Información general del proyecto: para qué sirve, cuál es la estructura general del diseño, qué grandes retos de diseño enfrenta (i.e. ¿qué es lo difícil?). Deben incluir la URL para consultar el proyecto.

El proyecto consultado es Hibernate: https://github.com/hibernate/hibernate-search.git

¿Para qué sirve?
El proyecto "Hibernate Search" es una biblioteca de búsqueda de texto completo que se integra con Hibernate, un framework (ORM) para Java. Hibernate Search ofrece funcionalidades avanzadas de búsqueda y filtrado de texto completo para aplicaciones Java que utilizan Hibernate como capa de persistencia.

¿Cuál es la estructura general del diseño?
El diseño del framework se basa en capas de abstracción. Es decir, dentro del framework hay varias librerías que se encargan de funciones diferentes, pero que todas contribuyen a la interacción de bases de datos relacionales (ORM). Entre las capas podemos encontrar:

1. Capa de persistencia:
2. Capa de integración Hibernate Search:
3. Capa de consultas y resultados:
4. Capa de indexación y búsqueda:
5. Capa de motor de búsqueda

En este caso vamos a estudiar una sola de las capas del Framework, que será la capa de Hibernate Search, que se encarga de la búsqueda de texto como se especificó en el anterior punto.

¿Qué grandes retos de diseño enfrenta?
Uno de los grandes retos que veo del Framework es manejar la persistencia de datos, ya que el framework debe ser flexible, y por lo tanto debe ser adaptable a cualquier persistencia que se le ponga. con el fin de que pueda ser utilizado en varios entornos de trabajo. Así mismo, puede llegar a ser un reto ser flexible en este aspecto, ya que en la persistencia es en donde se conecta directamente con la base de datos, lo que implica que se debe ser cuidadoso con esta capa, y debe tener un diseño escalable y mantenible, cosa que se logra implementando un buen patron de diseno

Información y estructura del fragmento del proyecto donde aparece el patrón. No se limite únicamente a los elementos que hacen parte del patrón: para que tenga sentido su uso, probablemente va a tener que incluir elementos cercanos que sirvan para contextualizar.

Fragmento de codigo analizado:
https://github.com/hibernate/hibernate-search/tree/main/build/config/src/main/java/org/hibernate/checkstyle/checks/regexp

Como se mencionó anteriormente, el fragmento de código que se analizará del framework es la capa de Hibernate Search en la carpeta build. Este fragmento del framework se encarga de proporcionar funcionalidades avanzadas de búsqueda y filtrado de texto completo para aplicaciones Java que utilizan Hibernate como capa de persistencia. Para dar más detalles, Hibernate Search permite realizar búsquedas de texto completo en los contenidos de las entidades persistidas. Igualmente, la biblioteca gestiona automáticamente el índice de búsqueda para las entidades persistentes. En general, esta capa del framework se encarga de hacer búsqueda y procesamiento de textos en persistencia de datos diferentes, sin embargo, en este caso específico estamos utilizando el build, que es una parte del código que se encarga de suprimir datos dependiendo si es un string, o un comentario en la base de datos.

Información general sobre el patrón: qué patrón es y para qué se usa usualmente.
El patrón que se utiliza en este caso es ***Abstract Factory.*** Este patrón, a grandes rasgos permite producir familias de objetos relacionados, con el fin de evitar cambiar el código principal cada que haya un nuevo objeto que tenga unas características similares a otro ya creado. Generalmente se usa cuando tenemos en nuestro código familias de clases relacionados, pero no queramos que haya un dependencia de clases entre estos clases , sino más bien de una interfaz que generalice la familia de clases en común, lo que permite una futura extensibilidad en caso de que se requieran agregar más clases relacionadas a esa familia de clases. Ejemplo del patrón:

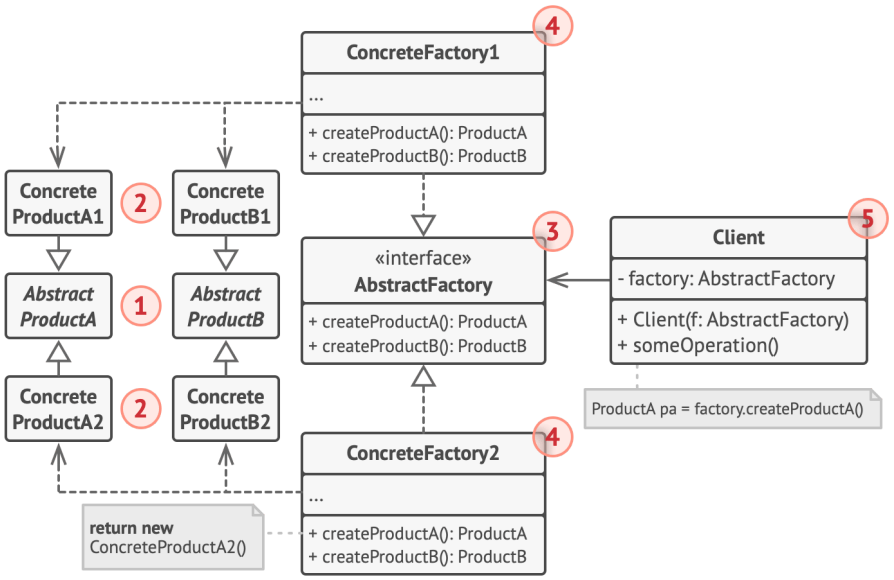


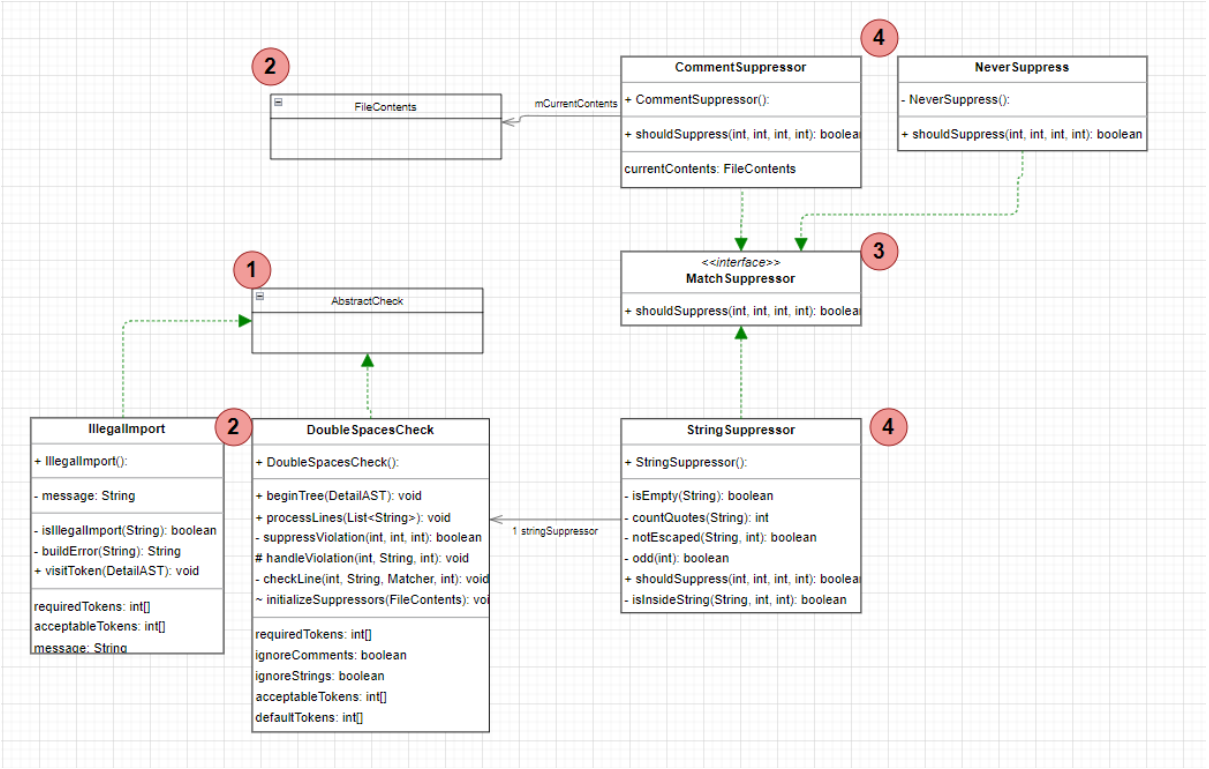
Imagen tomada de: https://refactoring.guru/es/design-patterns/abstract-factory

