**FIUBA - 7510**

**Técnicas de Diseño**

*Trabajo práctico 1: Logging*

1er cuatrimestre, 2014

(trabajo grupal)

Alumnos:

|  |  |  |
| --- | --- | --- |
| ***Nombre Completo*** | ***Padrón*** | ***Mail*** |
| Rodriguez, Marcelo | 90445 | marce\_yugo@yahoo.com.ar |
| Coco, Nicolas Alberto | 87212 | nicolascoco85@yahoo.com.ar |

*Fecha de entrega final*: Jueves 19 de Junio de 2014

*Tutora*: Federico Diaz

*Nota Final*:

**Tabla de contenidos**

Introducción

-Objetivo del trabajo

- Consigna general

Descripción de la aplicación a desarrollar

-Funcionalidad a implementar

-Objetivos

[-Entregables](#id.2s8eyo1)

[-Herramientas a utilizar](#id.lnxbz9)

Informe

-Supuestos

-Modelo de Dominio

[-Diagrama de paquetes](#id.1ksv4uv)

-Diagramas de clases

-Diagramas de secuencia

-Detalles de implementación

-Excepciones

-Nuevas funcionalidades

INTRODUCCIÓN

# Objetivo del trabajo

Aplicar los conceptos enseñados en la materia a la resolución de un problema, trabajando en forma grupal y utilizando un lenguaje de tipado estático (Java)

## Consigna general

Desarrollar la API completa, incluyendo el modelo de clases. La aplicación deberá ser acompañada por prueba unitarias e integrales y documentación de diseño. En la siguiente sección se describe la aplicación a desarrollar.

# 

DESCRIPCIÓN DE LA APLICACIÓN A DESARROLLAR

Se deberá implementar una herramienta que permita facilitar los diversos procesos de “Logging” que podrían requerir una aplicación.

***Funcionalidad a Implementar***

Definir la interfaz del usuario (programador)

● Proveer una API capaz de registrar mensajes/eventos con algún “nivel” determinado de prioridad.

● Proveer una API para configurar la herramienta.

Niveles de Logging

● Proveer los siguientes Niveles (ordenados por prioridad):

○ TRACE > DEBUG > INFO > WARN > ERROR > FATAL > OFF

● Permitir configurar a la herramienta para trabajar en algún nivel particular. Si por ejemplo

activamos el nivel INFO deberíamos registrar todos los eventos que se encuentran a su

derecha: INFO, WARN, ERROR, FATAL.

Destinos

● Permitir loguear a un archivo particular.

● Permitir loguear por consola.

● Permitir loguear a más de un archivo y/o consola en simultáneo

Formato Mensajes

Permitir si se desea definir el formato de todos los mensajes logueados por algun patron:

● %d{xxxxxxx} debería aceptar cualquier formato válido de SimpleDateFormat.

● %p debería mostrar el Nivel del mensaje.

● %t deberia mostrar el nombre del thread actual.

● %m debería mostrar el contenido del mensaje logueado por el usuario.

● %% debería mostrar un % (es el escape de %).

● %n debería mostrar el “separador” con el que el usuario configuró la herramienta o un

default a elección.

● %L line number.

● %F filename.

● %M method name.

Ejemplo:

%d{HH:mm:ss} ­ %p ­ %t ­ %m

21:12:09 ­ INFO ­ main ­ Se registró un nuevo usuario.

Configuración desde un archivo de Properties

Permitir levantar la configuración de la herramienta desde un archivo de properties.

## Objetivo

* Implementar una versión inicial de la herramienta con funcionalidad mínima.
* Aplicar las técnicas vistas en la teoría y en la práctica.

# *Entregables*

* Código fuente de la aplicación completa, incluyendo también: código de la pruebas, archivos de recursos
* Informe

***Herramientas a utilizar***

* Java >= 1.7
* Maven >= 3
* JUnit >= 4.11
* Git

INFORME

# *Supuestos*

* Habrá un único formato que será utilizado por todos los **destinations**.
* Se podrá filtrar por 2 criterios, los mismos son **level** y **active.**

***Modelo de Dominio***

En esta sección se mostrarán las clases implementadas y la responsabilidades que tienen cada una asignada.

