UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Máster en Ingeniería de Computadores y Redes

Trabajo Fin de Máster

MECANISMOS REUTILIZABLES PARA MEJORAR LA CONFIABILIDAD DE APLICACIONES DESARROLLADAS CON PYTHON

Autor: *Igor Dzichkovskii*

Director(es): *Juan Carlos Ruíz García*

10 de Julio de 2020

Abstract

Dependability is the property of computer systems that consistently performs according to its specifications. The need of mixing the necessary mechanisms of encryption and fault tolerance in system implementations with the functional code makes the resulting code very difficult to maintain and limits its ability to adapt and evolve. Open compilation, aspect orientation or reflective programming are only some of the alternatives available to develop software solutions with a clear separation between functional and non-functional mechanisms, such as those required for improving the dependability of such solutions.

Python is a general-purpose programming language widely used nowadays in many critical application domains where solutions require confidentiality and redundancy to tolerate the consequences of HW and/or SW faults. This master thesis explores the interest of using Python decorators as wrapping mechanisms to establish a clear separation between functional and dependability mechanisms (encryption and fault tolerance in our case) in applications developed with this language. The purpose is to ease the development of these mechanisms, their reuse, adaptation, and usage in disparate applications and in different contexts of use.

Resumen

La confiabilidad es la propiedad que atribuimos a aquellos sistemas informáticos en los que es posible depositar una confianza justificada en su uso. El problema es que los mecanismos de cifrado y tolerancia a fallos necesarios se mezclan en las implementaciones con el código funcional de las mismas, dificultando mucho su mantenimiento y limitando las capacidades de adaptación y evolución del sistema. La compilación abierta, la orientación a aspectos o la programación reflexiva son sólo algunas de las alternativas existentes para poder desarrollar soluciones informáticas con una clara separación entre mecanismos funcionales y no-funcionales, como lo son los orientados a mejorar la confiabilidad de dichas soluciones.

Python es un lenguaje de programación de propósito general muy utilizado actualmente en multitud de ámbitos aplicativos en los que se desarrollan soluciones con requerimientos de confidencialidad y redundancia para tolerar problemas de integridad o averías producidas por fallos de origen HW o SW. Esta tesina explorará el interés de utilizar los decoradores (decorators) de Python como mecanismos de encapsulación (wrapping) para establecer en las aplicaciones desarrolladas con este lenguaje una clara separación entre mecanismos funcionales y de confiabilidad (cifrado y tolerancia a fallos en nuestro caso). Se busca facilitar no sólo el desarrollo de estos mecanismos, sino también su reutilización, es decir, la adaptación y uso de los mismos a distintas aplicaciones en distintos contextos de uso.

Índice

Índice de Imágenes

CAPÍTULO 1: INTRODUCCIÓN GENERAL Y OBJETIVOS

Actualmente existen multitud de lenguajes de programación que ofrecen gran cantidad de soluciones tecnológicas que ayudan a los desarrolladores a realizar sus proyectos de manera más eficaz, rápida, y generando código cada vez más seguro y confiable. A pesar de estos requerimientos, el código generado debe ser fácilmente mantenible y debe poder evolucionar y adaptarse a las necesidades que el producto vaya a tener a lo largo de su ciclo de vida.

Para ello se deben utilizar tecnologías que ofrezca una clara separación entre mecanismos del sistema, del inglés *separation of concerns*. Gracias a ello los mecanismos existentes, funcionales y no funcionales, pueden combinarse y recombinarse con el objetivo de adaptar al sistema a distintos contextos de uso con distintos requerimientos funcionales o no-funcionales, por ejemplo, de confiabilidad. Una de estas tecnologías son los decoradores del lenguaje de programación *Python.* En este trabajo se estudiará la viabilidad de su uso para convertir soluciones no tolerantes a fallos a priori en soluciones con el nivel de redundancia y cifrado en las comunicaciones necesario para poder tolerar fallos HW y SW tanto de tipo accidental como malicioso.

1.1. Objetivos.

El objetivo principal de este trabajo es analizar y reproducir un ejemplo de uso de tecnología de decoradores con el fin de probar su implementación encapsulando y separando los diferentes mecanismos del código general de un proyecto. Los decoradores son mecanismos de encapsulación que pertenecen al lenguaje de programación *Python*.

Un ejemplo de uso es un sistema tipo cliente-servidor que se comunica por sockets, aplicando los decoradores como capsulas de los distintos mecanismos. Estos mecanismos pueden presentar un problema de entrelazado con el código del sistema,

es decir, que su código en lugar de centralizarse en una función, método o clase esté físicamente disperso a lo largo y ancho del código general de sistema. Encapsulando dichos mecanismos en los decoradores se obtienen beneficios de modularidad y reutilización que pueden ser explotados para ofrecer un aumento de tolerancia a fallos de todo el sistema.

Para abordar este estudio de viabilidad procederemos con los siguientes objetivos específicos:

- Definir qué son los decoradores, cómo se usan y preparar algunos casos de ejemplo.

- Desarrollar un sistema de tipo cliente-servidor de ordenamiento de una lista de números aleatorios a los que se aplicarán los decoradores con los mecanismos de tolerancia a fallos desarrollados. En efecto, no vamos a desarrollar un único mecanismos, sino varios, como un mecanismo de *logging u* otro de cifrado de comunicaciones, y a través de ellos evaluaremos los límites de los decoradores de Python a la hora de componer dichos mecanismos en base a las necesidades.

- Desarrollar otra solución cliente-servidor con el objetivo de mostrar hasta qué punto, y con qué premisas, es posible reutilizar los mecanismos de confiabilidad desarrollados para, y probados con, el primer sistema, en el segundo.

Estos objetivos demostrarán los beneficios que ofrecen los decoradores y justificarán su uso en los sistemas desarrollados en *Python* para mejorar su confiabilidad y seguridad.

1.2. Introducción, antecedentes, motivación y justificación

Los primeros lenguajes de programación no permitían el desarrollo de los grandes y complejos sistemas actuales, que manejan datos y al mismo tiempo hacen cálculos diferentes. Los primeros sistemas no requerían muchos mecanismos para, entre otras cosas, monitorizar su ejecución, tolerar caídas de servicio ofreciendo así una alta disponibilidad, o cifrar las comunicaciones entre sus componentes para robustecerlas frente a posibles ataques.

Los lenguajes de hoy en día son mucho más poderosos y ofrecen al programador la posibilidad de escribir un par de líneas de código para lograr un objetivo que antes requería varios días de trabajo. Hablamos de lenguajes como *Python,* que es flexible y que aúna beneficios de diferentes paradigmas de programación. Por tanto, los sistemas actuales son significativamente más potentes y pueden hacer una gran cantidad de cómputos difíciles. Sin embargo, cuanto más grande es el sistema, más problemas tendremos a la hora de enfrentarnos al proceso de desarrollo y uso.

Cada vez son más las empresas y los desarrolladores que utilizan *Python* implementando scripts escritos en este lenguaje o usándolo como lengua principal para proyectos. El portal *statista*, especializado en estadísticas, manifiesta que la popularidad de *Python* está en crecimiento, siendo ya la más alta de todos los lenguajes (imagen 1). [22]

Esto se debe a que *Python* es un sistema fácil de aprender. Además, al ser un lenguaje que aúna distintos paradigmas, ofrece una amplia variedad de tecnologías y herramientas de desarrollo. *Python* también tiene una comunidad sólida de desarrolladores que crea nuevas librerías para diferentes ámbitos de uso, apoyando así al desarrollo del lenguaje y de las librerías existentes.

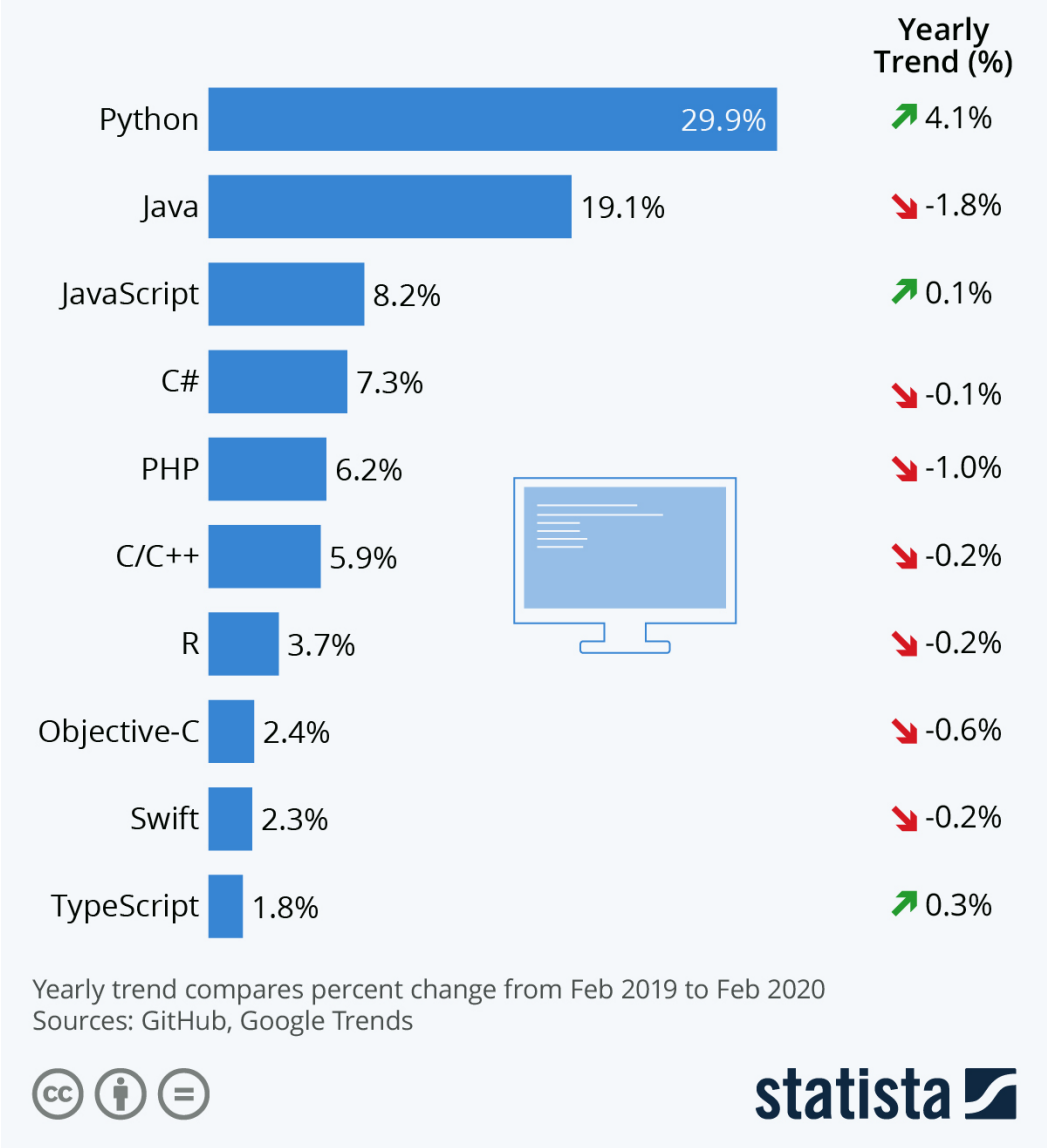


Imagen 1. Popularidad de lenguajes de programación en 2020.

Los sistemas desarrollados en *Python,* como otros sistemas escritos en otros lenguajes, son vulnerables tanto a distintos tipos de fallos, como los del HW sobre el que se ejecutan o los SW derivados de su programación, como para amenazas externas de tipo malicioso, que pueden afectar, entre otras cosas, a la integridad de la información intercambiada en la red o a su confidencialidad. El uso de mecanismos de tolerancia a fallos o de cifrado resuelve estos problemas, pero provocan otros. Uno de los problemas que ocurren por culpa del uso de estas tecnologías es el problema del entrelazado de código, que aparece al combinar distintos mecanismos que requieren de fragmentos de código implementados y dispersos en diferentes partes del código.

Para solucionar esta problemática se usa el concepto de encapsulamiento, del inglés *wrapping,* para capturar las invocaciones de métodos y poder aplicar sobre las mismas cierto pre- and post-procesamiento. *Python* está presentado como un mecanismo propio para ello, el mecanismo de decoración o en otras palabras los decoradores (*decorators* en inglés).

Los decoradores ofrecen un mecanismo de intercepción y control de llamadas muy similar a los mecanismos *before, around y after* que ofrece el paradigma de programación orientada a aspectos, y que permiten al desarrollador inyectar código antes, alrededor y después de las llamadas para adaptar la ejecución de estas a las necesidades. Es así como se puede realizar una separación de mecanismos (del inglés *separation of concerns*) evitando el problema del entrelazado de código anteriormente mencionado.

Cabe señalar que el estudio de esta tecnología de encapsulación es, en primer lugar, de interés profesional, porque el concepto de *wrapping* no solo se usa en *Python,* sino también en otros lenguajes. Por ejemplo, *Java* dispone de la herramienta *AspectJ* que está enfocada a hacer labores similares a los decoradores que nos referimos en este trabajo. Por tanto, la experiencia y enseñanzas que se deriven de este trabajo tendrán una utilidad en otros contextos de desarrollo SW.

En segundo lugar, este estudio es de interés académico, porque se han podido emplear diferentes conocimientos adquiridos durante los estudios del máster en el ámbito de soluciones tolerantes a fallos adaptativas, además de ampliar otros para llegar a la consecución del proyecto.

Finalmente, el estudio de los decoradores de Python tiene también detrás una fuerte motivación e interés personal. En mi opinión, el paradigma de programación orientada a aspectos es una tecnología con gran proyección, pero también con uso actual amplio, ya que permite la escritura de diferentes módulos a través de la inyección de comportamiento dentro de las partes de código donde sea necesario. Por otro lado, decimos que tiene mucha proyección porque los sistemas son cada vez más complejos y requieren cada vez más mecanismos y de una mayor capacidad para poderlos alterar

y adaptar. Así, la necesidad de los aspectos aumenta cada día, porque resultaría prácticamente imposible mantener un sistema con un único módulo donde se escriba todo el código y éste quede inalterado a lo largo de todo su ciclo de vida. Por lo tanto, aprender, analizar e implementar decoradores como tecnología para la adaptación de soluciones SW seguro que resultará de gran utilidad e interés para la futura carrera profesional y/o académica del autor de este documento.

1.3. Estructura del trabajo

Así, el presente trabajo se estructurará en base a la relación entre la parte teórica sobre los decoradores y las posibilidades de su uso práctico, que son resultado de la creación de sistemas propios de tipo cliente-servidor. Esto no significa que los decoradores sólo sirvan para mejorar la confiabilidad de este tipo de sistemas, lo que sucede es que el estudio se centrará en este tipo de sistemas porque requieren, de manera natural, de capacidades de tolerancia a fallos HW y SW para ser realmente útiles.

El **Capítulo 2** se centrará en las investigaciones realizadas y las soluciones disponibles actualmente para proporcionar una clara separación entre mecanismos funcionales y no funcionales, y solventar, o al menos mitigar, el problema del entrelazamiento de código. Por tanto, hablaremos de los compiladores abiertos, de la programación reflexiva y la metaprogramación, y de la programación orientada a aspectos, tecnologías estas ya integradas actualmente en lenguajes tan conocidos como *Java* o *Python*. Veremos que la clave para esta separación entre mecanismos es la encapsulación y explicaremos cómo un sistema puede robustecerse frente a fallos utilizando este tipo de soluciones.

En el **Capítulo 3** se explicará el problema de la separación de mecanismos en Python y nos centraremos en las peculiaridades del uso de decoradores, sus funciones y rasgos. En este apartado los aspectos teóricos estarán acompañados de ejemplos ilustrativos.

En base a las conclusiones presentadas en capítulos anteriores, en el **Capítulo 4** podremos empezar a crear nuestro propio sistema, es decir, hacer un desarrollo de

decoradores como mecanismos no funcionales. Empezaremos por observar el caso de ejemplo y entorno de desarrollo, después pasaremos a realizar tests a un prototipo en desarrollo. De la misma manera examinaremos los objetivos, el diseño y la implementación de varios mecanismos no funcionales encaminados a permitir las comunicaciones entre un cliente y un servidor (patrón de diseño *Proxy*), mejorar la robustez del cliente mediante un mecanismo de reintento, y la del servidor, mediante otros de redundancia y de gestión del estado, y finalmente, nos centraremos en el robustecimiento de las comunicaciones entre un cliente y su servidor mediante cifrado simétrico.

En el **Capítulo 5** pasaremos a crear un sistema distinto mediante el cual mostraremos el interés de los decoradores de cara a facilitar la reutilización de los mecanismos producidos. Cada paso de este segundo sistema se ilustrará a través de ejemplos.

Por último, en el **Capítulo 6**, presentaremos las principales conclusiones de nuestra investigación y analizaremos los resultados que hemos logrado en la creación de los sistemas.

CAPÍTULO 2: ESTADO DEL ARTE

En este capítulo trataremos el problema de entrelazamiento de código que puede pasar cuando un código requiere de mecanismos no funcionales, como son los de cifrado y tolerancia a fallos HW y SW. Este problema lleva a la generación de código difícilmente mantenible, muy poco modular y nada reutilizable Como ya hemos motivado en el capítulo anterior el concepto de separación de mecanismos (*separation of concerns* en inglés) ofrece la promesa de un código más modular, altamente configurable y con mejores características de adaptación a las necesidades. Veremos que esta separación de mecanismos se ha declinado de distintas formas, mediante compiladores abiertos y metaprogramación, con características propias de los lenguajes, como la reflexividad de Java y Python, o más recientemente con la orientación a aspectos. En esencia todas estas tecnologías basan su funcionamiento en algo muy sencillo, la intercepción (del inglés *wrapping*) de llamadas.

El uso de tecnologías de wrapping en sistemas distribuidos, y más concretamente, en el ámbito de los sistemas tolerantes a fallos no es nada nuevo. Los Interceptores de CORBA, un middleware para el desarrollo de sistemas distribuidos, ya utilizan este principio para ofrecer a los desarrolladores la capacidad de replicar, inhibir o derivar llamadas entre un cliente y un servidor. Lo mismo ocurre en el sistema tolerante a fallos FRIENDS, en el que se combinan la programación reflexiva, con la intercepción, para convertir programas no confiables, en programas que sí lo son.

Por tanto, los decoradores son una alternativa más a explorar para ofrecer una clara separación entre mecanismos en soluciones desarrolladas con Python. En otras palabras, los decoradores serán, en esencia, mecanismos de separación de código.

Cabe mencionar que, de aquí en adelante, en el presente trabajo se ha optado por el uso de las palabras “función” y “método” en referencia a un procedimiento de código que está definido con la palabra reservada *def*. La función es el procedimiento fuera de la clase y el método es el procedimiento dentro de la clase. En el contexto de este

trabajo, no hay diferencia observable entre estas dos definiciones, así que se ha optado por usar las dos palabras de manera indistinta.

2.1. Mecanismos funcionales versus Mecanismos no funcionales

Los mecanismos funcionales son los mecanismos que definen lo que las aplicaciones deben hacer, mientras que los no funcionales son los que adjetivan dichas funcionalidades. Así, por ejemplo, una aplicación puede monitorizar y controlar un determinado proceso físico (como en el caso de una caldera inteligente) o informático (como en el caso de una transacción bancaria). Y dichas funciones pueden diseñarse, o no, con tolerancia a fallos y a ataques, es decir asegurando su correcto funcionamiento no sólo en ausencia, sino también en presencia de problemas en los sistemas sobre los que se ejecutan (fallos HW), en su programación (fallos SW) o frente a intentos de manipulación de los activos informáticos utilizados (ataques a la integridad o la confidencialidad de los mensajes, por ejemplo).

Los mecanismos encaminados a robustecer la funcionalidad del sistema son mecanismos no funcionales y pueden estar o no, y proteger más o menos al sistema, en base al nivel de redundancia y al tipo de cifrado utilizado. En efecto, la protección del sistema frente a fallos y ataques no es gratuita.

Para tolerar los fallos HW es necesario introducir redundancia en la plataforma de ejecución, es decir, poner a disposición del sistema distintos recursos (elementos de procesamiento, almacenamiento, etc.) con vistas a replicar la ejecución y el estado de los programas. Así si falla uno de los recursos disponibles, sus réplicas pueden asumir la ejecución o el almacenamiento que realizaba el recurso con problemas hasta que éste sea reparado.

Para tolerar fallos SW, la ejecución del programa debe diversificarse de manera que la funcionalidad, o funcionalidades, crítica(s) del sistema se implementen utilizando distintos lenguajes de programación y/o distintos diseños para dicha(s) funcionalidad(es). Esto evitará, o al menos minimizará, la probabilidad de que un mismo

error de programación se de en las distintas replicas que, aunque ofrezcan la misma funcionalidad, habrán sido implementadas de distinta manera, e incluso llevando el principio de diversificación a sus últimas consecuencias, habrán sido desarrolladas por distintos equipos de desarrollo.

Para mejorar la seguridad en las comunicaciones es posible desplegar distintas estrategias, aunque sin duda el cifrado es actualmente la más extendida y soportada por todas las plataformas y lenguajes de programación. El cifrado de los mensajes puede ser simétrico o asimétrico, es decir con compartición de clave de cifrado privada o con el uso de una tupla de dos claves de cifrado, una pública y otra privada

Los diferentes mecanismos funcionales provienen del paradigma funcional de programación. Los decoradores tratados en este trabajo también pueden considerarse como un mecanismo funcional. Esta consideración se comprueba observando y analizando la programación funcional.

2.2. Problema del entrelazado de mecanismos funcionales y no funcionales

Actualmente la mayor parte de los proyectos software gestionan su complejidad sirviéndose de la orientación a objetos y del uso de patrones de diseño definidos partiendo de los principios básico que ofrece este paradigma de programación, como son las clases y sus instancias, los objetos, sus métodos y atributos, y las relaciones existentes entre clases, como la herencia o el polimorfismo.

El recurso de aprendizaje *Real Python* explica que la Programación Orientada a Objetos es un paradigma de programación que proporciona un medio de estructurar los programas para que las propiedades y comportamientos se agrupen en objetos individuales. Estos objetos son funciones y clases que pertenecen a los archivos (módulos) desarrollados en *Python*. [20]

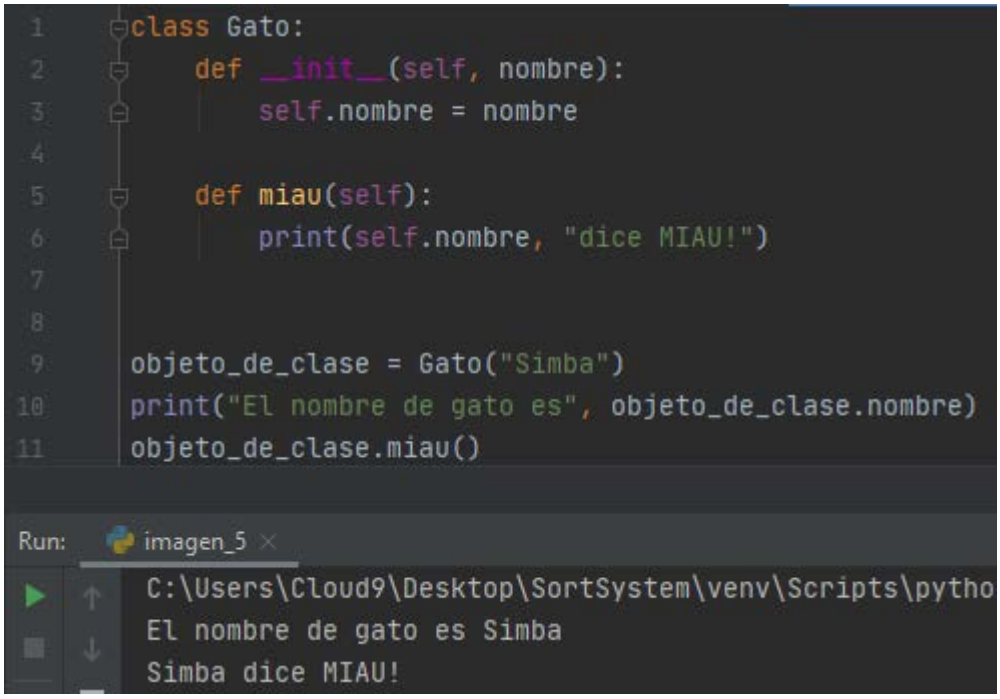


Imagen 2. Ejemplo de la programación orientada a objetos.

En el ejemplo presentado en la imagen 2 podemos observar que existe el objeto tipo clase *Gato,* que tiene el constructor de la clase *__init__*() y otro objeto tipo función *miau*. Al instanciar la clase pasando el nombre Simba a su constructor, obtenemos la instancia del objeto de la clase. Luego podemos usar esta instancia para obtener la propiedad de objeto que es *nombre* y ejecutar su comportamiento *miau*.

El problema de la orientación a objetos es que sólo permite clasificar a las entidades que pueblan el sistema por clase y no por el tipo de mecanismo que implementan o ayudan a implementar. Así la clase Gato permitirá crear distintos objetos de tipo Gato, e incluso, mediante herencia, sería posible instanciar distintos tipos de Gato (Persa, Siamés o Callejero, por ejemplo). Sin embargo cada uno de estos Gatos podrás ser o no diabético y tendrá por tanto una mayor o menor resistencia a problemas de tipo cardiovascular o de sordera. Por tanto, los tratamientos que deberían seguir para paliar su diabetes serán diferente.

En un sistema informático ocurre lo mismo, un cliente puede comunicar con un servidor y distintos clientes pueden requerir distintas aproximaciones para paliar la ausencia de un servidor en el momento de la conexión. Algunos clientes pueden simplemente no gestionar el problema. Otros tal vez generen un mensaje de error. Los habrá también que asuman que la red

puede estar congestionada y que tal vez el problema no es un problema del servidor, con lo que realizarán varios reintentos antes de dar al servidor por inalcanzable. Y finalmente, los habrá que además del error generen una alerta para que el servicio vuelva a ser lanzado.

Como vemos existen distintas aproximaciones para la resolución del problema y todas son válidas en la medida en la que se adecuen a los requerimientos del servicio establecidos y a las hipótesis de fallo del servidor asumidas. Además hay que tener en cuenta que estos requerimientos y estas hipótesis pueden variar a lo largo del tiempo por distintas circunstancias, con lo que el sistema debe poder evolucionar y adaptarse en consecuencia.

El problema es que el despliegue de las estrategias descritas involucra tanto al cliente como al servidor. En efecto, aunque el cliente sea el que detecte el problema y lo comunique, el servidor debe poderse relanzar, y además su estado (tanto el de sus atributos, como el de las invocaciones servidas) debe poderse restaurar, para que el problema se pueda solventar. Imaginemos que el servidor lleva la cuenta de las peticiones servidas y que factura en consecuencia a los clientes. Obviamente, en este escenario de ejemplo, relanzar el servidor olvidando las peticiones ya procesadas no es una opción.

Este ejemplo permite ya intuir la complejidad del despliegue de un mecanismo encaminado a asegurar una característica de confiabilidad (una alta disponibilidad) en un servicio. El problema es que el mecanismo de tolerancia a la caída del servidor requiere para de código que debe desplegarse tanto en los clientes como en el servidor, y por tanto, el mantenimiento y actualización de este código será más complejo que el código funcional del cliente, centrado en el cliente, y el del servidor, implementado en el código del servidor. Además la orientación a objetos, que permite estructurar muy bien la funcionalidad de los programas no permite capturar adecuadamente los mecanismos no funcionales que como hemos intentado explicar, suelen ser transversales, y por tanto, afectar a la funcionalidad de varios niveles del sistema o de varias entidades del mismo.

2.2.1. Cross-cutting and tangling concerns

Los *Cross-cutting concerns* son mecanismos transversales al sistema cuyas implementaciones requieren de varias partes implementadas en distintos módulos del sistema. Esto implica que distintos mecanismos, funcionales o no, coexistan en un

mismo código, lo que dificulta su mantenimiento y reutilización. Por ejemplo, en la imagen 3, se presenta el código de un método donde se entremezcla (del inglés *tangling)* código de monitorización de la actividad del método (*logging*), la lógica relativa a la gestión de una transacciones bancaria y el código propio de la operación a realizar, o código de negocio (*business*).

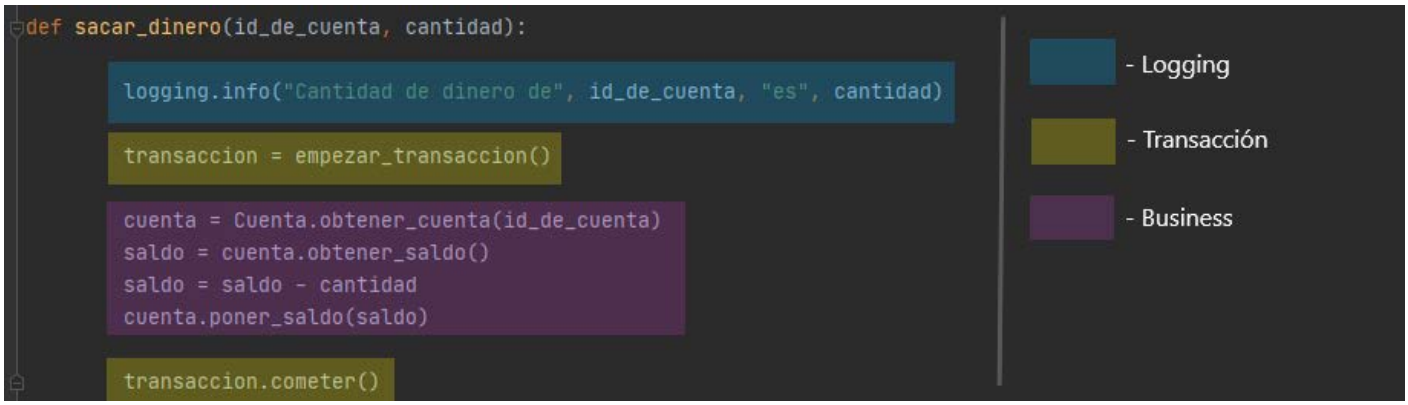


Imagen 3. Ejemplo de problema de entrelazado.

