
            mail_from="no-reply@example.com",
            mail_to=[email],
            mail_content=WELCOME_EMAIL_TEMPLATE % {
                "full_name", f"{first_name} {last_name}",
                "pasword": password
            })
    except UserAlreadyExists as e:
```

```
        logger.error(str(e))
    except EmailEnqueueError as e:
        logger.error(str(e))
```

En el ejemplo anterior ... ¿Tiene la función *signup_user* una única responsabilidad? La respuesta, en este caso sería ... no. La función se encarga de:

1. Almacenar los datos del usuario en la base de datos.
2. Generar una contraseña automática para el usuario.
3. Enviar el correo de confirmación.

Todos estos pasos forman parte del proceso de registro del usuario por lo que podríamos concluir erróneamente que la función, efectivamente, cumple una única responsabilidad pero ¿No conoce la función signup user demasiados detalles de la implementación? ¿Qué pasa si queremos cambiar la plantilla del correo ? o si queremos que el usuario introduzca el mismo su propia contraseña ? ¿Y si decidimos que el usuario debe indicar un número de teléfono para poder registrarse? Todo esto son motivos de cambio para la función que nos indica que conoce demasiados detalles sobre los diferentes pasos del proceso de registro. Todos estos motivos de cambio rompen el principio de responsabilidad única. Vamos a reescribir el código separando la responsabilidad

```
def generate_password()->str:
    letters = string.ascii_lowercase
    password = ''.join(random.choice(letters) for i in range(10))
    return sha256(password)


def create_user_with_password(email:str, password_hash:str, first_name:str=None,
last_name:str=None, )->User:

    return  User.create(
        email=email
        first_name=first_name,
        last_name=last_name,
        password=password

    )

def send_confirmation_email_for_user(user:User)->None:
```

```
    send_email(
        subject="Registro en example.com",
        mail_from="no-reply@example.com",
        mail_to=[user.email],
        mail_content=WELCOME_EMAIL_TEMPLATE % {
            "full_name", f"{user.first_name} {user.last_name}",
            "pasword": user.password
        })


def signup_user(username:str, email:str, first_name:str, last_name:str) -> None:
    try:
        password = generate_password()
        created_user = create_user_with_password(email, password, fisrt_name,
last_name)
        send_confirmation_email_for_user(created_user)
    except UserAlreadyExits:
        logger.error(str(e))
    except EmailEnqueueError:
        logger.error(str(e))
```

Una vez reescrito el código, la función *signup_user*, sigue teniendo la responsabilidad de crear al usuario y por lo tanto de recopilar todos los pasos necesarios para hacerlo (crear la contraseña, almacenar los datos, enviar el correo de confirmación), sin embargo hemos encapsulado los diferentes detalles de este proceso en otras funciones sin que el método "*signup_user*" conozca ningún detalle interno de las mismas.

¿Para qué nos sirve esto? Para empezar hace el código más legible y facilita la creación de test unitarios, ya que con la segunda versión del código podemos probar cada paso por separado. Además hace nuestro código más robusto y nos permite que las diferentes partes del proceso evolucionen de forma independiente del resto (Siempre que mantengamos intacta su interfaz pública) lo que significa que nuestro código tiene ahora partes separadas más cohesivas, y este debe ser siempre nuestro objetivo.

Una vez más hay que mencionar que hay que tener cuidado al separar responsabilidades ya que podemos caer en el error opuesto, dividir una responsabilidad que debería ser única en dos componentes diferentes. Podemos identificar este error, si siempre que utilizamos uno de los dos elementos, estamos obligados a utilizar también el otro. (no tienen sentido el uno sin el otro).

No devuelvas diferentes tipos de datos en una función

Aunque los lenguajes de tipado dinámico como python y javascript permiten devolver cualquier tipo de valor en una función, esto puede no ser una buena práctica. Por ejemplo, supongamos que tenemos una función que intenta recuperar datos de un recurso externo, por ejemplo ... un feed y tenemos el siguiente código.

```
def load_feed(feed_url):

    try :
        response = request_external_feed(feed_url)
        if response.status_code == HTTP_200_OK:
            return response.content
        else:
            return False
    except Exception as e:
        return "Error"
```

En caso de que todo vaya bien y la url exista, el sistema devolverá el contenido del feed en formato string, si el contenido no existe , devolverá un booleano y en caso de error devolverá un string. Este diseño tiene diversos problemas; Por una parte, el llamante debe conocer los detalles internos de la función para saber que tipo de dato devuelve en cada caso. Por otro lado, la cadena “Error” es un valor que puede interpretarse como True, otro detalle que debe conocer el llamante y que puede de no ser así, puede dar lugar a errores.

Creo que un mejor diseño, sería asumir que la mayor parte de las veces, el recurso externo que vamos a recuperar, es un recurso válido (de no ser así tendríamos un problema de diseño ya que la mayor parte de las veces se devolvería algún tipo de error que hay que controlar) por lo que la lógica a ejecutar cuando el feed no existe deberíamos delegarla a la aplicación llamante.

```
def load_feed(feed_url):
    response = request_external_feed(feed_url)
    if response.status_code == HTTP_200_OK:
        return response.content
    raise FeedRequestException(
```

```
        f"Error trying to reach the feed {response.status_code}" )
```

No aceptes el caso excepcional como una valor válido

Para ilustrar esta pauta, mostraremos un ejemplo de código, que no solo he utilizado yo mismo, sino que también me lo he encontrado con bastante frecuencia.

```
try:
    product_discount = product_row[3]
except:
    product_discotun = None
```

En el ejemplo anterior, tenemos una lista o array del que no conocemos su longitud exacta, por ejemplo porque podemos leerlo de un fichero csv, que puede tener diferentes formatos. Para evitar el error de que el índice pueda estar fuera de rango usamos una excepción vacía y asignamos un valor a la variable.

En este caso, la excepción actúa como un control de flujo en el que, sí existe el índice, lo asigna a la variable y si no , le asigna un valor por defecto (en este caso None). Como veremos más adelante, tanto por motivos de legibilidad como de rendimiento, las excepciones deben ser utilizadas como mecanismo de control de errores para casos excepcionales.

Si asumimos que el número de elementos en un “product_row” siempre va a ser pequeño , podríamos reescribir el código anterior de la siguiente forma

```
product_discount  = product_row[3] if len(product_row) > 3 else None
```

Podemos asumir que la primera implementación cumple el principio de *“Mejor pedir perdón que permiso”* ya que se asume que el código va a funcionar y luego tratamos la excepción. En mi opinión, esto dependerá del número de veces que esperemos que se

produzca el error. Si el error es frecuente entonces en un caso de control de flujo y no deberíamos tratarlo como una excepción.

Evita tener funciones con demasiados parámetros.

Tener una función con demasiados parámetros puede ser un indicio de alguna deficiencia en nuestro diseño. Una función con muchos parámetros probablemente indica un acoplamiento excesivo al código desde el que se llama, o tal vez, puede ser un signo de que dicha función tiene demasiadas responsabilidades, o ambos. Además, tener funciones con demasiados parámetros llevan a otros problemas, suelen ser poco legibles, es difícil recordar el orden de los parámetros y el código es más complicado de testear.

