

## Taller 5

- **Información general del proyecto**

El proyecto “*100-words-design-patterns-java*” tiene como propósito explicar ciertos patrones de diseño de la manera más simplificada y clara posible, para esto proporcionan descripciones cortas de cada uno de los patrones (De ahí el nombre, 100 palabras por descripción) e implementa pequeñas aplicaciones individuales que ayudan a entender el uso del patrón en un contexto aplicado.

Adicionalmente, el proyecto provee ejemplos reales de la aplicación de estos patrones, ya sea librerías oficiales, clases, métodos específicos, etcétera.

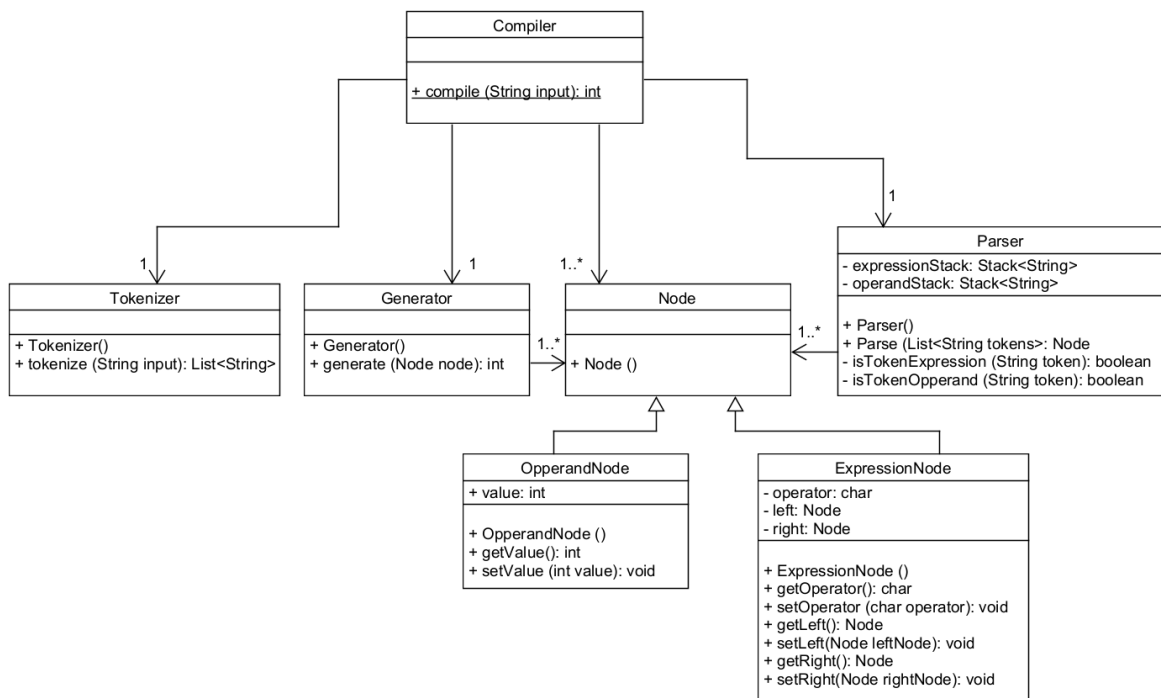
El proyecto se encuentra explícitamente dividido en folders que separan las distintas partes del proyecto, las cuales corresponden a las pequeñas aplicaciones que explican cada uno de los patrones abordados. Ninguna de estas aplicaciones está acoplada con alguna otra, las colaboraciones entre clases se dan únicamente entre las subdivisiones del proyecto (Las clases del patrón builder se conocen entre ellas y nada más, por ejemplo). Esta separación de componentes tan marcada genera diseños muy distintos, no son necesariamente incompatibles entre ellos, pero lucen algo inconsistentes y alejados uno de los otros.

URL del proyecto: <https://github.com/dstar55/100-words-design-patterns-java.git>

- **Información y estructura del fragmento del proyecto donde aparece el patrón**

Antes de ver las clases directamente involucradas con la fachada, es propicio explicar cuál es la estructura general del fragmento del proyecto correspondiente. La aplicación se define como los componentes más básicos de un nuevo lenguaje de programación. Este nuevo lenguaje está compuesto por un Lexer, que se encarga de tokenizar las expresiones que se reciben por medio de cadenas de texto, el parser, que verifica el aspecto sintáctico del nuevo lenguaje a través del uso de arboles de sintaxis abstractos y por ultimo los nodos, los cuales contribuyen al análisis realizado por el parser.

Las relaciones, colaboraciones y demás información de las clases se encuentran condensadas en el siguiente UML.



Como se puede observar en el UML, la clase que implementa la fachada se denomina “Compiler”, la clase compiler conoce todos los elementos necesarios para implementar el nuevo lenguaje de programación, pero no hay necesidad de que el usuario se interese por estas clases. Por esto, el usuario se involucrará únicamente con la abstracción del servicio prestado, un proceso que será facilitado por la fachada, la clase compiler.