Si el sistema no se monitorizara, el código de logging podría eliminarse, y si no se gestionaran las operaciones como transacciones, el relativo a las mismas también, quedando entonces el código del método sólo en su lógica de negocio. El problema de entrelazado ilustrado en la imagen 6 aparece aquí con el uso de *logging,* que no pertenece al propósito de negocio de la función. Si además se tuvieran que monitorizar todas las operaciones bancarias, el mecanismo de logging debería desplegarse en todos los métodos de la aplicación, con lo que cualquier alteración de este comportaría una complejidad importante, y un trabajo no despreciable, al tener que modificarse, al menos, tantas líneas de código como puntos de monitorización existieran en el programa.

El problema de entrelazado de mecanismos aumenta también la dependencia de los módulos y las tecnologías usadas en los mismos. Esto supone toda una dificultad para la legibilidad de código, porque, desde el momento en el que distintos mecanismos entrelazan sus códigos, las funciones disponen de más mecanismos que propósitos y se pueden generar fuertes dependencias entre mecanismos distintos. Además, el desarrollador, a pesar de entender para qué existe la función y/o clase y qué

procedimiento tiene, debe examinar, además, las tecnologías que hay dentro de estas, entenderlas y modificar su código evitando efectos colaterales indeseados.

Habiendo dejado clara la interdependencia que puede llegar a establecerse entre mecanismos funcionales y no funcionales en las aplicaciones, y los impedimentos que supone el entrelazado de código para la reutilización y mantenimiento de dichos mecanismos, centrémonos ahora en estudiar de qué manera es posible ofrecer lo que en inglés se denomina una buena *separation of concerns*, que no es sólo una separación entre mecanismos, sino también una separación de responsabilidades de desarrollo. De hecho, el objetivo es el de permitir que los especialistas en mecanismos funcionales desarrollen dichos mecanismos, y que los especialistas en mecanismos no funcionales se centren en la generación de estos otros mecanismos. El problema de cómo integrar posteriormente ambos mecanismos y de qué abstracciones ofrecer a cada uno para poder trabajar correctamente cuando se integras, se ha resuelto de manera distinta con distintas tecnologías que pasamos a estudiar.

2.3. Separation of concerns

Una solución fácil y elegante al problema de entrelazado es el concepto fundamental del desarrollo de *software*: *separation of concerns* (SoC) o separación de mecanismos/responsabilidades. Este concepto se describe fácilmente: No se trata de escribir un proyecto como un bloque único de código sino, en su lugar, dividir el código en pequeñas piezas finalizadas del sistema, cada una capaz de completar una simple tarea distinta.

2.3.1. Aspectos básicos

La separación de mecanismos y responsabilidades se divide en dos niveles generales: SoC de funciones de programación y SoC de módulos.

- ***Separation of concerns* de funciones de programación.**

La idea de este concepto consiste en evitar escribir funciones grandes. Cada función ha de tener algún propósito concreto y un código conciso, que esté enfocado en realizar este propósito. Si la función empezara a incrementarse obteniendo fragmentos de código que no cumplan con su propósito o que tengan un propósito distinto al suyo, entonces hablaríamos de la primera señal de violación del concepto de SoC.

- **Modelo del sistema manipulado en cada nivel**

Un programa que implementa el SoC correctamente recibe el nombre de programa modular. La modularidad y la separación de intereses, se logran encapsulando la información dentro de una sección de código. *Microsoft Patterns & Practices Team* manifiestan que los diseños por capas en los sistemas de información son otra implementación para la separación de intereses. Algunos ejemplos de estas capas son la capa de presentación, la capa de lógica *business*, la capa de acceso a datos etc. [13]

La modularidad o programación modular, por su parte, es otro concepto de diseño de software que está enfocado en separar la funcionalidad del proyecto en diversos módulos diferentes, independientes y reutilizables. Es decir, cada uno de ellos contiene todo lo necesario para ejecutar sólo un propósito o un comportamiento de la funcionalidad necesaria.

Estos tipos de *separation of concerns* nos permiten construir un sistema que sea manipulado a todos los niveles: manipulamos los módulos mantenibles y reutilizables y manipulamos las funciones y/u otros elementos (clases, por ejemplo) con cierto comportamiento encapsulado.

Si llevamos a un plano más concreto este concepto de SoC, nos encontraremos con dos tipos de mecanismos complementarios. Unos orientados al control del comportamiento de las entidades y otros orientados al control de su estado.

Los mecanismos de control del comportamiento son básicamente mecanismos de encapsulación (o wrapping en inglés) que siguen el patrón de diseño de *software* llamado *interceptor pattern*. El interceptor (imagen 4), elemento principal de este patrón, se utiliza cuando los sistemas quieran ofrecer una forma para cambiar o

potenciar su ciclo de procesamiento habitual. Es decir, usando el interceptor en el sistema podremos activarlo solo cuando se necesite procesar su comportamiento encapsulando, e incluso podemos llegar a inhibirlo o a reemplazar la activación del comportamiento esperado por otro alternativo, alterando así en el procesamiento de sistema.

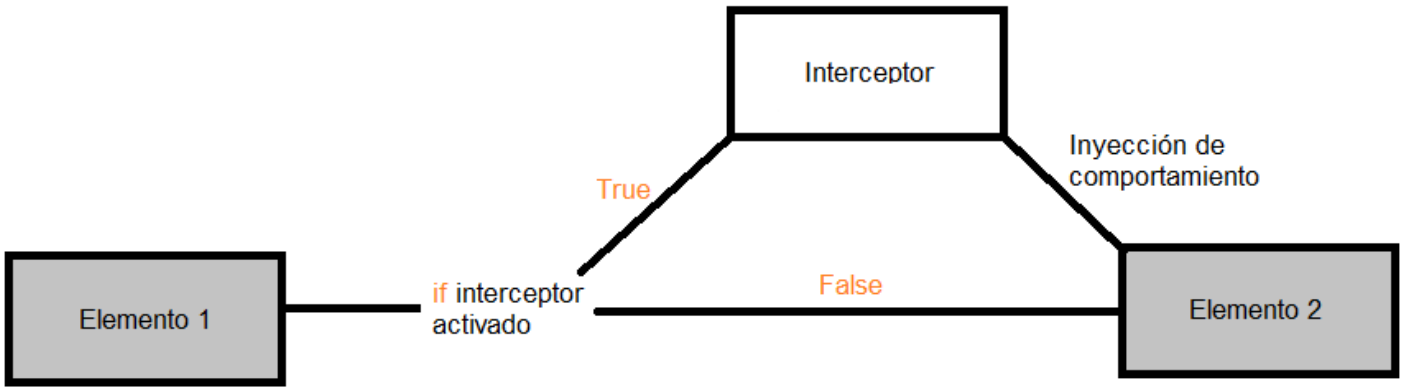


Imagen 4. Ejemplo de interceptor.

El uso de interceptores permite centrar la implementación de los mecanismos no funcionales en el propio interceptor ya que es posible alterar el comportamiento activado e incluso, tal y como muestra la imagen 7, inyectar un nuevo comportamiento si se desea.

Sin embargo, en ocasiones la alteración del comportamiento del sistema no es suficiente para desplegar mecanismos de confiabilidad o seguridad. Por ejemplo, el estado de un servidor debe poder restaurarse cuando el servidor se relanza a ejecución para tolerar un fallo, o los mensajes intercambiados entre entidades del sistema deben poder alterarse para poder ser cifrados. Para actuar sobre el estado del sistema existen distintos mecanismos reflexivos encaminados a permitir la introspección del estado y la intercesión o alteración de dicho estado.

El mecanismo reflexivo de introspección ofrece la capacidad a un programa de observar y por lo tanto razonar sobre su propio estado. En otras palabras, la introspección es la habilidad de sistema de investigar valores, metadatos, propiedades y/o funciones de un

objeto durante su propia ejecución. Por ejemplo, en *Python* la introspección permite obtener varios atributos diversos de las clases o funciones que llevan información consigo. Algunos de ellos son: "*__class__*", "*__doc__*", "*__hash__*", "*__init__*", "*__len__*", "*__sizeof__*", "*__str__*", "*count*", "*index*" y etc.

Por otro lado la intercesión es la capacidad de un programa de modificar su propio estado de ejecución o alterar su propia interpretación o significado. Es decir, la intercesión permite modificar el estado de la ejecución del programa.

Existen distintas soluciones que permiten utilizar de forma combinada los mecanismos de intercepción, introspección e intercesión con distintos objetivos. Pasemos a estudiarlos.

2.3.2. Metaprogramación y compiladores abiertos

Yannis Lilis y Anthony Savidis declaran que la metaprogramación es el concepto de programación que proporciona la habilidad a los programas de manejar a otros programas o a los elementos del código como si fueran su propia data. Esto permite analizar y transformar los sistemas existentes o generar otros nuevos. [9] La escritura de compiladores, ensambladores, intérpretes, *enlazadores, cargadores,* depuradores, *profilers* y etc. son elementos propios de la metaprogramación. Pero, a pesar de esto, este concepto se usa también para describir a los programas que operan sobre sí mismos, es decir, que analizan su estructura y la adaptan en consecuencia. Para ello se emplean las técnicas de introspección e intercesión anteriormente introducidas.

La metaprogramación suele funcionar a través de tres enfoques:

1. El primer enfoque consiste en exponer al código de permitir que un programa pueda personalizar la ejecución de otros programas a través de una serie de APIs que permiten actuar sobre el soporte de ejecución del sistema (su operativo, o las librerías del lenguaje necesarias para la ejecución de los programas).

2. El segundo enfoque es la ejecución dinámica de las expresiones que contienen comandos de programación, a menudo, compuestos por cadenas. También pueden estar compuestas por otros métodos que utilicen argumentos o contexto, como *JavaScript*. Así, los programas son capaces de escribir programas. [2]

3. El tercer enfoque consiste en salir del lenguaje por completo. Los sistemas de transformación de programas de propósito general, como los compiladores, que aceptan descripciones del lenguaje y llevan a cabo transformaciones arbitrarias en esos lenguajes, son implementaciones directas de la metaprogramación general. Esto permite que la metaprogramación se aplique a prácticamente cualquier idioma de destino sin importar si ese idioma de destino tiene alguna capacidad de metaprogramación propia. [1]

Un ejemplo de compilador desarrollado mediante la metaprogramación y que emplea la metaprogramación en su ejecución es el compilador *OpenC++*. Según Grzegorz Jakacki, OpenC++ es un compilador de tipo metacompilador [24]. Los metacompiladores son una herramienta de desarrollo software que se utiliza principalmente en la construcción de otros compiladores, traductores e intérpretes para otros lenguajes de programación. La entrada de un metacompilador es un programa escrito en un metalenguaje de programación, diseñado principalmente con el propósito de construir compiladores.

Grzegorz, en su artículo, manifiesta que el equipo de Shigeru Chiba y Gregor Kiczales a principios de los 90 definió un sistema de reflexión para el lennguaje *C++*, y lo implementó en un compilador de *front-end* de *C++* llamado *OpenC++.* Los objetivos principales de su trabajo era generar una herramienta que ofreciera implementaciones fáciles de extensiones de lenguaje, optimizaciones de código personalizadas y aumentos no intrusivos de código reutilizado. Algunas de estas ideas llegaron a conocerse posteriormente como programación orientada a aspectos (véase el apartado 2.3.3.). [3]

Jakacki, por su parte, explica la funcionalidad de este compilador afirmando que *OpenC++* parsea y analiza una unidad de *C++* a traducir, creando su representación abstracta en forma de objetos que representan entidades de programas de *C++.* Es

decir: expresiones, entornos, tipos, clases y miembros de clases. Todos estos objetos existen en un compilador de *OpenC++* (a nivel meta), y no deben confundirse con los objetos que existirán en el programa compilado durante la ejecución (nivel base). Para evitar confusiones, los objetos y clases de la *API* de introspección de *OpenC++* se denominan metaobjetos y metaclases respectivamente.

La API de introspección pone a disposición los *plug-ins* de *C++* metaobjetos, que pueden añadirse de forma dinámica o estática al compilador y ejecutarse durante la compilación. Los *plug-ins* pueden inspeccionar y modificar los metaobjetos. Una vez que los *plug-ins* hayan terminado, *OpenC++* produce el código fuente que refleja el estado actual de los metaobjetos, enviándolo después al compilador *back-end*. [24]

En el ámbito de la confiabilidad esta capacidad del compilador para analizar la estructura de un programa y transformarla fue utilizada para generar programas tolerantes a fallos en C++ partiendo de programas que no integraban características de confiabilidad [11]. La idea era simple. Los mecanismos de tolerancia a fallos se desarrollaban como metaprogramas que aplicaban transformaciones a los programas escritos en C++. De esta manera, por ejemplo, un fallo transitorio en el procesamiento de una función podría tolerarse si la llamada a dicha función se transformaba en tres llamadas sucesivas, realizando luego una votación mayoritaria sobre los resultados obtenidos de dichas llamadas. Este principio aplicado a funciones consideradas como críticas permite tolerar un error HW de tipo transitorio mediante redundancia temporal, es decir, la reejecución de la misma funcionalidad varias veces. Con 3 reejecuciones será posible tolerar un fallo transitorio como máximo, pero con 5 se podrán tolerar hasta 2.

Este mismo principio se ha utilizado en otros lenguajes, como en el caso de OpenJava [12], un compilador abierto de Java, e incluso con máquinas virtuales en las que la metaprogramación permite la modificación del comportamiento del cargador de clases del lenguaje, como en el caso de JavaAssist [7].

2.3.3. Reflexividad en lenguajes y orientación a aspectos

Como se ha observado anteriormente, la reflexibilidad a nivel de lenguaje de programación hace referencia a la habilidad de dicho lenguaje para poder modificar su propia estructura y comportamiento. La reflexibilidad otorga algunas características que son actualmente bastante comunes en lenguajes de alto nivel, como *Java* o *Python*. Algunos ejemplos de estas características son:

- Capacidad de encontrar y transformar bloques de código, protocolos, clases, funciones etc. como objetos de f*irst-class objects* en el *runtime*.
- Habilidad de convertir un *string* correspondiente al nombre simbólico de clase o función, una referencia a estas o su invocación.
- Capacidad de definir un *string* como un bloque de código base en el *runtime*.
- Habilidad de crear un nuevo interpretador para el *bytecode* de lenguaje con el fin de definir un nuevo significado o propósito para fragmentos del código, construcción de programación, etc.

Tanto Java como Python muestran maneras diferentes de explorar y examinar los atributos, clases y métodos en las clases. Los dos ofrecen mecanismos de introspección bastante interesantes.

En *Python* se usa el método *type*(variable), que muestra el tipo de variable pasándola como un argumento de este. En *Java,* para ello se emplea la construcción *.getClass*(), añadiéndolo justo después del nombre de la variable. Para determinar si la variable es una instancia de la clase o *child* de la clase específica, en *Python* se usa la función *isinstance*(variable, clase), donde el primer argumento es el elemento cuya instancia se comprueba y el segundo es el elemento con el que se comprueba la variable. Para determinar lo mismo en *Java* se usa la construcción variable *instanceof* clase.

Para obtener una lista de atributos diferentes al que lleva el elemento, en *Python* se utiliza la función *dir*(). Los detalles específicos se obtienen usando el método

getattr(variable, atributo). En *Java,* por su lado se utiliza *getFields*(), aunque para los métodos públicos se deba usar *.getDeclaredMethods*().

Si el atributo es ejecutable, puede ser llamado como una función ordinaria invocándolo al igual que la función en *Python* usando paréntesis (por ejemplo, *atributo*() es el elemento llamable). Para *Java* se emplea la construcción *atributo.invoke*().

La reflexión es un concepto crucial para la programación orientada a los aspectos (AOP del inglés *Aspect Oriented Programming*). Shigeru Chiba dice que la base de la programación orientada a aspectos es el concepto de la reflexión, definiendo a la AOP como la versión avanzada de esta [24]. El profesor Chiba explica que la AOP es un paradigma emergente para implementar mecanismos transversales al código de las aplicaciones, lo que se ha dado en llamar en esta memoria mecanismos no funcionales.

AspectJ es una herramienta de AOP soportada como proyecto por la *Eclipse Foundation,* y según su sitio web no es otra cosa que una extensión orientada a aspectos de *Java*. *AspectJ* permite una modularización de los mecanismos no funcionales, a través de la comprobación y el manejo de errores, la sincronización, el comportamiento sensible al contexto, la optimización del rendimiento, la supervisión y el registro, el apoyo a la depuración y los protocolos multiobjeto. [5]

La imagen 5 muestra el ejemplo de una clase que vamos a decorar con el aspecto que se da en la imagen 6. Esta clase es muy sencilla y tiene una función *hola*() que imprime la palabra “hola” por consola.

```
public class OrdinaryClass {

    public static void hola() {
        System.out.print("Hola ");
    }

    public static void main(String[] args) {
        hola();
    }

}
```

Imagen 5. Ejemplo de clase decorado con el aspecto.

El aspecto de la imagen 6 implementa la construcción *before*(), que se aplicará sobre cualquier método del programa cuyo nombre sea hola . El aspecto también despliega una construcción *around*(), que se comporta como un *wrapper* (véase el párrafo 2.3.4.) y que se aplica sobre cualquier llamada al método hola del programa. Finalmente la construcción *after*() actuará tras la llamada al método hola del programa. Por tanto, estas tres sencillas declaraciones permiten definir no sólo pre- (*before*)y post- (*after)* procesamiento a los métodos del programa, sino también interceptar (*around*) la ejecución de dichos métodos.

Lo más interesante es que los métodos en cuestión se definen mediante expresiones regulares, lo que ofrece una capacidad de expresión enorme. Así por ejemplo, podríamos coger un programa muy complejo y declarar que deseamos actuar sobre todos los métodos públicos existentes en el mismo, simplemente utilizando la expresión: `public *.*(..)`.

```
2  public aspect AspectExample {
3
4      before() : execution(* hola(..))
5      {
6          System.out.println("Me han hecho decir: ");
7      }
8
9      void around(): call(* hola(..))
10     {
11         System.out.println("[[Empiezo a hacer wrapper de la funcion]]");
12         proceed();
13         System.out.println("[[Termino de hacer wrapper de la funcion]]");
14     }
15
16     after() : execution(* hola(..))
17     {
18         System.out.println("Mundo!");
19     }
20
21 }
```

```
Console ✕  Cross References
<terminated> OrdinaryClass (1) [Java Application] C:\Program Files\Java\jdk-11.0.5\bin\javaw.exe (5 jul.
[[Empiezo a hacer wrapper de la funcion]]
Me han hecho decir:
Hola Mundo!
[[Termino de hacer wrapper de la funcion]]
```

Imagen 6. Ejemplo de aspecto para decorar la clase y su salida de ejecución.

Si el aspecto de la Imagen 6 se despliega sobre el código de la Imagen 5, la salida de este aspecto se observa en la parte inferior de la Imagen 6. Primero la ejecución llega a la *around*(). Una vez dentro, ejecuta el código del método cuya llamada ha sido interceptada usando la llamada *proceed*() y, al hacerlo, activa el código que hayamos ubicado en las declaraciones de *before*() y *after*() del aspecto.

2.3.4. *Wrapping*

Tal y como es posible entrever, el principio básico que subyace a la utilización de todas estas tecnologías de separación de mecanismos (recordemos que esta es nuestra traducción para *separation of concerns*) es el principio de intercepción o *wrapping* en inglés. Así, un compilador puede ser considerado abierto cuando el proceso de análisis sintáctico que aplica puede personalizarse, es decir, cuando es posible interceptar y reescribir las funciones que llevan a cabo dicho análisis. Lo mismo sucede con los aspectos, que permiten interceptar las invocaciones de los métodos con las contrucciones *before, after y around*, pudiendo incluso inhibir la llamada al método interceptado o modificarla para atender a un nuevo conjunto de requerimientos no funcionales.

Un *wrapper* es cualquier entidad que encapsula otro elemento del sistema. Se utiliza con dos fines primordiales: convertir los datos en un formato compatible y ocultar la complejidad de la entidad subyacente mediante la abstracción. Algunos ejemplos de técnicas de *wrapping* son los *wrappers* de objetos, los *wrappers* de funciones y los *wrappers* de controladores [4].

El *wrapper* de objetos es un elemento que encapsula un tipo primitivo de datos u otro objeto. Este elemento convierte el dato primitivo en una clase. Así, el desarrollador puede usar un método como *toUpperCase*() para modificar los datos.

El *wrapper* de funciones, por otro lado, envuelve uno o más métodos de sistema, pudiendo envolver una sola función para permitir que funcione con un código más nuevo o antiguo.

Por su parte, el *wrapper* de controladores permite que estos funcionen con un sistema operativo que de otro modo sería incompatible. Por ejemplo, es posible que una tarjeta gráfica antigua sólo admita controladores diseñados para Windows 7. Si no hubiera un controlador para Windows 10, el *wrapper* podría servir a modo de adaptador, permitiendo a la tarjeta gráfica utilizar el controlador de Windows 7 en Windows 10.

Un ejemplo de uso de *wrappers* con fines no funcionales son los interceptores de *CORBA* (*Common Object Request Broker Architecture*)

La imagen 7 muestra cómo funcionan los interceptores en el sistema CORBA. Estos interceptores envuelven los objetos, mensajes o datos que intercambian servidores y clientes para transformar o añadir nueva funcionalidades y/o comportamiento en estos [15].

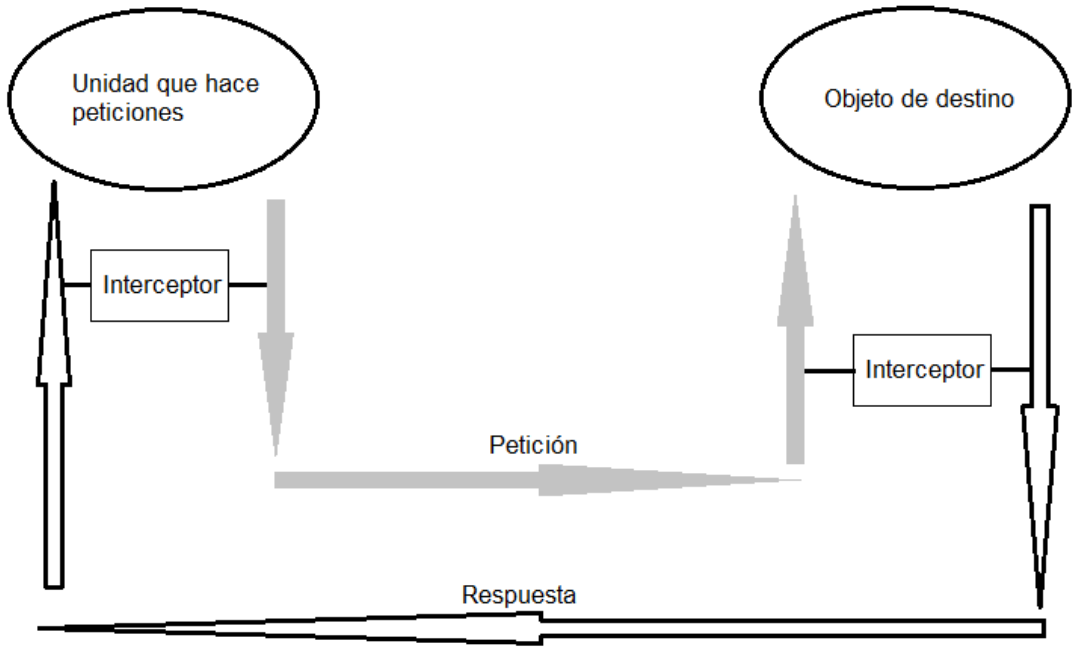


Imagen 7. Ejemplo de modelo de sistema *CORBA* con interceptores.

De hecho actuando en los extremos de la comunicación es relativamente sencillo cifrar las comunicaciones entre un cliente y un servidor. El cliente enviará un mensaje al servidor que será interceptado en origen, cifrado y reenviado a su destino. Cuando el servidor lo reciba se interceptará la comunicación, se descifrará el mensaje y se remitirá a su método receptor. Lo mismo, pero en sentido contrario se hará con el mensaje de respuesta. En este caso, cuando se intercepte la respuesta del servidor se cifrará y reenviará; y cuando el cliente la reciba, se interceptará la recepción, de descifrará el mensaje y se remitirá al cliente. Será así como el mecanismo no funcional de cifrado/descifrado actuará de manera transparente a los mecanismos funcionales que tanto cliente, como servidor implementen.

Este mismo principio de intercepción sirve de base a la definición de la especificación tolerante a fallos de CORBA, denominada FT-CORBA[26].

2.3.5. *Uso en sistemas tolerantes a fallos*

El sistema *FRIENDS* (*Flexible and Reusable Implementation Enviroment for your Next Dependable System*) desarrollado en el laboratorio de investigaciones LAAS-CNRS, es una arquitectura para el diseño y desarrollo de sistemas distribuidos tolerantes a fallos. Esta arquitectura aúna gran parte de los principios anteriormente introducidos con el objetivo de introducir mecanismos de tolerancia a fallos, seguridad y distribución a programas desarrollados en C/C++.

En su primera versión, FRIENDS v1 proporciona librerías de metaobjetos para la tolerancia a fallos, comunicaciones seguras y aplicaciones distribuidas basadas en grupos. El uso de metaobjetos proporciona una buena separación entre los mecanismos y las aplicaciones. Los metaobjetos pueden ser utilizados de forma transparente por las aplicaciones, y pueden estar confeccionados según las necesidades determinadas de aplicación y arquitectura, y según sus propiedades subyacentes [6]. Así se consiguen las siguientes características:

- Facilidad de uso: Los mecanismos que implementan la tolerancia a fallos y la comunicación segura deben ser utilizados y compuestos por los desarrolladores

de aplicaciones de manera sencilla. Esto también debería ser cierto para los propios desarrolladores de los mecanismos, para quienes, por ejemplo, los mecanismos de comunicación de grupo deberían ser fáciles de utilizar. Cuando se utilizan diferentes mecanismos en diferentes aplicaciones, el programador o el usuario de una aplicación determinada debería ser capaz de seleccionar el mecanismo adecuado, aunque su aplicación permanezca oculta.

- Reutilización: Desde el punto de vista del desarrollador, la reutilización puede adoptar una de las dos formas siguientes: la generalidad y la extensibilidad. La generalidad consiste en que los mecanismos pueden ser reutilizados sin modificación en nuevas aplicaciones. La extensibilidad, por otro lado, significa que parte de un mecanismo existente debe actualizarse y modificarse para obtener un mecanismo que cumpla diferentes requisitos.

  Los métodos y lenguajes de diseño orientados a los objetos constituyen una buena forma de obtener ambas propiedades: la abstracción y la encapsulación benefician a la generalidad, mientras que la herencia o la delegación permiten un incremento del desarrollo, garantizando la extensibilidad.

- Composición: Los autores explican que la composición como la capacidad de utilizar diversos mecanismos independientes dentro de la misma aplicación, dependiendo de las características y requisitos de esta. Por ejemplo, se pueden utilizar varias estrategias de tolerancia a fallos para los diferentes objetos de una misma aplicación sin que se produzca ningún choque. Los mecanismos también pueden ser de diferentes tipos: una aplicación puede necesitar tanto mecanismos de tolerancia a fallos como mecanismos de seguridad, y en función de cómo se compongan los mecanismos tendremos soluciones seguras tolerantes a fallos o soluciones tolerantes a fallos con características de seguridad.

Como vemos, todos estos principios se relacionan directamente con los conceptos de separación de mecanismos, intercepción, reutilización y mantenibilidad de código , sin perder de vista la confiabilidad de la solución informática producida.

Un sistema desarrollado con estos principios será indudablemente más económico para las empresas y requerirá menos tiempo de test y verificación.

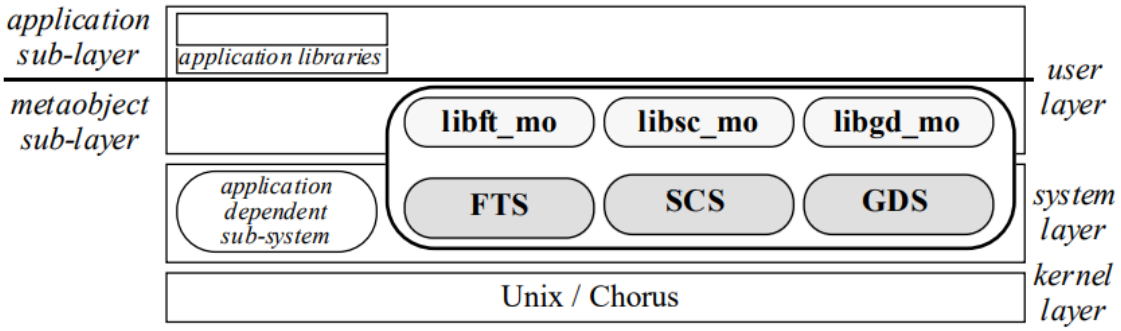


Imagen 8. Arquitectura general del sistema. [6]

En la imagen 8 se observa la arquitectura general de sistema *FRIENDS v1*. La plataforma proporciona un conjunto de subsistemas y bibliotecas (*libft_mo, libsc_mo, libgd_mo*) de clases de metaobjetos. La implementación de cualquier abstracción (tolerancia a fallos, comunicación segura, distribución) se divide así en una biblioteca de clases de metaobjetos y el subsistema correspondiente, abarcando así al menos parcialmente las capas de usuario y de sistema.

El problema de FRIENDS v1 es que la composición de mecanismos queda en manos del desarrollador final que puede tener unos conocimientos limitados de confiabilidad y seguridad y además el tipo de distribución que soportaba no era compatible con soluciones estandarizadas como las propuestas por la OMG a través de CORBA.

En la segunda versión de la plataforma, FRIENDS v2, trató de paliar estas deficiencias explotando las capacidades de compilación abierta de OpenC++ para automatizar el despliegue de mecanismos de seguridad y TaF sobre soluciones distribuidas de tipo CORBA [26].

El objetivo perseguido por los desarrolladores de FRIENDS era muy ambicioso y fue objeto de varias tesis doctorales. En esta tesina deseamos explorar la viabilidad de utilizar los mecanismos de intercepción (wrapping) de Python, llamados decoradores, nuestra traducción al término inglés decorators, con un objetivo similar al ya

mencionado: permitir el desarrollo de mecanismos no funcionales de TaF y seguridad en un contexto con necesidades propias de confiabilidad, como es el de un cliente interactuando con un servidor. Veamos en el siguiente capítulo qué son y cómo se pueden utilizar los decoradores de Python.