**Config**: Tiene la responsabilidad de mantener los parámetros de la configuración que le dará el usuario.

**ConfigConstants**: Contiene constantes de configuración, permite desacoplar este contenido y simplificar la clase Config.

**ConfigParser**: Esta clase será la responsable de interpretar el archivo de configuración.

**LogHandler**: Esta clase es la encargada de administrar las peticiones del log, administra cada eslabón de nuestra cadena de responsabilidad.

**Logger**: Es una interfaz de la cual depende la clase abstracta FormattedLogger que permite registrar los mensajes.

**FormattedLogger**: Es la clase abstracta que implementa la interface logger de ella extienden los tipos de destinos en los que se puede loguear, la consola o el archivo plano.

**PlainText**:Esta clase extiende de FormattedLogger y permite mostrar el contenido del mensaje en una archivo plano redefiniendo el método log.

**Console**:Al igual que PlainText extiende de FormattedLogger y es la encargada de mostrar el contenido del mensaje por consola, redefiniendo el método log.  
  
**Log**: Es la clase que contiene el punto de entrada API, interactúa directamente con el programador usuario a través de métodos estáticos

**Message**: Es la clase que maneja los datos del mensaje, como el método, el nivel, el thread etc.

**StringMessageFormatter**: Es una interfaz que contiene el método format que luego será implementado por cada formato para interpretar lo que el usuario necesita registrar en el mensaje de logging.

**Formatters**: Esta clase tiene la responsabilidad de la aplicación del modelo de formato a cualquier mensaje registrado por el usuario. implementa la interfaz MessageFormatter

**FileNameFormatter**: Implementa la interfaz StringMessageFormatter, redefiniendo el método format que permite a partir del patrón “F” mostrar el FileName del mensaje.

**MethodNameFormatter**: Implementa la interfaz StringMessageFormatter, redefiniendo el método format que permite a partir del patrón “M” mostrar el MethodName del mensaje.

**LineNumberFormatter**: Implementa la interfaz StringMessageFormatter, redefiniendo el método format que permite a partir del patrón “L” mostrar el LineNumber del mensaje.

**LevelFormatter**: Implementa la interfaz StringMessageFormatter, redefiniendo el método format que permite a partir del patrón “p” mostrar el LevelFormatter del mensaje.  
  
**UserMessageFormatter**: Implementa la interfaz StringMessageFormatter, redefiniendo el método format que permite a partir del patrón “m” mostrar el UserMessage del mensaje.

**ThreadFormatter**: Implementa la interfaz StringMessageFormatter, redefiniendo el método format que permite a partir del patrón “%t”mostrar el ThreadFormatter del mensaje.

**ScapeCharacterFormatter**: Implementa la interfaz StringMessageFormatter, redefiniendo el método format que permite a partir del patrón “%%”mostrar el ScapeCharacter “%” del mensaje.

**DefaultSeparatorCharacterFormatter**: Implementa la interfaz StringMessageFormatter, redefiniendo el método format que permite a partir del patrón “n” mostrar el Separador del mensaje, con el que el usuario configuró la herramienta o un default.

**DateFormatter**: Implementa la interfaz StringMessageFormatter, redefiniendo el método format que permite a partir del patrón “d” mostrar el Fecha del mensaje.

**FormattersBuilder**: Esta clase contiene una lista con los patrones utilizados para los formatos.

# *Diagrama de paquetes*

![](data:None;base64,)

## Diagramas de clases

![](data:None;base64,)

handlers

![](data:None;base64,)

loggers

![](data:None;base64,)

Formatter & Message

![](data:None;base64,)

## Diagramas de Secuencia

![](data:None;base64,)

![](data:None;base64,)

![](data:None;base64,)

## Detalles de implementación

config.properties

Partiendo de este archivo de muestra:

|  |
| --- |
| **destination**=Console,log1.txt,log2.txt  **active**=true  **format**= [%d{HH:mm:ss}] %m  **level**=DEBUG  **filter**=level(,FATAL,INFO)-active(true,true,false)  **logFileDir**=logs |

