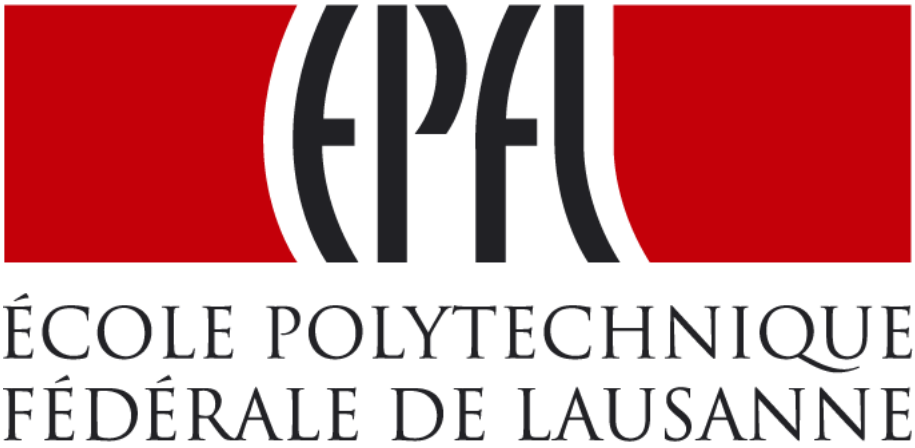
Information Technology Project

Conception



eBlast

Group 39 - Dieulivol David & Denoréaz Thomas

Academic Year 2010-2011

(May 26, 2011)

