PROJET 4A

Design Object

Framework de Logging

Equipe :

Cécile BETMONT

Adrien FRASCA

Laura HILLIQUIN

GitHub

Notre projet se trouve sur le répertoire GitHub à l'adresse :

https://github.com/danteAl/Betmont_Frasca_Hilliquin_Architecture

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').

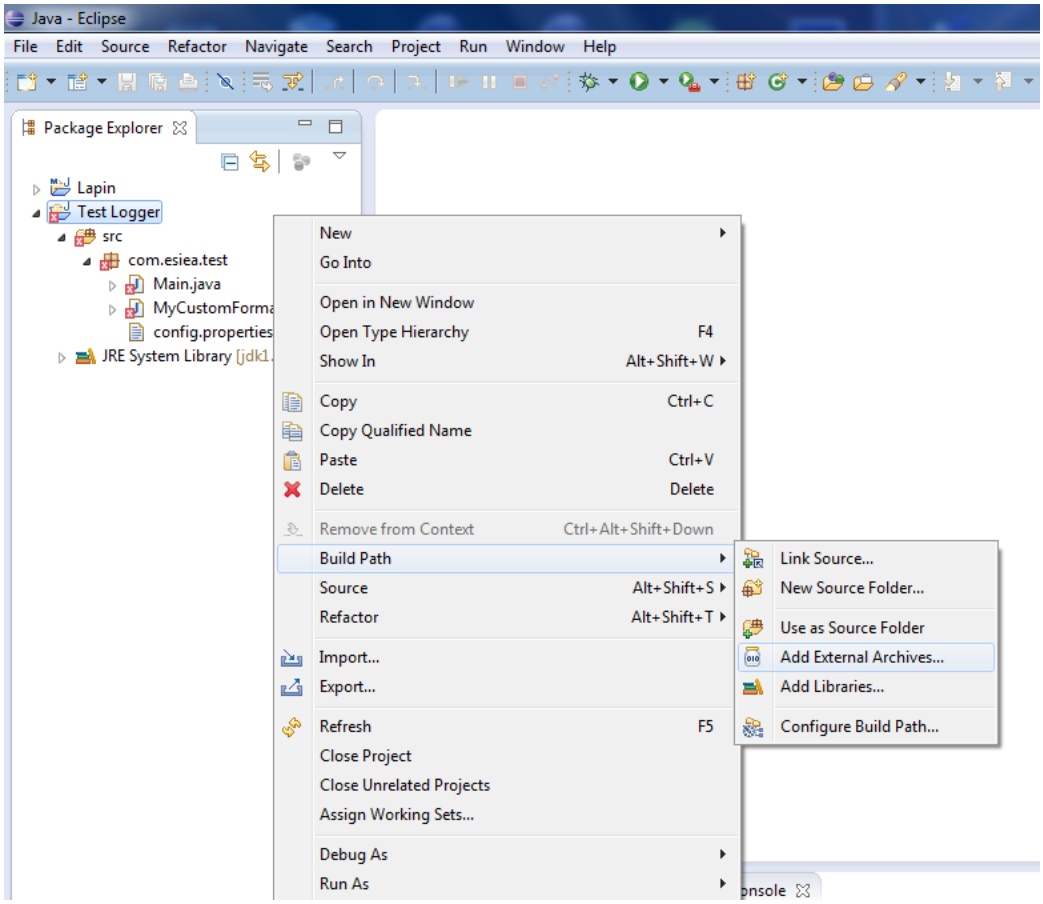
Installation

Vous trouverez sur notre répertoire GitHub plusieurs dossiers :
- Le fichier README
- Le fichier annexe du diagramme UML
- Notre dossier de travail Eclipse 'Projet Logger'
- Un dossier de test 'Test Logger' à importer sur Eclipse pour tester notre projet
- Le fichier logBFH.jar qui est ni plus ni moins que notre framework

