Structure du projet :

/analyser (source) :

Analyser.java : Définit la classe Analyser qui s'assimile à une liste de plugins à lancer, et une seule Configuration non mutable réglant l'ensemble des plugins. Cela fonctionne car Configuration est elle-même assimilable à un dictionnaire qui associe à un nom de plugin sa configuration (cf Configuration.java). Le but sera principalement d'appeler chaque plugin avec la bonne configuration, de stocker les résultats de chaque, et de les retourner sous une forme AnalyserResult appropriée.

AnalyserPlugin.java : C'est une interface qui donne la structure générale des plugins : des méthodes de lancement, des getters pour les résultats de type général Result (elle-même une sous-interface de AnalyserPlugin)

AnalyserResult.java : S'assimile à la liste des résultats de chaque plugin contenu dans la Configuration de l'Analyser (en tant que clés de dictionnaire). Est muni de méthodes pour la récupération de résultats et leur conversion en String ou en affichage HTML. Ces méthodes de conversion sont complexes et nécessitent que quelqu'un les explique.

CountCommitsPerAuthorPlugin.java : C'est un des plugins implémentant la structure générale de plugin offerte par l'interface AnalyserPlugin. Celui-ci est un exemple (donc à copier à volonté) qui s'occupe de compter les commits effectués par chaque auteur du projet. Il créera un dictionnaire qui à chaque utilisateur associe son nombre de commits, il est muni de méthodes de conversion String / HTML comme prévu, ainsi que d'une méhode processLog() auxiliaire qui épluche un log gitlab pour y recueillir l'information voulue.

/analyser (tests) :

TestCountCommitPerAuthorPlugin.java : Un test pour le plugin précédent. Lit 20 commits qu'elle attribue tour à tour à trois auteurs foo, bar et baz, et renvoie le nombre d'auteurs total, le nombre de commits par utilisateur, la somme de leurs commits, etc. Le but est de voir si les données obtenues sont bien celles attendues.

/cli (source) :

CLILauncher.java : crée un nouvel Analyser et va s'occuper de le remplir, lancer ses plugins, et afficher ses résultats, à partir d'une configuration lue depuis la ligne de commande. Pour cela, utilise la méthode makeConfigFromCommandLineArgs(String[] args) qui lit la ligne de commande et la décompose en arguments. Des ajouts attendus ici sont de créer nos propres options, d'autoriser l'import de fichier de config déjà existant ou de sauvegarder celui fourni (mais ça, c'est mon taff) à l'aide par exemple de Scanner(File f) et d'une méthode capable de générer une Configuration à partir d'un fichier texte brut passé en argument. À voir si une telle méthode a plus sa place dans Configuration ou dans CLILauncher.java, je pencherais plus pour Configuration.java : on aurait alors un constructeur à partir d'un simple fichier (ou mieux, d'une String donnant son chemin d'accès). Dispose d'un support d'aide primaire qui indique quand une commande est incorrecte (mais on peut mieux faire…). L'affichage des résultats est évidemment aussi important, puisque la coordination des sorties HTML des différents plugins peut ne pas être évidente.

/cli (tests) :

TestCLILauncher.java : Ce test tente de créer un CLILauncher, pour voir si la configuration est bien lue, et en demandant de compter les commits. Tente ensuite avec une option volontairement invalide. Les possibilités ici sont multiples, regarder les résultats avec des options correctes, tenter un import / export de sauvegarde de configuration, tester le support d'erreur, etc.

/config (source) :

Configuration.java : dispose d'un chemin d'accès non mutable vers le dépôt git, ainsi que d'un dictionnaire qui associe à un nom de plugin sa configuration de forme PluginConfig.

PluginConfig.java : interface vide pour le moment. Et c'est là le principal problème. Tout se lance bien sans config, mais un jour les plugins auront besoin d'une structure avec arguments, options variables. Ce qui signifie créer une structure, mais que garder en cas général (dans l'interface PluginConfig), ou en implémentation de l'interface ? (propre à chaque plugin, sauf si l'on crée des classes de plugin). Problème principal, mais qui viendra lorsque l'on commencera à créer nos propres plugins.

/config (tests) :

TestConfiguration.java : Test vide pour le moment, ce qui fait sens puisque les plugins n'ont pas encore de configurations différentes implémentées. Encore une fois, problème central mais qui viendra au fur et à mesure que l'on créera nos propres plugins.

/gitrawdata (source) :

Commit.java : Ce fichier implémente la classe Commit (ses attributs sont globalement les infos standards pour décrire un commit). Elle fournit également un outil parseLogFromCommand(Path gitPath) qui découpe les résultats de « git log » (console) en une liste de commits.
Détails rapides sur les nombreux types utilisés : ProcessBuilder est un constructeur de commande shell, qui prend en argument tous les mots de la commande et a un attribut directory pour savoir où être exécuté. Il est ensuite lancé sous forme de Process. Si la commande réussit, le stream d'output engendré par le processus (de type InputStream) est lu par un BufferedReader (qui semble être l'équivalent d'un Scanner), et est parsé par la fonction parseLog(BufferedReader reader) qui continue d'appliquer parseCommit sur le reader tant que commit.isPresent() (c'est-à-dire que commit est non vide, différent de Optional.empty()), après avoir récupéré la sortie de cette fonction (c'est-à-dire un commit finalisé, correctement construit).

Décrivons maintenant la fonction parseCommit(BufferedReader input), qui découpe le reader de la façon suivante : Si le reader est vide, termine en renvoyant un Optional.empty() (permettant à parseLog de terminer). Sinon, lit une ligne, la découpe selon le regex espace, vérifie qu'elle commence bien par « commit » sinon renvoie une erreur (par parseError() qui n'est rien qu'une fonction renvoyant une erreur), et construit un commit nommé builder, via CommitBuilder, à partir du deuxième élément de la ligne (découpée via espace). Le programme lit alors chaque ligne suivante, et l'identifie à une forme a:b qui lui donnent un attribut (ici a, qui sera Author, Merge, Date, etc, ignoré s'il n'est pas reconnu) et sa valeur respective b. Chacun des attributs est modifié dans builder via un setter. Ensuite, le programme s'arrête, ayant rencontré une ligne vide, et récupère toutes les suivantes, les concaténant pour former la description du Commit construit par builder. Le programme renvoie ensuit un Optional contenant le Commit de builder enfin construit.

Enfin, la classe implémente une fonction toString() qui renvoie la liste des informations du Commit.

CommitBuilder.java : Définit les attributs que prendra le builder (littéralement les mêmes que ceux d'un commit), dispose de setters pour éditer ces attributs, et d'un createCommit() qui construit un nouvel objet Commit de même attributs que le builder.

/gitrawdata (tests) :

TestCommit.java : Teste la lecture de log git. Utilise junit pour un test, qui vérifiera si le résultat attendu (hardcodé) est égal à ce que la fonction trouve. Il y a deux tests, un pour parseCommit() et un pour parseLog(). La structure des tests est assez complexe, puisque les variables appellent une succession de méthodes qui ne sont jamais vues auparavant. À détailler plus tard, donc.

git.log : log git pour tester.