En el archivo *config.properties* los **filtros** se aplican como un array de atributos que debe contener tanta cantidad de lugares como destinations haya:

|  |
| --- |
| level ( level para destination 1 , level para destination 2, ...)  active ( true|false , true|false , ...) |

De esta manera, podemos ver como lograr un comportamiento específico para cada destination. Del archivo de muestra notamos que sino se completa nada en una determinada posición del array de un filtro, se usará el valor por default. Para el caso particular ya descrito, el destination **Console**, tomará el level **DEBUG**.

Levels

Para la resolución de los distintos niveles de log se utilizó una variación del Patrón de Diseño ***Chain of Responsability***. Disponemos de una clase Handler que posee una colección de loggers y un sucesor. Al parsearse el archivo de configuración se construye la cadena de responsabilidades agregándole a cada Handler su colección de loggers.

A continuación podemos ver un ejemplo de cómo quedaría una cadena de Handlers armada con las referencias de quién apunta a quién como su sucesor:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **OFF** | **>** | **FATAL** | **>** | **ERROR** | **>** | **WARNING** | **>** | **INFO** | **>** | **DEBUG** | **>** | **TRACE** |
|  |  |  |  |  |  | -log1.txt  -log2.txt |  | -log3.txt |  | -Console |  |  |

Así las siguientes líneas de código:

|  |
| --- |
| log.warning(“warning !”);  log.info(“info !!”);  log.debbug(“debbuging…”); |

Provocarían el siguiente output:

|  |  |  |  |
| --- | --- | --- | --- |
| *log1.txt* | *log2.txt* | *log3.txt* | *Console* |
| warning ! | warning ! | warning !  info !! | warning !  info !!  debbuging.. |

Dado que al hacer log.warning(“mensaje”) se está llamando al Handler Warning que procesará el mensaje para sus loggers y luego le pasará ese mensaje al siguiente en la cadena (INFO) para que realice la misma actividad.

Siguiendo con este razonamiento log.debbug(“mensaje”), solo se vería reflejado en los loggers que tenga el Handler Debbug. Siendo éste el último eslabón de la cadena.

Respetando el principio de segregación de la Interfaz

Como se desprende de los diagramas de clases se dispone de una colección con comportamiento propio llamada **Formatters**. Es de suma importancia que la misma implemente la interfaz **MessageFormattter** que comunica que una clase es capaz de formatear un **Message**. Esto nos permitió dar un gran paso y lograr que una abstracción como **FormattedLogger** pase a depender de una interfaz y no de una clase concreta.

De la misma forma al crear la interfaz **StringMessageFormatter** logramos que cada uno de los formateadores concretos no dependa de métodos que no utilice (al no tener que implementar **format(Message message)** .

Lo mismo ocurre en el caso contrario: **Formatters** no depende de métodos que no utiliza ( no debe implementar **format(String formattedMessage, Message message)** )

Filtros Custom

[ Listar operadores y un ejemplo de modo de uso del usuario]

## Excepciones

* ConfigurationException: Esta excepción es lanzada cuando un atributo del propertiesFile no posee un valor válido.
* ConfigurationFileException: Esta excepción es lanzada cuando no es posible por la API generar un archivo propio de configuración. Por lo general debido a permisos de escritura denegados.

# *Nuevas funcionalidades*

En esta sección se mencionan las nuevas funcionalidades solicitadas y el modo en que fueron resueltas

# Binding a SL4J

Objetivo:

# Implementar un binding a SLF4J http://www.slf4j.org/ para que una library que use SLF4J pueda usar nuestro logger como backend.

# Implementacion:

Pendiente

# Nuevo Nivel

Objetivo:

# Se requiere agregar el nivel TRACE > DEBUG > INFO > WARN > ERROR > FATAL > OFF.

Implementacion:

La incorporación del nuevo Nivel significó un cambio para nada costoso dado que el diseño original absorbió de muy buena manera el cambio. Únicamente fue necesario agregarle un eslabón más , donde la cadena es construida.