```
def create_user(username, password, first_name, last_name, street, city,
       postal_code, address, email, phone):
       # ...
```

Por norma general una función debe tener como máximo cuatro parámetros. En caso de exceder este número, tal vez signifique que los parámetros pueden agruparse en objetos que posteriormente pueden pasarse a la función o que la función puede dividirse en varias de ellas. En el ejemplo anterior hemos unido ambos escenarios, una mejor versión de este código sería

```
@dataclass
class User:
   username : str
   password : str
   first_name: str
   last_name : str

@dataclass
class Address:
   street: str
   city: str
   postal_code: str
   country : str
```

```
@dataclass
class ContactData
   email : str
   phone : str


def create_user(user:User):
    # ...

def create_user_address(user_id:int, address:Address):
    # ...

def create_user_contact_data(user_id:int, contact_data:ContactData):
    # ...
```

En esta nueva versión no solo hemos reducido los parámetros de la función agrupándolos en clases más cohesivas, sino que además hemos dividido la responsabilidad de la función original.

Con bastante frecuencia, suelo encontrar códigos en los que, para no enviar muchos argumentos a una función, se agrupan en un diccionario. Debemos tener cuidado al intentar agrupar los parámetros utilizando diccionarios o estructuras de datos dinámicas de este tipo ya que, por un lado, no sabemos exactamente cuales son los valores que contendrá dicho diccionario sin leer todas las partes del código donde se llame a esta función y además añadimos la posibilidad de enviar valores no esperados que puedan provocar comportamientos indeseados. Debemos intentar establecer un contrato entre las funciones que creamos y los clientes que las utilizan.

No utilices flags para modificar el comportamiento de una función

Si necesitamos modificar el comportamiento de una función mediante el paso de parámetros, estamos rompiendo el principio de responsabilidad única, ya que asumimos que la función puede tener comportamientos diferentes según el valor de dichos parámetros (tienen más de un motivo para cambiar)

```
def calculate_amount_with_taxes(amount:Money, using_igic:bool):

    if using_igic :
       return amount + amount * 0.7
    else:
       return amount + amount * 0.21


 amount_with_taxes = calculate_amount_with_taxes(cart_amount, True)
```

Podemos hacer el código anterior más legible cuando llamamos a la función calculate amount si dividimos la responsabilidad

```
def calculate_amount_with_igic(amount:Money):
   IGIC = 0.07
   return amount + amount * IGIC

def calculate_amount_with_iva(amount:Money):
   IVA = 0.21
   return amount + amount * IVA

amount_with_taxes = calculate_amount_with_igic(cart_amount)
```

No fuerces un único return por función

Si cuando encontramos un resultado para una función, esperamos para devolverlo en la última línea de ejecución de la misma, estamos arriesgándose a que esa variable sea modificada por el camino o a que se produzca un error inesperado que evite la devolución del valor correcto. Cuando intentamos mantener un único return por función nos vemos obligados a utilizar condicionales y estructuras de control que complican el código y su legibilidad.

La recomendación de tener un único punto de retorno proviene de los inicios de la programación estructurada para mejorar el seguimiento del flujo de las aplicaciones y evitar errores en la gestión de memoria. Lo que hoy en día no tiene demasiado sentido.

COMENTARIOS

No comentes todo tu código

Hace algunos años se consideraba una buena práctica el comentar todo tu código y aclarar cada decisión de diseño que tomaras para que fuera comprensible para el próximo que pasase por allí (o para uno mismo unos meses más tarde). La realidad es que estos comentarios pasan a ser código que mantener y generalmente no aportan nueva información y acaban quedando perdidos y obsoletos olvidados cuando se modifica el código, lo cual generalmente empeora su legibilidad y ayuda a la introducción de errores.

Lo mejor es comentar sólamente aquellas partes de código que sea necesario y que no puedan explicarse simplemente mediante una buena nomenclatura bien porque son algoritmos complejos, porque son un “workaround” de un bug de la infraestructura etc...
Este principio no incluye aquellos comentarios que se generan para el uso de herramientas automáticas para generar documentación, por ejemplo “Swagger”. Aunque también hay que tener cuidado al generar comentarios para este tipo de herramientas.

No escribas comentarios tipo TODO

En mi experiencia, los comentarios *TODO* intentan registrar dos cosas. O bien ideas de funcionalidades potenciales no solicitadas y que podrían ser interesantes de implementar (YAGNI) o por otro lado, registrar carencias en el momento de escribir el código asumiendo que en algún momento del futuro sacaremos tiempo para escribirlo mejor. Estos comentarios solo ensucian el código y no aportan nada. En lugar de escribir comentarios de este tipo puedes

- Si es una funcionalidad que podría ser interesante, crea una tarea en tu backlog, de forma que el equipo pueda analizarla (o puedas presentarla al cliente)
- Si hay algún tipo de carencia en el código, refactoriza y corrígela, es probable que nunca tengas tiempo para arreglarlo en el futuro.

No escribas comentarios tipo FIXME

Utilizar FIXME, es asumir deuda técnica y aceptar que el código incompleto o de mala calidad es aceptable. Cuando tomamos decisiones de este tipo, estamos, por un lado, ensuciando el código con comentarios que no aportan nada a su compresión y por otro lado asumimos un cierto grado de deuda técnica como aceptable. En lugar de utilizar comentarios de este tipo, refactoriza el código y elimina la deuda técnica y si , por cualquier motivo no puedes asumir ese coste en tiempo, crea una tarea en Jira o tu gestor de tareas para que quede constancia de la necesidad y pueda ser resulta en el futuro.

No escribas comentarios redundantes

Los comentarios que no aportan información solo ensucian el código. Los siguientes comentarios son claramente innecesarios ya que no nos aportan ninguna información adicional que no podamos ver en el código.

