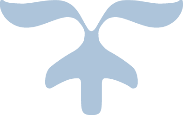


EJERCICIO GUIADO 3

Eduardo Faro González - 100475239 Mario Salvador Camacho - 100475293 Ignacio Tordable Mestres - 100475359



Contenido

[FUNCIÓN 1 2](#_heading=)

[FUNCIÓN 2 2](#_heading=)

[FUNCIÓN 3 3](#_heading=)

[**Singleton 3**](#_heading=h.lbcx1i2vq3cp)

[Pylint 3](#_heading=h.ma7vgkej2chm)

[ARTÍCULO 1 4](#_heading=h.gjdgxs)

[ARTÍCULO 2 5](#_heading=h.30j0zll)

[ARTÍCULO 3 6](#_heading=h.1fob9te)

[BIBLIOGRAFÍA 6](#_heading=h.3znysh7)

## FUNCIÓN 1

El nuevo código de la función 1 (room\_reservation) refactorizada es sencillo, primero creamos el objeto de HotelReservation con los datos de entrada, después creamos un objeto de la nueva clase ReservationJsonStore y llamamos al método de esa clase add\_item, el cual está dividido en tres pasos. Comienza llamando a self.load\_list\_from\_file, que genera la lista del JSON de reservas y la actualiza, a continuación añade a la lista el parámetro pasado como atributo y por último actualiza la lista, todo desde métodos de la súper clase JsonStore. Estos métodos han sido sacados del código anterior con “Extract method”.

Para poder validar cada uno de los datos que nos pasará el cliente, además de el localizador y la room key que generará el código, hemos creado un package. Esto nos permite simplificar la validación de estos datos. Inicialmente, hay un fichero padre (attribute) que servirá para la validación del formato de los distintos atributos. Este incluye un método validate, que se encarga de comprobar que los datos cumplen con el patrón especificado, si no saltará el error correspondiente.

Posteriormente, tendremos una clase hija por cada uno de los atributos: arrival date, credit card, id card, localizer, num days, name surname, phone number, room key y room type. En cada una de ellas se especifica el formato de los datos que deben cumplir, el mensaje de error específico y se ejecuta el padre. No obstante, para las funciones id card y credit card, ampliaremos su funcionalidad mediante un método validate, que en primer lugar ejecuta la validación del padre y posteriormente en el caso de credit card, comprueba que se cumple el algoritmo de luhn, y en el de id card que se cumple el algoritmo de la letra del NIF.

## FUNCIÓN 2

Inicialmente, hemos creado un package llamado parser para poder leer ficheros de diccionarios de tipo json. De este fichero leído con el parser podemos extraer my\_id\_card y my\_localizer. El fichero arrival\_json\_parser le proporciona los datos necesarios de arrival( id card y localizer), además del mensaje de error, al padre. El padre (json\_parser), extrae id cardy y localizer del fichero proporcionado y comprueba que sean iguales y las valida, si no imprime el mensaje de error.

Una vez sacados el localizador y el IdCard podemos llamar a la clase HotelStay. En el innit de esta clase hemos añadido una llamada al classmethod de HotelReservation que se encarga de crear una reserva de alguien que ha llegado al hotel y de comprobar su existencia en los archivos JSON. Este método recibe como entrada el localizador del objeto HotelStay y verifica si la reserva con ese localizador existe, creando un objeto de ReservationJsonStore y usando los métodos necesarios de la súper clase, a continuación crea un objeto de la clase HotelReservation, compara sus valores y lo devuelve.

Una vez comprobados todos los posibles problemas en el innit de HotelStay, se crea el objeto dentro de la función guest\_arrival llamado my\_checkin, con el localizador y IdCard sacados previamente. Para insertar este objeto en la lista JSON de checkins hace falta crear un nuevo objeto llamado checkin\_store de la clase StayJsonStore. Una vez creado, se aplica el método add\_item para añadir my\_checkin a la lista de arrivals.

Por último la función devuelve el room\_key del objeto de HotelStay.

## FUNCIÓN 3

Con la refactorización hemos conseguido reducir la función de guest\_checkout a una sola llamada al classmethod get\_stay\_from\_room\_key, la cual devuelve un booleano que será devuelto por la función.

Este método comienza ejecutando el atributo RoomKey para comprobar si la llave es válida. A continuación crea un objeto de la clase StayJsonStore y saca la lista de las estancias del hotel para más tarde comprobar si la llave de la habitación coincide con alguna de la lista.

Continuamos con la creación de my\_checkout, objeto de la clase HotelDeparture inmediatamente después creamos un objeto de CheckOutJsonStore para poder hacer un add\_item de my\_checkout a la lista JSON de checkouts. Por último si el método se ha ejecutado sin ninguna excepción devuelve True.

## Singleton

El patrón Singleton se utiliza en situaciones en las que debe existir exactamente una única instancia de una clase en toda la aplicación, y en las que el acceso centralizado a esa instancia es beneficioso para administrar recursos compartidos, promover la coherencia y mejorar el rendimiento.

Este patrón es muy restrictivo y hay que utilizarlo con criterio. Por ello, se ha implementado tanto en hotelmanager como en las clases hijas de acceso a los almacenes. También, se ha incluido un fichero para comprobar que se realiza bien el patrón (test\_singleton\_tests).