CAPÍTULO 3: SEPARACIÓN DE MECANISMOS EN PYTHON

Como ya hemos visto, al programar algunos mecanismos, estos no pueden encapsularse perfectamente en objetos, sino que deben estar dispersas por todo el código existente. Hablamos de mecanismos de, por ejemplo, traza y depuración, el uso de proxies a servidores y soluciones de cifrado y tolerancia a fallos. Todo ello genera problemas del entrelazado y otros obstáculos que pueden convertirse en problemas para los desarrolladores .

Python, según el blog oficial del su creador, Guido van Rossum, es un lenguaje interpretado de alto nivel orientado a objetos con semántica dinámica [25]. Python, está sujeto a los mismos problemas de entrelazado de código que ya hemos introducido. Sin embargo, integra un tipo de solución, los decoradores, *decorators* en inglés, bastante prometedora para inducir esa separación de mecanismos tan necesaria cuando se desarrollan soluciones de alta confiabilidad.

A través de los decoradores, aprovechando sus ventajas de interpretación, podremos modificar los elementos de los proyectos. En otras palabras, se trata de modificar dinámicamente las funciones y las clases durante la interpretación del código, encapsulando el comportamiento que afecta a las clases y funciones, o, en ciertas situaciones, ejecutando el código original sin realizar cambios .

Por lo tanto, en este capítulo nos referiremos a los decoradores como una solución posible a los problemas de entrelazado de código y la confiabilidad de aplicaciones aplicándolos como una tecnología para la separación de mecanismos en el lenguaje interpretado Python.

3.1. Interpretación de código

La interpretación de código se realiza mediante la ejecución secuencial, es decir la ejecución de instrucciones escritas línea por la línea directamente sin compilar el código fuente a instrucciones en lenguaje de máquina. De hecho, en Python a esto se dedica el programa intérprete, traduciendo cada instrucción en varias subrutinas de código máquina. Además, este lenguaje de programación emplea el intérprete estándar de bytecode CPython, escrito por el equipo de desarrolladores de Guido van Rossum en C y Python. Algunas ventajas que proporcionan los lenguajes interpretados son las siguientes:

Multiplataforma: Los lenguajes interpretados no requieren el uso de una plataforma en particular. Por ejemplo, el intérprete de Python puede ser instalado en diferentes sistemas operativos tipo Windows, Linux o MacOS. También existen varias opciones para instalar los entornos de desarrollo integrado con intérpretes instalados para iOS y Android.

Reflexión: Como se dijo en el capítulo anterior, durante la ejecución de código, los ordenadores procesan los datos y los modifican sin alterar su propio código. Sin embargo, en la mayoría de las arquitecturas informáticas modernas, el código se almacena como lo hacen los datos, y en algunos lenguajes de programación es posible procesar el código del programa como si fueran datos, lo que ofrece la posibilidad de poder cambiar el código del programa durante su ejecución. A esto lo denominamos reflexividad en el lenguaje.

Tipos dinámicos (Introspección de tipos): Python no requiere indicar explícitamente el tipo de la variable definida porque la introspección de tipos le permite determinar el tipo de un objeto durante la ejecución del programa (imagen 9).

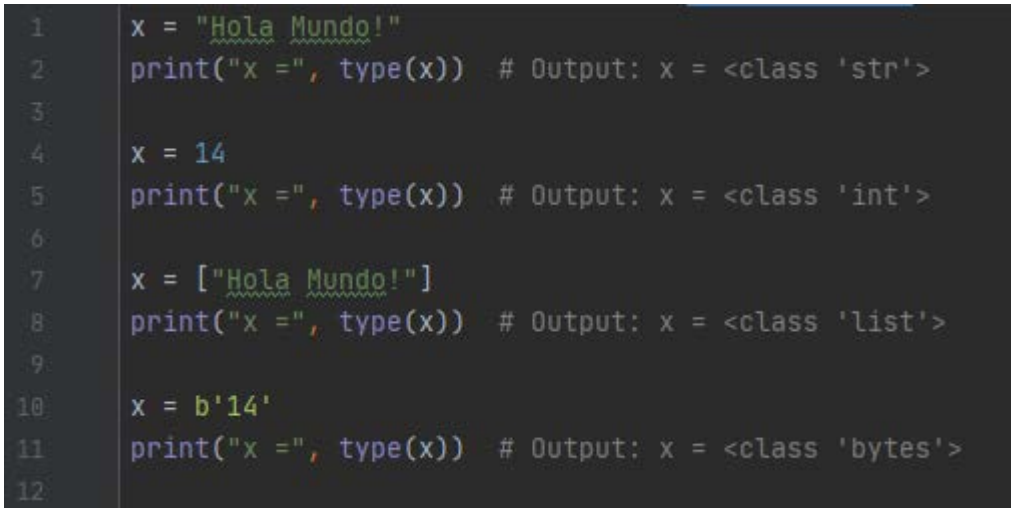


Imagen 9. Asignación de tipo de variable "x" sin indicar explícitamente su tipo

3.2. Uso de Decoradores

Como se ha mencionado, el uso de decoradores representa una solución a posibles problemas en sistemas que utilicen mecanismos no funcionales (de naturaleza transversal) y que deban ser reutilizables. Para poder hacer uso de los decoradores debemos entender primero qué son, para qué sirven y cómo funcionan.

3.2.1. Qué son los decoradores

Mark Lutz, en su libro *Learning Python,* describe a los decoradores como herramientas muy generales cuyo beneficio principal no es otro que la posibilidad de agregar diversos tipos de lógica a las funciones. El autor manifiesta que los decoradores pueden ser utilizados para administrar tanto las funciones como las llamadas posteriores a estas [10].

Luciano Ramalho, por su parte, define los decoradores de una manera todavía más concreta. El autor observa que un decorador es un elemento *llamable* y que toma como argumento otra función (la función decorada). El decorador puede realizar algún procesamiento con la función decorada, y la devuelve o la reemplaza por otra función u otro objeto *llamable* [18]. Sunil Kapil describe los decoradores como una utilidad de

Python que ayuda a añadir dinámicamente el comportamiento necesario a las funciones u objetos sin hacer cambios de estos [8].

Se puede concluir que los decoradores son una herramienta útil de Python con potencial para aplicar el concepto de reflexión con el objetivo de aumentar la confiabilidad de las aplicaciones gracias a la posibilidad de cambiar dinámicamente el comportamiento de los elementos decorados. Esta herramienta es imprescindible para separar los mecanismos desarrollados. De esta manera, se previene el efecto de entrelazado, toda una amenaza para la mantenibilidad del proyecto.

El proceso de desarrollo del sistema puede requerir a su programador la implementación de mecanismos de traza y depuración. Uno de estos mecanismos es el mecanismo de *logging,* que obtiene el tiempo de ejecución de las funciones, almacenándolo en un archivo o imprimiendo la información por consola. Una manera de implementar *logging* es importando librerías y añadiendo líneas de código de log dentro de estas funciones. Esta metodología, sin embargo, suele resultar incómoda y poco práctica a la hora de modificar el código, ya que requiere de la modificación de todos los métodos del programa que se deseen monitorizar.

En su lugar, resulta más simple añadir el decorador encima de estas funciones o clases. Así el programador protege el sistema de posibles errores que podrían ocurrir dentro del código en caso de no aplicar los decoradores. Además, al encapsular el *logging* dentro de un decorador separado, el programador se asegura de la alta mantenibilidad del proyecto al proponer la reutilización de dicho decorador.

Se garantiza entonces la mantenibilidad gracias a la posibilidad de cambiar el código en un único modulo en lugar de hacerlo en diferentes partes de código del sistema para hacer cambios en el mecanismo implementado. Como vemos, los decoradores sirven fundamentalmente para mejorar la modularidad de los programas en Python.

3.2.3. Cómo se utilizan los decoradores

Por su naturaleza, los decoradores pueden utilizarse como clases o funciones diferentes. Hay dos maneras ligeramente distintas de escribir decoradores, pero ambas producen el mismo resultado: el cambio del comportamiento de la función o clase decorada.

Los decoradores-clases se escriben en el código de la misma manera que se escriben los decoradores-funciones. Para aplicar un decorador a un elemento del sistema se necesita poner la arroba y el nombre del decorador encima del elemento decorado. En otras palabras, anotarlo (imagen 10).

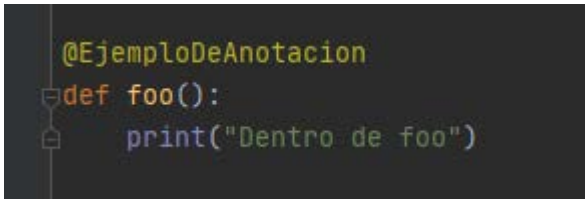


Imagen 10. Aplicación de decorador-clase *EjemploDeAnotacion* sobre la función *foo*().

En este ejemplo decoramos la función *foo()* escribiendo el decorador *EjemploDeAnotacion* encima de *foo()*. Las funciones en Python, según la documentación oficial del lenguaje [17], son objetos de primera clase. Esto significa que las funciones pueden utilizarse como argumentos y ser pasados a otras funciones como cualquier otro objeto tipo *string, int, float* y etc. Así que, en el *runtime,* la ejecución que llegue a la función *foo*() observa que esta función está decorada y pasa esta función como argumento al decorador EjemploDeAnotacion.

El decorador EjemploDeAnotacion es un decorador-clase, es decir, se define como una clase y, por tanto, tiene un constructor de clase, *__init__*(), y una función *__call__*() que es propia de la librería Python (imagen 11). El constructor realiza la decoración, y la función *__call__*() se aplica al llamar a la función decorada.

```python
class EjemploDeAnotacion:
    # El decorador-clase

    def __init__(self, decorated_function):
        # Ejecutamos el código de función __init__ al instanciar la función decorada

        print("Dentro de EjemploDeAnotacion.__init__")
        decorated_function()  # Ejecutamos la función decorada.

    def __call__(self):
        # Ejecutamos el código de función __call__ cuando llamamos a la función decorada

        print("Dentro de EjemploDeAnotacion.__call__")


@EjemploDeAnotacion
def foo():
    print("Dentro de foo")


print("Terminamos de decorar la función foo")
foo()  # Llamamos a la funcion
```

Imagen 11. El código del decorador *EjemploDeAnotacion*.

Al ejecutar el código descrito anteriormente obtenemos la salida siguiente:

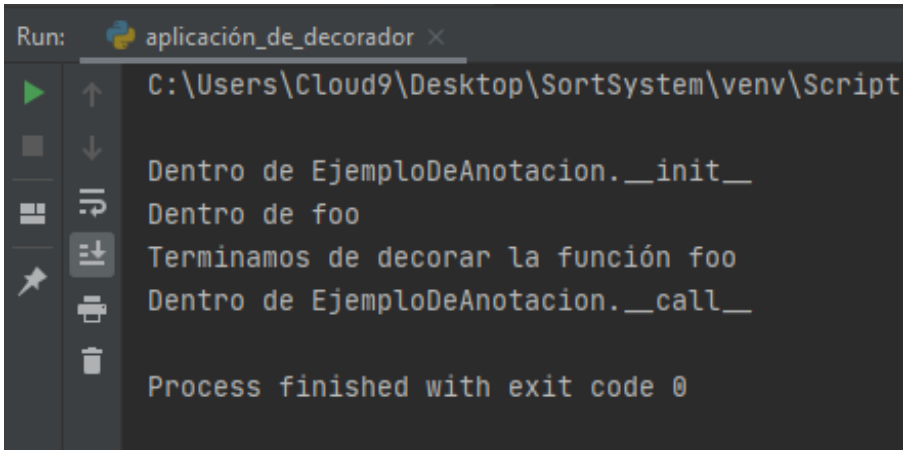


Imagen 12. Salida de la ejecución de función decorada.

Anotando el decorador sobre la función *foo*() esta función pasa al constructor *__init__*() como el argumento *decorated_function* que tiene el objeto de esta función (Imagen 13).

```
def __init__(self, decorated_function):
    print("Dentro de EjemploDeAnotacion.__init__")
    print("decorated_function =", decorated_function)

    # Output: Dentro de EjemploDeAnotacion.__init__
    #         decorated_function = <function foo at 0x02FF5388>
```

Imagen 13. Recibimos el objeto <function foo> en constructor del decorador.

Es posible ejecutar el argumento *decorated_function,* que tiene el objeto de función ejecutable, y, en este caso, pasamos a ejecutar el código de la función decorada. Además, podemos capturar la ejecución de esta función en una variable, si esta función retorna valores.

Cuando llamamos a *foo*() después de decorarlo, obtenemos completamente otro comportamiento de esta función, porque en lugar de ejecutar el código original ejecutamos el código del método *__call__*. Esto ocurre porque la decoración sustituye a la función original con su propio resultado. Entonces, llamando a la función *foo,* desde fuera sustituimos su código con el código del método *__call__,* que pertenece al decorador *EjemploDeAnotacion.*

Los decoradores posibilitan la implementación del código antes y después (al estilo de lo que sucedía con *before()* y *after()* en AspectJ) de la ejecución de la función decorada (Imagen 14).

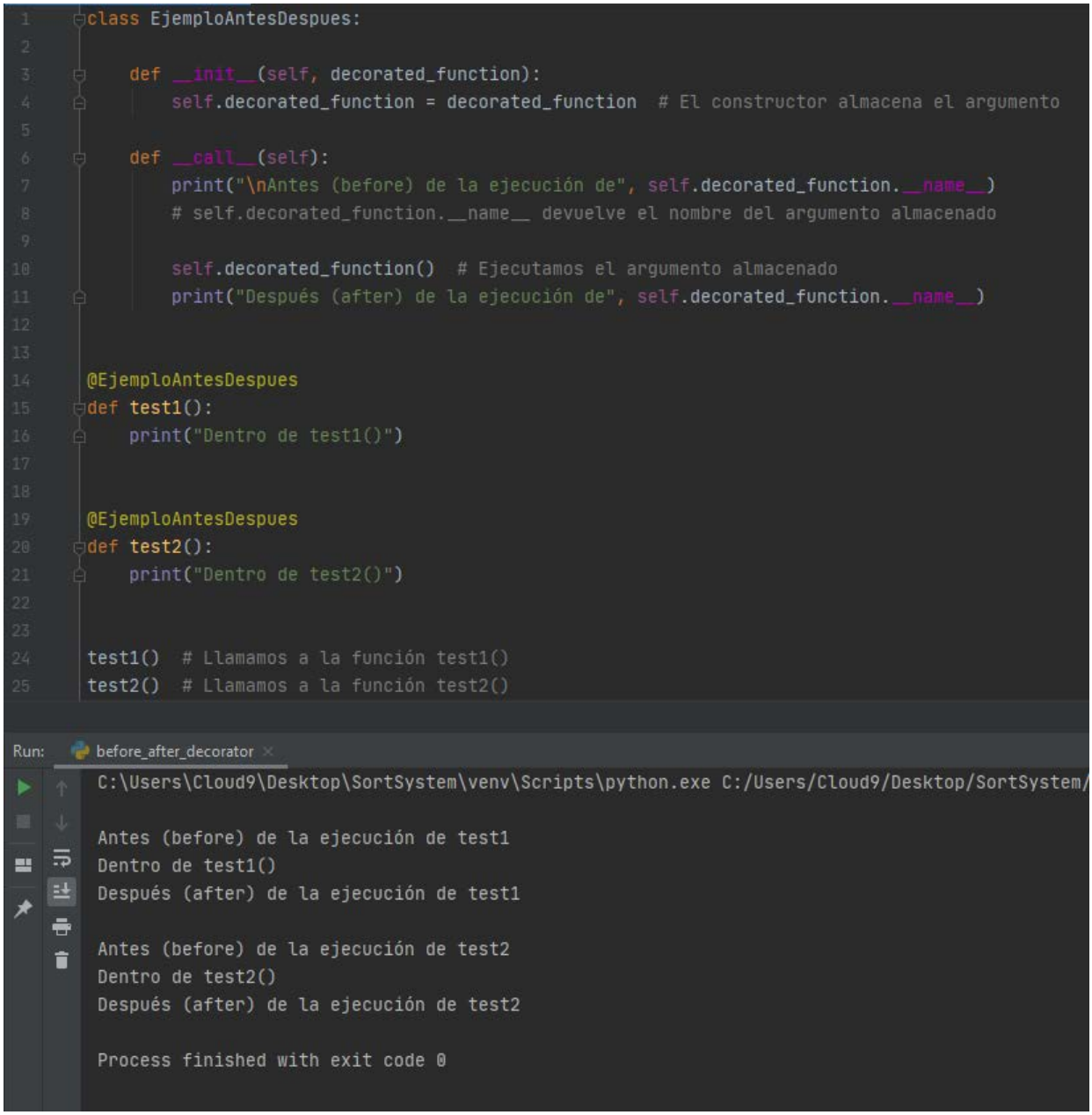


Imagen 14. Decorador definido como una clase de Python.

La Imagen 14 también nos muestra que al instanciar las funciones para decorarlas, estas se almacenan en la variable propia de la clase que definimos dentro del constructor. Esta variable es *self.decorated_function,* que contiene el objeto de la función decorada. Al llamar estas funciones se ejecuta el código del método *__call__*(), en el cual imprimimos que estamos “antes de la ejecución” de la función capturada. Más tarde, ejecutamos el objeto almacenado en el constructor y luego lo escribimos en la consola que estamos “después de la ejecución”. El atributo *__name__* de la función devuelve el

nombre de esta función, que se encuentra almacenada en la variable propia de la clase *self.decorated_function*.

Una vez introducido el concepto de decorador-clase en Python, es decir, un decorador definido como un clase más del programa, pasamos a ver cómo podemos definir un decorador-función, es decir una función Python para decorar otras funciones o métodos del programa. Los decoradores-funciones tienen una estructura parecida a los decoradores-clase. Al igual que los de decoradores-clase, los decoradores-función usan una función interior *wrapper()* que envuelve la función original (Imagen 15). El comportamiento de este tipo de decoradores es parecido a la ejecución del método *__call__()* de decoradores-clase en el sentido de que el decorador no se activa al instanciar la función en el mismo módulo del proyecto.

Una vez la función *wrapper()* está definida dentro del decorador, esta se devuelve desde la función de decoración definida, *decorator_function().* De este modo el mecanismo del decorador puede usar este objeto en lugar de la función decorada. Este proceso de devolver en una función otra función recibe en Python el nombre de closure o cierre. Este tipo de mecanismo es complejo y no vamos a describirlo en detalle porque excede el propósito de esta memoria. Simplemente señalaremos aquí que la función *wrapper()* se considera en este contexto un objeto *closure* y que el lector interesado podrá encontrar más detalles al respecto en [10]. Conceptualmente hablando, lo que ocurre es que la llamada a la función de decoración devuelve un objeto *closure*, el objeto asociado a la función *wrapper()*. Por tanto, la llamada a *test1()* activa la función de decoración *decorator_function()* que devuelve la función *wrapper()* que es la que finalmente se activa. Además, el argumento *decor_func* de la función de decoración hace referencia al método o función decorada, con lo que puede utilizarse dentro de la función *wrapper()* si se desea activar dichas función/método decorado. Como ya se ha mencionado, todos los detalles pueden consultarse en la Imagen 15.

Hay que tener en cuenta el tipo de datos que retorna el programador desde los decoradores, porque la función original puede tener un tipo de datos diferente de los datos que intenta devolver el programador desde su decorador. Por tanto, esto puede llevar a errores en la ejecución de aquellas funciones que trabajen con el valor devuelto.

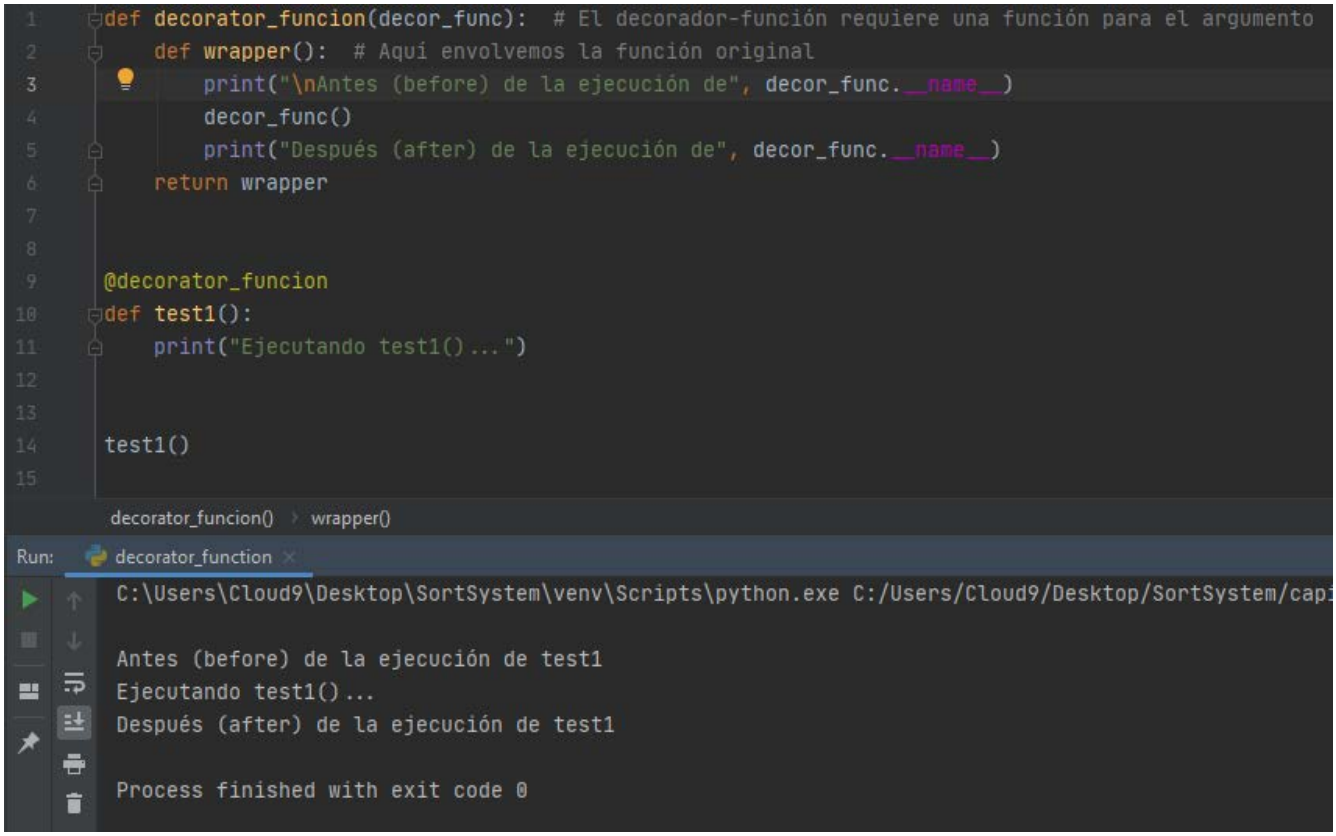


Imagen 15. Decorador definido como una función de Python.

Si la función original contiene argumentos, también podemos pasarlos al decorador. Para ver cómo hacerlos, empleamos el decorador-función que se define en la Imagen 16. Como vemos, los argumentos pueden ser de dos tipos:

- ***args***

  Los **args* es la sintaxis especial que se usa para pasar un número indefinido de argumentos a una función. Los argumentos que pasan a la función por **args* son la lista de valores que sean non-keyworded (no **kwargs). Por defecto esta sintaxis suele emplear la palabra *args.* El uso del asterisco informa a la función que, por este argumento, pueden pasar varios valores. Dentro de los decoradores la variable *args* también se usa con un asterisco.

- ****kwargs***

  Los ***kwargs* también tienen una sintaxis especial. En este caso, se usan para pasar la lista de valores que son *keyworded*. Esto significa que cada valor que pasamos

debe tener una llave o clave. En otras palabras, la lista es un diccionario clave-valor que mapea cada clave con un valor. El asterisco doble informa a la función que van a pasar los datos tipo clave-valor. El uso de esta sintaxis dentro de la función es igual que el uso de los **args*.

```
def decorator_funcion(decor_func):  # El decorador-función requiere una función para el argumento
    def wrapper(*args, **kwargs):  # Aquí envolvemos la función original y pasamos sus argumentos al decorador
        print()
        print("Args son:", args)
        print("Kwargs son:", kwargs)
        decor_func(*args, **kwargs)
    return wrapper


@decorator_funcion
def test1(string, **kwargs):
    print(string, "Mundo!")

    for clave, valor in kwargs.items():  # clave = Hello, valor = World!
        print(clave, valor)


test1("Hola", Hello="World!")
```

```
Run:  decorator_function
C:\Users\Cloud9\Desktop\SortSystem\venv\Scripts\python.exe C:/Users/Cloud9/Desktop/SortSystem/capitulo_3/decora

Args son: ('Hola',)
Kwargs son: {'Hello': 'World!'}
Hola Mundo!
Hello World!

Process finished with exit code 0
```

Imagen 16. Obtenemos los argumentos de la función *test1*().

Con independencia de si se implementan como funciones o como clases, los decoradores se pueden clasificar también en base a si tienen o no argumentos. Los ejemplos de decoradores que hemos estudiado hasta ahora pertenecen al tipo de decorador sin argumentos.

Pasar los argumentos al decorador produce cambios en su comportamiento. Esto puede resultar útil cuando el programador no quiere sobrecargar los argumentos de la función original pasando los datos necesarios únicamente para el decorador (véase el capítulo 4, donde se describe la implementación del sistema con decoradores que requieren los argumentos). Para pasar los argumentos al decorador-clase hace falta describirlos en el

constructor __*init*__() (imagen 17). Si tenemos los argumentos para decorador-clase, el método __*call*__() se llama una vez como la parte del proceso de decoración. En este caso, la función para decorar no se transfiere al constructor, sino al __*call*__() a modo de argumento único.

```
class DecoratorClaseConArgs:
    def __init__(self, arg1, arg2, arg3):  # La función para decorar no se pasa aquí, aquí pasan los argumentos
        print("Dentro de DecoratorClaseConArgs.__init__()")
        self.arg1 = arg1
        self.arg2 = arg2
        self.arg3 = arg3

    def __call__(self, funcion):  # La función para decorar pasa aquí
        print("Dentro de DecoratorClaseConArgs.__call__()")

        def wrap(*args):  # Envolvemos la función original
            print("Dentro de wrap()")
            print("Argumentos de decorador:", self.arg1, self.arg2, self.arg3)
            funcion(*args)
            print("Después de funcion(*args)")
        return wrap


@DecoratorClaseConArgs("Aquí", "pasamos", "otros datos")
def test(arg1, arg2, arg3):
    print("Argumentos de test:", arg1, arg2, arg3)


print("Después de decorar")

print("\nLlamamos a función varias veces\n")
test("Hola", "Mundo!", "¿Qué tal?")
print("Despues de primera llamada\n")
test("Hello", "World!", "What's up?")
print("Despues de segunda llamada\n")
```

Imagen 17. Implementación del decorador-clase con los argumentos.

La ejecución del archivo instancia, en primer lugar, las funciones y clases almacenadas en este archivo, y así se empieza el proceso de decoración que llama al constructor e inmediatamente invoca a __*call*__(), que puede tomar un único argumento (el objeto de la función) y debe devolver el objeto closure de la función decorada que reemplaza a la original.

Este proceso se observa en la salida de la ejecución (imagen 18), donde las primeras palabras imprimidas en la consola son “Dentro de DecoratorClaseConArgs.__init__()”, que indican que el constructor está instanciado. “Dentro de DecoratorClaseConArgs.__call__()” instancia la función __*call*__(), pero no llega a la función interior *wrap*() y “Después de

decorar", que está después de la instancia de la función decorada y el decorador. Es decir, después del proceso de decoración.

En el ejemplo se puede observar que pasamos los argumentos "Aquí", "pasamos" y "otros datos" directamente al decorador, decorando la función *test()*. A continuación, estos argumentos llegan al constructor, almacenándolas en las variables de la clase *self.arg1*, *self.arg2* y *self.arg3*. El objeto *__call__*() ya no se puede utilizar como una llamada a la función decorada. Para capturar la llamada usamos la función interior *wrap*(), para que envuelva la función original, y así sacar los argumentos de esta. Dentro de la *wrap*() se usan los argumentos previamente almacenados y se ejecuta la función original con los argumentos de esta función.