```
# Hacer login con google
def login_user_with_google(user):
   # ...

class UserLoginForm(Form):
  """
    Formulario de inicio de sesión
  """
```

No mantengas comentado el código obsoleto

No comentes una funcionalidad como medida para eliminarla. Si vas a comentar alguna parte del código ya sea porque ha quedado en desuso o porque la has refactorizado, asegúrate de eliminar el código comentado antes de subirlo al repositorio o a producción. Si en algún momento quieres ver la versión anterior (la que has eliminado) siempre podrás hacerlo mediante tu herramienta de control de versiones.

5. Construyendo con Objetos

Introducción básica a la programación orientada a objetos

La programación orientada a objetos, surge con la necesidad de crear sistemas cada vez más complejos y que fueran más fáciles de diseñar , mantener y evolucionar. La programación orientada a objetos busca mejorar la comprensión y la escalabilidad del software haciendo uso de la encapsulación, abstracción, modularidad y jerarquización. Además de intentar reducir los costes de mantenimiento y desarrollo.

Ten presente que esto es una simple introducción y que la programación orientada a objetos es un paradigma de programación complejo, con mucha bibliografía al respecto.

Fundamentos de la POO

La programación orientada a objetos pretende construir sistemas de software complejos mediante la creación de un universo de clases y objetos cohesivos e independientes que se relacionan entre sí.

- **Una clase** es una descripción de los datos y las operaciones que describen el comportamiento de un cierto conjunto de elementos (objetos). Es algo similar a un plano de construcción o a un patrón de costura ya que sirve como plantilla para crear elementos similares.
- **Un objeto** es un ejemplar concreto de una clase, es decir. Es una clase cuando se han asignado valores concretos a sus atributos (un estado).

Las relaciones que se producen en un software que utiliza este paradigma de programación puede ser de dos tipos

- **Relaciones entre clases**
    - Herencia
- **Relaciones entre objetos**
    - Composición
    - Agregación
    - Asociación
    - Uso / Dependencia

Relaciones entre objetos.

Composición

La relación de composición se produce cuando un objeto de la clase B es atributo de la clase A. Decimos que B forma parte de la estructura básica de la clase A y su ciclo de vida está ligado al de A

Por ejemplo, imagina que queremos representar un coche de manera sencilla. Los coches tienen un motor que forma parte de su estructura básica por lo que las relaciones entre las clases Car y Engine será de composición.

```
class Engine:
    def start(self):
     ...

class Car:
    def __init__(self):
        self._engine = Engine()

    def start(self):
        self._engine.start()
        ...
```

Agregación

La relación de agregación es una relación muy similar a la composición de forma que, si tenemos un objeto B que forma parte de la clase A, el objeto B no necesariamente tiene que nacer con A ni morir con él. El objeto B puede compartirse con otros objetos. En la agregación los objetos agregados pueden irse añadiendo durante el ciclo de vida del objeto A.

Por ejemplo, imaginemos que tenemos que modelar una Empresa. Los empleados forman parte de la estructura básica de la empresa, pero no nacen ni mueren con ellas. Por lo que las relaciones entre las clases Company y Employee podría modelarse con agregación de la siguiente forma.

```
class Employee:
    def __init__(self, name):
        self.name = name

    def display_info(self):
```

```
        print("Empleado:", self.name)

class Company:
    def __init__(self, name):
        self.name = name
        self.employees = []

    def hire_employee(self, employee):
        self.employees.append(employee)
```

Los objetos Employee son agregados al objetos *"Company"*, cuando *"Company"* recibe el mensaje *"hire_employee"* con el empleado que debe contratar.

Asociación

La asociación es una relación duradera entre un objeto cliente y un objeto servidor. Existe una relación de asociación entre A (cliente) y B(servidor) si un objeto de la clase A utiliza frecuentemente servicios de la clase B para poder llevar a cabo su responsabilidad.

Por ejemplo, una persona tiene una relación de asociación con su teléfono ya que, por norma general, el teléfono se utiliza con bastante frecuencia para poder llevar a cabo las responsabilidades diarias, pero a su vez el teléfono no forma parte esencial de la persona, ni de su esencia.

```
class Phone:
    def __init__(self, number):
        self.number = number

    def call(self, person):
        print("Llamando al número", self.number, "desde", person.name)

class Person:
    def __init__(self, name):
        self.name = name
        self.phone = None

    def call(self, phone):
```

```
        self.phone = phone
```

Uso / dependencia

Las relaciones de uso o dependencia, son relaciones momentáneas entre un objeto cliente y un servidor. Decimos que existe una relación de uso o dependencia entre dos objetos de las clases A (cliente) y B(servidor), si un objeto de la clase A hace uso de alguna operación de la clase B en un momento determinado, para poder llevar a cabo su responsabilidad.

Por ejemplo. Una persona puede utilizar un taxi para llegar a su destino, pero el taxi no forma parte de la persona y puede ser utilizado por otras personas en momentos distintos.

```
class Taxi:
    def __init__(self, taxi_number):
        self.taxi_number = taxi_number

    def drive_to_destination(self, destination):
        print(f"Dirigiéndose hacia el destino: {destination}.")

class Passenger:
    def __init__(self, name, destination):
        self.name = name
        self.destination = destination

    def request_taxi_ride(self, taxi):
        taxi.drive_to_destination(self.destination)
```

Puede parecer un poco extraño que el pasajero reciba al taxi como parámetro y no al revés, pero debemos recordar que aunque la programación orientada a objetos modela el mundo real, no es una representación exacta y cada objeto de cada clase tendrá la responsabilidad de operar sobre los datos que contiene.

Relaciones entre clases (Herencia)

La herencia es un mecanismo mediante el cual, una clase padre transmite todo su código a una clase hija (con posibles limitaciones de acceso). La herencia puede darse por extensión o por implementación (interfaces).

La herencia nos permite transferir comportamientos entre diferentes clases, aunque la transferencia de estos comportamientos suponen , a su vez, la transferencia de una parte del código, no debemos tomar la herencia como un mecanismo de reutilización de código.

Antes de crear una relación de herencia debemos comprobar si el elemento que va a heredar es un miembro de la clase heredada, haciendo directamente la pregunta "¿Es un..?" Por ejemplo , si vamos a crear una clase Estudiante y queremos que herede de la clase persona podemos preguntar "¿Un estudiante , es una persona?" si la respuesta es positiva, podemos crear una relación de herencia, si es negativa, aunque técnicamente sea posible esto crea malas relaciones y ensucia el código.

También es conveniente mencionar que por norma general es aconsejable utilizar otro tipo de relaciones siempre que se pueda evitar la herencia, de esta forma creamos un código más flexible y moldeable.

Herencia por especialización

Herencia por especialización: Es cuando la clase derivada o hija añade funcionalidad a la clase base ya sea añadiendo nuevos métodos o redefiniendo los que ya existen Veamos el ejemplo de una clase Dog, que especializa a una clase Animal.

```
class Animal:
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        pass


class Dog(Animal):
```

```
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed

    def bark(self):
        return "Woof!"

    def make_sound(self):
        return self.bark()
```

En este sencillo ejemplo podemos ver que la clase derivada “Dog” añade un atributo “breed” (raza) y además especializa el método make_sound para que el sonido que devuelva sea el del perro.

Herencia por implementación

La herencia por implementación se produce cuando una clase hereda el comportamiento definido en una interfaz. La interfaz no tiene una implementación específica para dichos comportamientos y , por norma general , carece de atributos.

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

    @abstractmethod
    def move(self):
        pass


class Dog(Animal):
    def make_sound(self):
        return "Woof!"

    def move(self):
        return "Running"
```

Por norma general, cuando una clase hija, hereda un conjunto de operaciones sin implementación de una clase padre, decimos que existe una relación de implementación y la clase padre se debe representar con una interfaz (en lenguajes como python, no existe una sintaxis específica para el uso de interfaces)

Herencia por extensión (de clases abstractas)

Podemos entender la herencia por extensión como una combinación de las dos anteriores, pero antes de poder continuar debemos definir un concepto que no hemos mencionado hasta ahora. El concepto de “clase abstracta”

Una **clase abstracta,** es una clase que contiene al menos un método abstracto o lo que es lo mismo, un método sin implementación. Al definir una clase como abstracta, estamos impidiendo que se puedan instanciar objetos directamente de ella. Las clases abstractas se diferencian de las interfaces en que pueden contener atributos y además deben contener comportamientos definidos (métodos implementados). Su principal objetivo es establecer una estructura común y definir un contrato para las clases derivadas, promoviendo la reutilización de código y permitiendo la definición de comportamientos específicos en cada clase derivada.

