UNIVERSITE CATHOLIQUE DE LOUVAIN

ECOLE POLYTECHNIQUE DE LOUVAIN

Génération de feedback enrichis pour la plateforme INGInious

Promoteur : Charles Pecheur
Lecteurs : Pierre REINBOLD
Peter VAN ROY

Mémoire présenté en vue de l'obtention
du grade de master ingénieur civil en informatique
options
ingénierie logicielle & systèmes de programmation, et
sécurité & réseaux informatiques
par Pham Anh-Tuan LE

Louvain-la-Neuve
Année académique 2014-2015

J'adresse mes remerciements aux personnes qui m'ont aidé et accordé leur temps à ce mémoire.

En premier lieu, je remercie M. Charles Pecheur, professeur au département informatique à l'Université Catholique de Louvain. En tant que promoteur de mémoire, il m'a guidé dans la direction du travail et m'a aidé à trouver les solutions pour avancer.

Je remercie également M. Pierre Reinbold, membre du département informatique à l'Université Catholique de Louvain, pour ses précieux avis et idées partagés lors des réunions avec mon promoteur.

Je tiens aussi à remercier M. Guillaume Derval, développeur de la plateforme *INGInious* pour ses conseils et réponses techniques à mes nombreux questions.

Enfin, je remercie toutes les autres personnes qui m'ont aidé et épaulé d'une quelque autre façon.

INGInious est une plateforme permettant de récolter des projets informatiques impliquant du code source. Elle a notamment la possibilité d'évaluer automatiquement les codes des étudiants et de leur fournir un feedback dans le but d'apprendre plus efficacement.

Pour analyser et tester une soumission remise par l'étudiant, il est nécessaire de l'inclure dans un ensemble de code qui sera ensuite compilé et exécuté. À l'issue de ces phases de compilation et d'exécution, des messages d'erreurs peuvent survenir. Ces rapports d'erreur ne sont généralement pas assez clairs pour que l'apprenant comprenne son erreur.
Particulièrement, les messages d'erreur sont référencés à des numéros de ligne où ont lieu les erreurs. Étant donné que le code de l'étudiant est intégré dans un ensemble de code, ces lignes d'erreur ne concordent donc pas avec ce qu'a produit l'étudiant.

Par conséquent, il est important de remédier à ce genre de problème et de créer un nouveau module pour la plateforme *INGInious*. L'objectif de ce mémoire sera de construire ce nouvel outil permettant d'analyser et de transformer les messages d'erreurs produits lors de l'évaluation du programme de l'étudiant.
Par ailleurs, cette analyse devra se faire de manière indépendante du langage de programmation afin que ce module puisse être adapté à n'importe quel autre type de langage informatique.