# Multiples Loggers

Objetivo:

# Permitir definir y utilizar loguers distintos en diferentes areas de una aplicacion. Es decir, se quiere poder utilizar dos loggers con distintas configuraciones en lugares distintos.

# Ej:

# “en el Modulo A”

# logger = xxxx.getLogger(“module­A­logger”) ….

# “en el Modulo B”

# logger = xxxx.getLogger(“module­B­logger”)

# Y cada uno de esos Loggers tienen potencialmente una configuración distinta.

Implementacion:

Este nuevo pedido implicó la mutación del patrón Singleton hacia uno Multiton. El cambio no fue costoso, dado que no se cambió radicalmente la estrategia pensada originalmente sino que se la extendió de 1 → N. Los patrones son muy parecidos y ofrecen la misma solución pero para diferentes cantidades.

# Logger API

Objetivo:

# Agregar la posibilidad de recibir una exception (Throwable) como parametro de la api.

# Ej: logger.info(‘un mensaje’, exception)

Implementacion:

Esta nueva solicitud implicó el agregado de nueva funcionalidad. Dado que nuestra clase se encontraba Clausurada ante cambios, pudimos incorporarla sin inconvenientes, adicionando código.

# Filtros

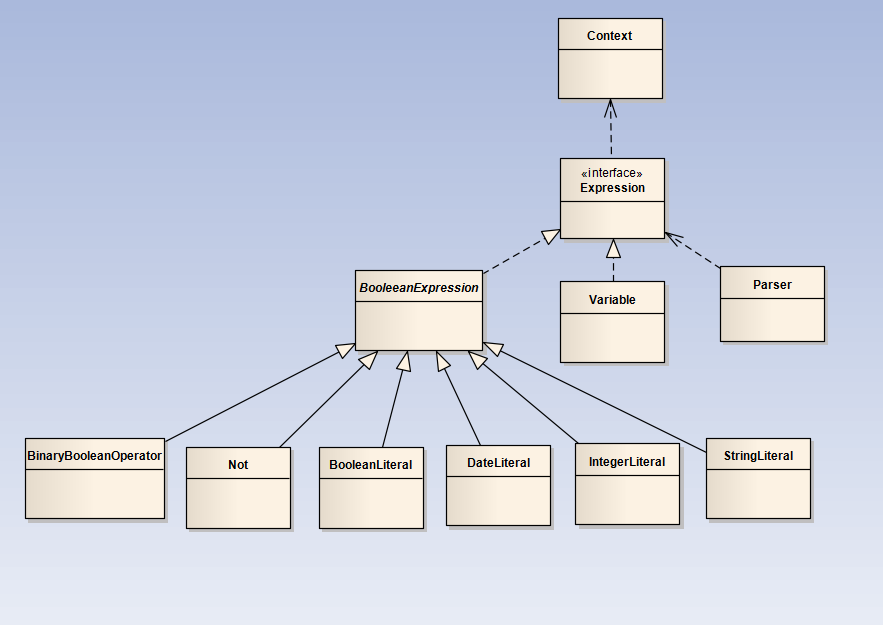
Objetivo:

# ● Permitir configurar un filtro a un Logger particular que filtre aquellos mensajes que cumplan con algun patrón de regular expression.

# ● Permitir a un usuario definir y configurar un Filtro Custom, que le permita decidir en base a todas las propiedades de un mensaje (contenido, nivel, fecha, etc) si lo desea loguear o no.