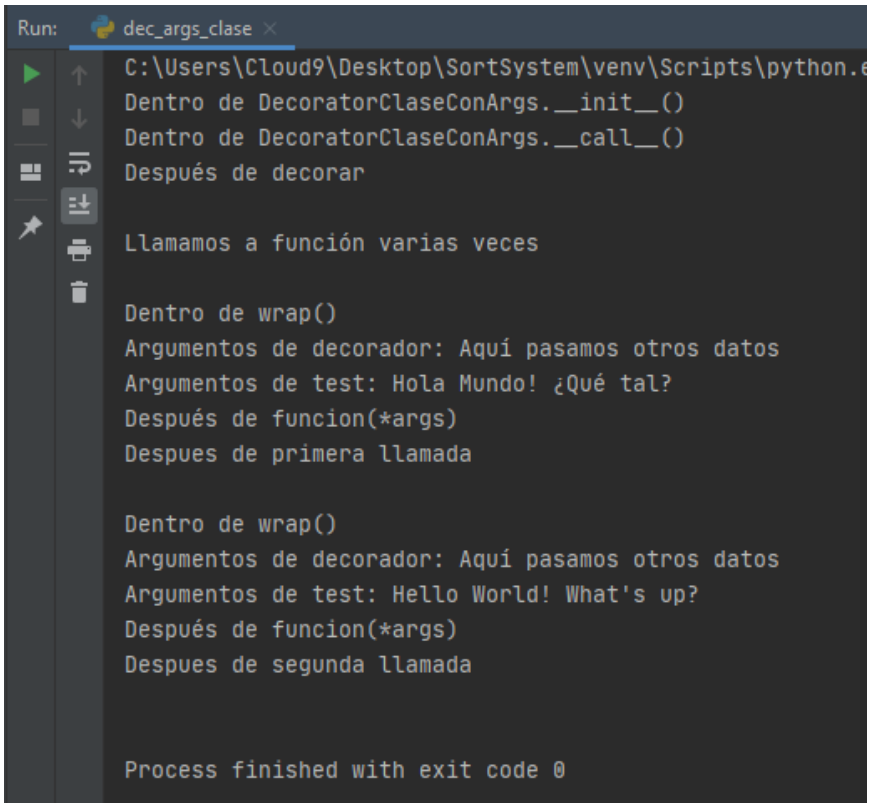
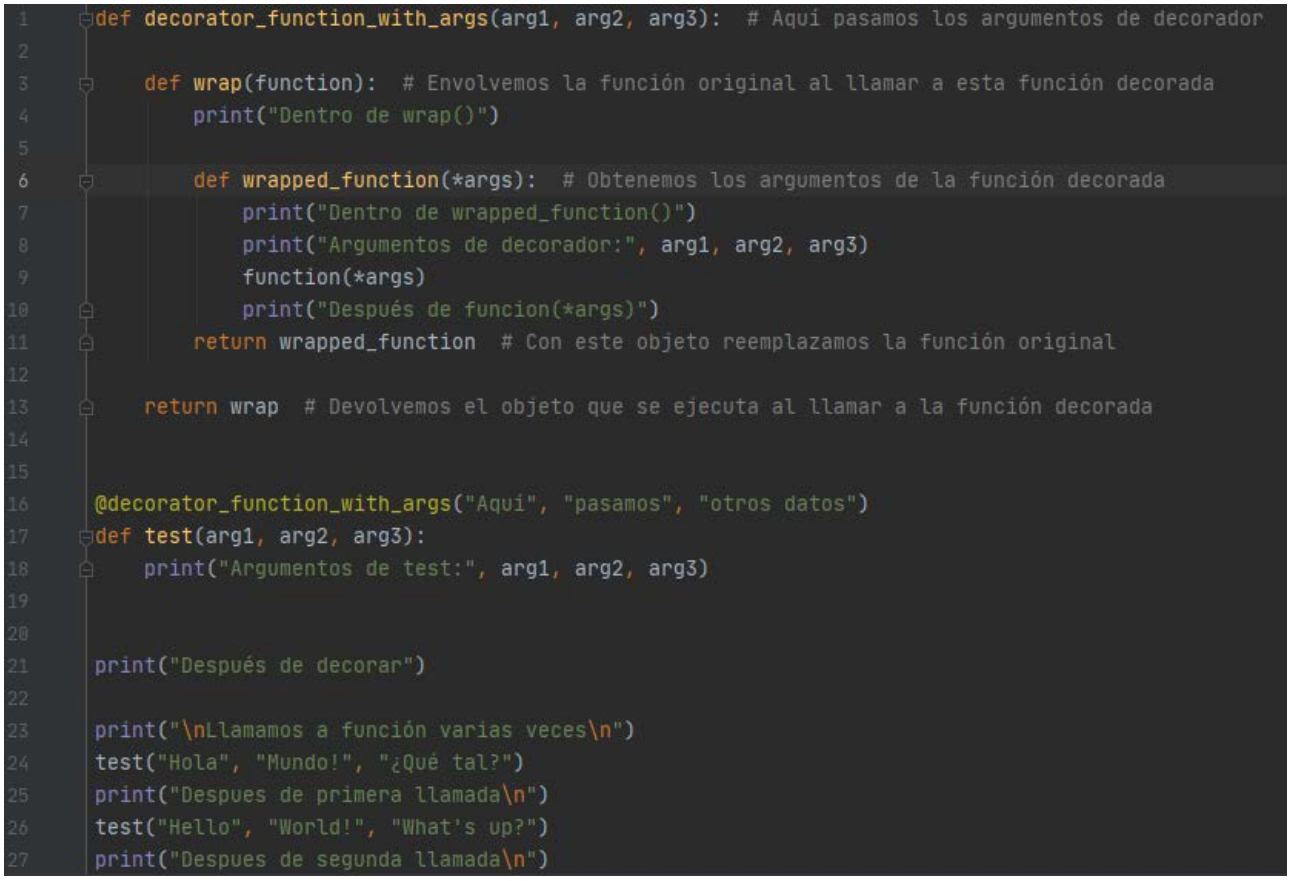


Imagen 18. Salida del ejemplo de decorador-clase con argumentos.

En referencia a los decoradores-funciones, estos decoradores se desarrollan de forma similar. Para pasar los argumentos a un decorador-función tenemos que definir una capa de métodos más (imagen 19) que dirigirá los argumentos del decorador.



```python
def decorator_function_with_args(arg1, arg2, arg3):  # Aquí pasamos los argumentos de decorador

    def wrap(function):  # Envolvemos la función original al llamar a esta función decorada
        print("Dentro de wrap()")

        def wrapped_function(*args):  # Obtenemos los argumentos de la función decorada
            print("Dentro de wrapped_function()")
            print("Argumentos de decorador:", arg1, arg2, arg3)
            function(*args)
            print("Después de funcion(*args)")
        return wrapped_function  # Con este objeto reemplazamos la función original

    return wrap  # Devolvemos el objeto que se ejecuta al llamar a la función decorada


@decorator_function_with_args("Aquí", "pasamos", "otros datos")
def test(arg1, arg2, arg3):
    print("Argumentos de test:", arg1, arg2, arg3)


print("Después de decorar")

print("\nLlamamos a función varias veces\n")
test("Hola", "Mundo!", "¿Qué tal?")
print("Despues de primera llamada\n")
test("Hello", "World!", "What's up?")
print("Despues de segunda llamada\n")
```

Imagen 19. Implementación del decorador-función con los argumentos.

Como se ve en la salida de la ejecución (imagen 20), obtenemos un resultado similar a los resultados de la ejecución del decorador-clase. Aquí el valor devuelto del decorador *decorated_function_with_args*() debe ser la función *wrap*(), que envuelve la función para decorar. Es decir, Python tomará la función wrap() y la llamará a la hora de la decoración, pasando a la función para decorar. Entonces, decimos que hay tres capas de funciones, y el capo interno *wrapped_function*() es la función de reemplazo real. Además, debido a los closures, *wrapped_function*() tiene acceso a los argumentos de decoración arg1, arg2 y arg3, sin tener que almacenarlos explícitamente como en la versión del decorador-clase.

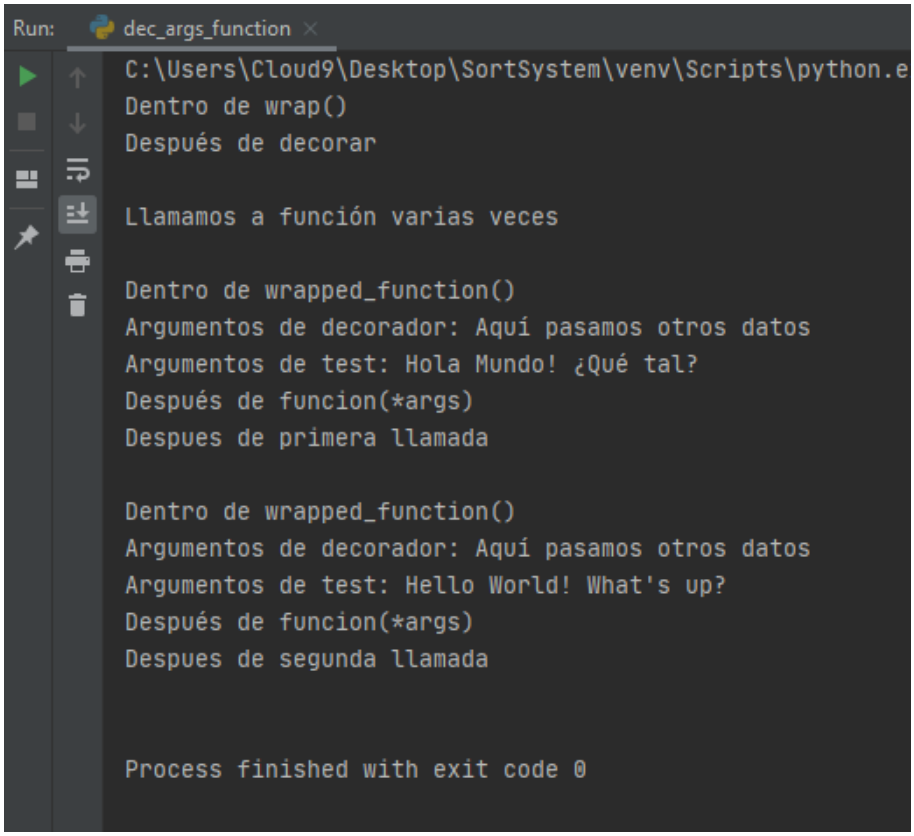


Imagen 20. Salida del ejemplo de decorador-función con argumentos.

3.3. Separación de Mecanismos a través de Decoradores

Como ya hemos comentado en varias ocasiones, el problema del entrelazado de código puede provocar que los proyectos que implementen muchos mecanismos diferentes puedan llegar a ser no mantenibles. Esto trae como resultado que los mecanismos, cuyas instrucciones estén por todo el código, en algún momento del desarrollo lleguen a ser muy complejos para ser seguidos y depurados. Aunque ello no fuera un inconveniente para proyectos de 3 archivos, si el programador se enfrentara a un sistema de más de 50 archivos diferentes, resultaría complicado encontrar allí el código entrelazado para cambiarlo. En este caso, consumiría mucho tiempo cada vez que se necesitara cambiar el código del mecanismo.

Por ejemplo, la librería “org.apache.tomcat” para Java tiene implementado el mecanismo *logging* cuyo código está en muchos módulos distintos del sistema (Imagen 21) [27].

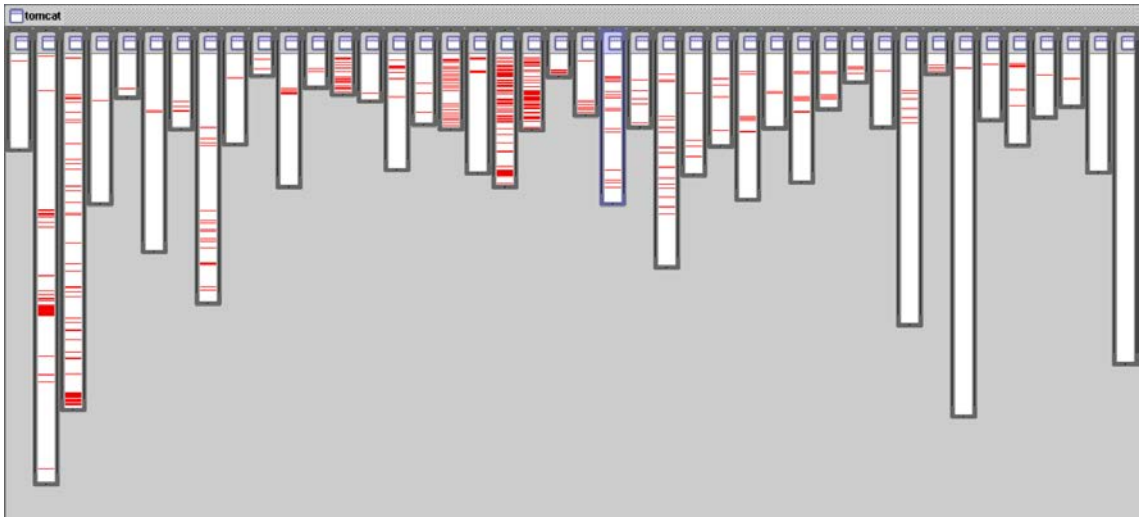


Imagen 21. Mecanismo *logging* en el código de la librería “org.apache.tomcat”.

La solución es bastante trivial: separar los mecanismos diferentes encapsulándolos fuera del código principal. Separando las tecnologías el programador se asegura de que no necesita volver a buscar por todo el código la instrucción para modificarla, sino buscarla en un archivo separado de todo el sistema. De este modo, se aumenta la mantenibilidad de todo el proyecto y, además, el programador obtiene un módulo reutilizable. Decimos que este módulo es reutilizable porque, al representar un módulo completo que tiene el mecanismo ya desarrollado, puede ser implementado para otra aplicación que usara este tipo de mecanismos de forma similar.

En Python, los decoradores, como las funciones y clases básicas, también pueden aplicarse en otra parte de proyecto, incluso en otro sistema. Esto permite escribir el mecanismo deseado como, por ejemplo el *logging*, dentro de un decorador inyectando este mecanismo en el código principal del proyecto y anotando las funciones cuyo comportamiento se necesite cambiar. El decorador, por su parte, se queda como un módulo reutilizable que se puede usar para el otro sistema que necesita implementar el mismo mecanismo.

Veamos a continuación cómo podemos servirnos de los decoradores para desarrollar mecanismos reutilizables de tolerancia a fallos y seguridad en soluciones Python.

CAPÍTULO 4: DESARROLLO DE DECORADORES PARA LA TOLERANCIA A FALLOS Y CIFRADO

Plantear el desarrollo de mecanismos concretos de tolerancia a fallos y cifrado con Python o con cualquier otro lenguaje sin definir previamente las hipótesis de fallo que consideramos no tiene sentido. En nuestro caso vamos a considerar una solución cliente-servidor en la que la caída del servidor se asimile a la caída del proceso que lo ejecuta, con lo que "levantar" el servidor consiste en volver a lanzarlo a ejecución, restaurando convenientemente su estado. En caso de fallo del servidor, los clientes aplicarán redundancia temporal, es decir, varios reintentos de invocación, antes de dar por "caído" al servidor. Además, el servidor tolerará los potenciales errores que pudiera tener su programación aplicando diversificación funcional en el servicio que ofrece. Finalmente, las comunicaciones entre cliente y servidor se deberán cifrar con el objetivo de mantener la confidencialidad e integridad de los mensajes intercambiados.

Una vez establecidas las hipótesis de fallo con las que vamos a trabajar, nos centramos en el ejemplo concreto con el que ilustraremos nuestro estudio. El servicio con el que trabajaremos permitirá a las aplicaciones ordenar una secuencia de ordenación de números de menor a mayor. Estos números podrán generarse aleatoriamente para evitar tener que introducirlos de manera manual El lenguaje de desarrollo del ejemplo será Python.

En los apartados siguientes describiremos detalladamente los siguientes elementos del sistema. Empezaremos por introducir nuestro caso de ejemplo. A continuación, y con el objetivo de cumplir con las hipótesis de fallo anteriormente descritas, desarrollaremos los siguientes mecanismos de cifrado y tolerancia a fallos:

- Mecanismo para la gestión de invocaciones remotas para los clientes (patrón de diseño Proxy). Por defecto, los clientes realizarán la ordenación de los valores con los que trabajen localmente. Se desarrollará un decorador que implementará el patrón de diseño Proxy, que permitirá que las peticiones de decoración locales se transformen en peticiones remotas remitidas hacia un servidor.

- Mecanismo para tolerar la caída del servidor. En caso de que la máquina sobre la que se ejecuta el servidor caiga, el proceso de ejecución del servidor desaparezca o el mismo servidor deje de responder, el cliente realizará varios reintentos de reconexión espaciados en el tiempo (redundancia temporal) con el objetivo de tolerar el problema, algo que sucederá si entre dos reintentos, el servidor es relanzado a ejecución. Del lado del servidor, este relanzamiento a ejecución requiere de poder salvar el estado del servidor cada vez que este se vea modificado y restaurarlo cuando el servidor sea relanzado.

- Mecanismos de tolerancia a fallos SW en el algoritmo de ordenación de valores. Como hemos mencionado el servicio que se ofrecerá consistirá en ordenar de menor a mayor un conjunto de valores. Se aplicará diversificación funcional para tolerar cualquier error de diseño e implementación del algoritmo de ordenación. Esto significa que se utilizarán distintos algoritmos y luego se votará la ordenación "más popular" de entre las resultantes, siendo esta la que se retornará al solicitante del servicio. .

- Mecanismos de cifrado simétrico para las comunicaciones cliente-servidor. Este mecanismo permitirá verificar la integridad de los mensajes que intercambien cliente y servidor y dificultará la interpretación de dicha información, que estará cifrada, si esta es capturada en tránsito.

Estos mecanismos de confiabilidad se implementarán utilizando decoradores como medio para separarlos del código funcional del cliente y del servidor.

La mayoría de los decoradores implementados en este trabajo son de tipo clase, porque este tiene dos métodos auxiliares __*init*__() y __*call*__(), que ayudan a manejar las

funciones y los datos de manera más fácil que a través del tipo función, que carece de la etapa de inicialización del decorador. Si dicho mecanismo llevara argumentos, el constructor __*init*__() ayudará a separarlos de la lógica de decorador que no es propia de los decoradores-funciones.

Además, los decoradores desarrollados en este trabajo serán interceptores que inyectarán el comportamiento necesario sólo si el booleano que controlará su comportamiento está puesto a *True.* Dicho de otro modo, todos los mecanismos podrán potencialmente desplegarse y será mediante un *flag*, que actuará activador, que podremos determinar si se aplican o no a la solución en ejecución. Además, al ser Python un lenguaje interpretado, podremos cambiar el valor de estos *flags* dinámicamente, con lo que el sistema adaptará el uso de estos mecanismos a los requerimientos de confiabilidad que vayamos definiendo. La lista de los *flags* de control que acabamos de mencionar se proporciona en la imagen 22. Esta lista está en el módulo "decorator_configuration.py" de la carpeta "decorators".

```
class DecoratorConfiguration:
    proxy_on = False
    profiling_on = False
    logging_on = False
    cyphering_on = False  # Server-client side
    FTRetry_on = False  # Client side
    FTBackup_on = False  # Server side
    FTDiversification_on = False  # Server side
```

Imagen 22. Controlador de decoradores-interceptores.

Algunos decoradores se pueden aplicar sólo en el lado del servidor o del cliente y otros se aplican sobre ambos. El decorador de cifrado se usa sólo para los dos, es decir, no va a funcionar al activarlo únicamente para servidor o para cliente.

4.1. Caso de Ejemplo

Todo el código ha sido desarrollado usando el lenguaje de programación Python, versión 3.8.2, creado por *The Python Software Foundation*. Se puede obtener dicha versión en https://www.python.org/downloads/. Para escribir el código, se ha optado por la IDE PyCharm de la versión 2020.1.2 (*Community Edition*) de la empresa JetBrains. Esta IDE ha sido elegida por la flexibilidad que ofrece su editor de código, por su alto rendimiento, distribución gratuita y por la disponibilidad de diferentes *plugins* útiles. La PyCharm puede ser recuperada en https://www.jetbrains.com/pycharm/.

Además de las librerías propias de Python, para el proyecto también se ha utilizado la librería pyfiglet. Esta librería nos permite escribir el texto usando la fuente "*bulbhead*". Esta herramienta resulta útil para crear los índices dentro de la *Command Line*. Es posible recuperar la pyfiglet desde https://pypi.org/project/pyfiglet/0.7/.

Para el decorador de cifrado se ha usado la librería externa "cryptography.fernet", por su codificación con el mecanismo propio de la librería *Fernet Token* (véase el párrafo 4.6.).

El caso de ejemplo es el sistema SortSystem de ordenación de un *array* (lista en Python) de números generados aleatoriamente (imagen 23). La aplicación dispone de un cliente y de un servidor, módulo del parseo de opciones de programa definidas por la *Command Line*, directorio del cifrado de mensajes, directorio para los decoradores, y directorio que contiene módulos auxiliares.

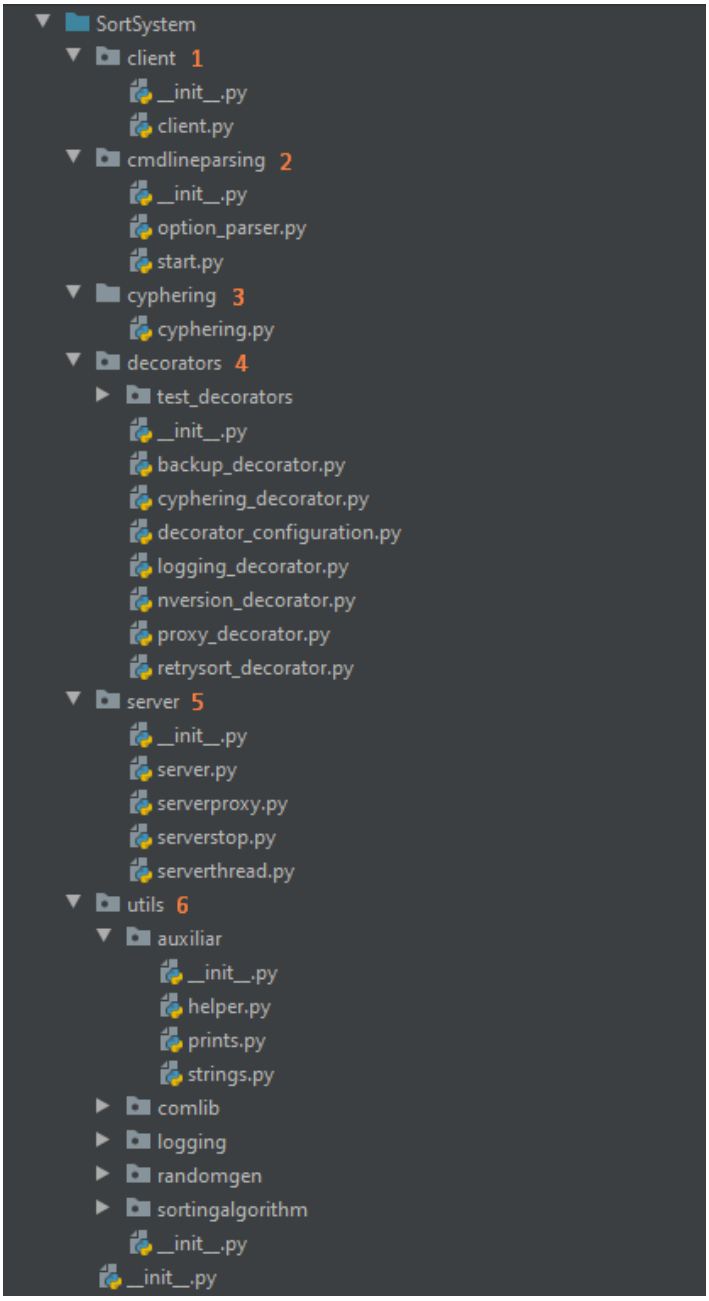


Imagen 23. El sistema de ordenación y su estructura. Dispone de directorio cliente (1), directorio para parsear opciones a *Command Line* (2), directorio para cifrar los mensajes (3), directorio de decoradores (4), directorio de servidor (5) y directorio de los módulos auxiliares (6).

Los módulos auxiliares son los siguientes:

- Clase *Helper* (en la carpeta “auxiliar”), contiene los booleanos necesarios para parar el bucle de servidor o para el *logging*.

- Implementación de la función propia de Python print (la cual es necesaria para activar el mecanismo de *logging*), en el directorio “auxiliar”. En los módulos donde se aplica esta función, *print* está reemplazado por *p.prints*.

- Contenedor de *strings* usados para pasar los mensajes por *sockets,* en “auxiliar”.

- Mecanismo de comunicación, ubicado en la carpeta “comlib”.

- Clase *MyLogger,* donde creamos programáticamente los archivos para *logs* y formateamos el texto para pasarlo a estos archivos. *MyLogger* se localiza en el directorio “logging”.

- Generador de números, o un *array* de números aleatorios en el directorio “randomgen” .

- Algoritmos para el ordenación que se sitúan en el directorio “sortingalgorithm”. Este directorio dispone de 7 algoritmos diferentes que son, *bubble sort, heap sort, insertion sort, merge sort, quick sort, selection sort* y el algoritmo propio de Python. También en esta carpeta se encuentra una clase base *SortingAlgorithm,* para las clases de algoritmos. Esta clase tiene los tipos de todos los algoritmos implementados y el método abstracto *sort*(), que se reescribe en cada uno de estos algoritmos. En su interior se encuentra también una *factory* que devuelve los objetos de los algoritmos.

En el archivo “decorator_configuration.py” están los *switchers* para activar los decoradores desde el archivo “option_parser.py”.

Se puede observar que en cada carpeta del proyecto está el archivo “__init__.py”. Este archivo se usa para informar a Python que el módulo donde se encuentra este archivo pertenece al proyecto, así como su ubicación en la estructura misma del proyecto. Sin “__init__.py” no se podría realizar los *imports* de métodos o clases, porque Python no define este módulo como parte del proyecto. El archivo está vacío por dentro, porque se ha empleado a modo de índice para el lenguaje y no requiere acciones por parte de un programador.

En el directorio de parseo de opciones está el archivo “start.py”, que arranca el sistema abriendo el archivo “option_parser” en command line (imagen 24).

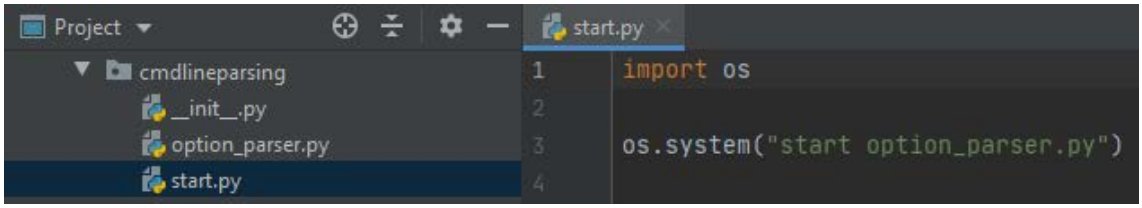


Imagen 24. Archivo “start.py” que abre el “option_parser.py” en consola.

El archivo “option_parser.py” contiene el código para iniciar los diferentes módulos imprimiendo el texto en la *Command Line*. El resultado de la ejecución de “option_parser.py” se muestra en la imagen 25. Desde aquí podemos pasar a iniciar el servidor y el cliente, eligiendo sus opciones (los decoradores a aplicar). También se puede parar la ejecución del servidor o salir de programa.

Imagen 25. *Command Line* donde se puede elegir el componente para iniciar. La consola dispone de inicialización del servidor (1), inicialización del cliente (2), opción para parar la ejecución del servidor (3) y opción para salir de la consola (4).

1. **Servidor.**

Al elegir la opción del servidor, se pasa a otra pantalla donde se puede elegir los decoradores que se desea aplicar. El comportamiento de los decoradores se describe detalladamente en los párrafos siguientes. Después de elegir las opciones necesarias, el parseo de opciones instancia el servidor (que está en el archivo "server.py" del directorio *server*) y ejecuta su método *server_start*().

Dentro del archivo de servidor se dispone de la clase *Server* que tiene varios métodos, mostrados en la imagen 26 y la imagen 27.

```
10  class Server:
11      def __init__(self):
12          self.port = 13534  # Número de puerto que usamos
13          self.client_number = 1  # El número del cliente que hace petición al servidor
14          self.host = '127.0.0.1'  # El nombre de host (127.0.0.1 por defecto)
15          self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # Instanciamos el socket
16          self.server.bind((self.host, self.port))  # Asignamos al socket el host y el puerto
17          self.server.listen(10)  # Escuchamos a clientes
18
19      def set_client_number(self, client_number):  # El setter para el número del cliente
20          @BackupClassDecorator("set", client_number)
21          def init_set_client_number(client_n=client_number):
22              self.client_number = client_n
23
24      def get_client_number(self):  # El getter para el número del cliente
25          @BackupClassDecorator("get")
26          def init_get_client_number():
27              return self.client_number
28          return init_get_client_number
29
```

Imagen 26. Constructor de la clase *Server*, *setter* y *getter* para el número de cliente.

Al instanciar el servidor, el constructor *__init__*() define y almacena los valores para puerto, host y número del cliente. Además, instancia el *socket*, le asigna el host, el puerto y se pone a escuchar a los usuarios.

Los *setter* y *getter* son necesarios para implementar el decorador que contiene el mecanismo de *backup* (véase el párrafo 4.4.2.). Los dos resultan triviales a la hora de realizar su tarea común: plantear y obtener el valor de la variable *client_number*.

```
29
30      def server_start(self):
31          client_number = self.get_client_number()
32          executor = ThreadPoolExecutor()
33          p.prints("> [Server.start] Hi!!! Starting the sorting server at port: {}".format(self.port))
34          while Helper.stop is not True:
35
36              if Helper.stop is True:
37                  break
38
39              p.prints("> [Server.start] Waiting for a new client ...")
40              client, addr = self.server.accept()
41              p.prints("> [Server.start] Processing request for client #{}...".format(client_number))
42              self.set_client_number(client_number + 1)
43
44              sst = ServerThread(client, SortingAlgorithmFactory.new_sa(alg_type.BUBBLESORT))
45              p.prints(">> Serving the client in a thread ...")
46              future = executor.submit(sst.run, client)
47              future.result()
48
49          p.prints("> [Server.start] Server is out of the do while loop ... bye!!!")
50          executor.shutdown()
51          p.prints("> [Server.start] All threads in the pool terminated ... bye!!!")
52          self.stop_server()
53
54      def stop_server(self):
55          p.prints("> [Server.stopServer] Stopping the server ...")
56          if self.server is not None:
57              p.prints("> [Server.stopServer] Closing the socket ...")
58              self.server.close()
59          p.prints("> [Server.stopServer] Setting stop variable to true ...")
60
61          print("Presione enter para continuar . . .")
62
63          exit_button = input()
64          if exit_button is not None:
65              exit(0)
```

Imagen 27. Métodos *server_start*() y *stop_server*() de la clase *Server.*

Al ejecutar el método *server_start*() se usa el *getter* para obtener el número del cliente. Luego, se instancia el ejecutor *ThreadPoolExecutor,* que es una clase que usa un conjunto de hilos para ejecutar llamadas asincrónicas. Después, va un bucle *while* que se para cuando hay el orden *stop* (desde el “serverstop.py”). Para verificar si hay una

orden de *stop* dentro del bucle, se usa la construcción *if* que comprueba si la variable auxiliar *stop* de la clase *Helper* está marcada como True. Si la variable *stop* es False, el servidor entonces ignora la construcción y se dispone a esperar las conexiones de usuarios usando la función *socket.accept()* (en caso de que el sistema fuera *self.server.accept*()). Cuando un usuario se conecta al servidor, *accept*() devuelve un par "(*conn, addr*)" en el que *conn* (*connection*) es un nuevo objeto socket utilizable para enviar y recibir datos sobre la conexión, y *addr* (*address*) es la dirección ligada al socket en el otro extremo de la conexión.

