Le 27 février 2015

Design Object

Framework de Logging

Equipe de projet :

Cécile Betmont

Adrien Frasca

Laura Hilliquin

**Explication de notre github**

Notre projet se trouve sur le répertoire Github à l’adresse   
<https://github.com/danteAl/Betmont_Frasca_Hilliquin_Architecture/commits/master>.

Ce travail a été réalisé par Cécile Betmont, Adrien Frasca et Laura Hilliquin.

Vous retrouverez les commits d’Adrien sous le nom de ‘DanteAl’, ceux de Laura sous le nom de ‘Hilliquin’ et ceux de Cécile sous le pseudo de cecile-betmont ou bien en tant que « unknown » (en changeant sa clé ssh, le répertoire github ne la reconnait plus en tant que cecile-betmont).

**Installation (Explique cette partie Adrien, je sais pas trop dans quelle format tu vas mettre la dernière version et je comprends pas comment un zip peut donner naissance à un jar, enfin je n’ai pas compris le truc de ROmain… ☺)**

Pour utiliser notre Logger, vous devez récupérer le dossier jar «» en effectuant un « git pull origin master » à partir d’un dosser git.

Dans un second temps, vous devez charger le dossier contenant notre Framework alors ainsi créé sur Eclipse. Ouvrez la classe « main » puis lancez le Framework.

**Fonctionnement du Framework**

Le Framework mis en place a pour but de récolter les différentes erreurs amenées par un programme et de les envoyer vers des cibles particulières, à un format spécifique. Le niveau et l’importance de ces erreurs sont classés par niveau.

Grâce au principe OCP, l’utilisateur de notre Framework se trouve en mesure de configurer ses logs à sa convenance, à partir d’un fichier « properties » ou bien directement de la console java.

Nous avons choisi d’introduire des paramètres par défauts, pour un utilisateur ne souhaitant pas configurer minutieusement son logger.

**Comment configurer son logger ?**

L’utilisateur instancie un logger spécifique à une classe en utilisant la méthode « getLogger » définie dans « LoggerFactory ». Son logger est alors lié à sa classe et stockée dans une liste présente au sein de la classe « LoggerFactory ». Pour chaque logger, l’utilisateur spécifiera, grâce aux méthodes de la classe « Configuration », le niveau qu’il veut donner à sa classe, le format de logs ainsi que chacune des cibles vers lesquelles il veut envoyer ses logs. Si l’utilisateur ne spécifie pas ces informations, par défauts, le niveau de logs sera en debug, la cible, un fichier d’extension .txt appelé « » enfin, le formateur sera de la forme : Wed Feb 25 19:29:20 CET 2015 [Name = Main, Level = DEBUG, Message = Test log1 ESIEA]

1. Ajouter un format de message

Pour cela, il se doit de créer son propre formateur : l’utilisateur crée sa propre classe héritée de la classe abstraite Formatter en redéfinissant la classe publique « format ».

L’exemple d’ajout d’un formateur dans notre classe main :

**private** **static** **class** MyCustomFormatter **extends** Formatter {

@Override

**public** String format(String message) {

StringBuffer sb = **new** StringBuffer();

sb.append("Prefix :\n");

sb.append(message + "\n");

sb.append("Suffix :\n");

**return** sb.toString();

}

Une fois le nouveau format spécifié, l’utilisateur l’ajoute aux configurations grâce à la la configuration java suivante:

Configuration.setLayout(MaClasse.class, new MyCustomFormatter())

1. La liste des cibles (handlers)

Nous avons conçu notre logger afin qu’il puisse gérer des cibles d’extension .txt simples et rotatifs.