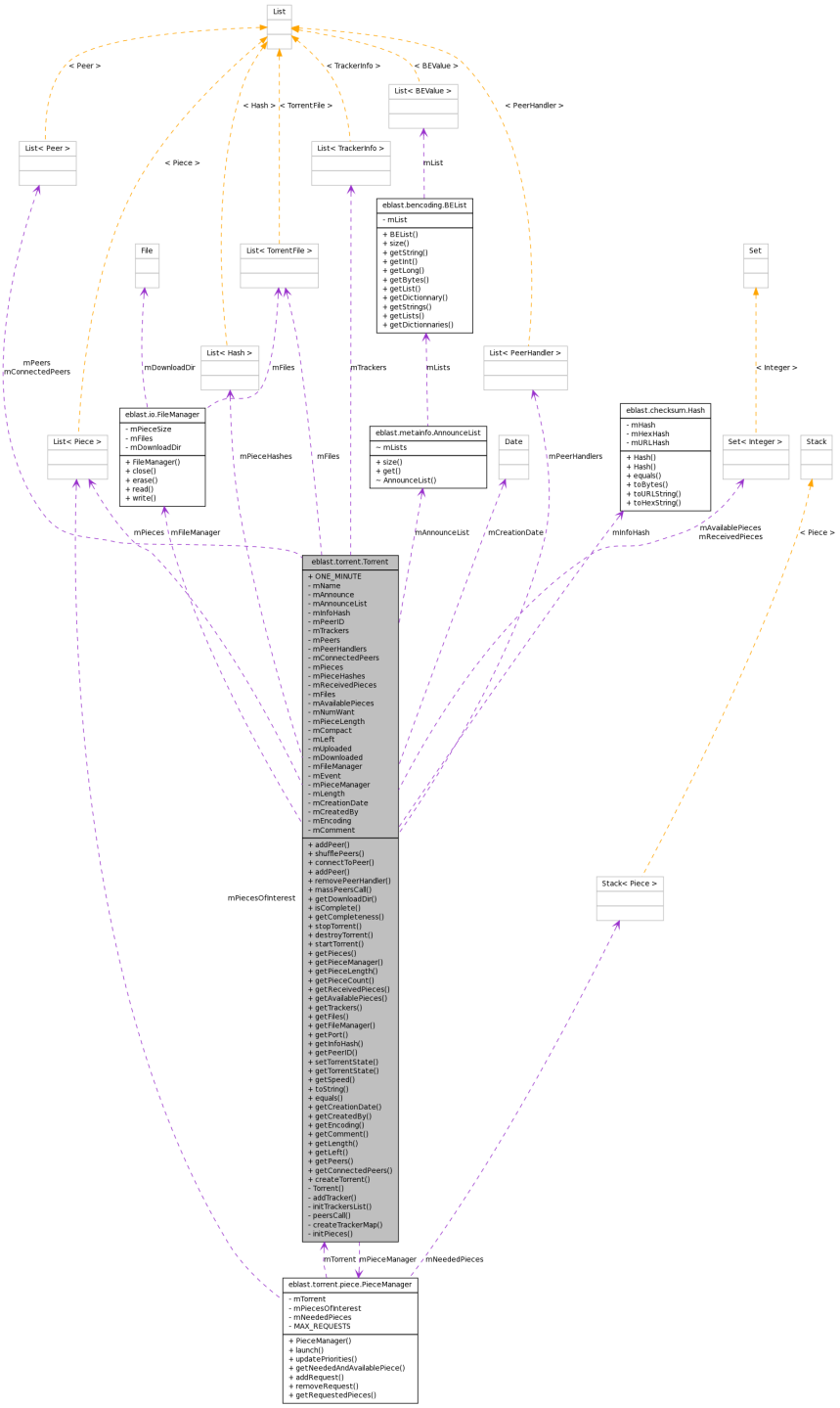
Contents

1 Package eblast

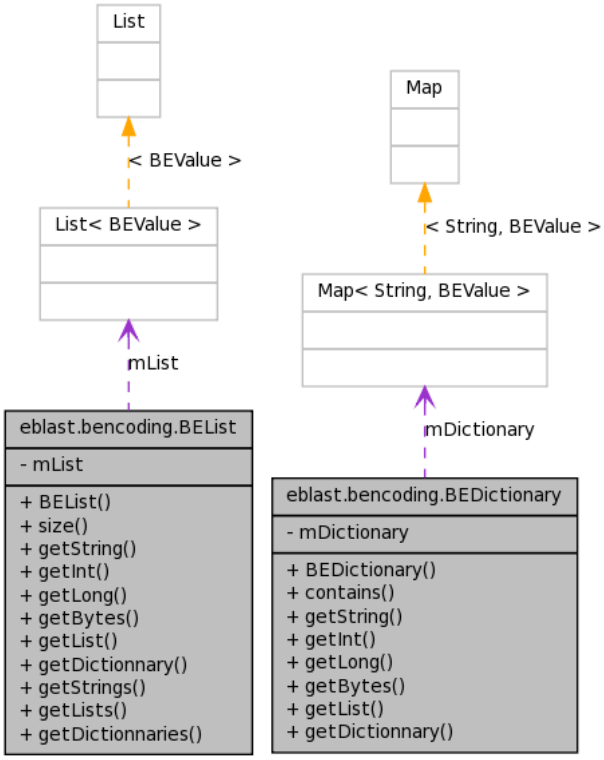
1.1 Concept

La classe utilitaire Convertor, comme son nom l'indique, nous permet d'effectuer des conversions de toutes sortes. Nous pouvons, par exemple, convertir un tableau de bytes en hexadécimal (et inversement), nous gérons également l'affichage correct des unités (par exemple au lieu d'avoir 30'000 KB nous aurons 30MB affichés).

La classe EBlast est la classe qui se charge du "look and feel" de l'interface ainsi que du lancement de la méthode main(). Elle fait également le lien entre le moteur du client et l'interface utilisateur.



2 Package eblast.bencoding



This package contains all the provided class to encode/decode and manage the BEncoding process.

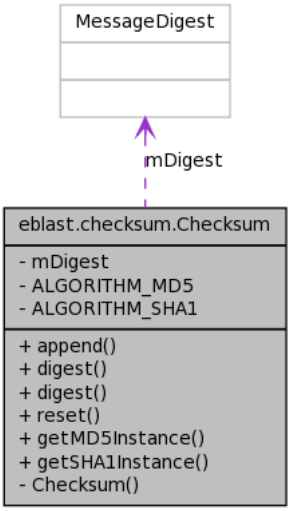
2.1 BEDictionary.java

This class encapsulate a *BEValue* and allows it to get all values inside this *Dictionary*. It has been created to counter the fact that we have to call two times each method to get a value. It makes the code clearer.

2.2 BEList.java

As for the *Dictionary*, this class do the same for a List of *BEValue*.

3 Package eblast.checksum

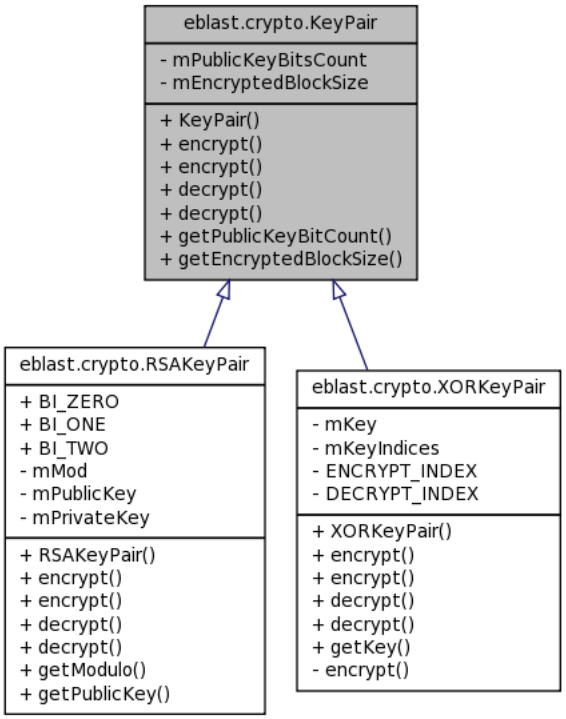


3.1 Concept

La classe Hash est une classe utilitaire permettant la manipulation aisée d'un hash. Notamment, nous pouvons à l'aide de cette classe retourner un hash dans un tableau de bytes, une String ou dans une chaine hexadécimale.

