Universidad De Buenos Aires
Facultad De Ingeniería

75.10 Tecnicas de diseño

Tp 1.1 Logging

Integrantes:

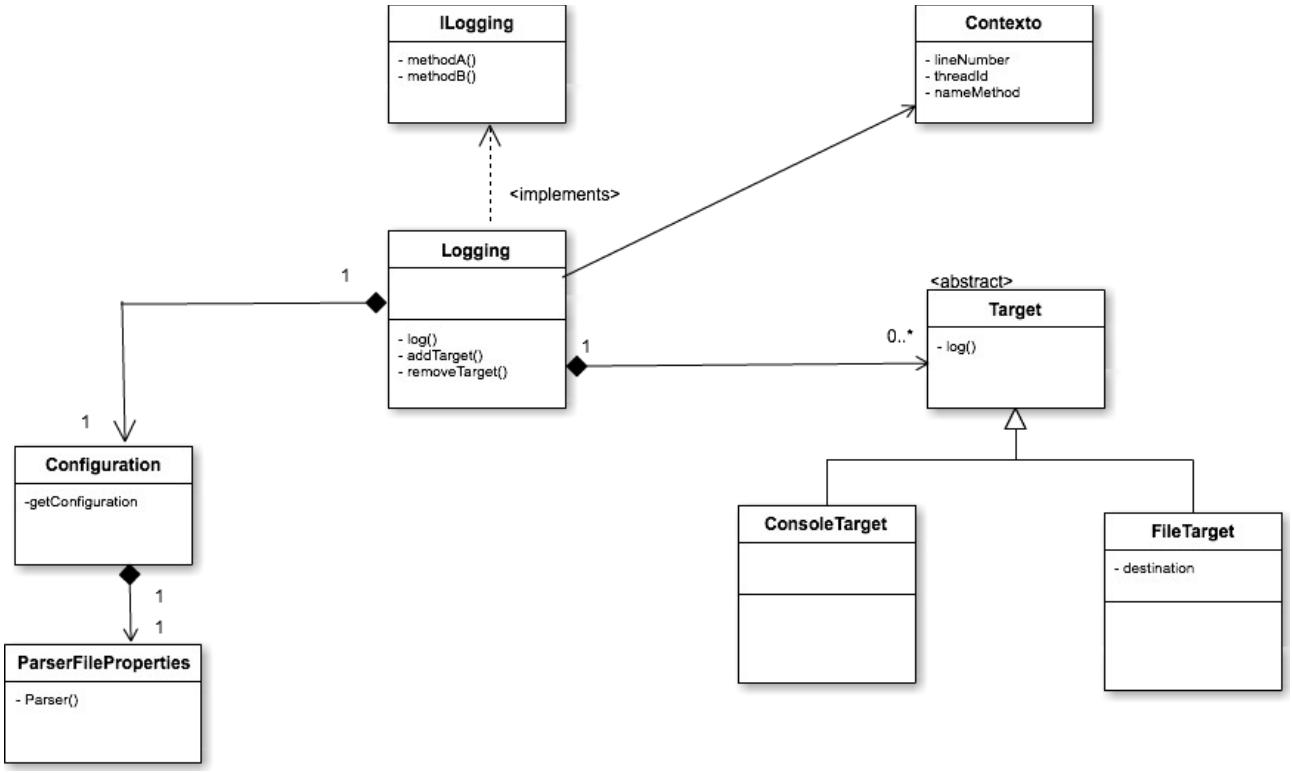
- S. Jorge
- S. Maximiliano
- O. Nicolás

Primer Cuatrimestre 2014

Indice:

Primer entrega

Diagrama de clases



Informe

A partir del enunciado expuesto en clase, se diseño y se desarrollo una herramienta que permita registrar mensajes de log a diferentes targets según un nivel y formato previamente definidos.

Se explican a continuación algunas decisiones de diseño implementadas con la finalidad de complementar la documentación y facilitar el posterior entendimiento del mismo para su correcta utilización, funcionamiento, modificaciones y extensiones que fueran necesarias.

1. Target: Se utilizo una clase abstracta haciendo que cada target implementado herede de ella con el fin de poder tratar a todos los targets como una misma cosa sin importar su naturaleza.
   En este contexto se podría haber utilizado, tambien, una interfaz en lugar de una clase abstracta.
   La decisión fue la primera ya que dicha clase posee comportamiento común a todos los targets.

2. Ilogging: se planteo que Logging implemente esta interfaz dado que en la misma se define la

Api que el usuario de nuestro Logging debería utilizar.
De esta forma, si en otra etapa se desea implementar un nuevo tipo de Logging, el mismo debería implementar tambien dicha interfaz y de esta manera mantenemos el acoplamiento bajo e incluso el código de quien utiliza nuestra librería clausurado ante cambios y se mantiene la inversión de la cadena de dependencia.

3. ConsoleTarget: Es la entidad encargada de realizar un log a una consola. Vale la pena destacar la forma en que se implemento el metodo equals del mismo puesto que, al devolver un valor constante, solo es posible (y por ende, admisible) una unica instancia del mismo en tiempo de ejecucion.