Posteriormente a la conexión del cliente, usamos el *setter* para preparar el número para la conexión de otro usuario al sistema, e instanciamos la clase *ServerThread* pasando al constructor de la clase *conn* y el algoritmo *bubble* para ordenar los números, usando la *factory* de algoritmos. Al almacenar los argumentos instanciando la clase, el objeto closure de la función *run* de *ServerThread* y el argumento *conn* de *accept*() se ponen en la función *submit*(), que programa la función para ser ejecutado como *function(*args **kwargs)* y devuelve un objeto *Future* que representa la ejecución de la función.

Después, se aplica la función *result*() sobre el objeto *Future*, que ejecuta el hilo creado. La función *result*() espera la terminación de ejecución del hilo. Si la función no esperara hasta que el código del hilo se termine, o si quitara la función *result*(), entonces el servidor volvería a esperar a las conexiones externas, provocando, tras la escucha del socket del servidor, un comportamiento inesperado por bloquear. Inmediatamente después de terminar el trabajo en hilo, se repetirá el bucle para otro cliente.

Si la variable auxiliar de *Helper stop* está marcada como True, el código se detiene saliendo del bucle, cerrando el ejecutor de hilos y ejecutando el método *stop_server*(). La función *stop_server()* comprueba que el socket existe y si es así lo cierra. En caso contrario, entonces seguirá ejecutando el código y seguidamente detendrá la ejecución terminando el proceso.

Al almacenar los valores en la *ServerThread* y al lanzar su función en hilo, el código empieza a ejecutar el método *run*() de esta clase (imagen 28). La función *run*() empieza instanciando la clase de comunicación *SocketCommunication* a la cual se transfiere el objeto de socket del servidor. Envolver el código en un bloque *try-except* ayuda a cerrar

la conexión antes de introducir el error que termina la ejecución, porque puede ocurrir que la conexión no se cierre automáticamente tras terminar el proceso por error, causando errores de sistema (habría que reiniciar la maquina o cambiar el puerto de socket) al volver a arrancar el SortSystem. Dentro de *try,* recibimos el mensaje, que es el array de números aleatorios del cliente, y comprobamos si se trata de la orden *stop*. En caso de ser así, se cambia la variable auxiliar de *stop* a True (que detiene el bucle del servidor) y se cierra la conexión. Si no fuera la orden de *stop*, entonces se comprueba que el mensaje no sea *None* (*null* en Python) y comprobamos que el algoritmo almacenado tampoco sea *None.* Si alguna de las comprobaciones fallara, se procederá a cerrar la conexión y se detendrá la ejecución de *ServerThread*.

```
class ServerThread:
    def __init__(self, client, algorithm):
        self.client = client
        self.algorithm = algorithm

    def run(self, conn):
        communication = SocketCommunication(conn)
        try:
            p.prints(">> Reading data to sort ...")
            message = communication.read_message_from_socket(Strings.from_client_to_server)
            if message == "stop":
                Helper.stop = True
                p.prints(">> Received stop order ... \n>> Stopping the server execution ...")
                conn.close()

            if message is not None:
                p.prints(">> This is the array to sort (server view) : {}".format(message))
                if self.algorithm is not None:

                    @NVersionDecorator
                    def get_data(data_to_sort=message):
                        ret_data = self.algorithm.sort(data_to_sort)
                        return ret_data

                    data = get_data(message)

                    p.prints(">> This is the sorted array (server view) : {}".format(data))
                    p.prints(">> Now writing the sorted array to the socket. \n")
                    communication.write_message_to_socket(Strings.from_server_to_client, data)
                else:
                    p.prints(">> Unable to sort because there is a problem with the algorithm to use")
            else:
                p.prints(">> Problems when reading message from socket ...")
        except socket.error:
            conn.close()
            raise socket.error("Socket error in ServerThread.run()")
        conn.close()
```

Imagen 28. Código de *ServerThread* y su función *run*().

Si el mensaje y algoritmo existen y no son *None*, se ordena el array en la función interior *get_data*() y lo envía por *SocketCommunication* al cliente.

Se puede comprobar que en el código, después de definir el método interno, se usa este método sin paréntesis. Esto es debido a que el comportamiento del decorador *NVersionDecorator* devuelve la lista de números en lugar del objeto de la función *get_data*() (véase el párrafo 4.5.). Ello implica que se emplee un método sin paréntesis, ya que no se trata una función, sino de una lista de números necesaria.

La *SocketCommunication* (imagen 29)*,* por su parte, es el módulo que proporciona la comunicación entre el servidor y clientes. Al instanciar esta clase, almacenamos la conexión de socket que luego usaremos para enviar y recibir los mensajes. Cuando se ejecuta la función *read_message_from_socket*() se asigna el mensaje llegado a la variable *data*. Luego, se comprueba el tipo de datos llegados en un bloque *try-except* y se devuelve el mensaje leído. En caso de que el método *loads*() no pudiera obtener el mensaje porque no es el *string* formateado JSON, la "json" mostrará el error JSONDecodeError, que se captura en *except* del bloque y, desde allí, se devuelven los datos "puros", es decir, los datos no pasados por la *loads*().

Cuando se ejecuta la función *write_message_to_socket*(), comprobamos el tipo de mensaje que queremos enviar. Si el mensaje fuera de tipo bytes, lo enviaremos usando la función estándar *send*(). Si no, primero lo convertiremos en el formato necesario usando el método *dumps*() de la librería "json" y luego lo mandaremos por la *send*().

Como se dijo anteriormente, por la *SocketCommunication* pueden pasar dos tipos de datos: bytes y lista. Los Bytes llegarán solo en el caso de estar activado el decorador *cyphering_decorator*. En cualquier otro caso, llegarán los datos de tipo *list.*

Luego, no es posible pasar una lista por el socket usando la función estándar de la librería "socket" *send*(), porque esta función requiere el tipo bytes. Para hacerlo, se ha optado por usar la librería "json", que ofrece los métodos de dichos *dumps*() y *loads*(). El *dumps*(), que se usa en *write_message_to_socket*(), serializa el objeto a un *string,* formateado a JSON usando una tabla de conversión especial. El método *loads(),* por su parte, decodifica el mensaje llegado. En el caso de que el *string* fuera de formato JSON, entonces este lo deserializa en el objeto inicial. De esta manera, usando la librería "json", se puede enviar un array de números que se convertirá en un string JSON y en un nuevo array de números al recibirlo.

El *string* formateado de JSON es muy parecido a un diccionario, porque su estructura tipo "{clave: valor}" es semejante a la estructura de los diccionarios. Como clave se ha empleado el mismo tag utilizado desde otras funciones. Por defecto, el tag puede ser el string auxiliar *from_client_to_server* si se pasan los datos de cliente a servidor, y el *from_server_to_client* si se hace lo opuesto.

```
class SocketCommunication:
    def __init__(self, s):
        self.socket = s

    @cyphering_decorator("read")
    def read_message_from_socket(self, tag):
        data = self.socket.recv(4096)
        try:
            data = json.loads(data)
            # json.loads: Deserialize s (a str, bytes or bytearray instance containing a JSON document) to a Python...
            # ...object using special conversion table.

            message = data.get(tag)
            return message
        except JSONDecodeError:
            return data

    @cyphering_decorator("write")
    def write_message_to_socket(self, tag, message):
        if isinstance(message, bytes):
            self.socket.send(message)
        else:
            data = json.dumps({"{}".format(tag): message})
            # json.dumps: Serialize obj to a JSON formatted str using special conversion table.

            self.socket.send(data.encode())
```

Imagen 29. Clase *SocketCommunication* y sus métodos *read_message_from_socket*() y *write_message_to_socket*()

2. Cliente.

Al elegir la opción de servidor, se pasa a otra pantalla donde se eligen los decoradores que se desea aplicar. Una vez elegidas las opciones necesarias, el parseo de opciones instancia el cliente (que está en el archivo "client.py" del directorio *client*) y ejecuta su método *main*().

El usuario dispone de la clase *Client* y dos métodos *sort*() y *main*(), que se muestran en la imagen 30. Al entrar en el método *main*() se genera un array de 10 unidades de

números aleatorios que no son más que 20. Al generar este array le asignamos la función *sort*(), que está decorada con el decorador *ProxyDecorator* (véase el párrafo 4.3.). Dentro de *sort*() se devuelve el array ordenado por el algoritmo *bubble* (el algoritmo por defecto) creado en la *factory*. Después de ordenar la lista de números, se comprueba si no es *None* y si no, se imprime array ordenado en la pantalla del cliente.

```
class Client:
    @ProxyDecorator
    def sort(self, args):
        return SortingAlgorithmFactory.new_sa(alg_type.BUBBLESORT).sort(args)

    def main(self):
        data = randomizer.random_data_array(10, 20)
        p.prints("> [Client.main] This is the array to sort (client view): {}".format(data))
        sort_data = self.sort(data)

        if sort_data is None:
            p.prints("> [Client.main] The sorted array returned None. Check whether the server is down !!!")
        else:
            p.prints("> [Client.main] This is the sorted array (client view): {} \n".format(sort_data))
```

Imagen 30. Clase *Client* y dos métodos *sort*() y *main*()

3. ServerStop.

El *ServerStop* tiene la clase *ServerStop,* que contiene el método estático *main*() (imagen 31). Dentro del método ejecutamos la función *stop(),* que pertenece a la clase *ServerProxy* (véase el párrafo 4.3.) y que crea una nueva conexión al servidor lanzando la orden *stop*. Después, se envuelve la función de *ServerProxy* en un bloque *try-except* para determinar si el servidor está en marcha. Si no fuera así, el método *stop*() introduce el error *ConnectionRefusedError* que capturamos en *except* y se procede a la ejecución fuera de *try-except*.

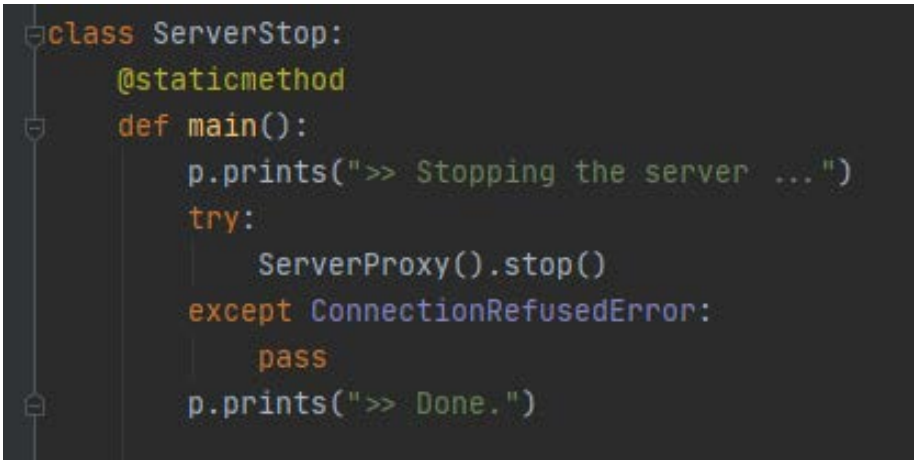


Imagen 31. Clase *ServerStop* y su método.

4.2. Test del Prototipo bajo Desarrollo

Durante los preparativos del prototipo del sistema de ordenación de números aleatorios se ha detectado algunos errores que habían paralizado el desarrollo y requerían correcciones para seguir programando. Como la depuración y el seguimiento de datos y elementos de sistema consumían un tiempo excesivo, se ha optado por un mecanismo de traza y depuración que ayuda a encontrar errores, implementando decoradores. Cabe señalar que mediante la verificación del correcto comportamiento del código eliminamos defectos (fallos SW de tipo lógico y léxico) en el prototipo con lo que código mejoramos su confiabilidad. El mecanismo de *logging* sirve a este propósito, con lo que puede considerarse como, y así se presenta, un mecanismo más para mejorar la confiabilidad del sistema.

El desarrollador puede seguir la ejecución a través de los *breakpoints,* pero la depuración a ciegas hará que todo el proceso consuma una excesiva cantidad de tiempo. El mecanismo *logging* puede proporcionar al desarrollador una tremenda ayuda al tratar de entender lo que el código en realidad hace, especialmente cuando alguien tiene que depurar y/o mantener el código de otra persona. El propósito de *logging* es hacer un seguimiento de los informes de errores y los datos relacionados de forma centralizada, también se encarga de registrar los eventos que ocurren durante el tiempo de ejecución.

- **Diseño.**

Por todo el sistema están las funciones de Python *print,* que imprimen en la consola en qué estado está la ejecución y por donde pasa el código. Algunas *prints* contienen los datos que el código está usando en un momento dado.

Una de las limitaciones de los decoradores es la necesidad de aplicarlos sobre las funciones o las clases creadas por el desarrollador. La función *print* es un *builtin* de Python, es decir, el método incorporado en el lenguaje que no puede ser cambiado por el desarrollador. Debido a esto, se ha decidido crear una función *print* propia (llamada *prints*()), para aplicar el decorador sobre esta. Así, cada vez que un código llegue a un *print* que contenga información de la ejecución, lo pasa por el decorador de *logging*.

- **Implementación.**

Como hemos mencionado, el decorador se usa sobre la función *prints*(), ubicada en el archivo "prints.py" en el subdirectorio "auxiliar" del directorio "utils".

El decorador-clase *LoggingDecorator* se encuentra en el archivo "logging_decorator.py", en el directorio "decorators". Este decorador se aplica sobre la función propia *prints*(), que está en el archivo "prints.py" en la carpeta "auxiliar".

El método *prints*() es un método suelto (las funciones en Python pueden ser sin clase) de tres líneas de código que se muestran en la imagen 32. Lo que hace este método se limita a pasar los strings que llegan a la función *print*() de Python.

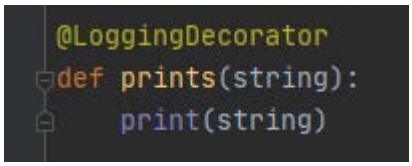


Imagen 32. Función *prints*()

El decorador *LoggingDecorator* (imagen 33) recoge el nombre de la función que llama a *prints*() y al nombre de su clase, formateándolos en un *string* que, a continuación, se asigna la variable *name_log_file*. Con la ayuda de la variable auxiliar *initialized,* el decorador comprueba si el *logger* ya está inicializado y formateado para usar. Si este no fuera el caso, realiza esta función con la ayuda del método *setup*() de la clase *MyLogger*.

Al inicializar el *logger* se comprueba si este *logger* existe. Se ejecuta entonces la función *prints*() y se hace un registro dentro de los archivos creados por *setup*(), utilizando el método *info*() de la librería "logger", que pasa los strings por filtros creados en la clase *MyLogger*.

```
class LoggingDecorator:
    def __init__(self, function):
        self.function = function

    def __call__(self, *args):
        if DecoratorConfiguration().logging_on is True:
            logger_tag = "logging"
            name_log_file = self.get_name_for_log_file()

            if not Helper.initialized:  # Checking if the logger has already been initiated.
                if DecoratorConfiguration().logging_on is True:
                    MyLogger(logger_tag).setup(name_log_file)  # Setting up the logger
                    Helper.initialized = True  # Changing helping variable to avoid multiple instantiation

            logger = logging.getLogger(logger_tag)
            if not logger:
                raise Exception("Logger does not exist. Please, check logging decorator")
            else:
                self.function(*args)
                logger.info(*args)
        else:
            self.function(*args)
```

Imagen 33. Clase *LoggingDecorator*, su constructor y *__call__*()

La clase *MyLogger* tiene el método *setup*(), donde se encuentra el código para definir el directorio y fichero donde se almacenarán los archivos del log. Esta carpeta en cualquier caso se crea en un nivel superior al directorio del sistema, es decir, en el directorio de todo el proyecto, donde también se encuentra la carpeta "backups" para almacenar las copias de seguridad de la aplicación (véase el párrafo 4.4.2.). A pesar de esto, el *setup*() crea los archivos ".txt" y ".html" y emplea los filtros para formatear los *strings* que pasan por la función *info*() de la librería "logger". Los archivos creados después de ejecutar el sistema se muestran en las imágenes 34 y 35.

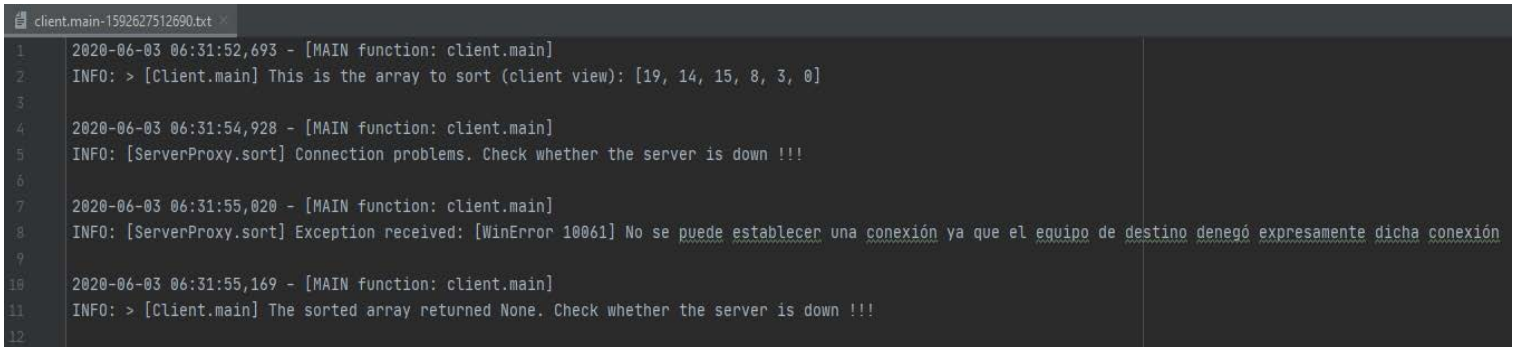


Imagen 34. Ejecución del cliente logueado en el archivo “.txt”.

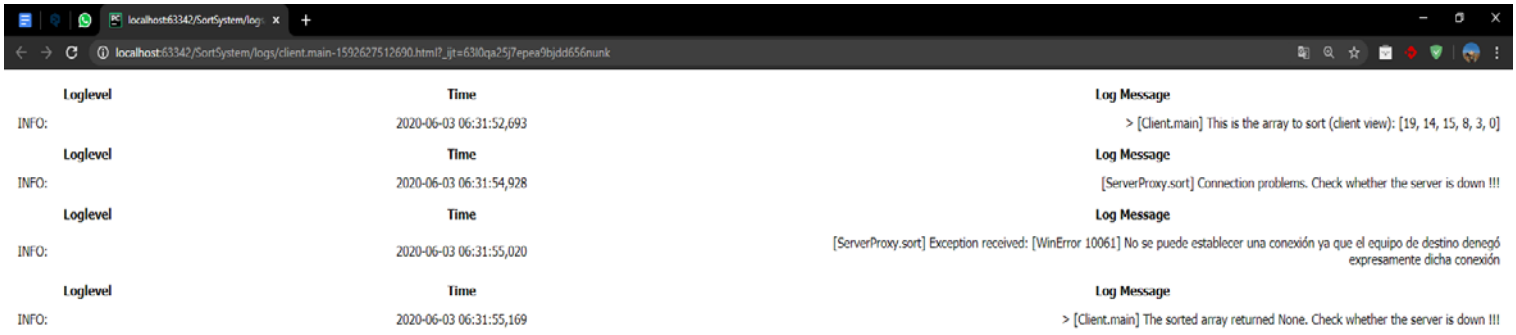


Imagen 35. Ejecución del cliente logueado en el archivo “.html”.

4.3. El Patrón Proxy para la Gestión de Invocaciones remotas

El *proxy* es un patrón de diseño estructural que permite a un objeto utilizar un servicio como si fuera local, a pesar de implementarse en otra máquina. La idea es que el *proxy* gestione toda la complejidad de la comunicación con el servidor (como la apertura del socket de comunicación y su uso para el envío y recepción de mensajes) y permita a cualquier otro objeto del sistema acceder al servicio ofrecido y utilizarlo como si se trabajara con un objeto local [21]. Desde esta perspectiva, el proxy oculta el hecho de que el servicio es remoto a los objetos que lo utilizan.

En el caso de nuestra aplicación de ejemplo, el servidor es el que ofrece la lógica de ordenación remota que podría utilizar el cliente. Aunque este mecanismo no es per se un mecanismo de tolerancia a fallos permite ocultar al cliente la necesidad de ubicar el servidor en una máquina distinta para así independizar el servicio de sus usuarios. Además, el proxy ofrece una oportunidad también para centralizar la aplicación de los

distintos mecanismos que afectan a los clientes, como los relativos al cifrado de las comunicaciones (gestionadas a través del proxy) o los reintentos de conexión (también gestionados desde el proxy) en caso de caída del servidor.

- **Diseño.**

El patrón proxy empleado para el presente trabajo ha sido el desarrollo de un servidor *proxy* que se activa al ejecutar el decorador *proxy* (figura 1). El cliente utiliza su propio método *sort*(), que activa un decorador de arranque del servidor *proxy* que, una vez puesto en marcha, invoca la función *sort*(), que se comunica con el servidor original a través de los sockets. Luego, los datos ordenados regresan por la misma ruta.

El decorador *ProxyDecorator* se usa sobre la función *sort()* de la clase *Client,* que está en el archivo "client.py" del directorio "client".

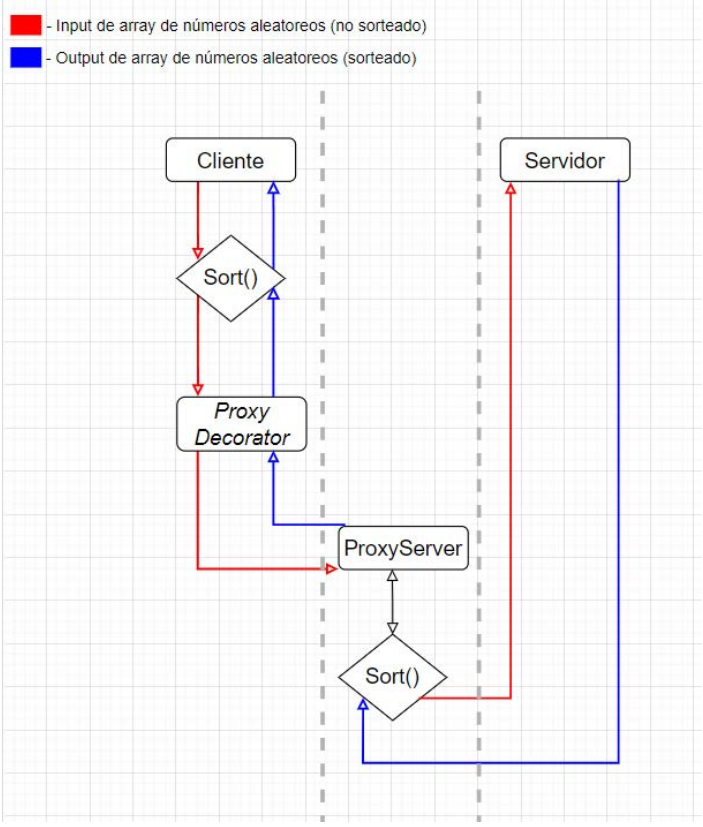


Imagen 36. Diseño del mecanismo proxy.

- **Implementación.**

El uso del cliente se describe más detalladamente en el párrafo 4.1.

El patrón *proxy* es de tipo clase que contiene *__init__*() y *__call__*() (imagen 37).

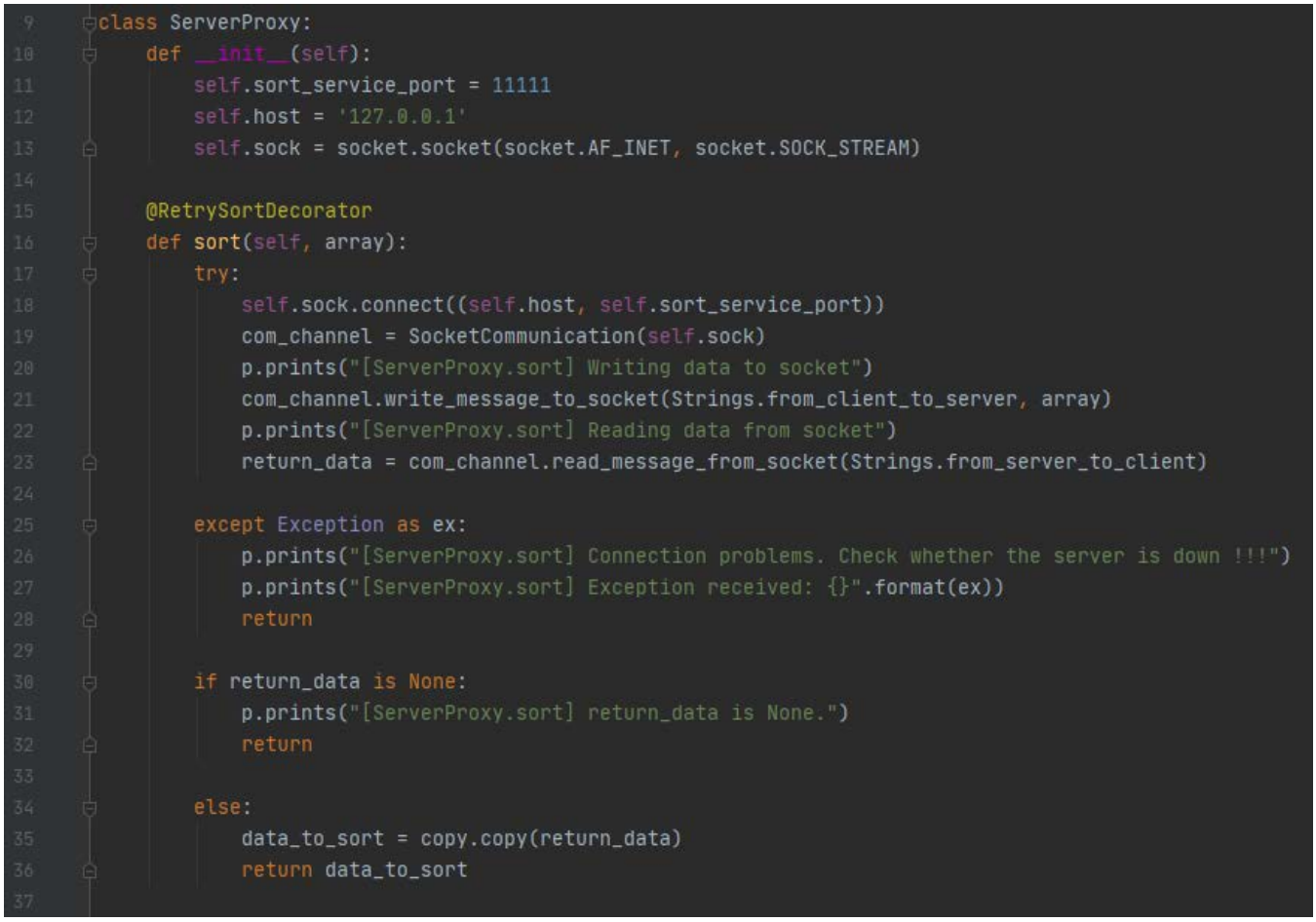
```
class ProxyDecorator:
    """
    It enables a seamless connection between a client class and a SortingServer
    """

    def __init__(self, function):
        self.function = function

    def __call__(self, *args):
        if DecoratorConfiguration().proxy_on is True:
            sorted_array = ServerProxy.sort(ServerProxy(), *args)
            return sorted_array
        else:
            return self.function(*args)
```

Imagen 37. *ProxyDecorator*.

El decorador arranca el servidor *proxy* (imagen 38) y ejecuta su método *sort*(), pasando el objeto de clase (para la sintaxis *self*) y el array para enviarlo al servidor original.



```
class ServerProxy:
    def __init__(self):
        self.sort_service_port = 11111
        self.host = '127.0.0.1'
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    @RetrySortDecorator
    def sort(self, array):
        try:
            self.sock.connect((self.host, self.sort_service_port))
            com_channel = SocketCommunication(self.sock)
            p.prints("[ServerProxy.sort] Writing data to socket")
            com_channel.write_message_to_socket(Strings.from_client_to_server, array)
            p.prints("[ServerProxy.sort] Reading data from socket")
            return_data = com_channel.read_message_from_socket(Strings.from_server_to_client)

        except Exception as ex:
            p.prints("[ServerProxy.sort] Connection problems. Check whether the server is down !!!")
            p.prints("[ServerProxy.sort] Exception received: {}".format(ex))
            return

        if return_data is None:
            p.prints("[ServerProxy.sort] return_data is None.")
            return

        else:
            data_to_sort = copy.copy(return_data)
            return data_to_sort
```

Imagen 38. *ServerProxy* y su *sort*().

El *sort*() ubicado en el bloque *try-except* prueba la conexión al servidor original. En caso de no poder conectarse, entonces lanza la *Exception* y regresa al cliente. En caso de poder, entonces instancia la clase de *SocketCommunication,* pasándola a la conexión y escribe el mensaje con el array al servidor original. A continuación, este lee los datos que se devuelven y, si los datos no son *None* (que pasan por alguna excepción que se captura en un *except*), los copia y los devuelve al usuario.

También el *ServerProxy* dispone del método *stop*() (imagen 39) que se usa para detener la ejecución del servidor en “serverstop.py”. Lo que hace este método es, en un bloque *try-except,* intentar a conectarse al servidor para después enviarle una orden de *stop* y cerrar la conexión. Si este mensaje no se pudiera enviar, la explicación sería que el servidor ya se encontraba detenido.

```
def stop(self):
    try:
        self.sock.connect((self.host, self.sort_service_port))
        com_channel = SocketCommunication(self.sock)
        com_channel.write_message_to_socket(Strings.from_client_to_server, Strings.stop)
        if self.sock is not None:
            self.sock.close()

    except IOError as ex:
        p.prints("[ServerProxy.sort] Connection problems. Check whether the server is down !!!")
        p.prints("[ServerProxy.sort] Exception received \"{}\": {}".format(ConnectionRefusedError.__name__, ex))
        return
```

Imagen 39. Método *stop*() de *ServerProxy.*

4.4. Tolerancia de Fallos de Caída del Servidor

Mientras el cliente puede tolerar una caída de servicio, el servidor, lógicamente no es capaz de tolerar errores en la ejecución de su lógica de ordenación. Esto afecta negativamente a la mantenibilidad y a la inocuidad del sistema. Entonces, se ha decidido desarrollar un decorador con el mecanismo de reintentos de conexión para tolerar por un tiempo determinado la caída de servidor. Junto a este, se ha desarrollado un decorador de *backup,* para poder restaurar el estado del servidor, consiguiendo así un ahorro significativo de tiempo.