Chaque logger est muni d’une liste de fichiers cibles. L’utilisateur est alors en mesure de configurer cette liste de fichiers. Il peut alors, grâce à la methode addHandler de la classe « Configuration », ajouter une cible en spécifiant le fichier texte à compléter :

Configuration.addHandler(Maclasse.class, new FileHandler(String fichier);

On peut aussi spécifier la volonté d’un fichier rotatif : une fois la taille du fichier dépassée 1Ko, notre système crée automatiquement un nouveau fichier dans lequel sera sauvegardé la suite des logs. Pour cela on utilise Configuration.addHandler(MaClasse.class, new FileRotateHandler(String fichier);

L’utilisateur pourra, de plus, spécifier la taille maximum de son fichier rotatif en utilisant le constructeur FileRotateHandler(String fichier, int tailleMax).

1. Le niveau des logs

Notre logger comporte les 7 niveaux différents suivants, d’importance croissante :

All, Debug, Info, Warn, error, fatal.

L’utilisateur spécifie, grâce à la méthode addLevel de la classe configuration, le niveau qu’il veut donner à ses logs.

1. Fichier Properties

Un fichier propertie est un fichier .txt de la forme suivante :

logger.com.esiea.logger.Test.level = INFO   
logger.com.esiea.logger.Test.formater = com.esiea.logger.FormaterDefault logger.com.esiea.logger.Test.handler = com.esiea.logger.Console  
logger.com.esiea.logger.Test. handler = com.esiea..HandlerRotateFile log.txt  
logger.com.esiea.logger.Test. handler = com.esiea.logger.HandlerFile file.txt

Ce dernier permet, au même titre que les configurations java, de configurer le framework. L’utilisateur n’aura qu’à importer ce fichier texte dans l’application.

Une fois le logger configuré, l’utilisateur n’aura qu’à appeler les fonctions principales permettant l’envoie du message aux format, niveau et vers les cibles spécifiques au logger : « logger.debug(String message) ».

**Document d’architecture**

1 - Diagramme UML

Voir document pdf « log\_uml » joint sur le fichier gitHub.

2 – Explication de l’architecture

Notre Framework de LOG s’organise autour de deux packages.

Le plus important, formé de 10 classes, com.esiea.logger s’occupe du fonctionnement de notre Framework. Le deuxième, appelé test, contient la classe test « main ». L’utilisateur peut configurer son Framework en passant par cette dernière.

Nous avons utilisés les trois principes de la programmation orientée objet :

* L’encapsulation
* L’héritage
* Le polymorphisme

Nous avons choisi de rendre chacun des attributs de nos classe « private » ou « protected » afin d’empêcher l’utilisateur de les modifier extérieurement. Grâce au principe OCP, il pourra configurer lui seul son Framework sans avoir besoin de modifier l’architecture du programme grâce aux fonctions créées à cette fin, déclaréees en « public ».

Les arguments et fonctions de protection « protected » ne sont utilisable que par les sous classe et les classe contenus dans le même package. Nous nous sommes servis de ces propriétés pour protéger nos attributs de l’extérieur mais permettre aux autres classes du même package d’y accéder.

Nous avons utilisés le principe d’héritage et de polymorphisme grâce à la mise en place d’une architecture basée sur des classes et méthodes abstraites. Elles nous permettent de redéfinir une même méthode partagée par plusieurs classes recherchant une fonctionnalité différente :  
La classe Handler, nous permettant de créer différentes cibles est définie en tant que classe abstraite composée d’une méthode abstraite message(). Chacune de ses cibles héritent de cette clase abstraite et redéfinisse la méthode message() : cette dernière prendra par la suite une forme différente suivant la cible utilisée.  
De la même façon, chacun des formatter que l’utilisateur voudra ajouter au framework héritera de la classe Formatter et redéfinira la méthode format contenue dans cette dernière.

Pour créer une instance de Logger spécifique à une classe, nous utilisons le principe de Liskov grâce à une classe extérieur LoggerFactory qui instancie chacun des Loggers.

En ce qui concerne les différents niveaux gérés par le Logger, nous avons décidés de sécuriser l’application en implémentant une classe de type énumération. Notre Framework ne gérant que 6 niveaux différents, l’utilisateur n’est alors en mesure d’utiliser que ces derniers.

Grâce à cette architecture logicielle, notre Framework semble bien respecter toutes les contraintes imposées par le sujet.