Si creamos una clase derivada a partir de una clase abstracta podemos decir que estamos implementando una herencia por extensión. Al contener métodos y atributos definidos en la clase padre, podemos especializar la clase hija a la vez que al estar obligados a definir los métodos abstractos estamos implementando el comportamiento. En el siguiente ejemplo podemos ver cómo se implementa éste tipo de herencia en python

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

    @abstractmethod
    def move(self):
```

```
        pass


class Dog(Animal):
    def make_sound(self):
        return "Woof!"

    def move(self):
        return "Running"
```

Smell Codes en Herencia

Existen muchos ejemplos frecuentes del mal uso de herencia aunque por norma general los que con más frecuencia me he encontrado son los dos siguientes:

- **Herencia por limitación:** Éste tipo de herencia se produce cuando la clase hija no implementa todas las operaciones de su clase padre o sobreescribe operaciones existentes de la clase padre para dejarlo sin comportamiento. Éste tipo de herencia es antiintuitiva, rompe el principio de menor sorpresa e imposibilita el polimorfismo.
- **Herencia por construcción:** La herencia por construcción suele producirse cuando, con la intención de reutilizar el código, creamos una jerarquía de herencia donde debería existir algún otro tipo de relación como composición o agregación. Creamos este tipo de herencia cuando , por ejemplo, implementamos una clase coche, que hereda de una clase motor simplemente para que el coche pueda utilizar las operaciones de la clase motor.

Polimorfismo

El polimorfismo es una propiedad de la programación orientada a objetos mediante la cual, cuando hacemos referencia a una clase dentro de nuestro código, esa misma referencia vale tanto para la clase especificada como para todas sus clases derivadas.

Para hacer uso del polimorfismo, debemos establecer un contrato que tanto la clase base (o padre) como las derivadas deben cumplir.

```
class ScrapperEngine:

    def __init__(self, base_url):
        self._base_url

    def execute(self):
        ...


class ScrapperHTML(ScrapperEngine):

    def _parse_html(self):
        ...

    def execute(self):
        ...

        html = self._parse_html()

        ...


class ScrapperJSON(ScrapperEngine):
    def _parse_json(self):
        ...

    def execute(self):
        ...

        json = self._parse_json()

        ...
```

```
@task
execute_periodic_scrap(scrapper_engine:ScrapperEngine):
    escrapper_engine.execute()
```

En el ejemplo anterior, podemos imaginar que tenemos un sistema de cola de tareas que ejecuta la función "*execute_periodic_scrap*". El sistema de cola podría instanciar objetos de cualquiera de las clases *ScrapperEngine* , *ScrapperHTML* o ScrapperJSON y pasarlo como parámetro a "*execute_peridic_scrap*". La tarea no se preocupa de que tipo de dato es el que se le está pasando siempre que todos cumplan el contrato, en este caso el contrato es definir un método "execute". Tanto *ScrapperHTML* como *ScrapperJson* realizan tareas distintas dentro del método execute pero eso no le importa a nuestra tarea programada ya que forma parte de la encapsulación del objeto.

Mixins

Un mixin es una clase base que define comportamientos que pueden agregarse a otras clases. Estas otras clases pueden compartir este comportamiento sin tener una relación de herencia. Por ejemplo podríamos crear una clase *JSONMixin* que defina los métodos *to_json* y *from_json* para leer y convertir datos en formato json y cualquier clase que quisiese encapsular este comportamiento podría heredar de JSONMixin.

```
def JSONMixin:
    ...

    def to_json(self):
        return json.dumps(self.data)

    def from_json(self, json_data):
        self.data = json.load(json_data)

def TicketInvoice(JSONMixin, Invoice):
     ...

def CarData(JSONMixin, Vehicle):
    ...


car_data = CarData()
```

```
car_data.to_json()
car_data.from_json(json_car_data)
```

Las *TicketInvoice* y *CarData* no tienen ninguna relación entre sí, pero ambas reciben la "habilidad" de leer y devolver datos en formato json, heredando del mixin *JSONMixin*

Al igual que con la herencia, debemos tener cuidado al reutilizar mixins. Los mixins deben utilizarse con el objetivo de proporcionar comportamientos determinados a otras clases, no como mecanismo de reutilización de código. Por norma general para que una clase determinada herede de un Mixin, debe utilizar (ella o sus objetos) todos los comportamientos proporcionados por dicho mixin. Si no es así, tal vez debería encontrarse otro tipo de relación que permita obtener los comportamientos necesarios

SOLID

Los principios SOLID son 5 principios de código limpio recopilados de diferentes autores. Estos principios han sido asociados tradicionalmente a la programación orientada a objetos pero realmente pueden aplicarse a cualquier paradigma de programación. Sus siglas son una regla mnemotécnica , extraída de sus nombres en inglés

1. Single Responsibility Principle(S)
2. Open close principe (O)
3. Liskov substitution principle (L)
4. Interface Segregation principle (I)
5. Dependency inversion principle (D)

1. Principio de responsabilidad única (SRP)

Este principio nos dice que cada componente del software que estamos desarrollando debe tener un único motivo para cambiar o lo que es lo mismo, cada componente debe encargarse de una sola cosa y tener sentido por sí mismo. Ya mencionamos este principio al hablar de funciones, cuando indicamos que debían tener una única responsabilidad. Lo que es una responsabilidad variará dependiendo del nivel de

abstracción en el que nos encontremos. No es lo mismo la responsabilidad de un método o función que será algo más técnico (limpiar una cadena, generar un password...) que a nivel de módulo o microservicio (gestionar usuarios, gestionar facturas...). Este principio es lo mismo que hablar de cohesión. Veamos un ejemplo

```
class Employee:
    def __init__(self, name, id, salary):
        self.name = name
        self.id = id
        self.salary = salary

    def calculate_salary(self):
        # Lógica para calcular el salario
        pass

    def save_to_database(self):
        # Lógica para guardar los datos del empleado en la base de datos
        pass
```

En el ejemplo anterior, la clase Employee tiene múltiples responsabilidades, ya que se encarga de realizar cálculos con los datos del empleado (en este caso solo hemos puesto calcular salario pero podría tener más comportamientos) y de almacenarlo en base de datos. Lo que le obliga a conocer no solo los datos del propio empleado sino además los datos de la infraestructura donde se van a almacenar (Sistema de base de datos, Esquema de las tablas etc.) Cada uno de estos acoplamientos es un motivo para cambiar por lo que la clase Employee no cumple el principio de responsabilidad única. Para hacer que cumpla este principio y que la clase sea cohesiva, debemos dividirla de la siguiente forma

