Trabajo Práctico N° 1 - Logging

Técnicas de Diseño (7510)
1er Cuatrimestre 2014

Grupo 17

Pablo Castarataro (89683)
Ricardo Gilberto (80713)
Flavio Ferro (87187)

Documentación de Usuario

Configuración de la Herramienta

1. Crear un archivo XML con el siguiente formato:

```
<?xml version="1.0"?>
<logger>
	<log>
		<level>Debug</level>
		<baseformat>%%%%%F %t %n %L %n %M HOLAAA %F %n %m %n %%  %p
%d{yyyy} %d{M} %d{yyyy-MM}</baseformat>
		<outputstring>console:</outputstring>
		<delimiter>:</delimiter>
	</log>
	<log>
		<level>Info</level>
		<baseformat>%d{HH:mm:ss}-%p-%t-%m</baseformat>
		<outputstring>file:log1.txt</outputstring>
		<delimiter>-</delimiter>
	</log>
</logger>
```

La raíz del documento debe ser un tag <logger>. Un logger está compuesto de uno o más logs.
En cada <log> se deberá especificar:
<level>: mínimo nivel de loggeo. Debe corresponderse con alguno de los definidos en el enumerado *LevelPriority*.
<baseformat>: formato del mensaje que se desea imprimir. (Los patrones válidos que se reemplazaran son aquellos definidos en *FormatterRepository*).
<outputstring>: este parámetro especifica la salida de impresión del mensaje. Son válidas las definidas en el *ConcreteOutputFactory*.
<delimiter>: define el o los caracteres que se consideran separadores.

2. Obtener un Logger a partir del archivo de configuración utilizando la clase LoggerFactory

```
LoggerFactory factory = LoggerFactory.getInstance();
```

```
Logger logger = factory.createLogger("config.txt");
```

Utilización de la Herramienta

Una vez ejecutados los pasos de configuración se debe utilizar la herramienta de la siguiente manera:

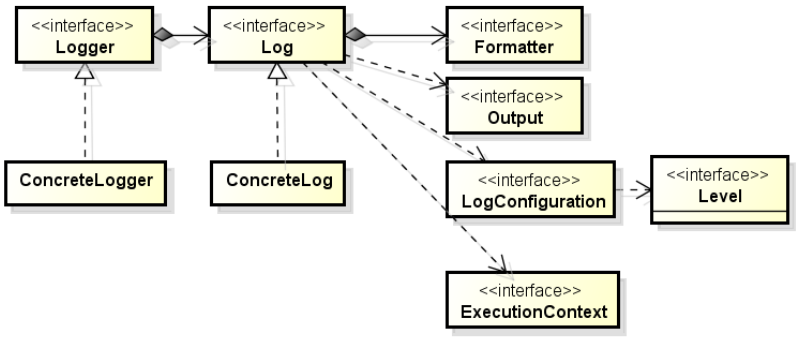
```
logger.log(LevelPriority.OFF, "MSG1");
logger.log(LevelPriority.DEBUG, "MSG2");
logger.log(LevelPriority.ERROR, "MSG3");
```

Documentación Técnica

Este documento sirve como referencia para interpretar como fue pensado e implementado el sistema de logging.

Visión general

A continuación se agrega un diagrama de clases que describe a grandes rasgos la arquitectura utilizada para cubrir con la funcionalidad pedida.



Una vez que creado el Logger a partir del archivo de configuración, el usuario tendrá disponible un método que le permite realizar el logueo correspondiente indicando el nivel y el mensaje. Esta interfaz es lo único a lo que el usuario tiene acceso.

Internamente, el logger itera sobre su colección de Log, que ya habían sido especificados en el archivo de configuración, y por cada uno de ellos manda a ejecutar el logueo correspondiente.

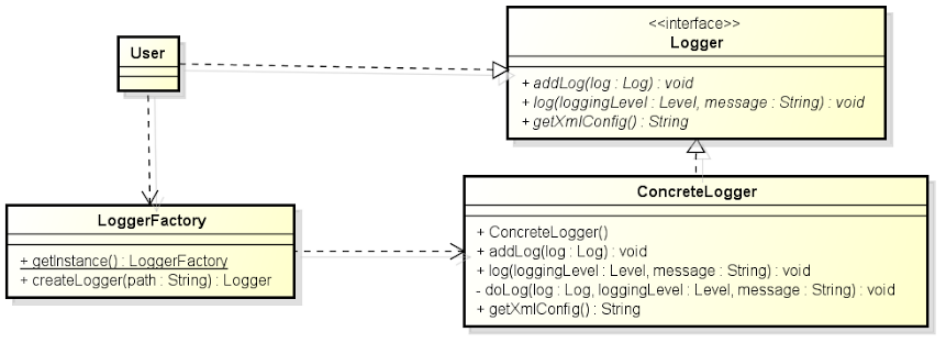
Por su parte, cada Log se encargará de:

- Formatear el mensaje según el formato especificado en la configuración.
- Verificar que el nivel mínimo de Log permite imprimir el mensaje. En caso de no serlo, ignora la impresión.
- Obtener datos del contexto en el que fue ejecutado el método.
- Imprimir el mensaje formateado, en la salida configurada.