La classe utilitaire Checksum nous permet de générer un checksum en choisissant l'algorithme à l'aide des méthodes statiques getMD5Instance() et getSha1Instance(). Elle nous permet donc d'avoir un seul objet pour générer des checksums indépendamment de l'algorithme utilisé.

4 Package eblast.crypto

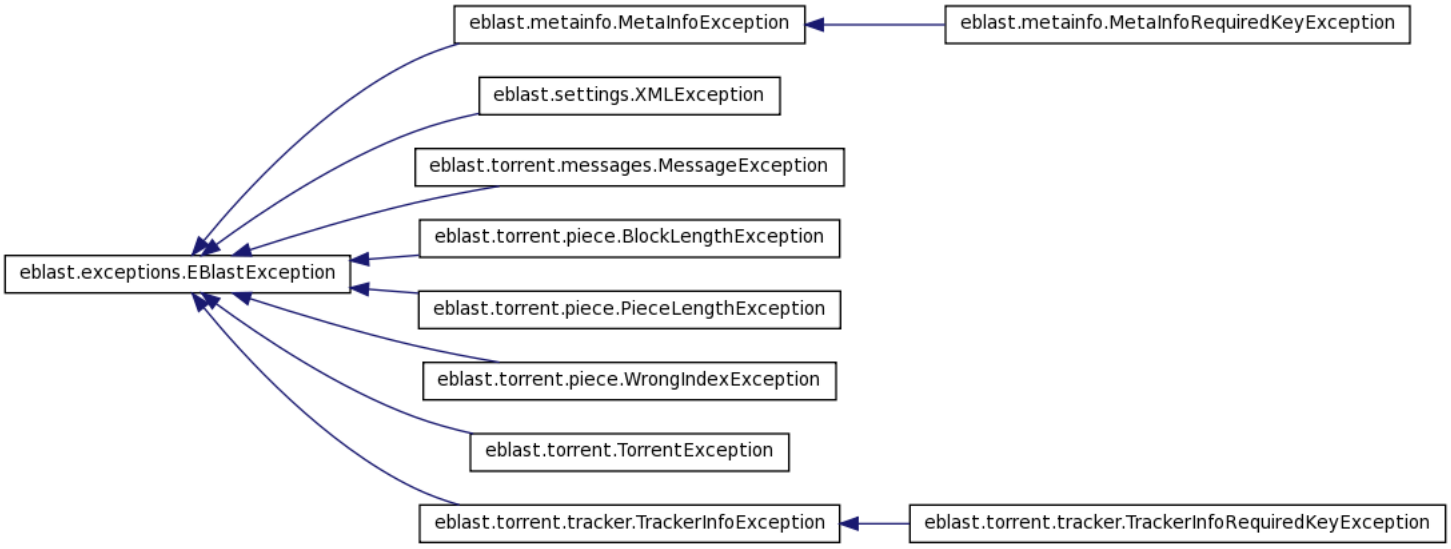


4.1 Concept

Ici une des majeures différences avec le projet proposé est que nous avons choisi de créer une classe abstraite Keypair avec une méthode d'encryptage et une autre de décryptage, toutes les deux indépendantes de l'algorithme utilisé. Nous avons donc dérivé de cette classe les classes RSAKeyPair et XORKeyPair.

Dans la pratique, nous utilisons la classe KeyGenerator pour créer un "Keypair" RSA ou XOR, selon les besoins.

5 Package eblast.exceptions



5.1 EBlastException

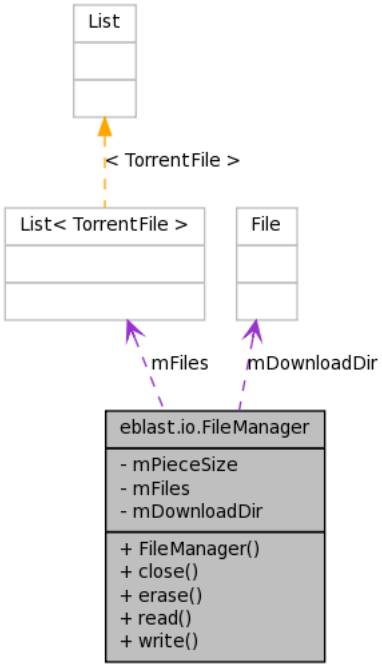
Cette Exception mère a été créée dans la continuité du système de log. Toutes nos exceptions "maison" héritent de cette EBlastException et l'avantage premier est qu'en appelant le constructeur de la classe parent (càd EBlastException) via super(message), cette exception sera automatiquement loggée dans notre système.