## Pylint

En cuanto al pylint, hemos conseguido eliminar todos los avisos que saltaban. Sin embargo, hubo tres avisos en los que hemos tenido que aplicar #pylint: disable para eliminarlos, ya que el código no se podía modificar, y las reglas de pylint tampoco. Lo hemos utilizando usando estos tres disable:

* # pylint: disable=invalid-name
* # pylint: disable=too-few-public-methods
* # pylint: disable=too-many-arguments

## ARTÍCULO 1

URL: <https://www.sciencedirect.com/science/article/pii/S0164121220300881>

El artículo[[1]](#footnote-0) con título “*Code smells and refactoring: A tertiary systematic review of challenges and observations*” realiza un análisis de la relación que tienen los code smells y el refactoring. A lo largo del ensayo, se explican las razones del ¿por qué los code smells afectan de manera negativa a la mantenibilidad y la comprensión del software? Los conceptos que explican son los siguientes:

1.- Definición de code smells

2.- Enfoques de detección de olores de código

3.- Herramientas de detección de olores de código

4.- Refactorización

Finalmente, demuestran que los olores de código y la refactorización tienen una fuerte relación con atributos de calidad, es decir, con comprensibilidad, mantenibilidad, complejidad, funcionalidad y reutilización. Además de que lanzan 13 preguntas abiertas para orientar a futuros trabajos de investigación, indicando que hay necesidad de mejores herramientas de detección y técnicas de refactoring más eficientes.

Consideramos que este artículo es relevante porque hace mucha labor en explicar cómo el refactoring puede ser utilizado estratégicamente para mejorar la calidad del software al abordar los olores de código. Además, hace un llamamiento a la comunidad para advertir de lo necesario que es encontrar nuevas prácticas y herramientas que sean mejores para mantener y asegurar la evolución del software en la industria.

## ARTÍCULO 2

URL - <https://www.sciencedirect.com/science/article/pii/S0950584919302605>

Este artículo[[2]](#footnote-1) de la revista “*Information and Software Technolog*y” presenta un estudio que compara la identificación de code smells de manera colaborativa e individual por parte de desarrolladores novatos y profesionales. El estudio demuestra cómo la identificación colaborativa puede mejorar la precisión y la exhaustividad (recall) en la detección de estos “olores”. Para probarlo, se realizó un experimento con 34 desarrolladores, tanto novatos como profesionales, quienes tenían que identificar olores de código en sistemas con los que no estaban familiarizados, usando métodos colaborativos e individuales.

Los resultados demostraron que la identificación colaborativa era mejor que la individual, siendo 27% mejor en precisión y 36% mejor en exhaustividad. En la conclusión, el artículo sugiere que las empresas deberían implementar la identificación colaborativa de code smells como una práctica estándar.

Este estudio es relevante ya que proporciona evidencia empírica (mediante experimentación y observación) que apoya el refactoring colaborativo como una estrategia eficaz para mejorar la calidad del código. Y dado que el refactoring es fundamental para mejorar la calidad del software, comprender cómo la colaboración puede mejorar este proceso ayuda a las organizaciones a optimizar sus esfuerzos de mantenimiento de éste.

## ARTÍCULO 3

URL: <https://www.sciencedirect.com/science/article/pii/S0164121219302031>

El artículo[[3]](#footnote-2) investiga las técnicas y herramientas para el refactoring y el seguimiento de clones de código (code clones). Explica, que los clones de código son porciones idénticas (o casi) en la base de código de un programa, los cuales tienen impactos tanto buenos como malos en el mantenimiento del software. El documento propone gestionar estos clones a través del refactoring y el seguimiento (tracking). Para ello, presenta una encuesta sobre el estado de las técnicas en esta área. El estudio categoriza los clones de código en cuatro tipos basados en su similitud:

1.- Clones Tipo 1: los fragmentos de código son idénticos, sin tener en cuenta sus comentarios y sangrías.

2.- Clones Tipo 2: Se crean a partir de clones de Tipo 1 debido al cambio de nombre de los identificadores y al cambio de tipos de datos.

3.- Clones Tipo 3: se crean a partir de clones de Tipo 1 y Tipo 2 debido a la adición, eliminación o modificación de líneas de código fuente.

4.- Clones Tipo 4: Si dos o más fragmentos de código realizan la misma tarea, pero se implementan de diferentes maneras, son de Tipo 4.

Después, se revisan diversas técnicas y herramientas para el refactoring de clones, que incluyen métodos automáticos y semi-automáticos. Finalmente, se examina cómo el seguimiento de clones puede ayudar a mantener la coherencia del código durante su evolución. Consideramos que este estudio es relevante, ya que destaca la necesidad de nuevas herramientas más efectivas. Además, de que aborda cómo el refactoring y el seguimiento de clones pueden utilizarse para mejorar la calidad del software.

## BIBLIOGRAFÍA

Lacerda, G., Petrillo, F., Pimienta, M., & Guéhéneuc, Y. G. (2020). Code smells and refactoring: A tertiary systematic review of challenges and observations. *The Journal of Systems and Software*, 0-36.

Mondal, M., K. Roy, C., & Kevin A, S. (2019). A survey on clone refactoring and tracking. *The Journal of Systems and Software*.

Oliveira, R., de Mello, R., Fernandes, E., Garcia, A., & Lucena, C. (2020). Collaborative or individual identification of code smells? On the effectiveness of novice and professional developers. *Information and Software Technology*.

1. La cita se encuentra en la bibliografía [↑](#footnote-ref-0)
2. La cita se encuentra en la bibliografía [↑](#footnote-ref-1)
3. La cita se encuentra en la bibliografía [↑](#footnote-ref-2)