- **Información general sobre el patrón**

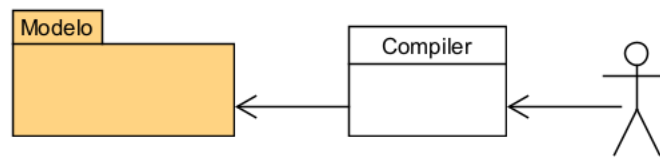
Fachada o *facade* es un patrón de diseño estructural que se encarga de dar fácil acceso y entendimiento a un programa, librería o framework complejo a través de una interfaz simplificada. El patrón fachada se implementa con la intención de proporcionar un acceso directo, limitado y simple a las funcionalidades del sistema que el cliente requiera específicamente, evitando de esta manera que se involucre o confunda con la complejidad del sistema mismo.

Este patrón es habitualmente usado cuando no se requiere que el cliente interactúe directamente con el sistema, más bien, se necesita que se involucre de la manera más intuitiva con los servicios que este ofrece, razón por la cual, una interfaz simple que esconde las complejidades de la lógica del proyecto es ideal.

Adicionalmente, el patrón fachada puede ser utilizado para separar el sistema de la aplicación en distintas capas o sectores. Implementando distintas fachadas se proporcionaría acceso a este conjunto específico de funcionalidades, separándolas así de otros subconjuntos y sus fachadas correspondientes, de esta manera, se crea una segregación exitosa que permite una colaboración menos acoplada.

- **Información del patrón aplicado al proyecto**

El patrón se aplica en el proyecto por medio de la clase compiler, la clase compiler conecta con todos los componentes del sistema que estructuran el nuevo lenguaje de programación. En primer lugar, la clase compiler se encarga de instanciar el parser, posteriormente hace uso del Lexer para convertir en tokens la cadena de texto que proporciona el usuario. Posterior a esto, la fachada se encarga de invocar el análisis sintáctico que será realizado por el parser, finalmente, tras haberse comprobado la sintaxis adecuada del input pedirá al generador que proporcione la respuesta correspondiente (En este caso, el nuevo lenguaje de programación no es más que una calculadora simple, por lo que el resultado es simplemente un entero correspondiente a la operación solicitada)



Desde la perspectiva del usuario, simplemente se ingresó una operación aritmética simple y se retornó el resultado de evaluar dicha expresión. La fachada se encargó de dar al usuario un acceso directo a la funcionalidad que necesitaba, ahorrándole así la necesidad de trabajar explícitamente con el sistema de tokens y análisis de la expresión, ya que estas partes del sistema no son de interés para el usuario.

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

La complejidad de un sistema de análisis sintáctico y traducción a tokens no era parte de las funcionalidades que interesaban al usuario de la calculadora, tampoco lo era el diseño de los nodos del árbol abstracto o el generador de respuesta, al usuario únicamente le importaba el resultado que se generaba por la colaboración de todas estas clases. También es necesario tomar en cuenta que el caso de estudio con mayor tendencia a pasar incluía a un usuario que no conozca cómo funcionan ninguna de estos elementos, ni en que orden se deberían instanciar o invocar.

Por todo lo anterior tiene sentido haber implementado una fachada. A diferencia del usuario la fachada si conoce el funcionamiento del sistema, sabe en que orden y cómo se deben seguir los flujos de la aplicación. La fachada da al usuario acceso a la funcionalidad de interés, la cual satisface los requerimientos que este solicitó. La ventaja que adquirimos es evidente, aislamos la complejidad del sistema lejos del alcance del usuario a la par de que cumplíamos con lo que este solicitó de nuestro programa.

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

La desventaja más evidente se refleja en el UML de la aplicación previamente expuesto. Para poder funcionar la fachada debe ser capaz de conocer, regular y colaborar con todas las clases que se involucren en las funcionalidades que la fachada desea satisfacer, en

nuestro caso específico, se podría decir que la fachada conoce todo el sistema. Esta enorme cantidad de colaboraciones puede llegar a ser difícil de controlar, pues la clase tiene demasiadas responsabilidades, deteriorando la coherencia de la fachada. Si en un punto se decidiera extender la aplicación es posible que estos problemas de acoplamiento y coherencia solo empeoren, dificultando el mantenimiento de la aplicación.

- ¿De qué otras formas se le ocurre que se podrían haber solucionado, en este caso particular, los problemas que resuelve el patrón?

Posiblemente, se habría podido incluir más clases intermediarias que ejecutaran las labores de la fachada. Un controlador podría instanciar el parser y ordenar al Lexer que tradujera cierto input ingresado por parámetro, una clase adicional se encargaría de relacionarse con el usuario y proporcionar al controlador el input que necesita el Lexer para iniciar su labor. Para evitar dañar la coherencia del parser adicionalmente se debería crear una clase que se encargue de mediar entre el Lexer, el parser y el generador, que conozca la clase nodo, esta clase suministraría al parser el argumento de entrada ya traducido a tokens y luego proporcionaría esta respuesta, que está en nodos, al generador, quien indicará cuál será la respuesta final del procedimiento. Se trata de ilustrar este escenario a través de un UML.