```
class Employee:
    def __init__(self, name, id, salary):
        self.name = name
        self.id = id
        self.salary = salary

    def calculate_salary(self):
        # Lógica para calcular el salario
        pass

class EmployeeModel:
```

```
    def save_to_database(self, employee):
        # Lógica para guardar los datos del empleado en la base de datos
        pass
```

En este caso la clase Employee solo tiene la responsabilidad de conocer y operar con los datos del empleado mientras que la responsabilidad sobre la lógica de almacenamiento la hemos llevado a otra clase.

En éste ejemplo la clase Employee solo tiene un método, pero hay que tener cuidado ya que tener una responsabilidad no significa tener un método. En este caso la clase tiene la responsabilidad de operar sobre los datos que pertenecen directa e inequívocamente al empleado y solo debe conocer detalles sobre el mismo. Cualquier detalle como la infraestructura de base de datos , es algo que está más allá de la frontera de la clase empleado y debe encapsularse y utilizarse mediante colaboradores.

Para entender lo que es una responsabilidad hay que conocer perfectamente el dominio del problema en el que estamos trabajando, de forma que podamos dividirlo en diferentes partes lo más cohesivas posible. Por norma general la responsabilidad se va dividiendo durante el proceso de refactorización ya que de no ser así es posible caer en el error contrario, la responsabilidad compartida, donde una responsabilidad se divide en dos o más elementos que no tienen sentido por sí mismos (no pueden funcionar los unos sin los otros)

2. Principio de Abierto Cerrado de Bertrand Meyer (OCP)

El principio abierto cerrado, dice que "*Un componente debe estar cerrado para su modificación pero abierto para su extensión*". Es decir, que un componente debe permitir cambiar su comportamiento, sin alterar su código.

Éste principio nos anima a utilizar de forma correcta las relaciones entre objetos para crear sistemas más robustos y fiables.

Decimos que un código está **abierto** cuando permite ampliaciones mediante herencia, composición o algún otro método de extensión (prototype en js, monkey patching en python ...).

Decimos que un código está **cerrado** cuando es utilizado por otros módulos ya que su comportamiento está acoplado a ellos y por lo tanto , debe mantener su contrato (interfaz pública) sin modificar.

El objetivo de este principio es impedir que al intentar cambiar el comportamiento de los elementos del sistema, para ajustarse a nuevos requerimientos, añadamos complejidades innecesarias a las funcionalidades existentes o aún peor, rompamos otras partes del código.

3. Principio de Sustitución de Liskov (LSP)

El principio de sustitución de Liskov dice que *"Dado un tipo base o supertipo, cualquier subtipo puede ser intercambiado por su supertipo"* Aunque pueda parecer un poco confuso, este principio está hablando de polimorfismo. El principal uso de este principio es el de evitar el "typecasting" o "comprobación de tipos"

El LSP dice que los subtipos de una clase deben cumplir tres reglas

- **Regla de la firma**: Los métodos de la subclase, deben tener los mismos parámetros que su superclase
- **Regla de los métodos**: Cada método de un subtipo debe preservar el comportamiento de su supertipo. Es decir, un subtipo de list que implementa "append" no debería colocar los elementos al principio de una lista, ya que no estaría respetando el comportamiento del supertipo.
- **Regla de las propiedades**: Cada tipo tiene unas propiedades que son la base de su funcionamiento y estas propiedades deben preservarse en sus subclases. Por ejemplo, si creamos un subtipo del Set, en Python, no debemos permitir que existan elementos repetidos.

4. Principio de segregación de la interfaz (ISP)

Este principio dice que el conjunto de métodos que componen una interfaz (clase, módulo, paquete...) debe ser minimalista, de forma que los consumidores o clientes de la interfaz utilicen todos sus métodos.

Este principio está muy relacionado con el principio de responsabilidad única y básicamente nos indica que si tenemos un componente y diferentes consumidores utilizan diferentes subconjuntos de su funcionalidad, debemos dividir ese componente en funcionalidades más pequeñas.

5. Principio de inversión de la independencia (DIP)

Tal vez sea el principio más complejo de enunciar y comprender. El principio de inversión de dependencia nos dice que *"Un componente con un nivel de abstracción alto (cercano a la lógica de negocio) no debe depender de un componente con un nivel de abstracción más bajo (cercano a la infraestructura)"*

El objetivo fundamental de este principio es el de desacoplar las partes de nuestro código que implementan lógica de negocio, de aquellas partes de nuestro código que forman parte de la infraestructura que utilizamos (frameworks, librerías etc..)

Una de las herramientas que tenemos para ello es la denominada *inyección de dependencia.*

Cuando queremos inyectar dependencia en un componente A , lo que hacemos es sacar la lógica de instanciación de objetos de dicho componente a un componente externo B, y posteriormente, pasar el objeto instanciado al componente A

Por ejemplo, imaginemos que tenemos una clase vehículo, como parte de esta clase tenemos un atributo motor, que instancia un objeto de la clase motor

```
class Vehicle :

    def __init__(self):
        self.engine = new Engine()
```

En el código anterior, nuestra clase Vehicle, depende directamente de la implementación concreta de la clase Engine. Ya que la instancia directamente y por lo tanto está acoplada a ella. Por ejemplo, ¿Qué pasa si nuestro vehículo puede tener diferentes tipos de motor ? En el ejemplo anterior no podríamos tener vehículos con diferentes tipos de motor ya que Vehicle está acoplado (depende) de Engine. Para mejorar este diseño, podemos pasar una instancia de Engine, como dependencia inyectada, de forma que Vehicle pueda utilizar el objeto engine, sin necesidad de conocer su implementación concreta