4.4.1. Tolerar los fallos del servidor mediante reintentos.

- **Objetivo.**

Según la documentación oficial de Microsoft para la tecnología Asure, los reintentos son un concepto concebido para manejar fallos transitorias cuando el servicio trata de conectarse a un servicio o recurso de red. Así, se reintenta de manera transparente una operación fallida, mejorando la estabilidad del sistema [14]. La idea que subyace es sencilla, la congestión de la red puede llevar a una pérdida de paquetes o a que la operación de conexión lleve más tiempo del esperado. El reintento mitiga el efecto en el sistema de este tipo de problema. También en el caso de que el servidor caiga y se detecte su caída, los mecanismos de restauración del servicio tardarán un tiempo en desplegar de nuevo el servidor (tal vez en otra máquina) y llevarlo a un estado anterior

conocido y correcto, seguro e inocuo. Los reintentos también sirven en el caso de que, durante todo este proceso, un cliente intente utilizar el servicio suministrado y el servidor aún no se encuentre en condiciones de ofrecerlo.

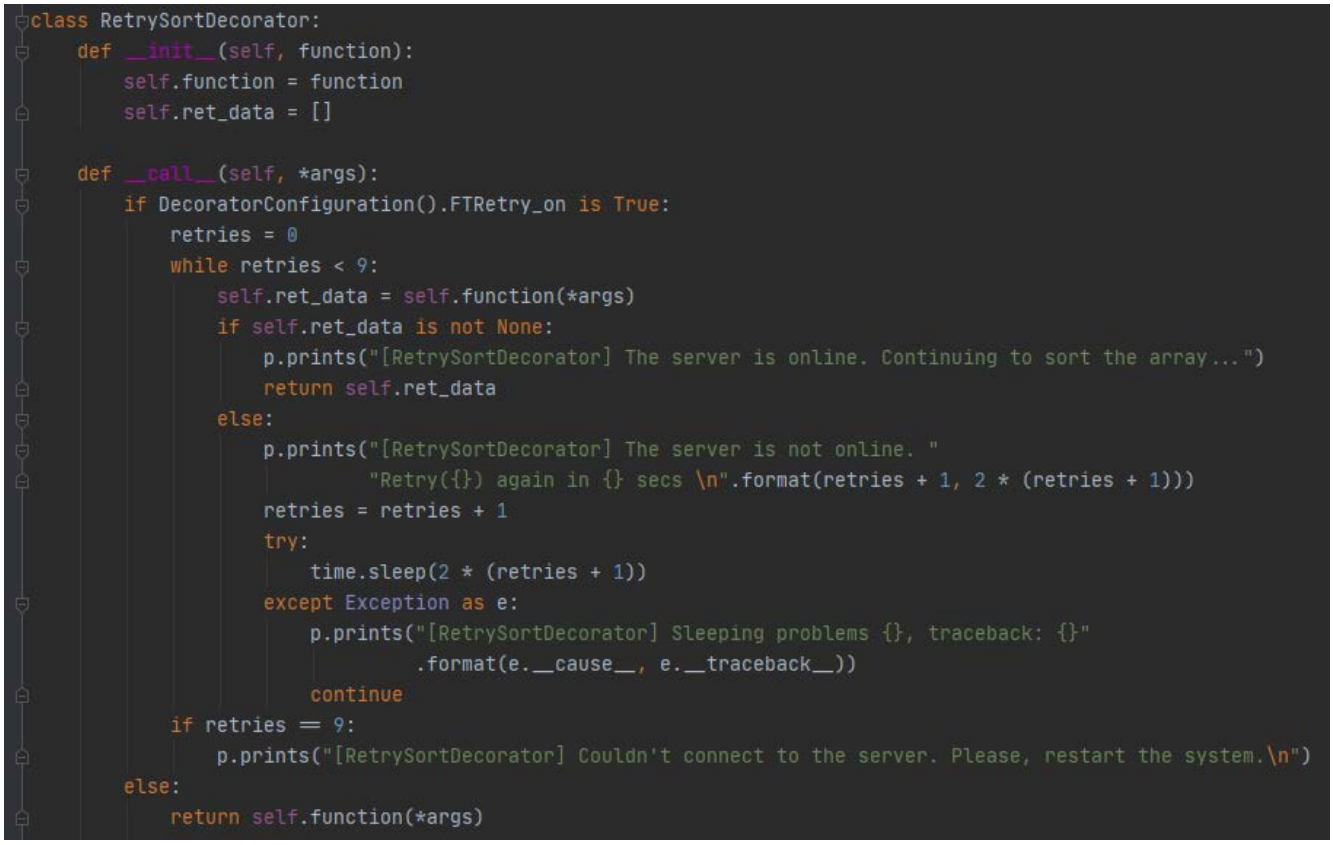
Obviamente, el mecanismo no es infalible y, si los reintentos no son lo suficientemente numerosos y no están espaciados en el tiempo, puede fallar.

- **Diseño.**

El diseño es elemental: consiste en aplicar el decorador que hace reintentos de conexión sobre el método *sort*(), que conecta el servidor *proxy* al original.

- **Implementación.**

RetrySort es el decorador-clase típico de las funciones *__init__*() y *__call__*() (imagen 40). Cuando llaman a *sort*() de *proxy,* el decorador entra en un bucle de 9 intentos. Al entrar al bucle intenta a ejecutar el *sort*(). Si este método devuelve el array, entonces sale del bucle devolviendo los datos a *ProxyServer*. Si no llegara nada, entonces se calcula el tiempo para hacer un reintento, repitiendo así el bucle.



```
class RetrySortDecorator:
    def __init__(self, function):
        self.function = function
        self.ret_data = []

    def __call__(self, *args):
        if DecoratorConfiguration().FTRetry_on is True:
            retries = 0
            while retries < 9:
                self.ret_data = self.function(*args)
                if self.ret_data is not None:
                    p.prints("[RetrySortDecorator] The server is online. Continuing to sort the array...")
                    return self.ret_data
                else:
                    p.prints("[RetrySortDecorator] The server is not online. "
                             "Retry({}) again in {} secs \n".format(retries + 1, 2 * (retries + 1)))
                    retries = retries + 1
                    try:
                        time.sleep(2 * (retries + 1))
                    except Exception as e:
                        p.prints("[RetrySortDecorator] Sleeping problems {}, traceback: {}"
                                 .format(e.__cause__, e.__traceback__))
                        continue
            if retries == 9:
                p.prints("[RetrySortDecorator] Couldn't connect to the server. Please, restart the system.\n")
        else:
            return self.function(*args)
```

Imagen 40. Decorador *RetrySort.*

4.4.2. Mecanismo de backup

- **Objetivo.**

El *backup* se usa para guardar y restaurar el estado previo del servidor y las configuraciones de decoradores desde el archivo creado por el decorador. Este mecanismo permite restaurar el servicio del servidor consumiendo poco tiempo en hacerlo.

- **Diseño.**

La estructura del mecanismo backup se muestra en la imagen 41. Instanciando el sistema, la consola pedirá elegir las opciones para activar los decoradores. Si el *backup* está activado, el decorador pedirá al programador escribir el nombre para el archivo del *backup* y procederá a procesar el código. Para restaurar las opciones de los decoradores, se requiere escribir "-file" y el nombre de archivo donde están los valores para restaurar.

La siguiente aplicación del mecanismo está en el servidor sobre sus métodos *set_client_number*() y *get_client_number*(). Al ejecutar el *setter,* el decorador guarda el número del cliente para restaurarlo al usar el *getter* después de caída.

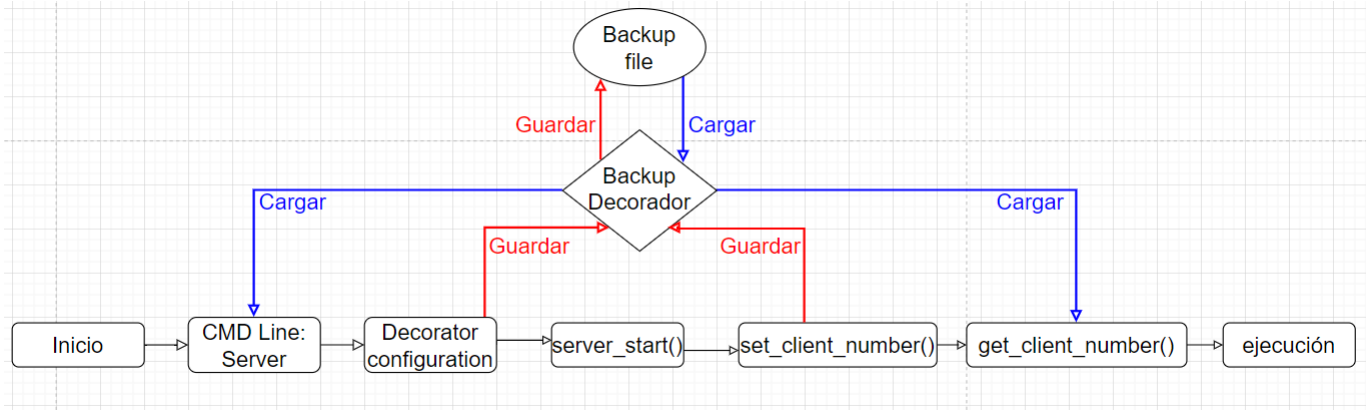


Imagen 41. Diseño del *backup.*

- **Implementación.**

Implementando el mecanismo de *backup* se ha encontrado la limitación de los decoradores. Para no sobrecargar la función original con los argumentos que se necesitan solo en un decorador, hemos optado por desarrollar el decorador con argumento.

En el caso del sistema, es necesario pasar el "tag" de la función por el decorador, porque el *setter* requiere el argumento para almacenarlo y *getter* no. Entonces, dentro del decorador estas funciones tendrían que pasar por lógicas diferentes.

Como se observa en el capítulo 3, el decorador-clase con argumentos pierde la etapa de inicialización de este decorador, porque, si tiene argumentos, estos tendrían que pasar por el método *__init__*(), y la función en este caso pasaría por *__call__*(), ejecutándose en ese preciso instante.

Esto ha sido un problema, pues el decorador, por estar arriba de la función, se inicializaba e intentaba ejecutar la función no desde donde se llamaba sino desde donde se inicializaba. En otras palabras, el decorador intentaba ejecutar el método cuando todavía no tenía argumentos, provocando así errores. Para solucionar ello, se ha optado

por dividir los *setters* y *getters* en dos funciones (imagen 42) para inicializar primeramente la función inferior que no esté decorada. En este caso, la interpretación no llegaba a ejecutar el decorador antes de la llamada a la función.

```
def set_client_number(self, client_number):  # El setter para el número del cliente
    @BackupDecorator("set", client_number)
    def init_set_client_number(client_n=client_number):
        self.client_number = client_n
```

Imagen 42. El *setter.*

El decorador se usa en el servidor sobre las funciones *init_set_client_number*() y *init_get_client_number*(). Las funciones relacionadas con decorador *save_options*() y *get_options*() se aplican en el parseo de opciones después de activar el *backup*.

El decorador-clase *BackupDecorator,* mostrado en la imagen 43, tiene la función *wrapper*(), que, al llamar a las funciones del servidor, consigue el nombre de la ruta donde los archivos de *backup* tienen que guardarse. Su carpeta se encuentra en el mismo directorio donde está la carpeta del *logging*.

Luego, el *wrapper* lee el *tag* de la función. Si es “set”, el decorador abre el archivo y guarda en su interior el número del cliente. Si el *tag* es “get”, el decorador busca el archivo y carga desde allí el número guardado. Si el archivo no existe, el decorador crea uno nuevo y pone el número 0. Las funciones *save_options*() y *get_options*() funcionan de la misma manera, y tienen el input para guardar el *backup* de las opciones con el nombre proporcionado por el usuario.

```python
class BackupDecorator:
    def __init__(self, type_of_func=None, args=None):
        self.args = args
        self.type_of_func = type_of_func

    def __call__(self, func):
        @wraps(func)
        def wrapper():

            dir_name = get_dir_name()

            if self.type_of_func == "set":
                saved_args = self.args
                file = open(dir_name + "saved_args", "w", encoding="UTF-8")
                saved_args = str(saved_args)
                file.write(saved_args)
                func(self.args)

            elif self.type_of_func == "get":
                try:
                    file = open(dir_name + "saved_args", "r", encoding="UTF-8")
                    data = int(file.read())
                    return data

                except FileNotFoundError:
                    file = open(dir_name + "saved_args", "w", encoding="UTF-8")
                    file.write('0')
                    file.close()
                    file = open(dir_name + "saved_args", "r", encoding="UTF-8")
                    data = int(file.read())
                    return data

        if DecoratorConfiguration.FTBackup_on is True:
            return wrapper()
        else:
            if self.type_of_func == "set":
                func(self.args)
            elif self.type_of_func == "get":
                return func()
```

Imagen 43. El decorador-clase *BackupDecorator.*

4.5. Tolerancia de Fallos SW en el Servidor

Algunos algoritmos del sistema pueden tener errores de código introducidos por el programador o de diseño relativos a fallos en el algoritmo utilizado este tipo de problemas se pueden solventar mediante mecanismos de diversificación, por ejemplo generando alternativas de servicio utilizando algoritmos distintos. También es posible utilizar distintos lenguajes de programación para cada alternativa, lo que evita los fallos en modo común propios de un lenguaje, o incluso, si llevamos esta lógica al extremo, podríamos solicitar a distintos equipos de desarrollo que generasen cada uno una de las alternativas. Esto permitirá que si en determinadas circunstancias, una de las alternativas fallara, el voto de la alternativa "más popular" permita tolerar ese fallo.

- **Objetivo.**

El objetivo de la diversificación funcional es detectar y enmascarar fallos residuales de implementación y diseño durante la ejecución de un programa, previniendo la avería de la aplicación y continuando su operación. Cuando este concepto se aplica a un programa o algoritmo, generando varias alternativas del mismo, el mecanismo recibe el nombre de "N-Version Programming".

Para este trabajo se han implementado 7 algoritmos distintos de ordenación de números. Para tolerar un fallo transitorio que afecte al cálculo de uno de los algoritmos, un error en la programación de alguna alternativa, o en el diseño de algún mecanismo de ordenación, todas las versiones (algoritmos) se ejecutan simultáneamente y cada una proporciona su resultado. A continuación, se comparan los resultados obtenidos y se determina el resultado correcto a través de una votación por mayoría. Contando con 7 algoritmos, el sistema conseguirá tolerar errores en la ordenación que realicen hasta 3 de los 7 algoritmos disponibles Si fallaran más versiones, el sistema será incapaz de tolerar el fallo masivo que se produzca y el error resultante será fatal, es decir, se devolverá un conjunto de valores desordenado o no se devolverá nada.

- **Diseño**.

Este mecanismo se emplea solo en la parte del servidor, pues no guarda relación alguna con el cliente, el cual no realiza ningún calculo. Para ser capaz de tolerar los fallos que lleguen desde los algoritmos, “N-Version” tiene que aplicarse sobre la función *sort*() del servidor para decorar el ordenación de un algoritmo.

Aquí nos hemos encontrado ante una limitación de decoradores que ya habíamos observado durante el desarrollo del mecanismo de *logging*: Es imposible aplicar el decorador sobre la llamada de la función (como “*x = socket.accept*()” o “*print*(*string*)”), porque la sintaxis de anotación de los decoradores requiere su aplicación solo sobre clases o funciones creadas. Entonces nos hemos obligado a tener que envolver la llamada al *sort*() en la función, para evitar esta limitación y poder aplicar el decorador.

La estructura del mecanismo *N-Version* está presentada en la imagen 44.

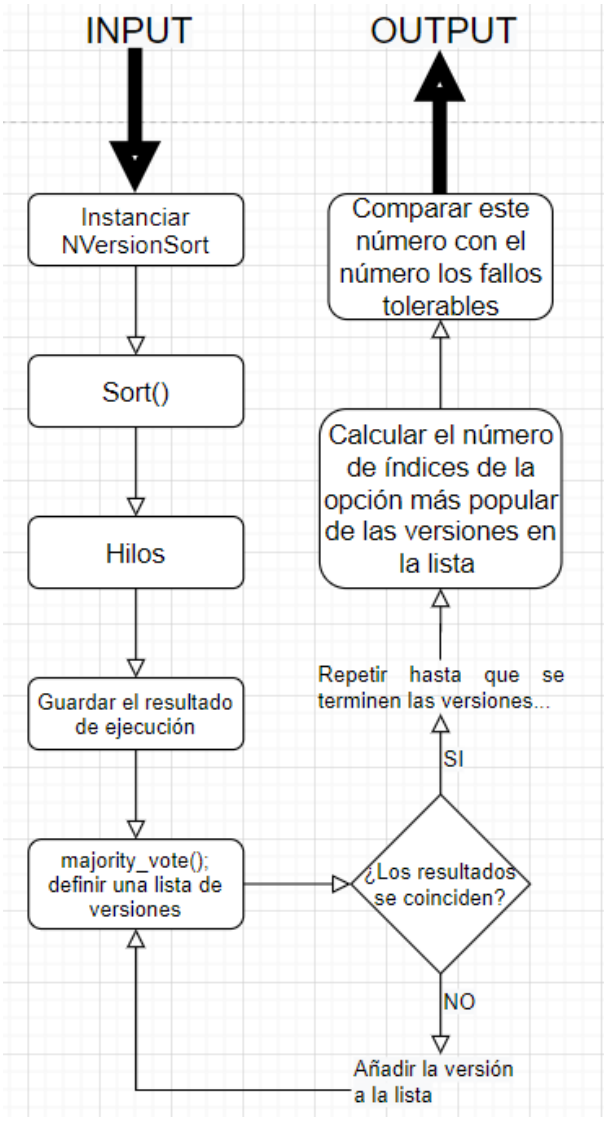


Imagen 44. La estructura del mecanismo *NVersion*.

En primer lugar, el mecanismo necesita instanciar la clase *NVersionSort* y llamar al *sort*(), donde se preparan los hilos para ejecutar los algoritmos. Tras lanzarlos al procesamiento, se guardan los resultados y se envían al método *majority_vote*(), donde se define una lista con las versiones obtenidas por el resultado de la ejecución de algoritmos. Luego, en la función *index_result_list*(), se verifica que los resultados de la ejecución coinciden con los datos de las versiones de la lista. Finalmente, se calcula el número de los índices más populares para las versiones, se comparan con el número de fallos permitidos y, si todo pasó bien, se devuelve el *array* ordenado.

- **Implementación**.

El decorador-clase *NVersionDecorator* (imagen 45) se aplica sobre la función *sort*(). Cuando la llaman desde fuera, primero el decorador instancia el algoritmo especial *NVersionSort* (que está en el archivo "nversión.py" del subdirectorio "sortingalgoritm" del directorio "utils") que ejecuta los otros algoritmos en un bucle *for.* Después, comprueba sus resultados y, si el *NVersion* ha podido tolerar los fallos, entonces devuelve los datos.

```
class NVersionDecorator:
    def __init__(self, function):
        self.function = function

    def __call__(self,  *args, **kw):
        if DecoratorConfiguration().FTDiversification_on is True:
            nv = SortingAlgorithmFactory.new_sa(alg_type.NVERSIONSORT)
            try:
                if args is None:
                    print("[NVersionDecorator.wrapper] Trying to sort a None type array of ints")
                    raise TypeError("NVersionDecorator recieved a None type array")

                else:
                    print("[NVersionDecorator.wrapper] Sorting array {}".format(*args))
                    if nv is not None:
                        output_data = nv.sort(*args)
                    else:
                        output_data = None
                    if output_data is None:
                        print(">> Hey a SORTING ERROR occured, please do your algorithm more Fault-Tolerant!!!")
            except Exception as ex:
                print(traceback.format_exc())
                raise ex

            if output_data is None:
                return None

            else:
                return copy.copy(output_data)
        else:
            return self.function(*args)
```

Imagen 45. El decorador *NVersionDecorator*.

Dentro del bloque *try-except* se comprueba que la función *sort*() tiene los argumentos. Si no los tuviera, introduce la excepción *TypeError*. Pero, si *sort*() los tuviera, la ejecución verifica que la *factory* ha creado el objeto de la clase *NVersionSort*. En caso afirmativo, se ejecuta el código del método *sort*() en esta clase. Si, por el contrario, no se haya

creado nada, el decorador entonces devuelve *None* para, a continuación, loguear el error pasado.

Más adelante se encuentra el *except* indefinido de *Exception*. Como la clase de ordenación tiene una gran cantidad de métodos y cualquiera de estos es capaz de producir errores distintos e inesperados, se usa el *traceback* para determinarlos. En el caso de que todo estuviera bien, el decorador hace copia del array y la devuelve al servidor.

NVersionSort dispone de 3 clases diferentes y 12 funciones distintas. La primera clase es *NVersionSort* y es subclase de *SortingAlgoritm,* que contiene 8 funciones. Su propio constructor *__init__*() tiene un argumento, pero su uso no es obligatorio porque tiene el valor por defecto *None* (imagen 46).

Dentro del constructor se preparan el masivo *versions* de los hilos con los algoritmos para ejecutar, y el masivo *data,* con los resultados de la ejecución de los algoritmos que luego usamos en el *setter* *__setdata__*(). El comportamiento esperado del usuario es pasar a *NVersionSort* o un *array* de algoritmos, o un algoritmo o ninguno en absoluto. Se trabaja con este comportamiento preparando un bloque *if-else* que comprueba el tipo del argumento, y define la variable *algorithms.* Si no ha llegado nada al constructor, este prepara una lista de instancias de todos los algoritmos.

```
class NVersionSort(SortingAlgorithm):
    def __init__(self, algorithms=None):
        self.data = []
        self.versions = []

        if isinstance(algorithms, list):
            # Checking if the user pass an array...
            self.algorithms = algorithms

        elif algorithms is None:
            # ...or nothing...
            self.algorithms = [SortingAlgorithmFactory.new_sa(alg_type.QUICKSORT),
                               SortingAlgorithmFactory.new_sa(alg_type.HEAPSORT),
                               SortingAlgorithmFactory.new_sa(alg_type.MERGESORT),
                               SortingAlgorithmFactory.new_sa(alg_type.BUBBLESORT),
                               SortingAlgorithmFactory.new_sa(alg_type.SELECTIONSORT),
                               SortingAlgorithmFactory.new_sa(alg_type.INSERTIONSORT),
                               SortingAlgorithmFactory.new_sa(alg_type.PYTHONARRAYSORT)]

        else:
            # ... or an algorithm.
            self.algorithms.append(algorithms)

        self.versions = self.__setitem__(self.algorithms)
```

Imagen 46. Constructor de la clase *NVersionSort*.

A continuación, se ejecuta la función *setter,* propia de la clase *__setitem__*() (imagen 47), donde se crea un *array* de los hilos para ejecutar los algoritmos, y lo asigna a la variable *versions*. El *setter* requiere o un algoritmo o un masivo de algoritmos. Si no hubiera llegado nada o si hubiéramos recibido un array vacío, la ejecución entonces sale de *__setitem__*(). En caso de haber llegado datos correctos, el *setter* ejecuta un bucle *for* para instanciar el objeto de la clase interior *NVersionThread,* que crea un hilo para cada algoritmo, añadiéndolo en la lista *versions* y devolviéndola posteriormente al constructor.

```
def __setitem__(self, algorithm):
    """
    Setter of the various instantiated algorithms
    """
    if algorithm is None or len(algorithm) == 0:
        return

    for i in range(0, len(self.algorithms)):
        # Any sort coming. For example quick sort.
        # algorithm: <auxiliar.sortingalgorithm.quick_sort.QuickSort object at 0x0074E508>

        nvt = NVersionSort.NVersionThread(self.algorithms[i])
        # nvt: <__main__.NVersionSort.NVersionThread object at 0x026DDAC0>

        self.versions.append(nvt)
    return self.versions
```

Imagen 47. *Setter __setitem__*().

La clase interior *NVersionThread* (imagen 48) dispone de un constructor donde se almacena el algoritmo y se define una lista *data* para los datos a ordenar, así como un *setter* que almacena en esta lista la copia del *array* de números. Además, esta clase posee un getter que devuelve la copia de *data* y el método *run*(), que lanza el proceso de ordenación del algoritmo y retorna su resultado.

```
class NVersionThread:
    """
    This inner class is the thread carrying out the sorting of the provided array of ints
    """

    def __init__(self, algorithm):
        self.algorithm = algorithm
        self.data = []

    def __setitem__(self, value):
        if self.algorithm is not None:
            self.data = copy.copy(value)

    def __getitem__(self):
        if self.algorithm is not None:
            return copy.copy(self.data)

    def run(self):
        if self.algorithm is not None:
            return self.algorithm.sort(self.__getitem__())
```

Imagen 48. *NVersionThread.*

El método principal de la clase que se llama desde el decorador es el método *sort*() que lo podemos encontrar en la imagen 49. Esta función se llama desde el decorador para ordenar los números usando el mecanismo de *NVersion*. El método requiere un argumento: la lista de datos.

```
def sort(self, array):
    if self.algorithms is None or len(self.algorithms) == 0:
        p.prints("[NVersionSort.sort] No versions available")
        return None

    p.prints("[NVersionSort.sort] Working with {} versions".format(len(self.algorithms)))
    thread = ThreadPoolExecutor()
    p.prints("[NVersionSort.sort] Transmitting the array to sort to all versions")

    for i in range(len(self.algorithms)):
        self.versions[i].__setitem__(array)
        p.prints("[NVersionThread.__setitem__] Cloning the data...")

    if len(self.versions) == 1:
        return self.versions[0].__getitem__()

    p.prints("[NVersionSort.sort] All versions are now under execution")
    p.prints("[NVersionSort.sort] Waiting for all versions to finish")

    for i in range(len(self.algorithms)):
        future = thread.submit(self.versions[i].run)
        d = future.result()
        self.__setdata__(d)

    for i in range(len(self.algorithms)):
        thread.shutdown(wait=True)

    p.prints("[NVersionSort.sort] Voting for the major result")
    data = self.majority_vote()
    return data
```

Imagen 49. Método *sort*().

Cuando la ejecución llega a esta función, en primer lugar, comprueba que la clase se ha inicializado y que ya tiene una variable de algoritmos definida y no vacía. En caso negativo, devuelve *None* para loguear el error. En caso afirmativo, instancia un *pool* de hilos y se hace copias de array, para ordenar cada algoritmo.

En segundo lugar, la ejecución comprueba si la versión a ejecutar es única. Si no lo fuera, usa el *getter* que devuelve el objeto del algoritmo. Luego, lanza los hilos de algoritmos para procesar pasando al ejecutor la función *run*() de *NVersionThread* y guarda las copias de los resultados en el *array data* usando el otro *setter,* *__setdata__*(), idéntico a *__setitem__*(). Más adelante, se cierran los hilos que quedaron abiertos y la ejecución los espera si no hubieran terminado de procesarse.

Después, la ejecución lanza el método *majority_vote*() mostrado en la imagen 50, para comparar las listas ordenadas.

```
def majority_vote(self):
    arr_list = []

    for i in range(len(self.versions)):
        index = self.index_result_list(arr_list, self.versions[i].__getitem__())
        if index == -1:
            arr_list.append(NVersionSort.ResultMatches(self.versions[i].__getitem__(), i))
        else:
            arr_list[index].matching_indexes.append(i)

    index = self.index_most_popular_option_in_list(arr_list)
    if index == -1:
        return None
    else:
        if len(arr_list[index].matching_indexes) ⩾ ((int((len(self.versions))/2))+1):
            if arr_list[index].result is None:
                return None
            else:
                return copy.copy(arr_list[index].result)
        else:
            return None
```

Imagen 50. Función *majority_vote*()

Cuando la ejecución llega a la función, esta crea una lista *arr_list* para guardar las versiones con los resultados del procedimiento de algoritmos. El bucle *for* define un *index* que comprueba si los datos están en la lista creada usando el método *index_result_list*() presentado en la imagen 51. La primera versión que pasa por este *for* entra en *index_result_list(),* donde en el primer *if* obtiene el índice "-1", porque la lista de resultados todavía está vacía y, al sacar este índice, esta versión se guarda en el

arr_list con su resultado como la instancia de la clase de almacenamiento *ResultMatches.*

```
@staticmethod
def index_result_list(arr, result):
    if len(arr) == 0:
        return -1
    for i in range(len(arr)):
        rm = arr[i]
        if result == rm.result:
            return i
    return -1
```

Imagen 51. Función estática *index_result_list*().

La función *index_result_list*() requiere el array *arr_list* y el resultado de ordenación de un algoritmo. Después del primer *if* que sirve para definir la versión primaria, la ejecución llega en un bucle *for*. En este bucle la lista guardada *arr* se compara con el resultado de ejecución de otro algoritmo.

Si el resultado coincidiera con la versión guardada, la función devuelve "0" y en el método inferior (*majority_vote*()) se sitúa el índice del resultado coincidido dentro de la instancia de la versión usando su lista *matching_indexes.* Es decir, lo envía dentro de la variable del objeto *ResultMatches.*

Si, por el contrario, el resultado no coincidiera con la versión guardada, el bucle *for* seguirá buscando coincidencias, rastreando todos los objetos del *array* de versiones. Si ni con esas las encontrara, entonces devuelve el índice "-1". Este índice significa que la función intentó comparar las listas presentadas, pero que sus valores resultaron diferentes. Entonces, se añade en el array de versiones la versión nueva, que tiene su propio resultado.