6 Package eblast.http

6.1 HTTPGet

This class perform a HTTP Get Request, it will return an *InputStream* connected to the client. It is also possible to download the result into a file.

7 Package eblast.io : Nouveautées

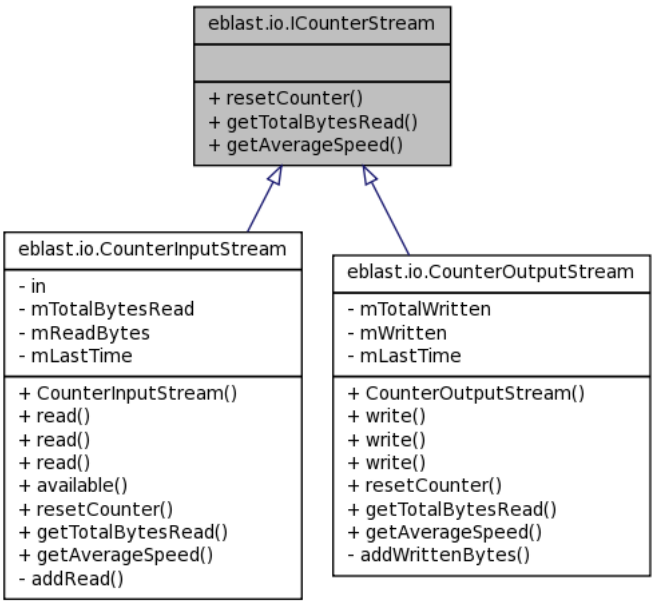


7.1 FileManager

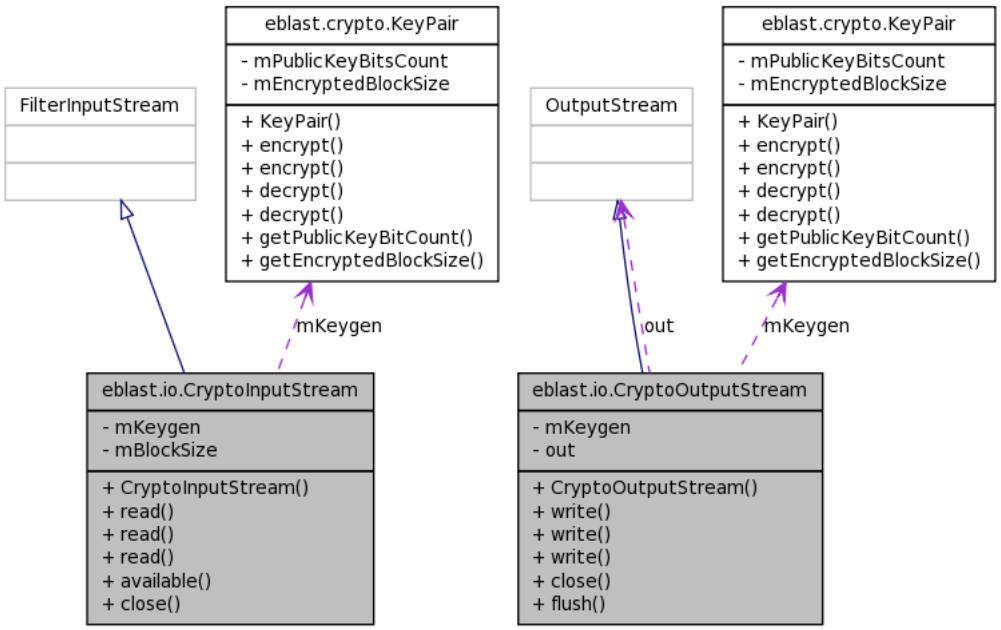
Cette classe est l'intermédiaire principal entre le système de fichiers et notre programme. Elle nous permet de lire et d'écrire des blocs dans un fichier sur la machine hôte. Cette classe est d'autant plus importante par le fait que nous avons choisi de ne pas stocker les blocs en mémoire mais de directement les écrire sur le système de fichiers.