4. FileTarget: Es la entidad encargada de realizar un log a un archivo. Posee una referencia interna del nombre del archivo al cual debe loguear. Es posible tener varias instancias del mismo siempre y cuando no exista ningun par de FileTargets que apunten a un mismo archivo.

5. Context: es encargado de guardar toda la informacion relativa al entorno de ejecucion en un momento dado, previo a la invocacion al metodo log.

6. Level: es el encargado de guardar el nivel de funcionamiento actual y mantener y validar los niveles permitidos en los que el log puede funcionar.

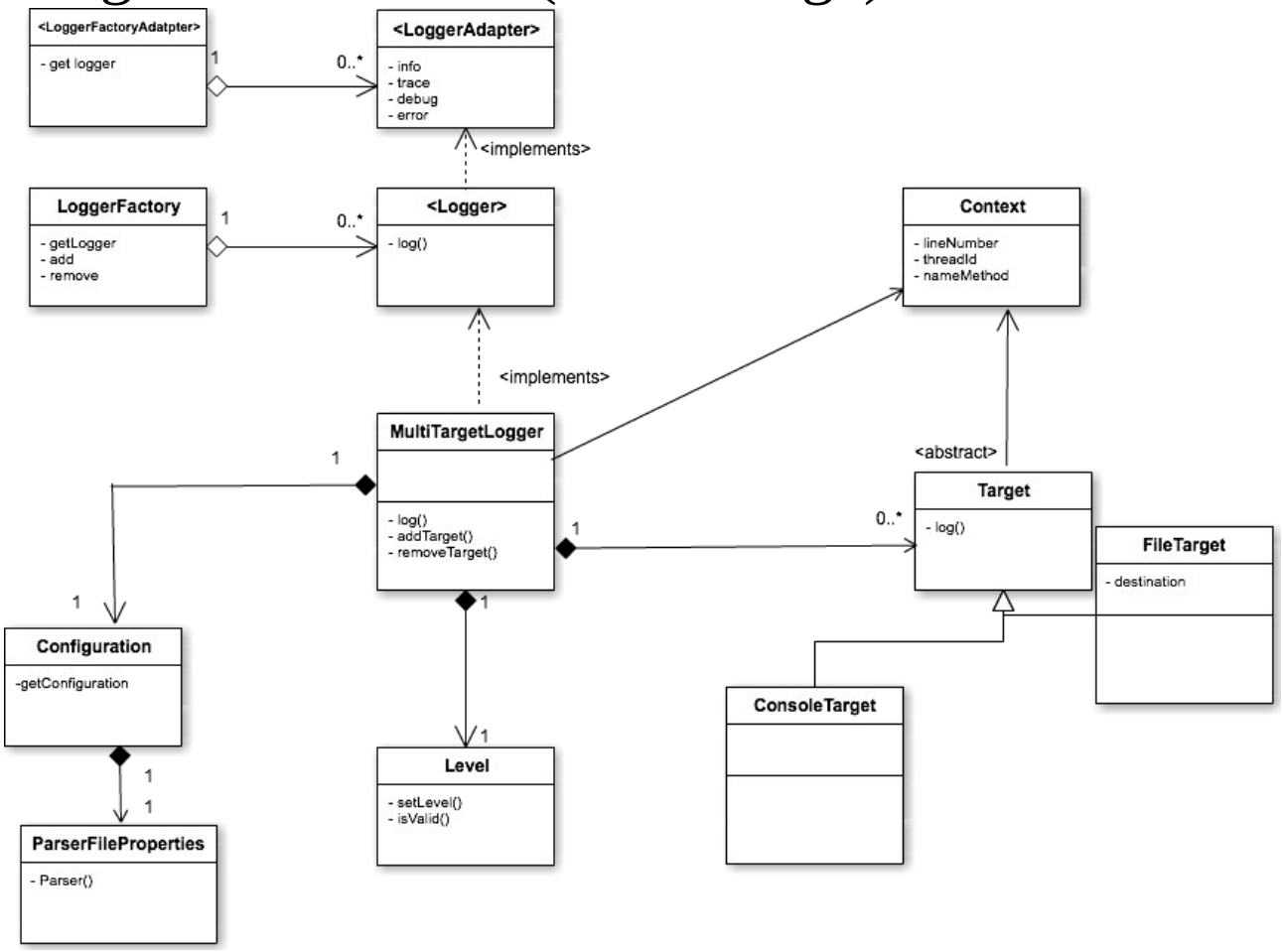
7. ParserFileProperties: Es la entidad encargada de procesar el archivo properties, para obtener una configuración inicial de los parámetros deseados a la hora de loguear.

8. Configuration: Es la que guarda todas las configuraciones que se leyeron de disco, y permite cambios manuales a los campos deseados a travéz de setters.

9. Logging: es el encargado de administrar y delegar la responsabilidad de todos los componentes que conforman el log para el que mismo pueda cumplir su correcto funcionamiento.