Para la configuración de la herramienta, se optó por la utilización de un archivo XML por sobre uno properties debido a que planteamos una estructura de árbol para la creación de varios Logs dentro de un mismo Logger.

Configuración de la herramienta

Para poder configurar la herramienta, se provee un Factory que, a partir de un archivo de configuración con el formato especificado (*), genera una instancia válida que implementa la interfaz *Logger*.

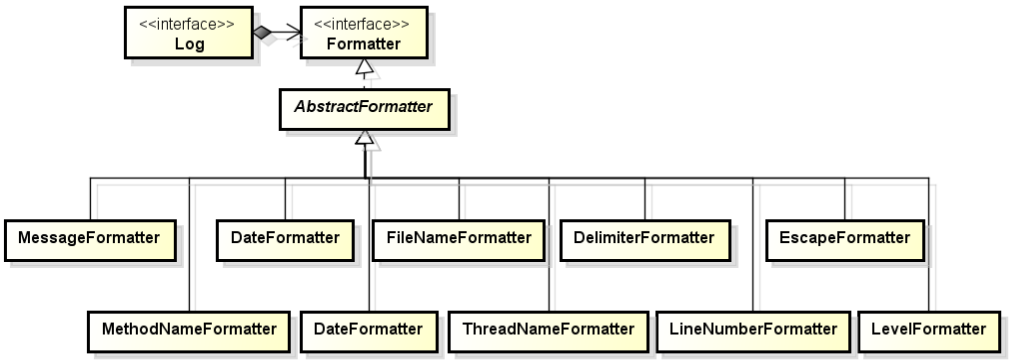


(*) ver formato en Documentación de Usuario.

Formatos

Para procesar los distintos formatos, el Log delega este comportamiento a cada tipo de Formatter que se haya agregado en *FormatterRepository*. Para agregar un nuevo tipo de formato, se deberá implementar la interfaz *Formatter* y luego agregarlo a este repositorio.
El procesamiento de los formatos consta de dos etapas. Un pre proceso, donde el formato podrá reemplazar el patrón por el valor deseado siempre que este no entre en conflicto con un patrón existente, y un post proceso donde se concluye con el formateo del mensaje a devolver. Lo importante es que luego de aplicar estos dos procesos todas las apariciones del patrón sean reemplazadas por el valor correspondiente.

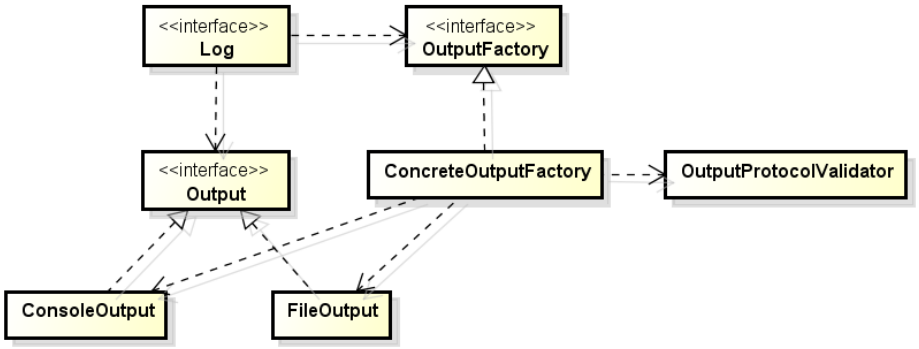
La idea de utilizar herencia para los formatters es evitar repetición de código innecesario al momento de post procesar el mensaje.