Dans ce package nous avons quelques flux qui peuvent s'avérer très utile :

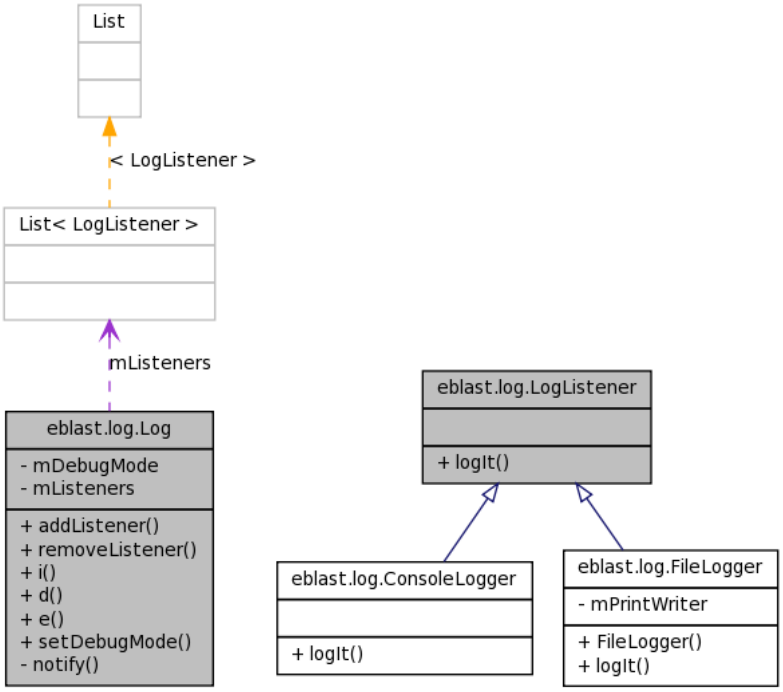
- CounterInputStream/CounterOutputStream vont respectivement compter le débit entrant et le débit sortant de données. Ceci s'avère très pratique pour déterminer la vitesse de téléchargement.



- CryptoInputStream/CryptoOutputStream : La classe abstraite Keypair du package eblast.crypto nous permet d'avoir un niveau supplémentaire d'abstraction. Grâce à cette classe nous avons créés des flux d'entrée/sortie uniques qui supporte plusieurs types de cryptographie (nous avons implémentés RSA et XOR). Une seule classe pour une infinité d'algorithmes, là est la clé de l'abstraction avec KeyPair.



8 Package eblast.log



8.1 Concept

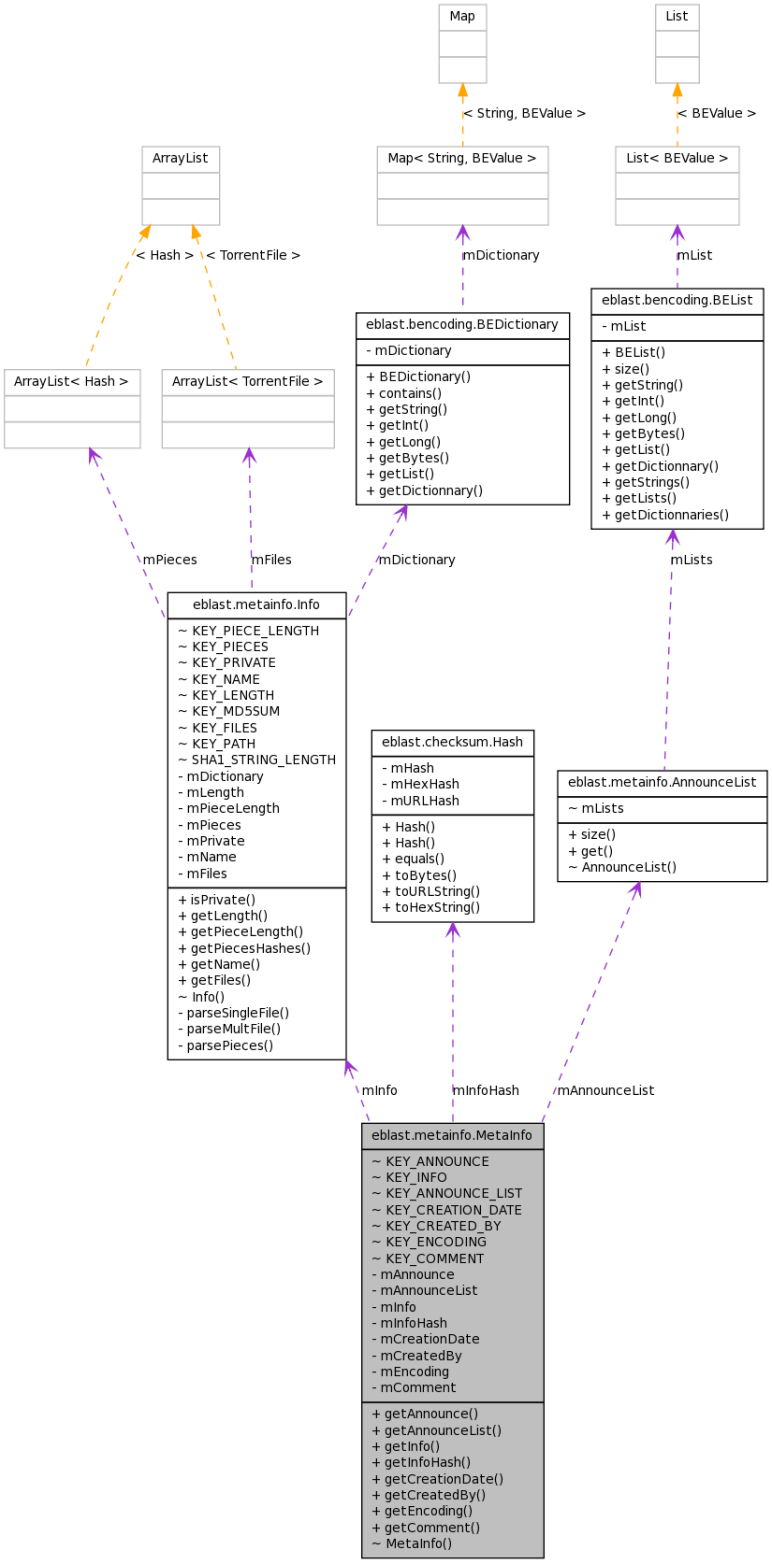
Plusieurs classes ont été créées dans le but de simplifier l'affichage des messages. Cette architecture est basée sur le Design Pattern Observer. Plusieurs idées ont été apportées :