Implementación

Se optó por definir un lenguaje simplificado de consultas para que un usuario pueda definir una expresión lógica a ser evaluada por cada mensaje, y para evaluar dichas expresiones, se realizó una implementación del patrón ***Interpreter*** cuyo diagrama de clases se encuentra a continuación:



# Lenguaje:

* Operadores
  + OR (disyunción lógica)
  + AND (conjunción lógica)
  + NOT (negación lógica)
  + = (igualdad para números, strings y fechas)
  + > (mayor para números, strings y fechas)
  + >= (mayor o igual para números, strings y fechas)
  + < (menor para números, strings y fechas)
  + <= (menor o igual para números, strings y fechas
* Operandos posibles:
  + thread (nombre del thread)
  + line (número de línea)
  + message (contenido del mensaje)
  + method (nombre del método)
  + class (nombre de la clase)
  + date (fecha y hora en el que el mensaje fue logueado)
  + level (nivel con el que fue logueado el mensaje)
  + string literal (una cadena de caracteres cualquiera)
  + entero literal (un número entero cualquiera)
  + fecha literal (una fecha cualquiera formateada)
* Modificadores de precedencia
  + Se podrá modificar la precedencia de los operadores mediante el uso de paréntesis para crear expresiones anidadas

Parséo e interpretación

# Debido a que el patrón elegido resuelve de manera eficiente la evaluación de una expresión en notación polaca inversa y que consideramos que es mucho más sencillo para el usuario evitar esta misma, nos encontramos ante la necesidad de crear un parser que se encargue de analizar una expresión en notación cotidina y transformarla para que pueda ser interpretada por nuestra implementación sin problema. Para esto utilizaremos una implementación del algoritmo *Shunting Yard* creado por Edsger Dijkstra.

Esto último nos ha quedado pendiente y por eso no hemos podido aun incluir los filtros custom, sin embargo, ya contamos con una implementación del patrón que pasa con éxito todas las pruebas unitarias.

# Configuración

Objetivo:

# Permitir leer la configuración desde un archivo XML.

Implementacion:

Se desarrolló la clase ConfigParserXML encargada de esta labor. El impacto fue el esperado: nuevas funcionalidad agregando nuevo código sin modificar el existente.

# Inicialización

# Objetivo:

# La herramienta deberá intentar leer la configuración automáticamente de la siguiente manera:

# 1. Si existe el archivo logger­config.properties, leerlo y utilizar esa configuración.

# 2. Si existe el archivo logger­config.xml, leerlo y utilizar esa configuración.

# 3. Si no existe ningún de los dos anteriores, utilizar una configuración default (por ejemplo que se utilice siempre el mismo Logger, con nivel INFO y un formato de mensaje específico).

Implementacion:

Teníamos diseñado que al no encontrar el archivo de properties, genere uno por default. Lo que hicimos ahora fue incorporar en el medio de esa cadena el intento de búsqueda de un archivo XML de configuración. Para ello se creó la clase ConfigParserXML a la cual se le delega esa labor.

# Nuevo Formato

Objetivo:

# Se quiere poder formatear los mensajes a un JSON del siguiente estilo:

# {‘datetime’: ‘2001­07­04T12:08:56.235­0700’, ‘level’: ‘INFO’, ‘logger’: ‘LoggerName’ ,‘message’: ‘processing ...’}

Implementacion:

Para realizar esta implementación se decidió utilizar una librería, que fue agregada en pom.xml

<dependency>

<groupId>org.codehaus.jackson</groupId>

<artifactId>jackson-mapper-asl</artifactId>

<version>1.8.5</version>

</dependency>

Se creo una nueva clase JsonFormatter donde se implementó el método format que realiza el cambio de formato a tipo JSON.

# 

# Formato Mensajes (Pattern de la entrega anterior)

Objetivo:

# Agregar una nueva pseudo­variable:

# ● %g debería mostrar el nombre del Logger que emitió el mensaje.

Implementacion:

El impacto fue un poco más que el deseado, al decidir agregar el atributo nombre del Logger actual. Esto implicó el simple agregado de un nuevo formateador LogNameFormatter.java , pero también repercutió en la clase Message la cual ahora almacena dicho atributo.

# Destinos Custom

Objetivo:

# Permitir al usuario definir y configurar sus propios destinos.

# Ej: (pueden usar otra estructura)

# <...>

# <implementor>org.foo.bar.MailAppender</implementor>

# <param1>192.168.1.1</param1>

# <param2>juan­perez­123</param2>

# </...>

# y “org.foo.bar.MailAppender” es escrita por el usuario, no existe en la herramienta.

Implementacion:

# Pendiente.