Salidas

Para poder imprimir los mensajes resultantes en los distintos tipos de salida, se planteó la creación de un Factory que a partir de un protocolo definido en el archivo de configuración decide que tipo de *Output* deberá instanciar. Esta fábrica utiliza el patrón de diseño “Cadena de responsabilidad” para determinar cuando el *Output* es válido.
Para agregar nuevos tipos de Output se debe:

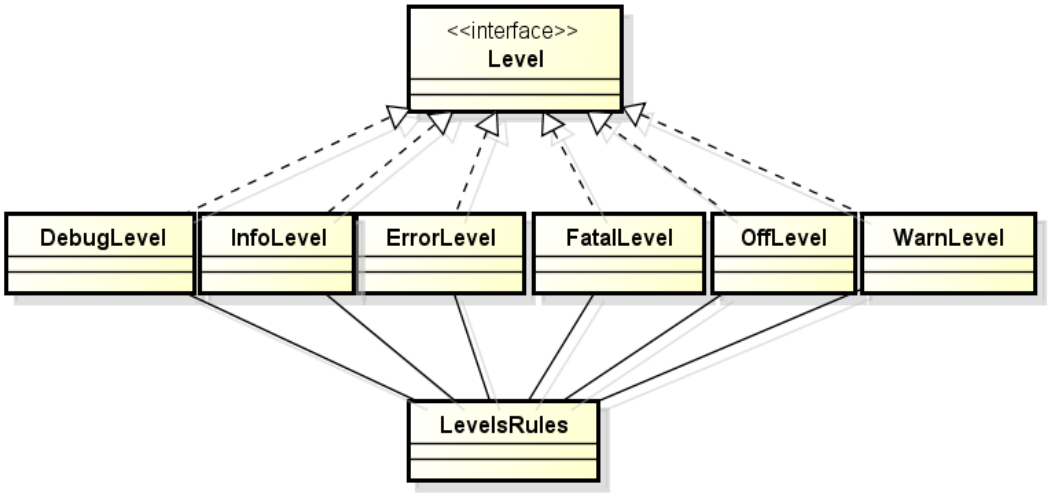
- Crear una clase que implemente Output.
- Agregar en OutputFactory el nombre de la clase anteriormente creada.



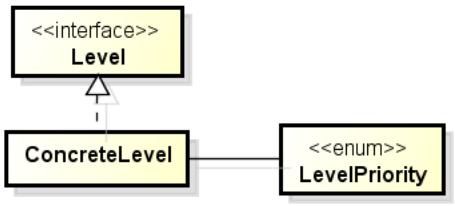
Niveles

Inicialmente se había pensado una solución que utilizaba una matriz para definir cuando un nivel estaba por arriba de otro. En la segunda entrega implementó una simplificación que, al no ser necesario realizar reglas complejas de comparación, utiliza un enumerado para comparar los niveles según su prioridad.

Antigua Implementación



Nueva Implementación



Las ventajas que presenta la nueva implementación es que permite agregar nuevos niveles sin la necesidad de crear nuevas clases, simplemente se debe agregar al enumerado en la posición deseada según su prioridad.

Entrega Parte 2

Funcionalidades Requeridas:

- Binding a SLF4J.
- Nuevo Nivel.
- Multiples Loggers.
- Logger Api que soporte Throwables.
- Filtros por RegEx.
- Filtros Custom.
- Configuración desde Properties, xml o default.
- Formato JSON.
- Formato %g que muestre el nombre del Logger.
- Destinos Custom.

A continuación hacemos un breve resumen con el impacto que trajo la introducción de cada uno de estos nuevos requerimientos al sistema.

Binding a SLF4J
Se implementaron las clases necesarias para realizar el binding, delegando en las clases de nuestro framework la creación y el logging.
Para poder lograrlo, realizamos una leve modificación en el diseño original del sistema trasladando la propiedad de nivel mínimo de logueo desde el *Log* hasta el *Logger*. Esto además, se vio reflejado en los correspondientes archivos de configuración.

Antes

```
<?xml version="1.0"?>
<logger>
	<log>
		<level>Warn</level>
		...
```

Después

```
<?xml version="1.0"?>
<logger name="Main" level="Warn">
	<log>
```

. . .

Nuevo Nivel
El diseño flexible de la herramienta permitió agregar un nuevo nivel de Logueo de manera sencilla, modificando únicamente el enumerado *LevelPriority* para tener en cuenta el nuevo nivel y su prioridad.

Multiples Loggers
Para que sea posible se modifico el factory original ademas de los archivos de configuración para admitir múltiples loggers con su nombre identificatorio.

Logger Api que soporte Throwables
Se agregaron métodos especiales a la interfaz *Logger* que soportan *Throwables*. El stack trace del *Throwable* se adjuntará al mensaje formateado final de logueo.

Filtros por RegEx
Especificado en la property *filterpattern* se incorpora filtros por regular expression.