1. Tout d'abord nous séparons les messages en trois catégories classiques : Informations, Debug et Erreurs.
2. Nous ne sommes pas limités à afficher le texte dans la console. Grâce à l'interface LogListener, toutes les classes implémentant cette interface pourront afficher les messages de la façon dont elles le souhaitent. Cette interface nous demande d'implémenter la méthode logIt(), qui pourrait s'implémenter de plusieurs façons : message dans la console, Ecriture dans un fichier, dans une interface, etc.
3. La classe Log référence toutes les classes souhaitant reçevoir des messages de log. Elle contient trois méthodes statiques :i(), d(), e().

Imaginons que nous souhaitions écrire le message "Bonjour" sur tous les canaux de log, ceci se passe de la façon suivante :

1. Log.i("Tag", "Bonjour")
2. cette méthode va appeler pour toutes les classes souhaitant reçevoir l'information la méthode logIt()
3. Toutes les classes ont à présent reçu le message.

9 Package eblast.metainfo



9.1 AnnounceList

The MetaInfo contains a list of lists of strings that represents the announce-list, because this is unusual, we decided to create a class to encapsulate this list, and then be able to get the list we need.

9.2 FileDictionary

In multiple-file implementation, each file contains a *BEDictionary*, *FileDictionary* is there to encapsulate this one, and return a TorrentFile with the informations.

9.3 Info

Info is a *BEDictionary* that contains all informations about the file(s) of the torrent. It contains the following fields:

- **piece length** (*integer*): Number of bytes in each piece
- **pieces** (*Byte String*): String consisting of the concatenation of all 20-byte SHA1 hash values, one per piece

And these are the optional fields:

- **private** (*0 or 1*): It defines whether the torrent is private or not.

9.4 MetaInfo

This is the main *BEDictionary*, it contains:

- **info** (*BEDictionary*): Described above.
- **announce** (*String*): URL of the Tracker