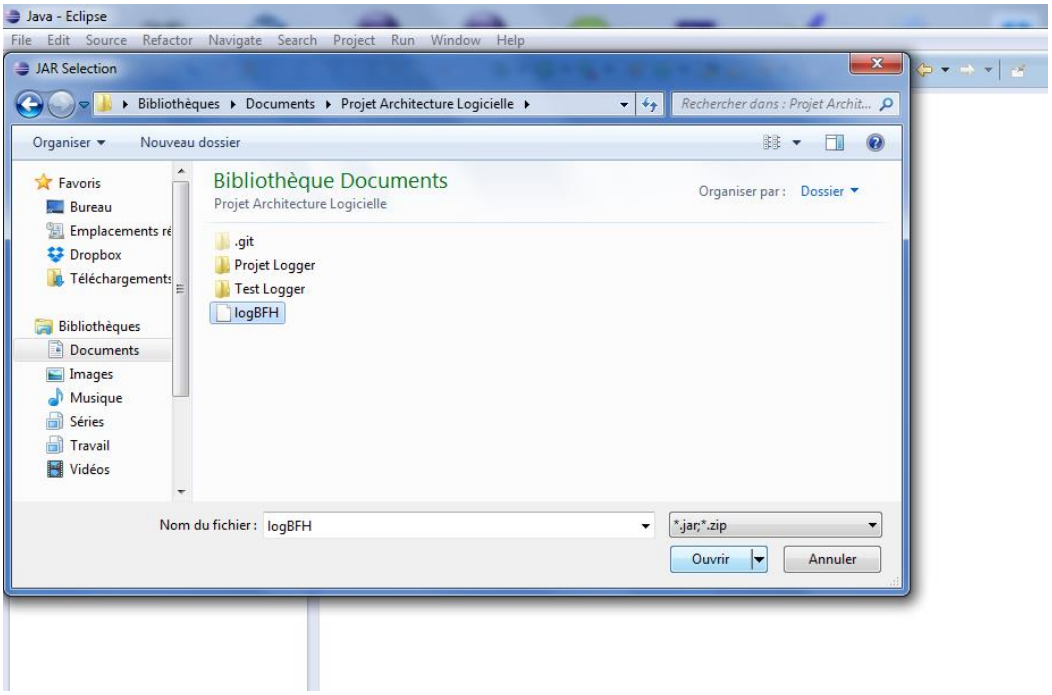
Pour procéder à l'installation de notre framework, télécharger avant tout le dossier **'Test Logger'** et le fichier **'logBFH.jar'** de notre répertoire GitHub.

Importer ensuite le projet **'Test logger'** sur Eclipse. Ce projet contient une classe Main afin de tester notre framework avec différente méthode, une classe **MyCustomFormatter** qui est un exemple de format réaliser pour les log et un fichier **config.properties** qui est un exemple d'utilisation d'un fichier properties pour notre framework (voir détails **d) Fichier Properties**).
Notre projet importé contient actuellement des erreurs mais cela est normal car nous n'avons pas encore importé notre framework à ce projet.

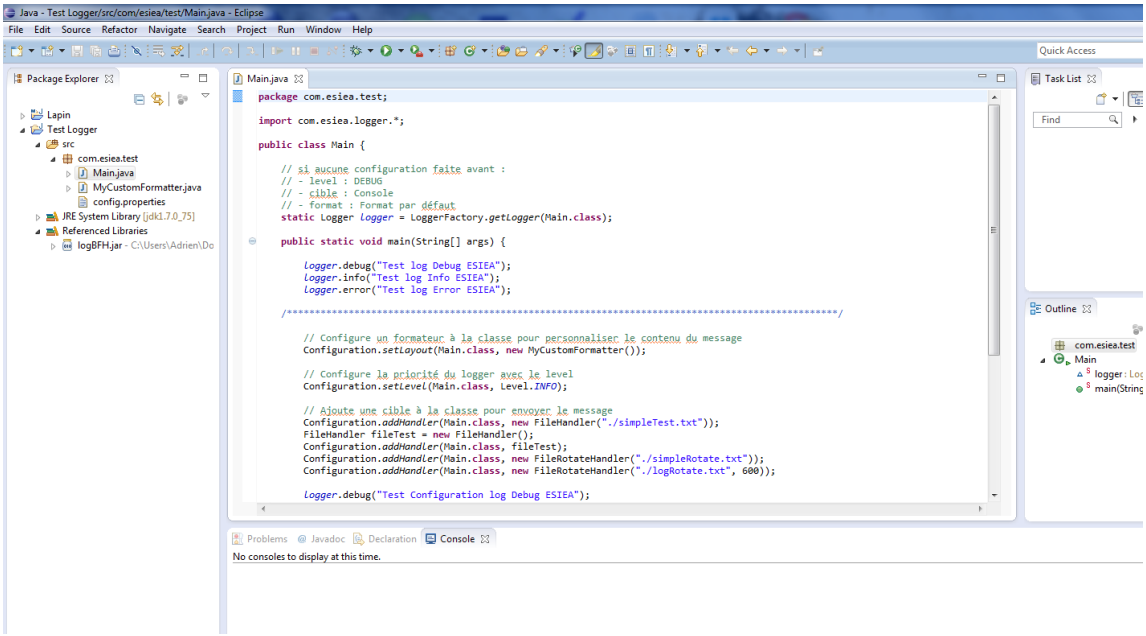
Pour réaliser cela, il suffit de faire un **clic droit** sur projet, sélectionner **'Build Path'** et sélectionner **'Add External Archives…'**



Ensuite on sélectionne notre fichier **'logBFH.jar'**



Après avoir cliqué sur ouvrir, on constate que l'importe à bien été effectué et que notre projet ne contient aucune erreur.