La clase interior de almacenamiento *ResultMatches* (imagen 52) guarda los resultados de la ejecución de las diferentes versiones de algoritmos y prepara la lista de los índices

de los algoritmos cuyos resultados coinciden con el guardado. Es decir, si hay tres versiones con resultados diferentes, la clase *ResultMatches* se instancia tres veces guardando cada versión, creando sus propias listas para cada uno.

```
class ResultMatches:
    def __init__(self, result, i):
        self.result = copy.copy(result)
        self.matching_indexes = []
        self.matching_indexes.append(i)
```

Imagen 52. *ResultMatches.*

Al preparar todos los índices y versiones, el código ejecuta el método *index_most_popular_option_in_list*(). Esta función presentada en la imagen 53 verifica la lista que llega por el argumento. Si el resultado es *None,* devuelve "-1", que más adelante loguea el error. Si la lista tiene un argumento, devuelve 0, que luego se usa para sacar los datos de la única versión que hay. Y si tiene más versiones, entonces las compara entre sí para encontrar a una que tenga más *matches* que el resto y así devolver el número de índices que coincidieron con esta versión.

```
@staticmethod
def index_most_popular_option_in_list(arr_list):
    if arr_list is None:
        return -1
    if len(arr_list) == 1:
        return 0
    most_popular_index = 0
    for i in range(len(arr_list)):
        if len(arr_list[most_popular_index].matching_indexes) < len(arr_list[i].matching_indexes):
            most_popular_index = i
    return most_popular_index
```

Imagen 53. Método *index_most_popular_option_in_list*().

El índice de los resultados coincidentes se devuelve a *majority_vote*(), donde primero se comprueba que este no es "-1" y, si no es, lo compara con el número de los fallos que puede tolerar el sistema. Los fallos tolerables se calculan según la formula "*(versions/2)*

-1" donde la división se redondea. Como para este trabajo se ha decidido que el número de fallos posibles sea 3, entonces el sistema sigue funcionando bien, sin caídas, en el caso de haber 4 versiones correctas. Al ejecutar *NVersionSort* se devuelve la copia del *array* ordenado y el sistema tolera hasta 4 fallos.

4.6. Comunicaciones Seguras Mediante Cifrado Simétrico

Los atacantes que quieran conseguir los datos que pasan por el sistema podrían hacerlo a través de la conexión. El acceso de estos a datos puros y claros puede afectar tanto a la confidencialidad, como a la integridad de dichos datos.

- **Objetivo.**

Se ha implementado un decorador de cifrado que cifra los datos de tal manera que resulte muy complejo su descifrado. Este decorador hace que el trabajo de los atacantes sea mucho más largo y pesado, forzando a los ciberdelincuentes a tener que dedicar una mayor cantidad de tiempo y dinero en conseguir la información deseada. De esta manera, se aumenta la seguridad del proyecto, porque el coste del hackeo aumenta, acercándose al coste de datos, haciendo que los ataques al sistema sean menos rentables y relevantes.

- **Diseño**.

El diseño se basa en la implementación del decorador sobre las funciones de la *SocketCommunication*. Al escribir el mensaje lo codifica usando el método *encrypt*() de la librería "cryptography.fernet", que encripta los datos. El resultado de esta encriptación se conoce como *Fernet Token* y tiene fuertes garantías de privacidad y autenticidad. Este token representa la autenticación del usuario. Los datos dentro de un *fernet* están protegidos con claves propias o de cifrado simétrico. El mensaje cifrado tiene el tipo *bytes,* representando una cadena de símbolos codificados. Al recibir el mensaje el decorador lo decodifica y devuelve el resultado a la función que llamó a los métodos de comunicación.

- **Implementación**.

El decorador-función *cyphering_decorator*() (imagen 54) tiene el argumento que define la función decorada. Es decir, también es un *tag* al igual que lo que tenemos en *BackupDecorator*. Todo lo que hace este decorador es determinar el tag de la función decorada y codificar o decodificar mensajes debido a este tag.

```
def cyphering_decorator(decorator_argument):
    def decorator(function):
        def wrapper(*args, **kwargs):
            if DecoratorConfiguration.cyphering_on is False:
                return function(*args, **kwargs)
            else:
                if decorator_argument == "write":
                    encrypted_message = Cyphering().encrypt(args[2])
                    encrypted_message = function(args[0], args[1], encrypted_message)
                    return encrypted_message

                elif decorator_argument == "read":
                    encrypted_message = function(*args, **kwargs)
                    decrypted_message = Cyphering().decrypt(encrypted_message)
                    return decrypted_message
        return wrapper
    return decorator
```

Imagen 54. Decorador de cifrado.

4.7. Discusión

Cada decorador implementado en el sistema ha conseguido reproducir el comportamiento deseado. Esto justifica y comprueba el verdadero valor práctico de los decoradores como la tecnología para separar los mecanismos no-funcionales bajo estudios de los funcionales que son propios a aplicaciones cliente-servidor.

Pero, a pesar de las ventajas que presentan los decoradores, no nos hallamos ante una tecnología que podamos calificar como ideal. Los decoradores tienen sus limitaciones al igual que su lenguaje de implementación, *Python*. Estas limitaciones no son excesivas, pero aun así pueden resultar incómodas para el desarrollo de grandes aplicaciones.

Para este trabajo, la principal limitación de los decoradores ha sido la de permitir la manipulación de los argumentos de las llamadas. Los argumentos cambian significativamente el comportamiento de la tecnología, obligando al ingeniero a tener que hacer introducir cambios en el código funcional de la aplicación.

Es posible resolver este problema sobrecargando los argumentos de métodos decorados obteniendo el valor necesario, como la parte de la función. Esto lleva a los argumentos sin uso a no implementarse para nada más, excepto a ser pasados por el decorador.

Como no es la solución al problema, existe otra manera que consiste en recoger los datos necesarios dentro del decorador, gracias a la reflexibilidad y a la ventaja de introspección de *Python*, investigando el *stack* que muestra la información común donde el programador tiene que encontrar lo que necesita y sacarlo programáticamente escribiendo el código. Esto lleva a un problema de reutilización del decorador porque, al escribir más líneas de código, el mecanismo llega a ser más específico y complejo, lo que limita su reutilización. Finalmente, se ha determinado solucionar este problema usando la función auxiliar que tiene en su interior la función original ya decorada, tal y como se ha ilustrado para el decorador de Backup en la sección 4.4.2.

Otro problema encontrado durante el desarrollo es la limitación de uso de los decoradores sobre las llamadas a las funciones (imagen 55).

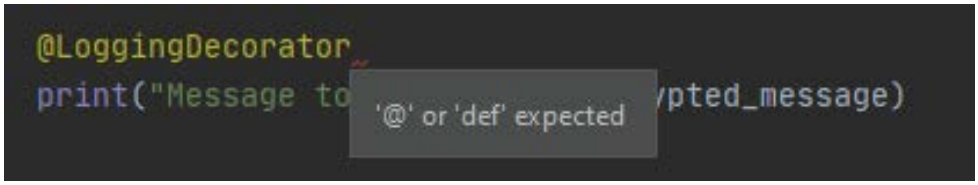


Imagen 55. Error que impide el uso de los decoradores sobre las llamadas.

Por lo demás, los decoradores pueden aplicarse solo sobre las funciones y clases creadas por el programador. Las funciones propias de *Python* tampoco permiten el uso de los decoradores.

CAPÍTULO 5: ADAPTACIÓN Y REUTILIZACIÓN DE MECANISMOS

En este capítulo trataremos la adaptación y reutilización de los decoradores ya desarrollados e implementados en un sistema. La tecnología de la decoración, al igual que los mecanismos relacionados con la programación orientada a objetos, está enfocada a la separación de las funcionalidades de la aplicación. Su objetivo principal es ejecutar una lógica externa al código base inyectando el comportamiento necesario en determinadas partes del código. Este propósito de los decoradores les permite adaptarse a diferentes sistemas, porque el mecanismo está desarrollado ya y solo falta realizar algunos ajustes para el nuevo sistema y su futura reutilización en otros sistemas.

Para demostrarlo, se explicará el concepto de orquestación de mecanismos, concretamente la composición y adaptación de los mismos, y se mostrará un ejemplo de la ordenación de array de enteros. Más adelante, trataremos la reutilización de los mecanismos no funcionales en otros ejemplos donde se detallarán los requisitos para esta reutilización y se mostrará el despliegue de estos en otra aplicación de tipo servidor-cliente que “juega” al Ping-Pong.

5.1. Orquestación de Mecanismos no funcionales

Ramirez y Corrales manifiestan que la orquestación es el proceso de definir en detalle el flujo de datos y el flujo de control entre los servicios compuestos que permiten la generación de un servicio ejecutable. En otras palabras, el objetivo de la orquestación no es otro que el de secuenciar los servicios y proveer la lógica adicional para procesar datos, en nuestro caso, procesar la composición entre mecanismos no funcionales [19].

5.1.1. Composición y Adaptación

Como se puede intuir, la definición de orquestación se acerca al término de la composición. La composición describe una clase que hace referencia a uno o más objetos de otras clases en variables de instancia. Es decir, la composición es un concepto que modela una relación *has-a*.

Todo ello permite la creación de tipos complejos combinando objetos de otros tipos. Esto significa que una clase Composite puede contener un objeto de otra clase Componente. Esta relación implica que un componente pertenece a compuesto. La adaptación de la funcionalidad se proporciona aprovechando la composición de un grupo de objetos encapsulados. Esta encapsulación ofrece los módulos autónomos que pueden usarse ajustándose un poco para el uso en otros sistemas.

5.1.2. Ejemplo de Ordenación de Array de Enteros

En este bloque mostraremos la salida del sistema desarrollado aplicando los decoradores de proxy, reintentos y cifrado. En el siguiente, estos decoradores se implementarán para otra aplicación, demostrando la adaptación y la reutilización de mecanismos.

La ejecución original sin ningún mecanismo lo encontramos en la imagen 56. Los procedimientos que se observan en la figura son los esperados. El cliente no conecta con el servidor. Simplemente ordena el vector de enteros que se genera aleatoriamente de manera local. Para utilizar el *proxy*, y por tanto el servicio del servidor, necesitaremos utilizar el decorador desarrollado a tal fin. En el ejemplo mostrado, el cliente simplemente invoca su propio *sort*(), un método local.

Por su parte, el servidor se lanza a ejecución y se queda esperando a que algún cliente le solicite conectarse, algo que a priori no sucederá.

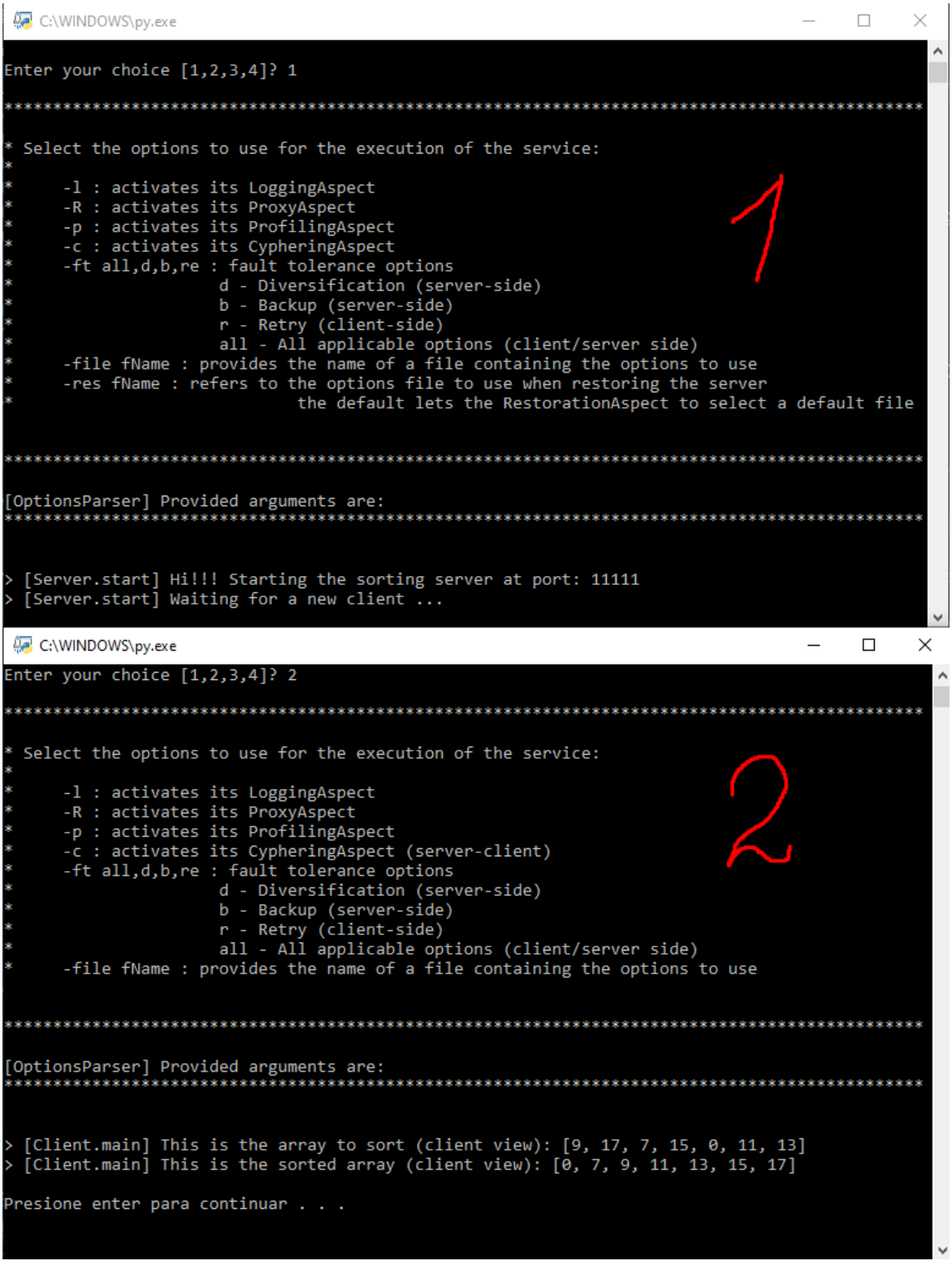


Imagen 56. Ejecuciones de servidor (1) y cliente (2).

El siguiente paso es arrancar solo el cliente y activar los decoradores *ProxyDecorator* y *RetrySortDecorator,* para ver el comportamiento del patrón *proxy* y observar cómo los reintentos toleran la caída de servicio. Cabe señalar que el decorador *ProxyDecorator*

podría aplicarse sin el mecanismo de reintento, pero en caso de caída del servidor el fallo no se podría tolerar. La salida obtenida se muestra en la imagen 57. En la imagen se observa que la ejecución activa los decoradores, entrando en la función *main*() del cliente. Al obtener el array para ordenar *ServerProxy* se intenta conectar al servidor. Para ello, cuenta con nueve reintentos, cada uno dura dos segundos más que el intento anterior. Al conectarse, el cliente continúa con su rutina. En caso de no poder haber establecido la conexión, se avisa al cliente del error en el servidor.

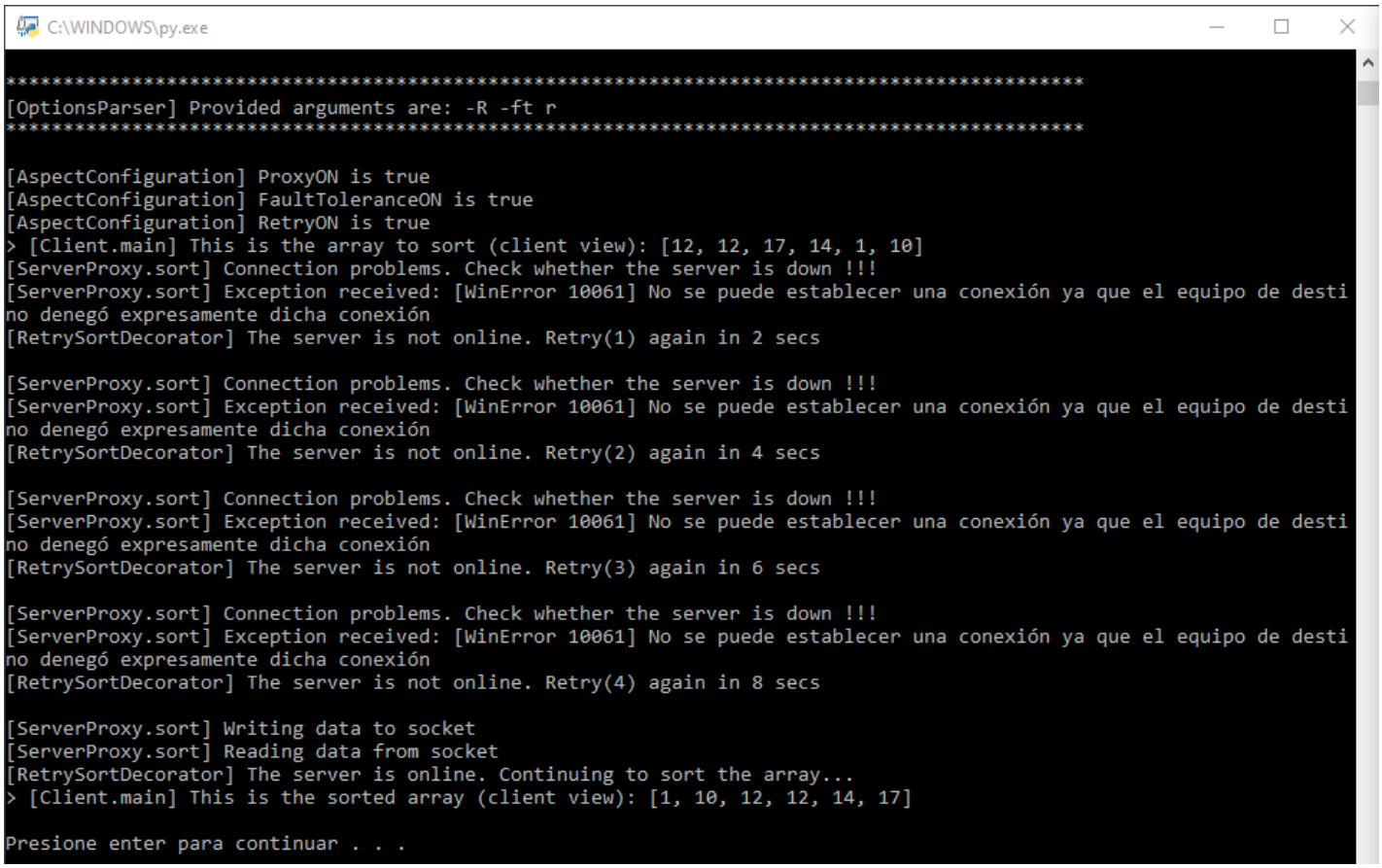


Imagen 57. La salida esperada de la ejecución solo del cliente aplicando los decoradores de *proxy* y reintentos.

Como podemos apreciar, no se observa los cambios en el comportamiento del servidor. El servidor hace la ordenación y vuelve a esperar las otras conexiones o a la orden de stop. Esto es una buena señal, porque los decoradores aplicados son para implementarse en la parte del cliente, entonces el servidor no debería ser afectado (imagen 58).

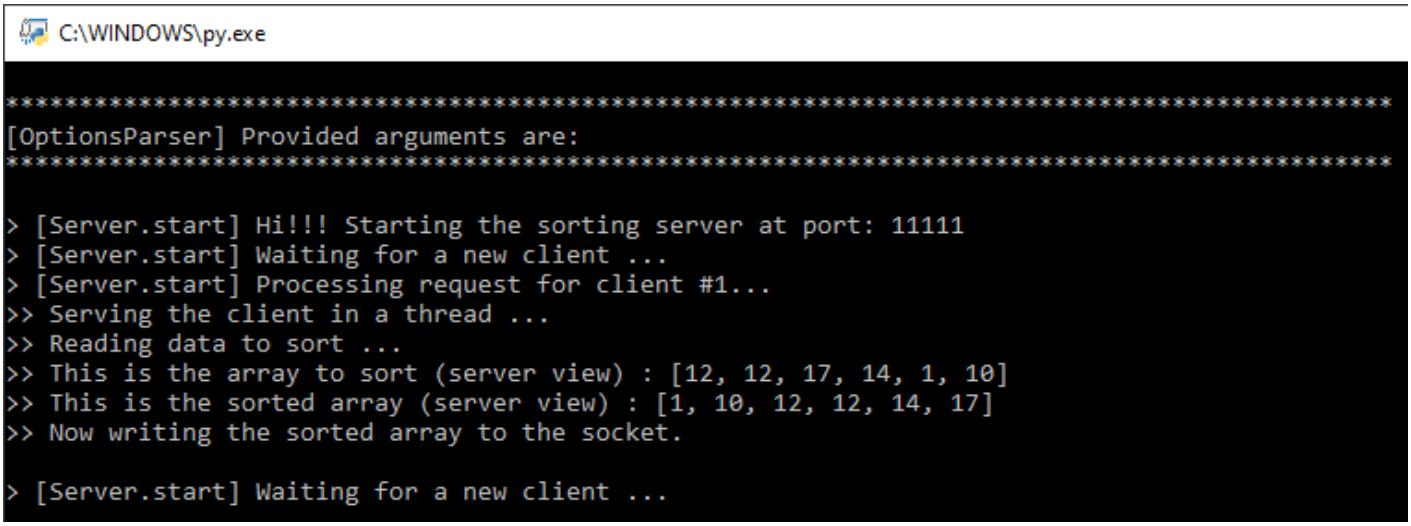


Imagen 58. El comportamiento del servidor.

El decorador de cifrado no está pensado para realizar funciones de imprimir para no dar información extra a los potenciales atacantes. Pero se puede observar la ejecución del decorador, sus valores y variables, que pasan por dentro en el modo de depuración de código. Esto está ilustrado en la imagen 59.

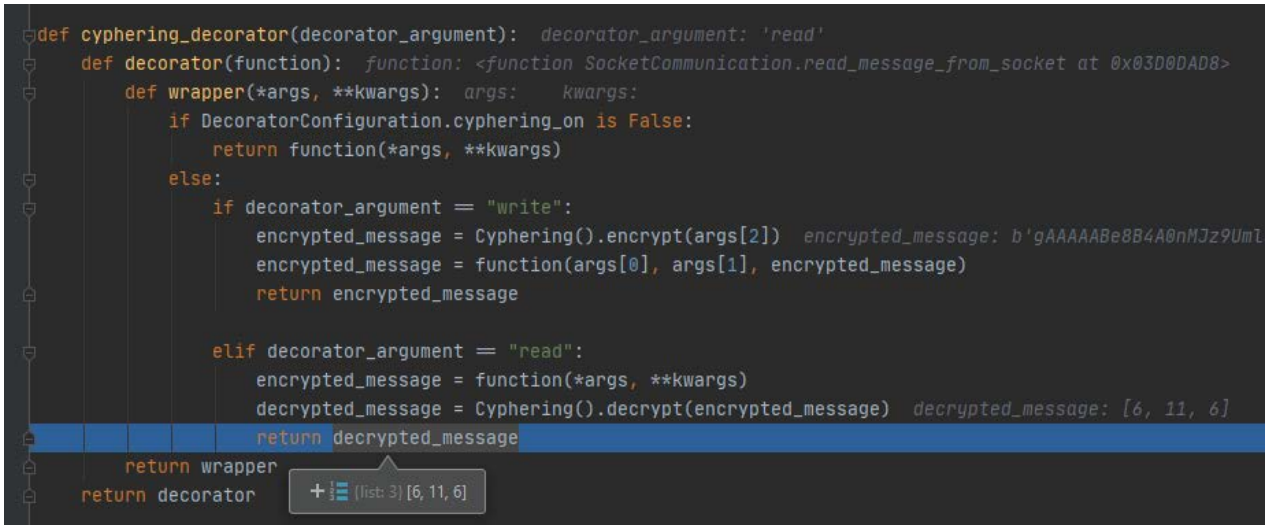


Imagen 59. Depuración de *cyphering_decorator*().

Hablando del servidor, el servidor caído no produce ningún tipo de actividades ni imprime la información por la consola porque está desactivo. El servidor reiniciado restaura el estado de sistema siguiendo a trabajar con el mismo cliente, ejecutando el resto de los decoradores activados con sus mecanismos (imagen 60). En este ejemplo se

observa que el servidor al reiniciarse sigue trabajando con el cliente #18 implementando los mecanismos de *backup*, cifrado y *N-Version*.

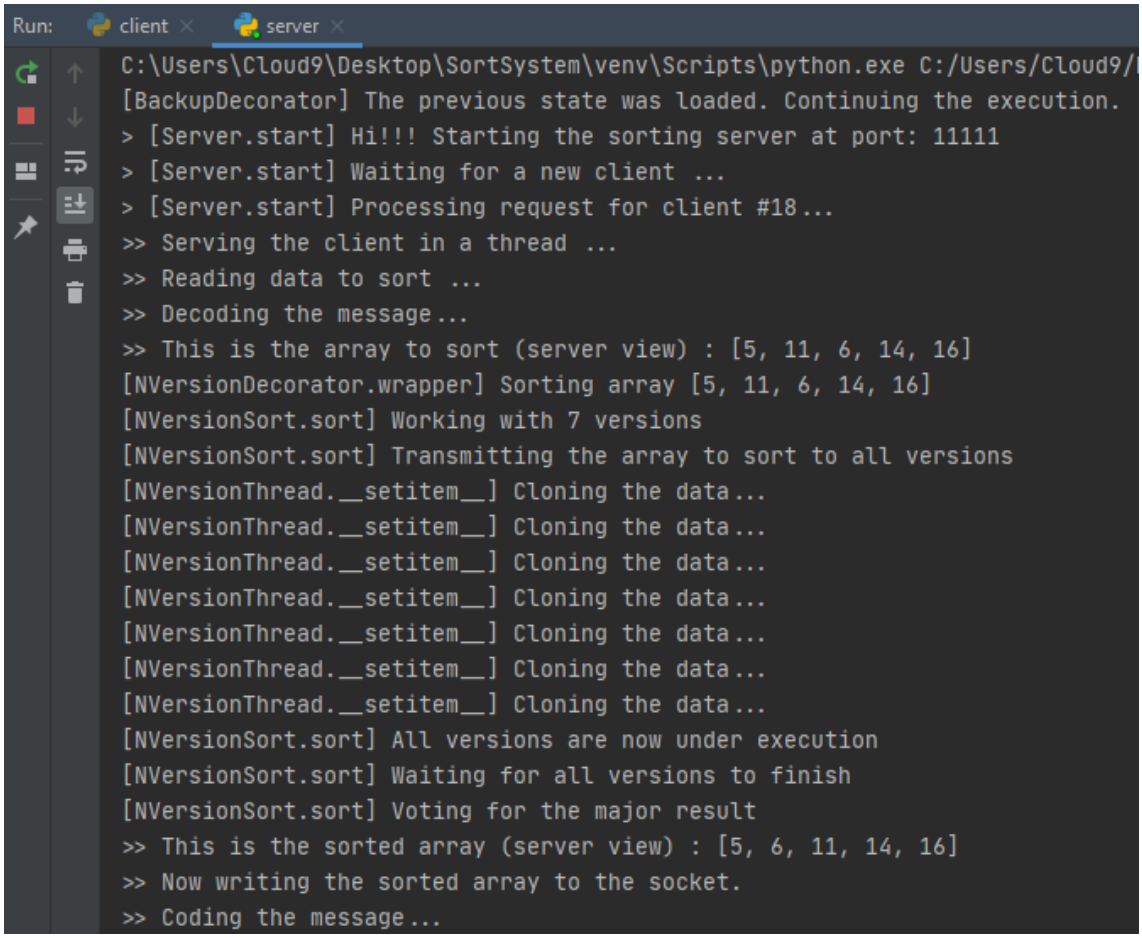


Imagen 60. Salida de servidor reiniciado.

Si se produce un error durante la ejecución, por ejemplo el error de cifrado de mensajes que está ilustrado en la imagen 61, el servidor indica si ha pasado algún problema, donde se ha ocurrido este problema y vuelve a escuchar a los clientes.

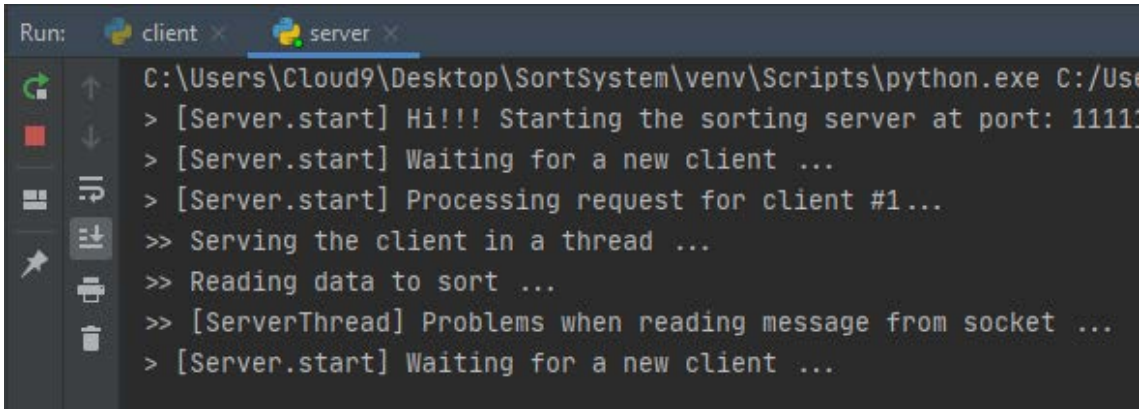


Imagen 61. Ejemplo de error de cifrado.

Si el problema no ha sido detectado, que puede pasar por los errores que generan las librerías usadas, el servidor mete la excepción ordinaria (relacionada con el problema) y para su ejecución.

El código inicial del cliente prácticamente no se modifica al incluir los dos decoradores. Todo lo que necesita hacer el desarrollador es anotar las funciones que quiere decorar con las etiquetas de decoradores necesarios. En las imagen 62 está ilustrada la versión del cliente sin decoración.