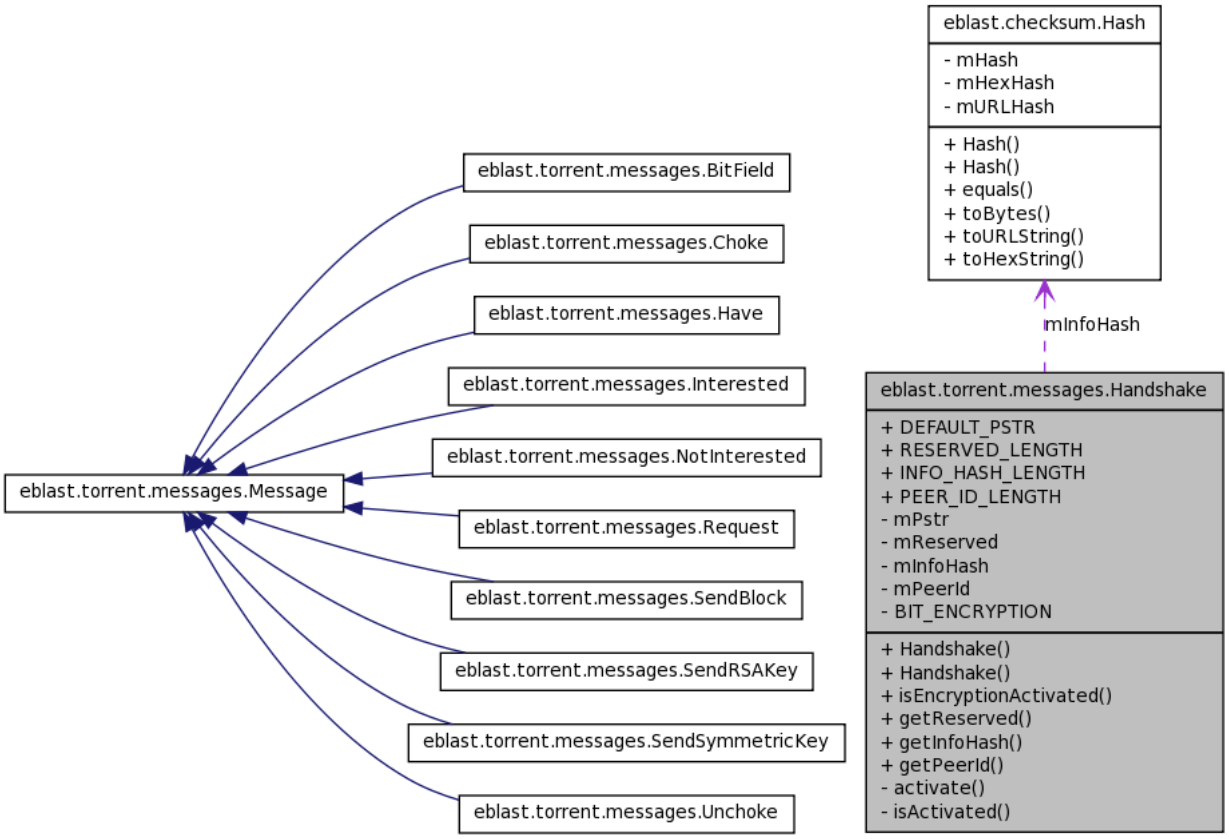
And these are the optional fields:

- **announce-list** (*AnnounceList*): Described above.
- **creation date** (*integer*): Creation time of the torrent in UNIX format
- **comment** (*String*): Comment about the torrent.
- **created by** (*String*): Who created this torrent.
- **encoding** (*String*): String encoding format used to generate the pieces (inside the *Info* dictionary).

9.5 MetaInfoReader

This class allows to create an instance of a MetaInfo object.

10 Package eblast.torrent.messages

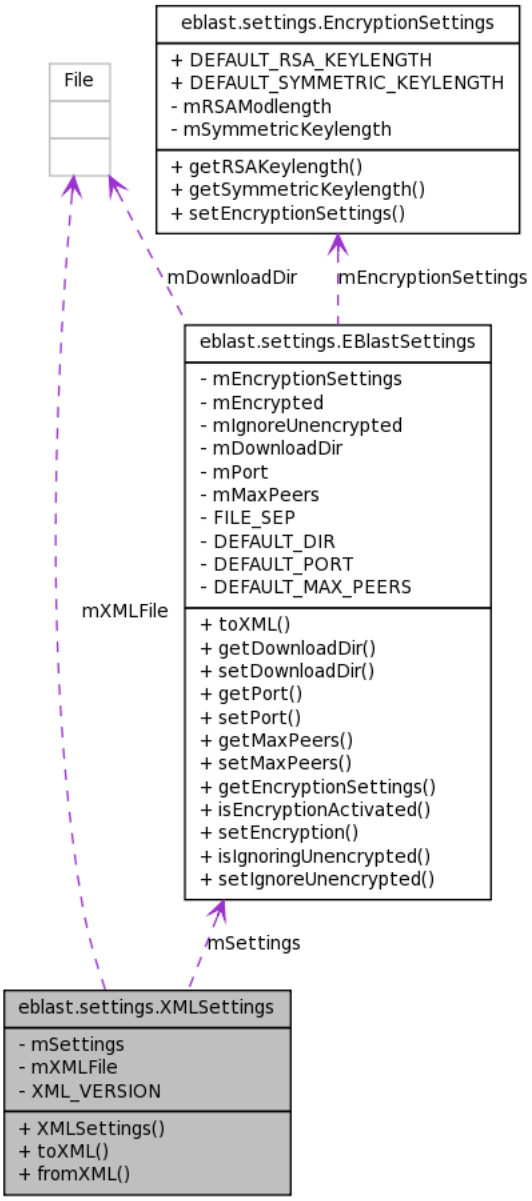


10.1 Concept

Nous avons suivi le Design Pattern Visitor pour cette étape de façon classique. nous avons aussi créé, pour plus de simplicité à l'utilisation, des flux MessageInputStream/MessageOutputStream qui vont respectivement pouvoir lire/écrire des Messages et des Messages Handshake dans un flux de données passé en paramètre à ces dernières.