Si le projet contient toujours des erreurs c'est que la dépendance de notre version java (jre 1.7.0_75) n'a pas été faite. Il vous suffit pour cela d'importer votre version java au projet (clique droit -> build path -> add librairies -> jre System librairy).
Cette opération peut être réalisée sur n'importe quel projet java Eclipse

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 Framework ?

L'utilisateur instancie un logger spécifique à une classe en utilisant la méthode « getLogger » définie dans « LoggerFactory » : Logger logger = LoggerFactory.getLogger(MaClasse.class);
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 sera la console et enfin, le formateur sera de la forme : Le 27/02/2015 à 19:29:20 [NAME= com.esiea.test.Main, LEVEL=DEBUG, MESSAGE= Test log ESIEA]

a) 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("Message du fichier :\n");
                sb.append(message + "\n");

                return sb.toString();
        }

}
```

Une fois le nouveau format spécifié, l'utilisateur l'ajoute aux configurations grâce à la méthode suivante:

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

b) La liste des cibles (handlers)

Nous avons conçu notre framework 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, grâce à la méthode *addHandler* de la classe « Configuration », ajouter une cible en spécifiant le fichier texte à compléter :

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

Si aucun argument n'est passé en paramètre du constructeur, *new FileHandler()*, cela créera un fichier texte par défaut « loggers.txt » ce trouvant à la racine du projet.

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 :

Configuration.addHandler(MaClasse.class, new FileRotateHandler(String path));

L'utilisateur pourra, de plus, spécifier la taille maximum de son fichier rotatif en utilisant le constructeur :

FileRotateHandler(String fichier, int tailleMax);

Si l'utilisateur souhaite supprimer une de ses cibles, il peut utiliser la méthode suivante :
Configuration.removeHandler(MaClasse.class, Handler nomCible);

c) 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 sa classe :

Configuration.setLevel(MaClasse.class, Level.WARN);

d) Fichier Properties

Un exemple de fichier .properties est de la forme suivante :

```
logger.com.esiea.test.Main.level = DEBUG
logger.com.esiea.test.Main.formatter = com.esiea.test.MyCustomFormatter
logger.com.esiea.test.Main.cible1 = com.esiea.logger.FileHandler,./log.txt
logger.com.esiea.test.Main.cible2 = com.esiea.logger.FileHandler
logger.com.esiea.test.Main.cible3 = com.esiea.logger.FileHandler,./test.txt
logger.com.esiea.test.Main.cible4 = com.esiea.logger.FileRotateHandler,./testRotateDefault.txt
logger.com.esiea.test.Main.cible5 = com.esiea.logger.FileRotateHandler,./testRotate.txt,500
#logger.com.esiea.test.Main.cible6 = com.esiea.logger.ConsoleHandler
```

Ce dernier permet, au même titre que les configurations java, de configurer le framework. L'utilisateur n'aura qu'à importer ce fichier .properties grâce à la méthode *getProperties* en passant en argument le chemin du fichier .properties :

```
Configuration.getProperties(String path);
```

Le format d'une ligne de commande dans un fichier .properties est très spécifique. Initialisation de l'Attribut de MaClasse avec l'Objet, et les Argument1 et Argument2 de son constructeur si besoin (séparé par **une virgule**) :

```
logger.MaClasse.Attribut = Objet
logger.MaClasse.Attribut = Objet,Argument1
logger.MaClasse.Attribut = Objet,Argument1,Argument2
```

Il y a 3 types d'attributs prédéfini :
- level : configure le level d'un logger
- formatter : configure le format d'un logger
- cibleX : configure la cible où sera envoyé le message

Une fois le framework configuré, l'utilisateur n'aura qu'à appeler les fonctions principales permettant l'envoie du message aux format, niveau et cibles spécifiées :

```
logger.debug(String message)
logger.error(String message)
```

Document d'architecture

1 - Diagramme UML

Voir document pdf « Log UML» joint sur le fichier GitHub.

2 - Explication de l'architecture

Notre Projet Logger 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 le 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.