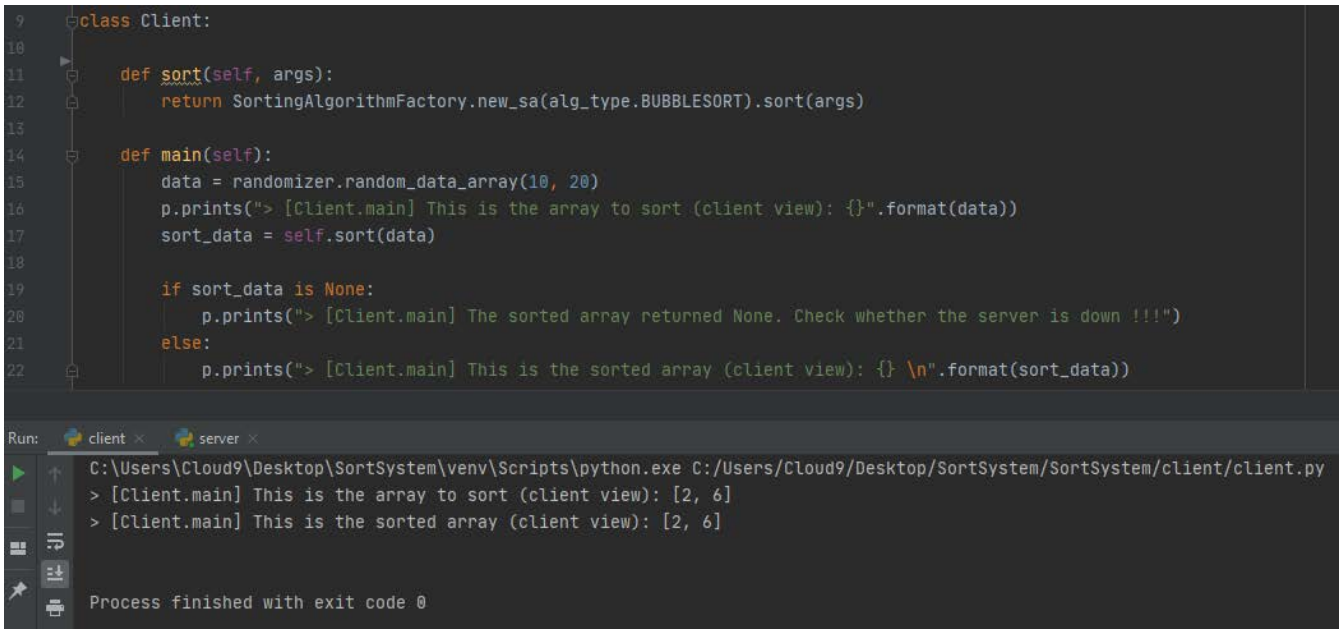


Imagen 62. Código inicial de cliente.

En este ejemplo el cliente no hace petición al servidor proxy, que hace gestiones de invocaciones remotas, y está ordenando los números localmente, es decir, el cliente ordena los números por sí mismo, que se observa en la salida de ejecución.

Para cambiar el comportamiento del cliente solo se necesita anotar la función necesaria (es decir, literalmente escribir 2 líneas de código más) con los decoradores de proxy y *N-Version* (imagen 63). Usando los dichos decoradores el usuario hace petición cual se gestiona en el servidor proxy, y ordenación local y remota pasan por el mecanismo de diversificación.

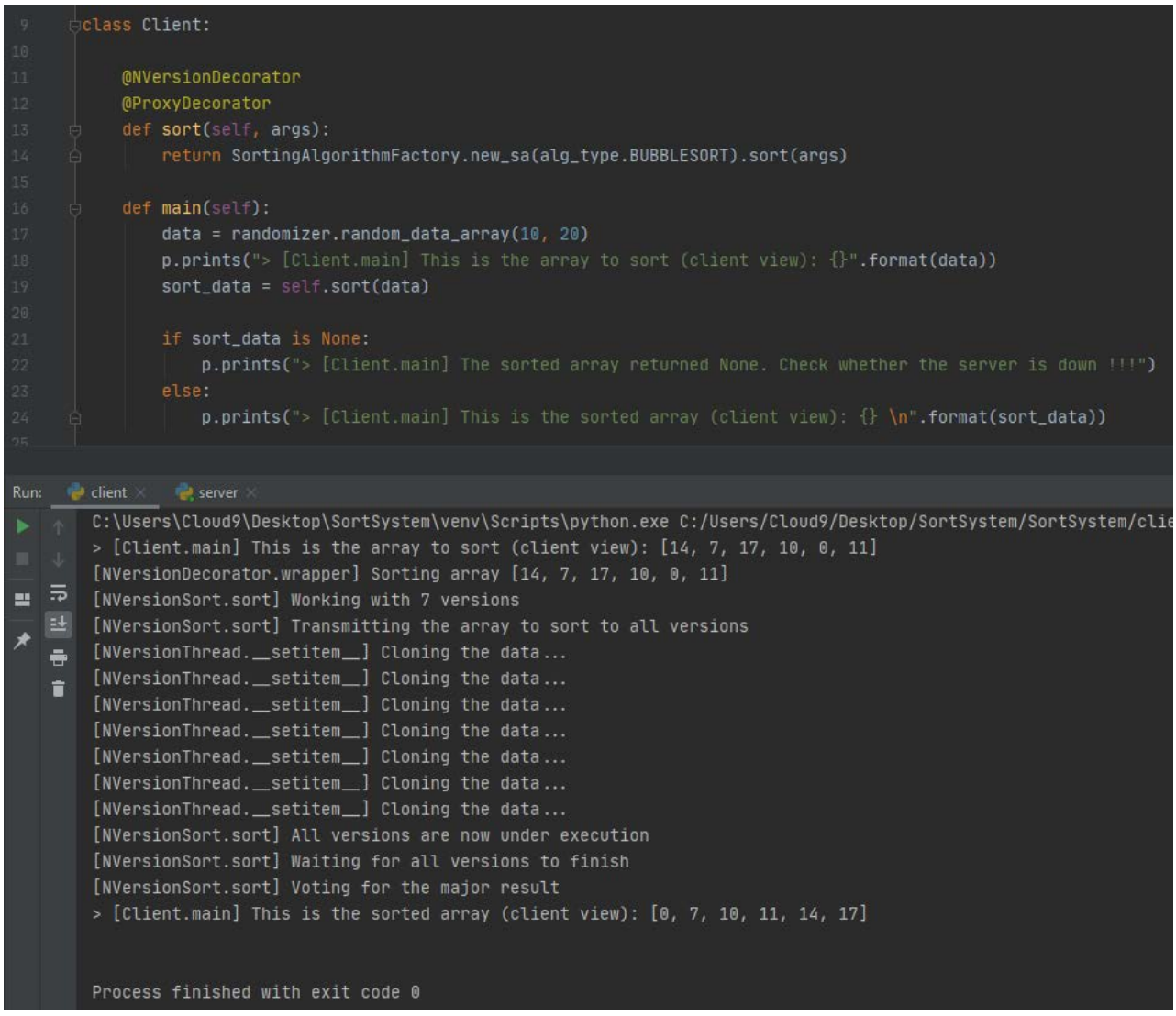


Imagen 63. Cliente decorado y su salida.

5.2. Reutilización de Mecanismos en Otros Ejemplos

Como se dijo anteriormente, los mecanismos implementados gracias a la orquestación y a la composición pueden ser empleados otra vez en otros sistemas. Los decoradores desarrollados en este trabajo son módulos satisfactorios para demostrar la reusabilidad y la adaptación de estos módulos encapsulados. En este párrafo se desarrolla otro sistema muy trivial de un servidor y un cliente que juegan a *Ping Pong.* Es decir, mandan y reciben 5 mensajes, aplicando los decoradores de reintentos y cifrado.

5.2.1. Requisitos de Reutilización

Como se ha podido apreciar anteriormente, los módulos de los mecanismos no pueden ser ideales y tienen que ajustarse algo cada vez que pasan a usarse en otra aplicación. Los decoradores nuevos se pueden conseguir copiando los existentes, porque así se puede conseguir una similitud de sistemas más adecuada que desarrollarlo desde *scratch.* Los requisitos de reutilización en caso de este proyecto han sido:

- Borrar las funciones de otras clases de módulos del sistema original, porque la aplicación nueva no disponía de librerías de cifrado, ni el servidor *proxy*, ni comunicación de sockets, ni etc.
- Reproducir los métodos de librerías en los decoradores de manera rápida y compacta.
- Borrar todos los *prints* que había porque no tenían relación alguna con el sistema nuevo.
- Borrar los *ifs* que eran para cambiar las configuraciones de los decoradores.

También se ha encontrado un obstáculo de la librería que se ha usado para hacer la codificación y la comunicación de sockets. Se ha empleado la construcción *if-else* para poder ser capaz de manejar errores que puedan ocurrir (imagen 64). Como el cifrado de *fernet* token codifica el valor según su propio algoritmo, este valor no puede ser

codificado o decodificado por las funciones de sockets *encode*() y *decode*(). Esto causaría errores. No pasa nada si el decorador está activado, pero si no, se necesita cifrar y descifrar los *strings* usando los métodos dichos. Para solucionar este problema optamos por comprobar si el mensaje es un *string* decodificado o un cifrado dentro de la función original. Esto afectó negativamente a la confiabilidad de la aplicación por proporcionar menos reusabilidad y adaptación de los módulos.

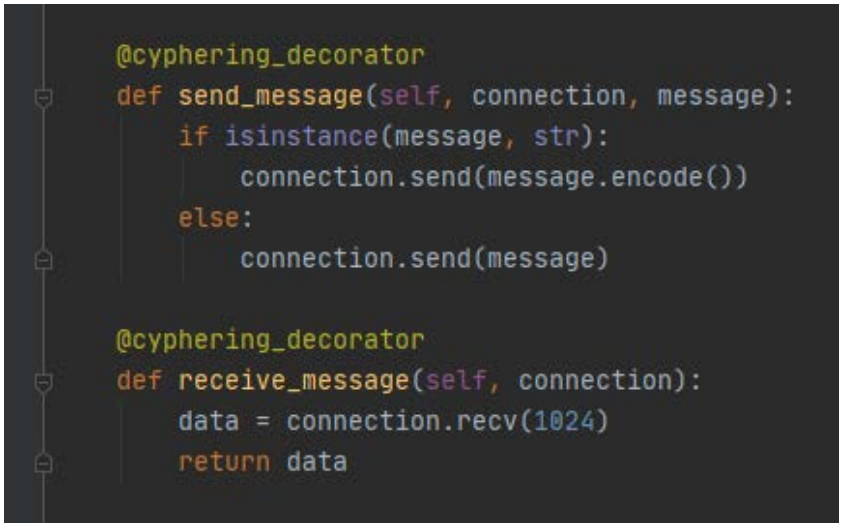


Imagen 64. Construcción *if-else* dentro de la función propia del servidor. El cliente para comunicarse usa los mismos métodos, que están en su clase.

5.2.2. Ejemplo de Ping Pong

El ejemplo de *Ping Pong* es el sistema de cliente servidor parcialmente descrito en apartados anteriores. Es la aplicación que comparte los mensajes y sirve para testear la reutilización y adaptabilidad de los mecanismos de reintentos y cifrado.

- **Servidor.**

El servidor que se observa en la imagen 65, dispone de tres métodos y un constructor. En el constructor se crea y almacena el socket y la ejecución se empieza por el método *start*(). En esta función se asigna el número de “disparos” que hacen los “jugadores”. Más tarde el servidor se pone a escuchar a los clientes. Se entra en el bucle *while* que

controla el número de *shots* y empieza a jugar enviando y recibiendo mensajes usando los métodos *send_message*() y *receive_message*(). Estas funciones están decoradas por *cyphering_decorator*()

```
class PingPongServer:
    def __init__(self):
        self.port = 4444
        self.host = '127.0.0.1'
        self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server.bind((self.host, self.port))
        self.server.listen(10)

    def start(self):
        print("\nLet's play some Ping Pong with the client!")
        shots = 1
        conn, addr = self.server.accept()
        while shots != 6:
            data = self.receive_message(conn)
            print("\nRonda #{}".format(str(shots)))
            print("Client attacks:", data.decode())
            print("Server has an answer: PONG!")
            self.send_message(conn, "PONG!")
            if shots == 5:
                print("\nThe game is over!")
                break
            shots = shots + 1

    @cyphering_decorator
    def send_message(self, connection, message):
        if isinstance(message, str):
            connection.send(message.encode())
        else:
            connection.send(message)

    @cyphering_decorator
    def receive_message(self, connection):
        data = connection.recv(1024)
        return data
```

Imagen 65. El servidor de *Ping Pong*.

- **Cliente.**

Los métodos de comunicación del cliente son iguales que las funciones del servidor. La única diferencia está en el constructor y la clase *start*() (imagen 65). El usuario tiene la conexión estándar de cliente de sockets.

- **Mecanismo de reintentos.**

El decorador-clase prototipo es más compacto que el módulo original que se observa en la imagen 66, pero tienen la misma lógica de intentos usando la variable *retry* y un bucle *while*.

```python
import time


class RetrySortDecorator:
    def __init__(self, function):
        self.function = function
        self.ret_data = None

    def __call__(self, *args):

        retries = 0

        while retries < 9:
            self.ret_data = self.function(*args)

            if self.ret_data is not None:
                print("\nThe server was sucessfully connected!")
                break
            else:
                print("\nRetry({}) again in {} secs.".format(retries + 1, 2 * (retries + 1)))
                retries = retries + 1
                time.sleep(2 * (retries + 1))
```

Imagen 66. El decorador-prototipo de reintentos.

El resultado de la ejecución del decorador es la imagen 67. El comportamiento del decorador es el esperado cuando, al llamar a la función decorada el *RetrySortDecorator,* entra en un bucle e intenta a ejecutar la función. Si este método no devuelve nada, el decorador vuelve a conectarse una vez más esperando una pausa de reconexión.

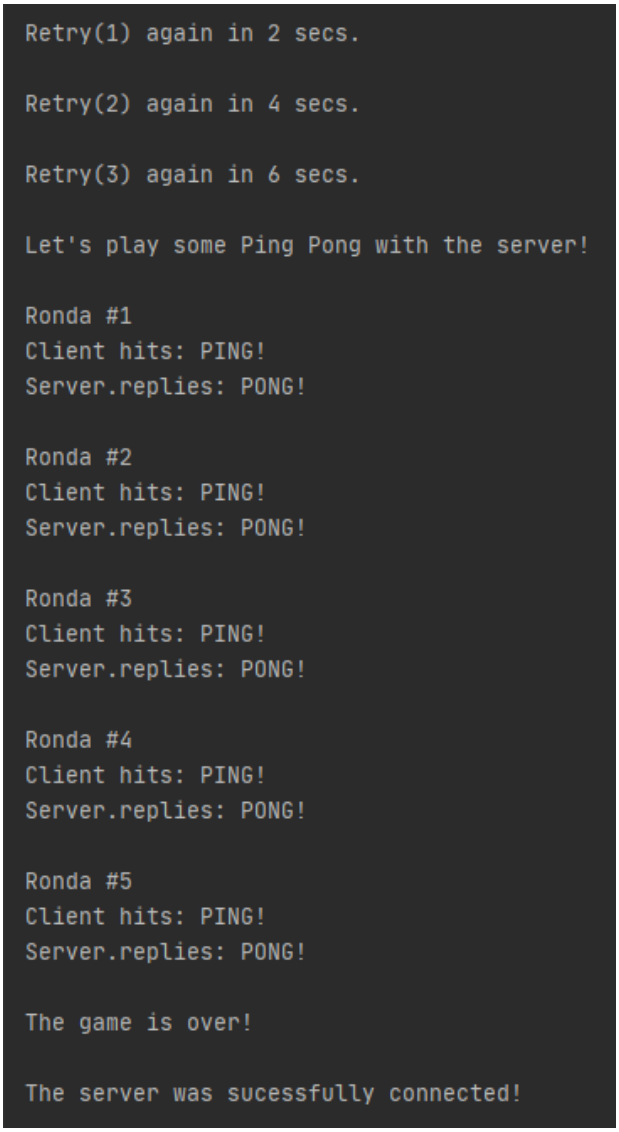
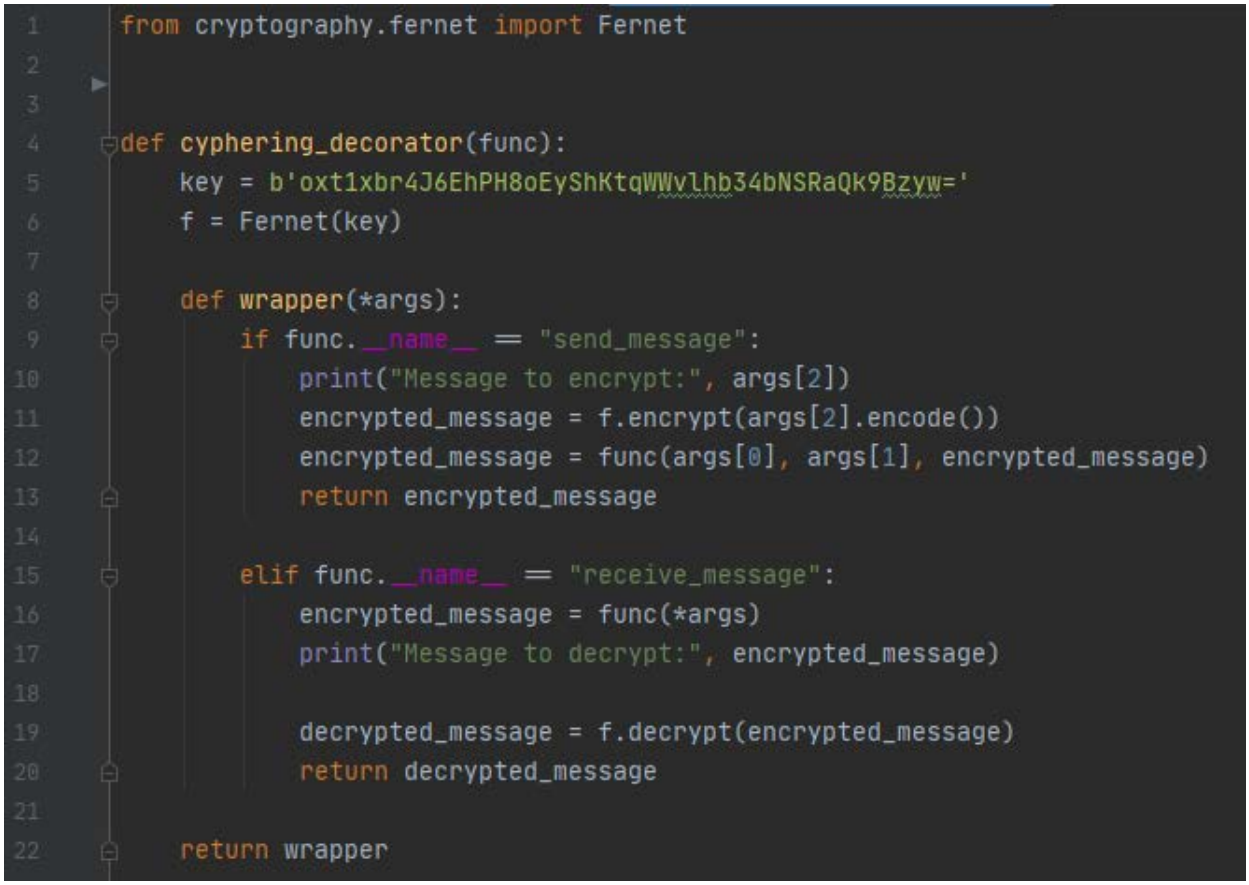


Imagen 67. Salida de la ejecución de prototipo *RetrySortDecorator* para el *Ping Pong*.

- **Cifrado de las comunicaciones.**

El decorador-función *CypheringDecorator* es el otro decorador que se ha decidido usar para demostrar la reusabilidad de los módulos y mecanismos. A pesar de que se ha encontrado un obstáculo durante el desarrollo del sistema, su comportamiento ha sido satisfactorio. El decorador manejaba y codificaba los datos correctamente sin causar errores (imagen 68).



```python
from cryptography.fernet import Fernet


def cyphering_decorator(func):
    key = b'oxt1xbr4J6EhPH8oEyShKtqWWvlhb34bNSRaQk9Bzyw='
    f = Fernet(key)

    def wrapper(*args):
        if func.__name__ == "send_message":
            print("Message to encrypt:", args[2])
            encrypted_message = f.encrypt(args[2].encode())
            encrypted_message = func(args[0], args[1], encrypted_message)
            return encrypted_message

        elif func.__name__ == "receive_message":
            encrypted_message = func(*args)
            print("Message to decrypt:", encrypted_message)

            decrypted_message = f.decrypt(encrypted_message)
            return decrypted_message

    return wrapper
```

Imagen 68. El decorador prototipo *cyphering_decorator().*

Este decorador es de tipo función, y lo primero que hace al instanciarse es preparar la llave para un objeto “Fernet” y crear este objeto. Más adelante, al llamar a la función, comprobamos el nombre de la función y ejecutamos la lógica de la librería “Cyphering”. La salida relacionada se puede observar en la imagen 69.

```
Let's play some Ping Pong with the server!
Message to encrypt: PING!

Ronda #1
Client hits: PING!
Message to decrypt: b'gAAAAABe78d81EV5ybidlmHM7TfNMnRRipAlu45cbPG96aewpCEK0f7juwu5yx0ao4clUiHL5k8lnLQiyvht88DcS_iYYqU7Bw=='
Server.replies: PONG!
Message to encrypt: PING!

Ronda #2
Client hits: PING!
Message to decrypt: b'gAAAAABe78d8RjFa4t8JDcfp5Feh0wKlMhIr8HJQgwG4fkVuGOVeuxeak3bhhcAPlbWsBaQphsPaptMVMHoLoVbfGjB3TeFwzQ=='
Server.replies: PONG!
Message to encrypt: PING!

Ronda #3
Client hits: PING!
Message to decrypt: b'gAAAAABe78d8Mtya2086EtB47l48x9gfV8PBCZJmzv6ZlKmAAAAx0ZXitfzf0N0KevVt7seS611TIwdhnotWjh00n2Hqwn0h_Q=='
Server.replies: PONG!
Message to encrypt: PING!

Ronda #4
Client hits: PING!
Message to decrypt: b'gAAAAABe78d8CgNxGPXtUi5qj5ZRidkcc2q11SrGq9wb4KlL0hCAE8_qlhL7F7kBo8KRkGbFMtOFjFgb5JEpw6twChYG5NARfA=='
Server.replies: PONG!
Message to encrypt: PING!

Ronda #5
Client hits: PING!
Message to decrypt: b'gAAAAABe78d86HIJLlyc8xFpBrcOfz7XDvn7DFUhM8vRw4u6JsWlCvmJ6OyTsSlH4g_V6pcW1W02FxdNhIjLzh5xgbK-N3t1Qg=='
Server.replies: PONG!

The game is over!

The server was sucessfully connected!

Process finished with exit code 0
```

Imagen 69. Salida del decorador de cifrado.

CAPÍTULO 6: CONCLUSIONES

En primer lugar, hay que decir que se ha logrado satisfactoriamente el objetivo de analizar y reproducir a través de un ejemplo la separación de mecanismos funcionales y no funcionales en programas Python con ayuda de los decoradores.

Los ejemplos de uso mostrados en el capítulo 3 funcionan correctamente, son descriptivos y detallados y explican los diferentes usos y formas de implementar los decoradores en el sistema.

Luego, en el capítulo 4, se alcanza el objetivo principal y el objetivo adicional de desarrollar un sistema de tipo cliente-servidor de ordenación de una lista de números que se aplicarán los decoradores con los mecanismos de tolerancia a fallos por desarrollar un sistema implementando los decoradores que encapsulan los siguientes mecanismos:

- *Logging*
- *Proxy*
- Reintentos
- *Backup*
- *N-Version programming*
- Cifrado simétrico

Estos mecanismos que están encapsulados en los decoradores del proyecto y que están inyectados en el código principal tienen el comportamiento esperado, y no provocan errores o fallos entre sí al ejecutarse con el sistema.

Más adelante, en el capítulo 5, se ha cumplido el objetivo adicional de desarrollar otro ejemplo de solución cliente-servidor en la cual se han reutilizado algunos de los decoradores ya existentes para demostrar y justificar la flexibilidad y reutilización

posible de dichos mecanismos. Supuestamente, los decoradores debían de ser ajustados al sistema nuevo por requisitos y módulos que tiene este sistema, pero su código base no ha sido alterado en absoluto, como ha sido investigado en el capítulo 5, demostrando que nos hallamos ante una tecnología de alta reutilización.

Para finalizar, desde un punto de vista personal, este trabajo ha significado una puerta al mundo de la programación orientada a aspectos, el cual es de mi interés. En mi opinión, ese ámbito no solo es vigente, sino que experimentará un gran desarrollo en los próximos años. Además, este proyecto me ha resultado muy práctico a la hora de aumentar mis conocimientos en varios aspectos de la informática. A pesar de haber aprendido el lenguaje *Python* de manera más profunda y profesional, este proyecto me ha ayudado a comprender la gran utilidad de conceptos esenciales, como la separación de mecanismos, la programación orientada a aspectos, o el *wrapping* y aplicarlos a los ejemplos prácticos de dos soluciones distintas de tipo cliente-servidor.

Aunque yo no haya desarrollado el mecanismo de decoradores, esto no significa que no se haya tenido que realizar un profundo estudio de este para poder entenderlo a la perfección, para poder adaptarlo a nuestras necesidades, y de esta manera, para poder aplicar los decoradores como soporte a los mecanismos de tolerancia a fallos que se deseaban desarrollar y reutilizar, en nuestro caso, sobre dos sistemas con comportamientos diferentes.

BIBLIOGRAFÍA

1. Bartlett, J. (s. f.). *Introduction to metaprogramming*. Recuperado 11 de junio de 2020, de https://www.ibm.com/developerworks/library/l-metaprog1/

2. Britt, J., & Neurogami. (s. f.). *Documentación Oficial de Ruby, Class: BasicObject* (Ruby 1.9.3). Recuperado 11 de junio de 2020, de https://ruby-doc.org/core-1.9.3/BasicObject.html#method-i-instance_eval

3. Chiba, S. (2006). *Program Transformation with Reflection and Aspect-Oriented Programming*. Generative and Transformational Techniques in Software Engineering, 65-94. https://doi.org/10.1007/11877028_3

4. Christensson, P. (2019, August 6). *Wrapper Definition*. Recuperado 12 de junio de 2020, de https://techterms.com/definition/wrapper

5. Eclipse Foundation, Inc. (s. f.). *The AspectJ Project*. Recuperado 12 de junio de 2020, de https://www.eclipse.org/aspectj/

6. Fabre, J. C., & Perennou, T. (1998). *A metaobject architecture for fault-tolerant distributed systems: the FRIENDS approach*. IEEE Transactions on Computers, 47(1), 78-95. https://doi.org/10.1109/12.656088

7. Javassist by jboss-javassist. (s. f.). Recuperado de https://www.javassist.org/

8. Knoernschild, K. 8. (2001*). Java Design: Objects, UML, and Process* (1ª ed.). London, Reino Unido: Pearson Education.

9. Lilis, Y., & Savidis, A. (2020). *A Survey of Metaprogramming Languages. ACM Computing Surveys*, 52(6), 1-39. https://doi.org/10.1145/3354584

10. Lutz, M. (2013). *Learning Python, 5th Edition* (5.ª ed.). Sebastopol, EE.UU.: O'Reilly Media.

11. M.-O. Killijian, J.-C. Fabre, J.-C. Ruiz-Garcia, and S. Chiba. 1998. *A Metaobject Protocol for Fault-Tolerant CORBA Applications. In Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems (SRDS '98)*. IEEE Computer Society, USA, 127.
12. Michiaki Tatsubori, Shigeru Chiba, Marc-Oliver Killijian, Kozo Itano, *OpenJava: A Class-based Macro System for Java, Reflection and Software Engineering*, pp. 119--135, Springer-Verlag, 2000
13. Microsoft Patterns & Practices Team. (2009). *Microsoft® Application Architecture Guide*. Redmond, EE.UU.: Microsoft Press.
14. Microsoft. (2017, 23 de junio). *Retry pattern - Cloud Design Patterns*. Recuperado 17 de junio de 2020, de https://docs.microsoft.com/en-us/azure/architecture/patterns/retry
15. Oracle. (s. f.). *Introduction to CORBA Request-Level Interceptors*. Recuperado 13 de junio de 2020, de https://docs.oracle.com/cd/E13203_01/tuxedo/tux100/rli/rliover.html#:%7E:text=A%20request%2Dlevel%20interceptor%20is,components%20of%20a%20CORBA%20application
16. Pilgrim, M. (2010). *Dive into Python 3*. Nueva York, EE.UU.: Apress.
17. Python Software Foundation. (s. f.). *Python Documentation*. Recuperado 16 de junio de 2020, de https://docs.python.org/3/reference/datamodel.html#objects-values-and-types
18. Ramalho, L. (2015). *Fluent Python: Clear, Concise, and Effective Programming* (1.ª ed.). Sebastopol, EE.UU.: O'Reilly Media.
19. Ramirez, J. D., Corrales, J. C. (2015). *Automatic Orchestration of Converged Services on JSLEE Environment*. Revista Tecnura, 19(46), 15. https://doi.org/10.14483/udistrital.jour.tecnura.2015.4.a01

20. Real Python. (s.f.). *Object-Oriented Programming (OOP) in Python 3*. Recuperado 9 de junio de 2020, de https://realpython.com/python3-object-oriented-programming/

21. Refactoring.Guru. (2020, 19 abril). *Proxy*. Recuperado 17 de junio de 2020, de https://refactoring.guru/design-patterns/proxy#:%7E:text=Proxy%20is%20a%20structural%20design,through%20to%20the%20original%20object

22. Roper, W. (2020, 3 de marzo). *Python Remains Most Popular Programming Language*. Recuperado 5 de junio de 2020, de https://www.statista.com/chart/21017/most-popular-programming-languages/

23. Sanatan, M. (2020, enero). *Functional Programming in Python*. Recuperado 8 de junio de 2020, de https://stackabuse.com/functional-programming-in-python/

24. Software Developer's Journal, & Jakacki, G. (2013, 28 agosto). *OpenC++ - A C++ Metacompiler and Introspection Library*. Recuperado 11 de junio de 2020, de https://www.codeproject.com/Articles/13896/OpenC-A-C-Metacompiler-and-Introspection-Library

25. van Rossum, G. (s. f.). *What is Python? Executive Summary*. Recuperado 15 de junio de 2020, de https://www.python.org/doc/essays/blurb/ 3.

26. JC Ruiz, MO Killijian, JC Fabre, P Thvenod-Fosse, *Reflective fault-tolerant systems: From experience to challenges*, IEEE Transactions on Computers 52 (2), 237-254

27. Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J. M.; Irwin, J. (1997). *Aspect-oriented programming* (PDF). ECOOP'97. Proceedings of the 11th European Conference on Object-Oriented Programming. LNCS. 1241. pp. 220–242.