1. Los Productos Abstractos declaran interfaces para un grupo de productos diferentes pero relacionados que forman una familia de productos.
2. Los Productos Concretos son implementaciones distintas de productos abstractos agrupados por variantes. Cada producto abstracto (silla/sofá) debe implementarse en todas las variantes dadas (victoriano/moderno).
3. La interfaz Fábrica Abstracta declara un grupo de métodos para crear cada uno de los productos abstractos.
4. Las Fábricas Concretas implementan métodos de creación de la fábrica abstracta. Cada fábrica concreta se corresponde con una variante específica de los productos y crea tan solo dichas variantes de los productos.

Tomado de: https://refactoring.guru/es/design-patterns/abstract-factory

Información del patrón aplicado al proyecto: explicar cómo se está utilizando el patrón dentro del proyecto.

Dentro del proyecto, se está utilizando para suprimir datos en la persistencia. En este caso, hay varias clases diferentes que se encargan de suprimir diferentes tipos de datos dentro de la persistencia. Por ejemplo, para suprimir Strings hay una clase, y para suprimir comentarios hay otra. Por ello, se crea una interfaz que implementa el método de suprimir según la ubicación del dato, y esta interfaz funciona como la ***interfaz de fábrica abstracta***, ya que a partir de ella, se crean las dos familias, que en este caso vendrían siendo las ***fábricas concretas*** , que una de ellas se usa para eliminar Strings (**StringSupressor**) , y la otra para eliminar comentarios (**CommentSuppressor**).
En el caso del **StringSupressor**, encontramos que contiene un ***producto concreto***, llamado **DoubleSpacesCheck,** que se encarga de verificar si el String a verificar tiene dos espacios en blanco consecutivos. Luego, **DoubleSpacesCheck** extiende de la clase abstracta **AbstractCheck, q**ue funciona como un ***producto abstracto*** que completa la familia de clases para la parte de Strings.
Por otro lado, en el caso de **CommentSuppressor,** se puede ver que tiene un ***producto concreto*** a nivel de su familia, llamado **FileContents** que tiene una funcionalidad parecida al **AbstractCheck** .Igualmente, se presenta el diagrama de clases del fragmento de la librería, cada número está relacionado con la explicación del diagrama del patrón en el punto anterior.



Cabe aclarar que la clase abstracta **AbstractCheck** y **FileContents**, se encarga de proporcionar funcionalidades comunes para implementar verificaciones personalizadas en Checkstyle. Ambas hacen funcionalidades similares, ya que ambas pertenecen a la librería CheckStyle.

¿Por qué tiene sentido haber utilizado el patrón en ese punto del proyecto? ¿Qué ventajas tiene?

Una de las ventajas, es que se reduce el acoplamiento fuerte entre las clases que cumplen el papel de productos concretos. Además, el uso del patrón evita repetir código en clases que se relacionan en su funcionalidad, y así mismo ayuda a separar las clases que tienen un objetivo diferente, pero una funcionalidad parecida. Por otro lado, se puede mover el código de creación de productos a un solo lugar, lo que significa que en el caso de que se necesite implementar la supresión de un nuevo tipo de dato, sea más fácil de implementar con el patrón.

¿Qué desventajas tiene haber utilizado el patrón en ese punto del proyecto?

Al tener que crear una nueva familia para crear un nuevo producto, hace que el código se complique de más si se implementan nuevas interfaces y clases asociadas al patrón creado.

¿De qué otras formas se le ocurre que se podrían haber solucionado, en este caso particular, los problemas que resuelve el patrón?
Otra posible manera de implementarlo, sería sin la necesidad de crear una familia de clases, es decir, solamente utilizar la interfaz en común de todas las maneras de suprimir datos, y si hay mas maneras diferentes de suprimir datos, se van creando más interfaces que abarquen más cosas en común para la supresión de diferentes tipos de datos. Igualmente,también se hubiese podido implementar el patrón Inyección de dependencias (Dependency Injection), con el fin de proporcionar instancias de objetos a través de un contenedor de inversión de control, lo que permite desacoplar la creación de objetos de su uso y facilita la sustitución de implementaciones concretas.

Bibliografía

Abstract Factory. (2004). En *Software Architecture Design Patterns in Java*. Auerbach Publications.