Filtros Custom
Implementando la interface *Filter* es posible incorporar filtros custom que se verificarán al momento de loguear.

Configuración desde Properties, xml o default
Se incorpora la posibilidad de levantar *Loggers* de distintos tipos de archivos de configuración. (Properties - XML - Default). Desde *LoggerFactory* se verifica la existencia de un archivo de configuración del tipo Property, passando a XML en caso de no existir y por último a Default. La implementación de la interface *LoggerFactoryHandler* por parte de cada uno de los Factories específicos es necesaria para lograr este comportamiento.

Formato JSON
Mediante un nuevo *Formatter* la incorporación del formato de salida JSON no fue de gran complejidad, simplemente se implementó *JsonFormatter* ademas de agregarlo del *FormatterRepository.* El patrón escogido para imprimir este formato es %J.

Formato %g que muestre el nombre del Logger
De manera similar al formato JSON, se incorporó un nuevo *Formatter* llamado *LoggerNameFormatter.* Simplemente se implementa este nuevo *Formatter* y se lo agrega al *FormatterRepository.*

Destinos Custom

La incorporación de destinos custom estaba soportada en el diseño original. Los únicos cambios que se hicieron fueron en el archivo de configuración, donde se especifica el nombre de la clase que implementa el destino custom (Output).

El destino custom se indica en el archivo de configuración de la siguiente manera

```
<outputstring>ar.fiuba.tecnicas.logging.log.FileOutput:log2.txt</outputstring>
```

donde los parámetros necesarios para un destino custom en particular se especifican luego de ":". Cada Output tendrá su formato predefinido para recibir los parámetros y es cada uno de ellos el encargado de verificarlos.

Ejemplo Archivo de Configuración XML

```
<?xml version="1.0"?>
<loggers>
  <logger name="Main" level="Warn">
    <log>
      <baseformat>%d{HH:mm:ss}-%F-%L-%m</baseformat>
      <outputstring>ar.fiuba.tecnicas.logging.log.ConsoleOutput:</outputstring>
      <filters>
        <filterpattern>.*Mundo.*</filterpattern>
        <customFilter class="ar.fiuba.tecnicas.logging.filter.FilterMock">
          configuraciondefilter1
        </customFilter>
      </filters>
      <delimiter>;</delimiter>
    </log>
    <log>
      <baseformat>%d{HH:mm:ss}-%p-%t-%m</baseformat>
      <outputstring>ar.fiuba.tecnicas.logging.log.FileOutput:log1.txt</outputstring>
      <delimiter>-</delimiter>
    </log>
  </logger>
  <logger name="logger2" level="Warn">
    <log>
      <baseformat>%d{HH:mm:ss}-%p-%t-%m</baseformat>
      <outputstring>ar.fiuba.tecnicas.logging.log.FileOutput:logger2.txt</outputstring>
      <delimiter>-</delimiter>
    </log>
  </logger>
</loggers>
```

Ejemplo Archivo de Configuración Properties

```
loggers= org.slf4j.Logger,logger3
org.slf4j.Logger.logs= log1,log2,log3
org.slf4j.Logger.level=Debug
org.slf4j.Logger.log1.baseformat=%p %d{yyyy-MM}
org.slf4j.Logger.log1.outputstring=ar.fiuba.tecnicas.logging.log.ConsoleOutput:
org.slf4j.Logger.log1.delimiter=:
org.slf4j.Logger.log2.baseformat=%g%d{HH:mm:ss}-%p%n-%t-%m
org.slf4j.Logger.log2.outputstring=ar.fiuba.tecnicas.logging.log.FileOutput:log1.txt
org.slf4j.Logger.log2.delimiter=_
org.slf4j.Logger.log3.baseformat=%g%d{HH:mm:ss}-%F-%n%L-%m
org.slf4j.Logger.log3.outputstring=ar.fiuba.tecnicas.logging.log.FileOutput:log2.txt
org.slf4j.Logger.log3.delimiter=#
org.slf4j.Logger.log3.filters=filter1,filter2,filter3
org.slf4j.Logger.log3.filter1.regex=.*probando.*
org.slf4j.Logger.log3.filter2.class=your.custom.filter.implementation
org.slf4j.Logger.log3.filter2.conf=configuraciondefilter2
logger3.level=Error
logger3.logs= log1
logger3.log1.baseformat=%g%d{HH:mm:ss}-%F-%n%L-%m
logger3.log1.outputstring=ar.fiuba.tecnicas.logging.log.FileOutput:log2.txt
logger3.log1.delimiter=#
```