```
class Vehicle:

    def __init__(self, engine):
        self.engine = engine
```

En este segundo diseño, Vehicle puede tener diferentes tipos de motores y no está acoplado a la implementación concreta de uno de ellos. Al pasar el objeto “engine” como parámetro al constructor en lugar de instanciarlo directamente decimos que hemos *“inyectado la dependencia”.* Por supuesto, el objeto *engine* debe respetar el Principios de sustitución de Liskov, o lo que es lo mismo, debe implementar la misma api pública que Engine.

6. Control de Errores

Mejora la integridad de tu software

Uno de los elementos más importantes y más ignorados durante el desarrollo de software es el manejo correcto de los errores y las excepciones. Las excepciones son las herramientas que nos proporcionan los lenguajes de programación para poder gestionar los comportamientos inesperados en nuestro código, redirigiendo su flujo.

Aunque las excepciones permiten redirigir el flujo ante comportamientos inesperados, no se deben confundir con una herramienta de control de flujo, su objetivo es permitirnos determinar si , en caso de error, nuestro código debe continuar ejecutándose o debe detenerse por completo. En la mayor parte de los casos, utilizaremos excepciones para detener la ejecución del código, por lo que las excepciones deben utilizarse única y exclusivamente para gestionar comportamientos excepcionales.

A continuación veremos una serie de pautas que nos ayudarán a mejorar nuestro manejo de los errores y las excepciones.

No utilices excepciones para controlar el flujo de ejecución.

Como ya hemos dicho, las excepciones se utilizan para notificar errores y situaciones excepcionales, no para alterar el código según la lógica de negocio. Si utilizas excepciones para situaciones que son previsibles, tu código será menos legible y eficiente por lo que será más complicado de mantener. Utilizar excepciones para controlar el flujo de ejecución es similar a utilizar un go-to considerado una práctica desaconsejada.

Seleccionar el tipo de excepción correcto

Debemos seleccionar siempre el tipo de excepción que más se ajuste al comportamiento que queremos controlar. Utilizar siempre la excepción más genérica para controlar cualquier error que pueda producirse, es una mala práctica y señal de un diseño deficiente. Además, seleccionando tipos de excepciones concretos, se facilita la detección y corrección de errores que no hayamos previsto.

Escribe un mensaje descriptivo

Cuando utilizamos una excepción, debemos notificar el error de forma clara y precisa para mejorar la legibilidad en el momento de la identificación y la depuración del error.

No utilices excepciones silenciosas

Una excepción genérica, cuyo manejo se deja vacío no fallará nunca, incluso cuando sería aconsejable que lo hiciera. Este tipo de bloques de manejo de excepciones se utilizan o bien porque decidimos ignorar los errores que se producen en nuestro código o porque estamos utilizando este mecanismo para controlar el flujo de ejecución de nuestra lógica de negocio. Utilizar excepciones de esta forma, es una mala práctica que debilita nuestro código y nos indica un fallo de diseño.

Separación de responsabilidad

Cuando definimos una función o método, que lanza una excepción, estamos obligando al cliente (o llamante) a conocer este detalle de implementación para poder gestionarlo correctamente. Esto rompe el encapsulado del código ya que obliga a conocer detalles de la implementación de nuestro componente para utilizarlo correctamente. Teniendo esto en cuenta, podemos ver que , una función o método que lanza demasiadas excepciones posiblemente tenga demasiadas responsabilidades y deberían ser separadas en múltiples componentes.

No expongas las trazas

Tanto cuando informamos al usuario final de una excepción, como cuando propagamos dicha excepción a aplicaciones clientes, es muy importante evitar que se propaguen detalles sensibles que puedan comprometer la seguridad de nuestro código. Estos detalles pueden ser:

- Exponer la traza completa del error
- Mensajes de error que contengan detalles de implementación
- Mensajes de error que incluyen valores de estado de las variables en el momento de la excepción

Cuando exponemos una excepción al cliente, es mejor utilizar mensajes genéricos tipo “se ha producido un error” o “Algo no ha funcionado”, pudiendo dar más detalles del error pero sin incluir detalles de implementación. Los mensajes detallados deben enviarse solamente a los sistemas de logging y depuración de nuestro sistema.

Incluir excepción original

En ocasiones, como parte del manejo de una excepción, decidimos lanzar una nueva excepción. Esto puede ocurrir, por ejemplo, cuando utilizamos librerías de terceros y

decidimos sustituir sus excepciones por excepciones internas de nuestra aplicación. En este caso, es recomendable incluir en la nueva excepción los datos de la excepción original para facilitar su depuración.

Las excepciones deben ser gestionadas correctamente.

Una excepción define una previsión de que algo puede fallar en nuestro código. Debemos diseñar nuestras excepciones con la misma calidad que tiene el resto de nuestro código.Para ello, debemos planificar, con la mayor exactitud posible, cuál va a ser la reacción del sistema a dicho comportamiento excepcional.

En algunos casos será mejor dejar que la excepción siga escalando y capturarla en los componentes más externos del sistema, mientras que en otros podremos manejarla localmente o establecer mecanismos de tolerancia a errores.

Capturar una excepción cuya única gestión es imprimir en el log, puede ser considerado como una excepción vacía en la mayoría de los contextos. Asegúrate de que, si haces esto, la excepción ha sido gestionada adecuadamente y el usuario final conoce sus consecuencias.

Aserciones (asserts)

Las aserciones o asserts, se utilizan para controlar condiciones que nunca deben producirse en nuestro código. La expresión que se utilice en una sentencia assert debe ser imposible de cumplir ya que de cumplirse indica un error en el software y su ejecución debe detenerse ya que es un defecto grave en el funcionamiento.

Cuando establecemos una cláusula de este tipo y la condición no se cumple el sistema lanzará una excepción. Esta excepción **no debe ser controlada,** ya que el objetivo de los asserts debe ser detener la ejecución del sistema ante un error grave en su funcionamiento.

Los asserts suelen ser utilizados como cláusulas de guarda para evitar el mal uso de componentes del software por parte de los propios desarrolladores. Por ejemplo, para controlar que se cumple el contrato al llamar a una determinada función o componente y que no se está haciendo un uso indebido que pueda provocar errores.