Segunda entrega

Diagrama de clases (2da entrega)



Informe

A partir de la segunda parte del enunciado expuesto en clase, se diseño y se desarrollo una herramienta que cumpliese con todos los requerimientos solicitados a partir de la extension de la primera entrega.

Se analiza tanto la solucion planteada para cada uno de ellos como el impacto que tuvo en el diseño original.

- Binding a SL4J: Para poder efectuar el binding, se tuvieron que utilizar dos patrones: Adapter y Factory. El primero para transformar o adaptar instancias de nuestro logger a instancias de loggers que manejen las firmas propuestas por la librería de slf4j. El segundo para encapsular la creación de estas instancias, y obtener de manera organizada el logger necesario en cada momento.

El tp asimiló muy naturalmente estos cambios.

- Nuevo Nivel: Agregar el nivel trace a la implementacion original fue practicamente trivial dado que los mismos estaban implementados a partir de un enumerado.

- Multiples Loggers: Para poder desarrollar esta funcionalidad fue necesario agregar una clase llamada LoggerFactory, la cual posee metodos unicamente estaticos sin necesidad de instarciarse (unico punto de entrada). La misma es responsable de la creacion y destruccion de los de los logs. El impacto resulto en una extension del trabajo original casi sin ninguna modificacion del codigo ya existente.

- Logger API: La aplicación no sufrió grandes cambios a nivel estructural para poder implementar todo lo requerido para esta entrega. Se puede decir que el único cambio visible a nivel usuario con respecto a la primer versión, fue que ahora al construir un logger, debe recibir por parámetro el nómbre de éste, el resto de la funcionalidad vieja se mantuvo intacto.

- Filtros regex: fue necesario extender la clase MultiTargetLogger para que soportara esta funcionalidad (filtra según mensaje final). Se opto por modificar la misma en vez de extenderla a pesar de que esto viola el principio de abierto-cerrado por cuestiones de facilidad en la implementacion dado que el impacto sobre los metodos afectados era casi nulo.

- Configuración: La configuración en si no cambió mucho, se modificó el path a los archivos properties y xml, antes se recibían por parámetros y ahora tienen una ubicación default. Hubo que implementar el órden de carga de estos archivos, arrancando por el properties, luego si éste no existía buscar el xml, y por último una configuración defaul por si lo anterior fallaba.

Se optó por un único archivo properties y un único archivo xml para almacenar toda la información de los loggers. Para el primer caso se optó por concatenarle el nombre del logger a la clave leída del archivo. Para el segundo se localizan todos los atributos de ese logger mediante un tag con el nombre de éste.

- Inicialización: Como se mencionó anteriormente, la inicialización permanece bastante fiel a lo que era la primer entrega. Los cambios que hubo fueron, el agregado del nombre del logger (que es recibido por parámetro) y la eliminación del path de los archivos de configuración, ya que estos están en una ubicación default ahora.

- Nuevo Formato (json): El nuevo formato se adaptó perfectamente al tp, la implementación fue muy fácil y local, hubo que modificar solo un archivo.

- Formato Mensajes (Pattern de la entrega anterior - %g): Al igual que el level, la implementación de éste nuevo campo de formato fue casi trivial ya que los loggers ahora tenían un atributo más que era su nombre.

- Destinos Custom y Filtros custom: para implementar estas funcionalidades se tuvo que utilizar reflection dado que las mismas se cargan en tiempo de ejecucion.
Para poder utlizarlos correctamente y que su uso no sea impredecible se provee al usuario una clase y una interfaz (Target y FilterCustom respectivamente) de la cual los destinos custom deben heredar y los filtros implementar para poder agregar su funcionalidad.

Para poder utilizarlos correctamente una vez desarrollados se debe colocar en el mismo directorio la clase padre/interfaz y la nueva clase y desde una consola ejecutar el comando:
javac *.java

esto creara el archivo .class correspondiente a la clase custom que el usuario ha desarrollado. Luego debe copiar la misma en algun directorio dentro del classpath para poder utilizarla (las pruebas fueron realizadas en (target/classes/ar/com/grupo1/tecnicas/Logging/), luego para poder cargarlas en tiempo de ejecucion se debe invocar a los siguientes metodos de MultiTargetLogger según corresponda:

MultiTargetLogger.addTarget(String Name);
MultiTargetLogger.setCustomFilterClass(String Name);

Todas las clases custom deben pertenecer al paquete: ar.com.grupo1.tecnicas.Logging para funcionar correctamente. A la vez, el parametro con el que es invocado (Name) debe ser de la forma:

nombreDePaquete.NombreDeClaseCustom

Ejemplo:

MultiTargetLogger.addTarget("ar.com.grupo1.tecnicas.Logging.NetworkCustomTarget");

Los customs Target tuvieron un impacto bajo en el desarrollo puesto que hereda de target y cada MultiTargetLogger cuando efectivamente loguea lo hacia de modo polimorfico para cada destino, de la misma manera esto sigue funcionando para destinos custom.

Para los filtros custom en cambio fue necesario utilizar el patrón de template method donde existe un paso (el filtrado custom) que se redefine para cada clase custom.