La méthode readMessage() de MessageInputStream appelle directement FactoryMessage pour lire le message contenu dans un flux de données. Il en découle que la classe MessageReader n'avait plus aucune utilitée : nous l'avons donc supprimée.

11 Package eblast.settings



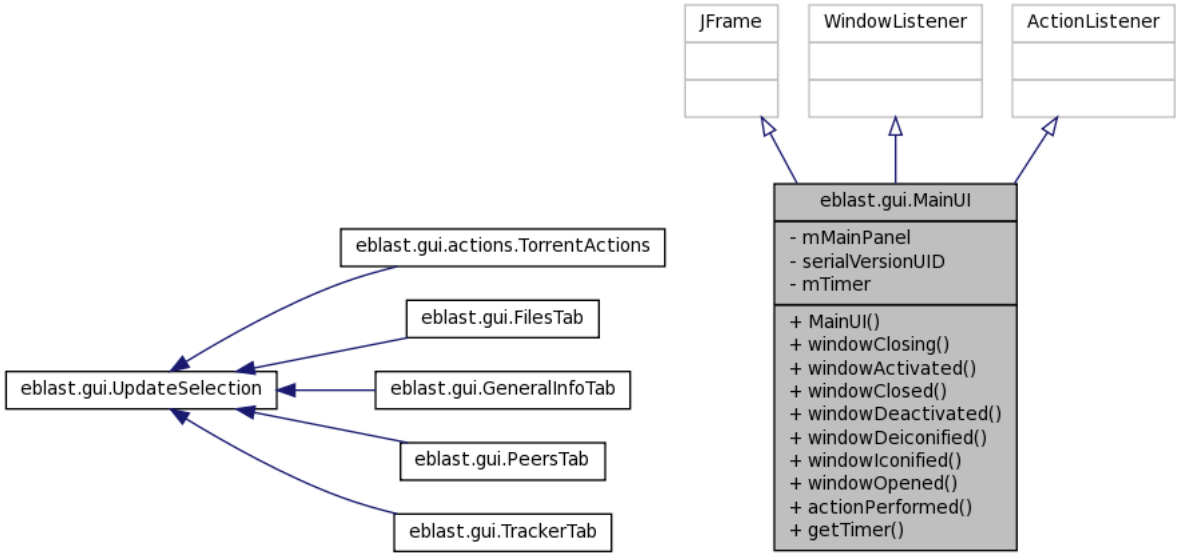
11.1 Concept

Ici nous avons introduit un moyen centralisé de configuration de notre application. Tout ce qui concerne la configuration du programme (hors interface utilisateur) est stocké dans la classe EBlastSettings.

Pour la persistence des données, nous avons aussi choisi de garder les données de configuration sur le système de fichiers du système hôte dans un fichier XML. Pour l'écriture en XML des données de configuration et pour le "parsing" d'un fichier de configuration XML nous utilisons la classe XMLSettings. En revanche, pour interpréter les données "parsées" nous utilisons la classe XMLHandler.

Pour finir la classe EncryptionSettings contient tous les paramètres concernant le cryptage dans notre application.

12 Package eblast.gui



12.1 Concept

L'interface est composée de deux parties : celle du haut contient tous les torrents en cours de téléchargement et celle du bas contient des informations relatives aux torrents sélectionnés dans la partie du haut.

D'un point de vue conceptuel, trois points sont à mentionner :

1. Un timer rafraichit un bon nombre de fenêtres à une durée fixe.
2. La partie du bas (infos sur le(s) torrent(s) sélectionné(s9) doit-être informée en tout temps de quels torrents sont sélectionnés, c'est pourquoi la classe ObserversTorrentList a été développée : elle permet à une classe souhaitant connaître les changements de sélection de torrents dans la Liste d'en être avertie instantanément. Comme son nom le suggère, cette classe se base sur le principe du Design Pattern Observer.
3. La classe TorrentActions nous permet de stocker des "actions" (telles que "ouvrir boite de dialogue", "mettre en pause un torrent", "supprimer un torrent") qui peuvent être appelées par n'importe quel composant (un bouton, un menu, un clic de souris) car ces actions sont basées sur le principe des ActionListeners.