Rapport de projet

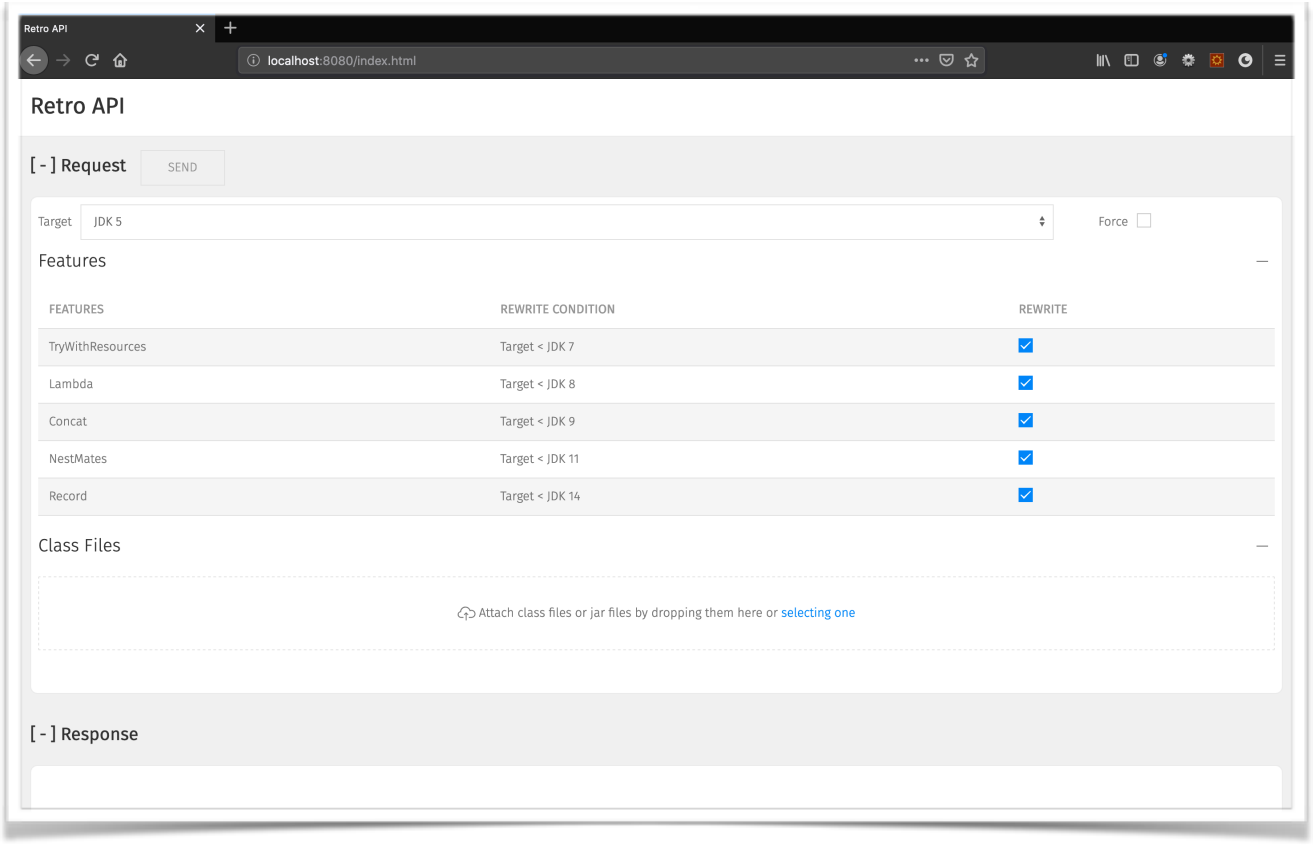
Java - Projet Retro 2019-2020

Mamadou Cisse - Sanchez Fernandes Stephane

Présentation 3
Choix techniques 4
Mise en place du projet 4
Design pattern visitor et façade 4
ClassTransformer et FeatureVisitor 5
MockerVisitor 5
Frontend 5
Problèmes rencontrés 6

Présentation

L'objectif de ce projet est la réalisation d'un programme permettant de downgrader le byte code d'une classe java tout en gardant la comptabilité de certaines features. Pour ce faire on nous a mis a disposition la librairie ASM qui permet de manipuler le byte code d'une classe.