```
def create_client_account_with_bank_api(client_id:str, iban:str):
      assert client_id , "El identificador del cliente no puede estar vacío"
      assert iban, "El iban no puede estar vacío"
```

En la mayor parte de los sistemas, los asserts son desactivados en producción y se utilizan solamente en los entornos de desarrollo. En Django, por ejemplo, los asserts se desactivan automáticamente cuando se desactiva el modo Debug

7. Desarrollo de Test

Evalúa la calidad y la integridad de tu código

Qué son los tests automáticos

Un test automático es una función o método, que ejecuta una parte de nuestra aplicación para verificar su funcionamiento de forma independiente (dependiendo del tipo de test) al resto del sistema. Por norma general, un test se compone de 3 partes

- **Arrange (organización) :** Consiste en preparar la parte de la aplicación que queremos probar (establecer el estado del sistema, introducir datos de pruebas en una base de datos ...)

- **Act (actuación)**: Consiste en realizar las acciones necesarias para la prueba que queremos realizar (generalmente llamar a una función o método)

- **Assert (afirmación):** Es la parte en la que observamos si el resultado de la acción o acciones realizadas es el esperado.

Por qué desarrollar Test

- Crea aplicaciones más seguras y robustas
- Mejora la estabilidad al modificar funcionalidades existentes ya sean desarrolladas por nosotros o por algún otro miembro del equipo
- Generan documentación implícita sobre las funcionalidades
- Facilitan la actualización de las versiones de framework y librerías permitiendo identificar los puntos conflictivos o de error.
- En momentos de conflicto en el desarrollo, recurrir a técnicas como el TDD pueden facilitar la mejora del diseño
- Ahorra tiempo de desarrollo ya que, al desarrollar test evitamos el estar constantemente realizando pruebas manuales tanto al desarrollar como al resolver bugs.

Tipos de tests

- **Unitarios:** Prueban funcionalidades específicas y aisladas, generalmente no deben tener acceso I/O (a base de datos, ficheros, apis externas...) no la necesitan.

- **Integración:** Prueban componentes enteros de aplicaciones y cómo se combinan entre ellos, pueden requerir de accesos a bases de datos, apis de terceros ...
- **Regresión:** Son pruebas diseñadas específicamente para reproducir errores históricos. Cada prueba es ejecutada para comprobar que el error ha sido corregido y no se ha vuelto a producir
- **Validación / Aceptación:** Los tests o pruebas de verificación y validación, tienen el objetivo de comprobar que el sistema cumple con su función, o dicho de otra forma, una prueba de validación es la que comprueba que se han implementado todos los requisitos de la tarea y funcionan correctamente.
- **Interfaz de usuario (End to End):** Los test de interfaz o pruebas de interfaz de usuario, son aquellos cuyo objetivo es comprobar que la interfaz de usuario y sus flujos, funciona correctamente, generalmente se realizan utilizando librerías que automatizan o simulan las acciones de un usuario sobre la interfaz de nuestra aplicación mediante el uso de eventos de teclado y ratón.

Los test unitarios son más rápidos y eficientes, a mayor cantidad de tests unitarios, menor cantidad de tests de integración.

Consideraciones básicas en la escritura de tests

1. **Todo lo que se puede romper debe tener test** (esto incluye modelos, vistas, formularios ...)
2. **Cada test debe comprobar únicamente una función**(como norma general)
3. **Keep it simple and stupid (KISS)** , los test también son código y requieren mantenimiento por lo que deben ser fáciles de leer y modificar, es una buena práctica mantener los test en un formato conciso y reducido.
4. **Ejecutar los test siempre** , antes de hacer un push, después de hacer un pull y antes de pasar código a preproducción y producción
5. **Separar los test en unidades mínimas o casos de uso concretos** que faciliten la identificación y corrección de errores. Si dudas entre un único método o múltiples, ve siempre a por los múltiples (siempre que las afirmaciones que quieras probar tengan sentido)

6. **Documentar los test puede ayudar a entender mejor una funcionalidad** (ojo, los comentarios , al igual que el código, representan conocimiento y por lo tanto también hay que mantenerlos.)
7. **No sigas los principios DRY** (Don't Repeat Yourself): Aunque el código de los test debe ser lo más limpio y mantenible posible, el objetivo fundamental de los test es validar una funcionalidad en concreto no la limpieza o reusabilidad del código, por esto salvo que aporte un beneficio claro, no sigas este principio.
8. **Utiliza nombres identificativos para tus test**
9. **Haz pruebas de error:** No escribas test solo para los casos de éxito , escribe test también para aquellos momentos en los que esperas que tu código falle (lance una excepción, rechace una petición...)
10. **Los test deben estar aislados y ser reproducibles en cualquier entorno:** Si un test requiere acceso a ficheros o recursos fuera del entorno de ejecución, estamos introduciendo la posibilidad de que no sean reproducibles y por lo tanto no podamos ejecutarlos siempre que lo necesitemos.
11. **Los test deben ser rápidos:** Al diseñar nuestros tests, debemos asegurarnos de que sean lo más rápidos posible, para lo que tendremos que poner esfuerzo en que estén bien diseñados y optimizados.

Código testable vs no testable.

El código testable, es aquel para el que podemos escribir test unitarios fiables sin demasiado esfuerzo. Éste código debe ser determinista, o lo que es lo mismo, es un código al que proporcionando una entrada determinada, podemos predecir su salida.

En un entorno ideal, un software se construiría en su mayoría con funciones puras. Una función pura es aquella que, para una determinada entrada producirá siempre la misma salida independientemente del estado del sistema. Sabemos que no es posible construir un software utilizando únicamente funciones puras ya que necesitamos poder almacenar estados para trabajar con ellos, pero sí que es una buena práctica el que , siempre que sea posible, las funciones o métodos que construyamos sean puros, ya que esto ayuda a que el código sea más fiable, fácil de testar y mantener.

Código no testable.

Imaginemos que queremos conectar dos sistemas que se encuentran en diferentes servidores. Para realizar esta conexión queremos enviar una firma, de forma que, el sistema que recibe la solicitud, pueda confirmar la identidad del emisor. Para ello, generamos una firma a partir de , una clave secreta que solo conocen emisor y receptor, y una clave pública formada por la marca de tiempo actual, el usuario que realiza la petición y la clave secreta.

```
def generate_signature_for(username:str):
    secret_key = os.environ.setdefault("SECRET_KEY", "")
    current_timestamp = datetime.now().isoformat()
    signature_key = f"{username}{current_timestamp}{secret_key}"
    return hmac.new(secret_key, msg=signature_key, digestmod=sha256).digest()
```

Antes de continuar, párate e intenta analizar el código anterior y determinar, qué problemas tiene para que podamos testear.

Para empezar vamos a comenzar por analizar qué problemas tiene el código anterior y por qué , a pesar de ser un código compacto y a priori, sencillo, es un mal diseño:

- **Está acoplado a todos los métodos de los que depende :** Conocer exactamente la variable de entorno que almacena la clave, el formato de fecha, los elementos que componen la firma o el algoritmo de cifrado, hacen que esta función conozca demasiado sobre los detalles de implementación del sistema y por lo tanto, esté acoplada a ellos.
- **Su funcionalidad no es intuitiva al ver su firma:** Si nos encontramos en el código llamante, no podemos intuir las dependencias de esta función por lo que , sin leer su código nos sería imposible realizar un test fiable.
- **Rompe el principio de responsabilidad única:** Como hemos dicho anteriormente, esta función tiene varios motivos para cambiar, en concreto 4. Cada una de sus líneas representa una dependencia que obligaría a modificar éste método, ya que conoce demasiado sobre las librerías e implementaciones del sistema. Veamos estos motivos de cambio en más detalle
    - *Si cambia el origen de la clave secreta*, esta función deberá cambiar, ya sea porque cambie el nombre de la variable de entorno, o porque cambie completamente el lugar de almacenamiento de la misma (por ejemplo si utilizamos un fichero)

- *Si cambiamos la librería o el formato de fecha*: Cambiar el formato para que no utilice el formato iso o cambiar el módulo mediante el cual obtenemos la fecha (por ejemplo para utilizar timezone en lugar de datetime) son motivos por el que deberemos modificar esta función
- *Si decidimos modificar los elementos que componen la firma*: Por ejemplo si decidimos añadir el email o algún otro dato, o generar firmas más complejas (Éste debería ser el único motivo de cambio de esta función ya que su responsabilidad es generar la firma).
- *Si deseamos modificar cualquiera de los elementos que general el has*h: Por ejemplo si deseamos cambiar el algoritmo por sha512.

Además de los problemas mencionados anteriormente, este código no es testable , ¿sabrías decir por qué ?.

Pues bien, el método no es determinista. Indicando una entrada concreta no tenemos forma de determinar su salida ya que el resultado que obtengamos depende de otros elementos, que no podemos proporcionar como entradas. Estos elementos son:

- **La clave secreta:** Ya que se almacena en una variable de entorno del sistema
- **La fecha y hora actuales**: La fecha y hora serán diferentes cada vez que ejecutemos el test por lo que es imposible determinar cuál será en el momento de la ejecución y por lo tanto predecir su salida.
- **El algoritmo hash que queramos emplear**: Tal vez este problema sea el menos evidente pero es algo que debemos conocer de antemano para realizar el test.

Generar un test para esta función implicaría que debemos hacer un mock de la clave secreta, un mock de la función que determina la fecha actual y un mock de la función que se encarga de generar el hash. No creo que haga falta mencionar que son muchos mocks y de elementos innecesarios

Por norma general, los mocks deben utilizarse exclusivamente para emular llamadas a recursos externos o en algunos casos, para mejorar el rendimiento de los test, si la única forma que tienes de probar tu propio código es mediante mocks, esto puede ser síntoma de un diseño deficiente , especialmente, si se requiere más de uno.

Código testable.

Vamos a arreglar el código para poder determinar su estado y por lo tanto poder crear un test unitario que nos permita probar su funcionamiento. Para modificar el código anterior tenemos varias alternativas, si en lugar de una función independiente fuese un método de una clase, podríamos utilizar el método de la inyección de dependencia y delegar en otros métodos o atributos de la clase, los diferentes pasos del proceso. En este caso , al ser una función independiente haremos uso de pasos de parámetros para eliminar algunas dependencias. Una primera aproximación al desacoplamiento podría ser

