**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

*Tutor*: 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 **contener la palabra clave en forma excluyente o no exclueyente.**

***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.

**Logueable:** Es la interfaz hacia el usuario de la aplicación hacia el usuario ya que le da acceso a los métodos que puede utilizar.

**Dispositivo:** Es una interfaces y el método imprimir es el que tienen que implementar las clases que usen esta interfaz, en nuestro caso hay dos clases que usan dispositivo Archivo y Consola, a futuro se pueden implementar nuevas clases.

**Archivo y Consola:** ambas utilizan Dispositivo, reciben un string e implementan distintas formas de procesar el mismo, ya sea imprimiendo por pantalla o guardar en archivo.

**OperadorDeDispositivos:** Su nombre es bastante descriptivo, posee dos métodos uno agregar dispositivo y el otro imprimir, guarda una lista de dispositivos y luego opera con ellos.

**Forma:** Es una interfaz y el método concatena es el que tienen que implementar las clases que usen esta interfaz, recibe un StringBuilder y devuelve el mismo. La idea es de tener la posibilidad de extender la funcionalidad del formateo del string que se imprime en los logs, las clases que la implementan son Hilo, Nivel y Tiempo, cada una con caracteristicas propias para darle forma al log.

**Formateo:** La responsabilidad de esta clase es la encargada de tomar un string y devolver el mismo con el formato que se va a imprimir, esta clase conoce a Forma ya que implementa una lista de formas de acuerdo, a las distintos formateos posibles, en caso de agregar nuevos tipos de formas, hay que extender desde forma y luego agregar en el constructor de Formateo para que implemente la misma. El método armar es el que toma el stringBuilder y lo devuelve con el formato correspondiente.Diagramas de clases

**DISEÑO DE LA SOLUCIÓN:**

En un principio nuestro análisis se inclinó hacia ver que los posibles cambios en la aplicación serian a partir que el log se iba a imprimir en otros lugares ademas del disco y de la pantalla, es por eso que nos inclinamos por buscar una solución de tener una interfaces llamada dispositivo que la tendrán que implementar la nuevas clases que se pueden agregar y esto no impactaría sobre el resto de la aplicación. Todo esto basado en los principios de buen diseño vistos a principios de la cursada y buscando que los cambios no impactarían en las clases clausuradas a cambios.

Por otra parte entendemos que nos quedo muy rígido todo lo relacionado con los niveles de logueo, es decir de agregar un nuevo tipo de nivel de log (DEBUG,INFO, WARN,ERROR,FATAL), esto nos impactaría en forma severa respecto al diseño definido. Por ejemplo patrones vistos en las últimas clases nos darían una mayor flexibilidad a la hora de encontrarnos con este tipo de cambio, por lo tanto este tipo de prácticas nos son muy útiles para poder ver que herramientas tenemos a la hora de decidir sobre nuevos diseños.

## Diagramas de Clases





## Detalles de implementación

config.properties y config.properties.xml

Partiendo de este archivo de muestra:

|  |
| --- |
| logLevel=WARN  archivo=/tmp/Aplicacion.log  formato=%d%p%t%g  aplicacion=TP1  loguer=Marcelo |

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Tests tipo="1">  <Config>  <LogLevel>INFO</LogLevel>  <Archivo>/tmp/AplicacionXML.log</Archivo>  <Formato>%d%p%t%g</Formato>  <Aplicacion>TP2</Aplicacion>  <Loguer>NICO</Loguer>  </Config> |

Levels

Completar implementacion

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **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.

**Principios de buen diseño destacados:**

* ***Inversión en la cadena de dependencia:*** Tanto forma como dispositivos ( Interfaces) son implementadas por las clases que indica el diagrama ya que estas clases poseen “métodos” comunes que son necesarias, en el caso de imprimir para dispositivo, independientemente de que sea un archivo, consola o registra. De forma análoga sucede con Forma y sus respectivas clases.
* ***Principio de Simple Responsabilidad:*** Debido a su bajo acoplamiento por ser clases con tareas especificas e independientes favorece su una mejor cohesión.
* ***Principio de Abierto-Cerrado:*** Este solución permite hacer frente a futuros cambios ya que de ser necesario se puede extender mediante uso de interfaces. Ej: Una clase “impresora” que implemente <dispositivo>, donde por ejemplo el método “imprimir” podría imprimir los log en una hoja.
* ***Principio de segregación de interfaces:*** La solución dada brinda solo lo que se necesita.

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

# 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 ­config.properties, leerlo y utilizar esa configuración.

# 2. Si existe el archivo config.properties.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:

completar

# 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 Logger.java , pero también repercutió en la clase Registrador 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.