Frontend du programme

Choix techniques

Mise en place du projet

Dès le début de notre projet, nous avons commencé par faire des schémas uml pour avoir une vue d'ensemble de l'architecture du projet.
Très vite nous avons décidé de nous partager les tâches. Un de nous devait mettre en place le projet en créant les packages, et data classes nécessaires (package fr.umlv.retro.models) et l'autre devait lire les différents docs (ASM, JVM…) pour l'expliquer une fois qu'il aurait compris.

À la suite de ça nous avons initialement choisi d'utiliser l'API tree d'ASM car ce dernier était plus facile comprendre et utiliser. Mais étant donné que ce dernier comportait certains défauts comme le fait qu'il met en cache trop d'informations, nous somme partis sur le core l'API qui est basé sur le pattern Visitor.

Design pattern visitor et façade

Comme expliqué plus haut, nous avons décidé à la suite de notre phase de réflexion d'opter pour une implementation basée sur le design pattern visitor. Étant donné que l'API de la librairie ASM est axée sur ce pattern, cela semblait être le meilleur choix.

Nous avons crée une class **Retro** servant de façade (design pattern façade) au programme. Cette classe contient principalement des méthodes qui une fois appelées sont déléguées à des méthodes de sous-api (comme le file system ou encore le logger). Cela nous permettait de ne pas avoir de singleton ou encore de ne pas créer des méthodes avec trop de paramètres. La quasi-totalités des classes de notre sont immutables.

La classe **Retro** est aussi le point d'entrée de l'API avec ses méthodes statiques (**exec**). Elles prennent en paramètre des chemins et les options de lancement du programme et transforment la classes et jar aux emplacements indiqués l'aide de la classe **ClassTransformer**. Les transformations sont enregistrées sur le disque (dans un sous-dossier **retro-output** relative aux chemins passés en option afin de ne pas modifier les fichiers originaux) uniquement si aucun problème ne survient (warning, error, exception).

ClassTransformer et FeatureVisitor

Une fois que la méthode exec de la classe Retro est appelée et que tous les arguments sont correctes, La classe ClassTransformer entre en jeu. Ce dernier hérite de la classe ClassVisitor de la librairie ASM et nous permet de parcourir les différents instructions dans un bytecode.

La méthode important dans ce dernier est la méthode privée « **visit** ». Cette dernier prend en paramètre la version target spécifié par l'utilisateur et parcourir dans le sens inverse les **FeatureVisitor** disponible à la création de **Retro** afin de faire en sorte que chaque visitor délègue ses appels aux méthodes visit (**visitField**, **visitMethod**…) au précédent afin de former une chaine. La chose important dans cet algorithme est que le dernier élément de la chaine est une ClassWriter ce qui permet d'écrire les modifications dans un tableau de byte qui sera écrit sur le disque à la fin.

Les classes FeatureVisitor redéfinissent les méthodes adéquates afin de détecter et réécrire les features.

MockerVisitor

Lors de la transformation des bytecode il est possible que ce dernier ne soit pas valide pour une certaine target même si les features sont réécrites à cause de l'utilisation d'API non disponible dans la target. Par exemple le package **java.util.function** n'est pas disponible sur une target < 8. Pour contourner ce problème, nous avons créer un visitor spécial « **MockerVisitor** » qui est appelés en premier (important car sinon il est impossible des instructions soit supprimés par un des visitors comme c'est le cas lors pour les dynamic invoke) dans la chaine des visitor afin de detecter l'utilisation de ces API. Une fois que la méthode visitEnd de ce visitor est appelée, nous générons relativement par rapport au dossier root (le dossier contenant les chemins passés en option au programme) Une copie du code des API en nous assurant qu'il n'y a pas de duplication, et que le bytecode des ces classes passent aussi par notre ClassTransformer afin de mocker récursivement les apis. Cela marche car initialement la classe façade de notre programme contient l'unique instance du MockerVisitor ce qui permet de garder son état et de ne pas avoir de problème de dépassement de la pile d'appels.

Frontend

Pour la partie front-end nous avons tous réalisé sans utiliser un des frameworks qu'il était possible de choisir.

Problèmes rencontrés

On n'a pas eu de problème majeur sur ce projet mis à part au niveau du choix de découpage des classes. On à répondu à la totalité du cahier des charges hormis le fait que nous avons pas réussis à récrire les **Records** et à detecter et réécrire les **TryWithResouces** par manque de temps.