```
def generate_signature_for(username:str, secret_key:str, current_timestamp:str):
    signature_key = f"{username}{current_timestamp}{secret_key}"
    return hmac.new(secret_key, msg=signature_key, digestmod=sha256).digest()
```

Ésta primera aproximación elimina parte del acoplamiento de la primera versión del código y nos permite testear la función ya que , dada una entrada concreta, podemos determinar su salida y no requiere de mocks. Lo que sí requiere este código es conocer el algoritmo utilizado para crear la clave hash de la firma, lo que hace que debamos acceder al código de la función para poder generar un test fiable (o de lo contrario, deberíamos generar un mock).

Podemos modificar éste código haciendo uso de *"funciones de orden superior"* para eliminar el acoplamiento al algoritmo de hash.

```
def generate_signature_for(
        username:str,
        secret_key:str,
        current_timestamp:str,
        generate_hash:Callable
    ):
    signature_key = f"{username}{current_timestamp}{secret_key}"
    return generate_hash(signature_key, secret_key)
```

En este último ejemplo hemos reducido la función a una única responsabilidad y todas las demás dependencias del proceso son “inyectadas” mediante el paso de parámetros. Esta última versión reduce la responsabilidad de la función generar la clave de firma y su único motivo para cambiar, es que dicha estructura cambie , siendo desconocedor de cualquier detalle del resto del proceso.

Saber qué probar

Diseñar test es algo complicado, y requiere de muchos otros conocimientos como patrones de diseño, arquitectura etc, si quieres ser un experto.

Para diseñar buenos tests, debes probar todo aquel código que forme parte de tu diseño, pero debes procurar no escribir test que comprueben funcionalidades del propio lenguaje, framework o librería que estés utilizando. Esta es una de las dificultades a la hora de decidir qué es lo que debemos probar.

También debes asegurarte de que cada test prueba una única cosa y que no estás combinando varias funcionalidades o casos de uso en el mismo test. Vamos a ver un ejemplo sencillo. Supongamos que tenemos el siguiente modelo en Django.

```
def generate_student_code():
  return generate_random_string(8)

class Student(models.Model):
    code = models.SlugField(max_length=8, unique=True, default=generate_student_code)
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    date_of_birth = models.DateField()
    email = models.EmailField(max_length=254)

    def __str__(self):
        return f"({self.code}) {self.first_name} {self.last_name}"
```

Este es un caso en el que hay que tener muy claro cuales son los elementos que pertenecen a nuestro diseño y por lo tanto lo que debemos probar, veamos que es lo que debemos y lo que no.

Lo que no debemos probar

No debemos probar que *code,* es un slug y que tiene un formato válido ya que Django hace esa comprobación por nosotros, del mismo modo que no debemos comprobar que *first_name* o *last_name* son cadenas de caracteres, que *date_of_birth* es una fecha o que email tiene un formato correcto de correo electrónico. Todos estos elementos forman parte de Django por lo que no debemos hacer test para comprobar estas afirmaciones (podríamos hacerlos pero estaríamos probando componentes proporcionados por el framework)

Lo que sí debemos probar

Hay elementos que forman parte de nuestro diseño, por ejemplo, que el campo *code* sea único y que se le asigne una cadena aleatoria de 8 caracteres cuando se cree un nuevo estudiante. Al ser decisiones de negocio y no formar parte del funcionamiento del framework. Otras decisiones de diseño son , que el nombre y el apellido tengan 100 caracteres como máximo o que el email tenga 254 . También es conveniente probar que la representación en formato __str__ funciona correctamente sobre todo en casos donde el usuario pueda contener caracteres especiales en nombre o apellidos (por ejemplo la ñ) .

Otro elemento susceptible de validar es que la fecha de nacimiento se encuentre en un rango concreto, por ejemplo que no podamos indicar que hemos nacido en una fecha en el futuro o hace más de 100 años.

Consideraciones en el uso de test

Hay que tener cuidado al escribir test ya que, tener demasiados tests no es una buena práctia. Lo mejor es diseñar una buena estrategia para alcanzar una cobertura de test de calidad. La cobertura ideal suele estar en torno al 80% del código.

En prácticamente todos los lenguajes y frameworks existen herramientas para comprobar la cobertura de test de nuestro código.

Anexos

¿Por dónde continuar?

¿Y ahora qué?

Si te ha gustado este documento y has visto las posibilidades que te ofrece el diseñar un código de calidad, limpio y mantenible, no te detengas en los conocimientos básicos que se han expuesto aquí. En la siguiente sección te dejaré enlaces a varios libros que me han ayudado a mejorar la filosofía con la que abordo cada problema y me han hecho un mejor diseñador de software. Espero que este documento te haya ayudado a aprender conceptos nuevos y que te genere curiosidad por seguir adentrándote en este mundo.

Bibliografía

- **Código Sostenible** - Carlos Blé Jurado
- **Clean Code in Python: Refactor your legacy code base** - Mariano Anaya
- **El programador pragmático** - Andy Hunt y Dave Thomas
- **The Art of Clean Code; Best Practices to Eliminate Complexity and Simplify Your Life** - Christian Mayer