Table des matières

1 Introduction **1**

2 INGInious **3**

2.1 Description d'INGInious . . . 3

2.2 Fonctionnement d'INGInious . . . 4

3 Énoncé du problème **7**

3.1 Problème de correspondance de ligne . . . 7

3.2 Problème de messages incompréhensibles . . . 9

4 Description de la solution **11**

4.1 Maperror . . . 12

4.1.1 Sources d'information nécessaires . . . 12

4.1.2 Output de `maperror` . . . 13

4.1.3 Comment s'utilise `maperror` ? . . . 13

4.1.4 Architecture de `maperror` . . . 14

4.1.5 Correspondance de ligne . . . 14

4.1.6 Transformation des messages d'erreur . . . 21

4.2 Feedbackaftermaperror . . . 26

4.2.1 Source d'entrée de `feedbackaftermaperror` . . . 26

4.2.2 Implémentation du Feedbackaftermaperror . . . 27

4.3 Script studentCodeHighlight . . . 28

5 Tests, évaluations et résultats **31**

5.1 Test global des outils . . . 31

5.1.1 Méthode . . . 31

5.1.2 Résultat . . . 32

5.2 Test de la pertinence du feedback . . . 33

5.2.1 Méthode . . . 33

5.2.2 Résultats bruts . . . 33

5.2.3 Filtrer les résultats bruts . . . 34

5.2.4 Analyse et interprétations des résultats . . . 35

5.2.5 Facteurs influençant les résultats 36

5.3 Conclusion des tests réalisés 37

6 Mise en pratique **39**

6.1 `maperror` 39

6.1.1 Utilisation de la commande 39

6.1.2 Fichier de règles de transformation 40

6.1.3 Étendre pour d'autres langages de programmation 42

6.2 `feedbackaftermaperror` 43

6.3 Changement nécessaire sur *INGInious* 43

6.3.1 Inclure le container `maperror` 43

6.3.2 Javascript `studentCodeHighlight.js` 44

6.3.3 Fichier de configuration d'*INGInious* 44

7 Conclusion **47**

A Architecture de `maperror` **51**

B API des outils **52**

B.1 API de `maperror` 52

B.2 API de `feedbackaftermaperror` 53

C Exemple d'un code Run **54**

D Fichier de transformation de règles **55**

D.1 Syntaxe du fichier de transformation de règles 55

D.2 Exemple de fichier de règles de transformation 56

E Exemple de test **58**

Chapitre 1

Introduction

La plateforme *INGInious* est une base de travail destinée aux étudiants en informatique afin qu'ils puissent soumettre leur travaux comportant du code source. Elle a d'ailleurs la capacité de procéder à un travail d'évaluation des codes soumis et de remettre un feedback en retour à l'étudiant.
INGInious est actuellement utilisé par de nombreux titulaires de cours pour évaluer la connaissance de leurs étudiants. Cet outil est une évolution des soumissions par papier et permet ainsi aux encadrements de diminuer leur charge de correction car une partie de l'évaluation se fait de manière automatique. Un autre avantage lié à l'utilisation de cette plateforme est la réduction de l'utilisation de papier.

Pour évaluer le code d'une soumission, il est nécessaire d'inclure ce code dans un ensemble de code afin de procéder à une batterie de tests. Lors de ces vérifications, une compilation et une exécution de code doivent être produites. Durant ces phases, des erreurs de compilation ou d'exécution pourraient émaner et provenir directement ou indirectement du code de l'étudiant. Par conséquent, il faut reporter ces messages à l'étudiant afin qu'il puisse comprendre ses éventuelles erreurs.

Cependant, les indications des lignes d'erreur reportées ne coïncideraient pas avec les réelles lignes du code soumis puisque celui-ci a été inclus dans un plus grand ensemble de code. Ainsi, ce problème de correspondance constitue un premier challenge à résoudre dans le cadre de ce mémoire.

Un autre souci provenant des messages d'erreur est la complexité et l'imprécision de certaines informations suivant le type de compilateur ou d'interpréteur du langage. Dans d'autres cas, certaines erreurs reportées touchent indirectement au code de l'étudiant. En effet, des erreurs rapportées pourraient d'un premier abord avoir aucun lien avec le code de l'étudiant. Toutefois, si nous analysons plus en profondeur, nous constatons que l'erreur provient directement du code de l'étudiant.

Le second objectif de ce mémoire sera donc de concevoir un module visant à faire évoluer les messages d'erreur de compilation ou d'exécution.

Ces deux objectifs désignés apporteront un certain avantage dans l'utilisation d'*INGInious* pour l'étudiant. Effectivement, si le professeur retournait les messages d'erreur bruts sortis de la compilation ou de l'exécution, l'étudiant débutant ne comprendraient quasi rien ou très peu de ces messages d'erreurs. Tandis que si ces erreurs sont retraitées dans le but d'une meilleure compréhension pour l'étudiant, l'apprenant tirera profit et apprendra d'autant mieux.

En résumé, dans le travail de mémoire, nous allons essayer d'implémenter un module capable d'une part de corriger le problème de correspondance des lignes d'erreur, et d'autres part, d'analyser les messages d'erreur et de les faire évoluer. Cet outil permettrait ainsi d'améliorer la qualité générale des feedbacks.

Chapitre 2

INGInious

Avant de concevoir le module permettant d'analyser et de transformer les messages d'erreur, il est nécessaire de comprendre l'environnement dans lequel nous allons travailler. Par conséquent, ce chapitre décrira tout d'abord la plateforme *INGInious*, avant d'expliquer son fonctionnement.

2.1 Description d'INGInious

INGInious est une plateforme informatique destinée aux étudiants et aux titulaires de cours informatiques. Elle permet aux élèves de soumettre leur code source de leur projet. Cette plateforme a été développée par des chercheurs du département INGI [1] afin de donner la possibilité aux professeurs et assistants d'évaluer automatiquement les codes de leurs étudiants, de faciliter la correction et de noter les travaux.

Pour accéder à la plateforme, il est nécessaire de se connecter sur le site d'*INGInious* [2] et de posséder un identifiant INGI ou un identifiant UCL [3]. Après s'être identifié, nous accédons à une page personnelle reprenant l'ensemble des cours auxquelles nous nous sommes enregistrés.
En cliquant sur un des cours, nous arrivons sur une page reprenant l'ensemble des exercices et projets disponibles pour ce cours qu'on appelle des "**tâches**". Une tâche peut être constituée de plusieurs sous-questions et suivant le type de tâche, il est possible de coder et de soumettre le code, ou bien de soumettre un fichier sur la plateforme.
Après une soumission, un feedback peut être fourni pour permettre à l'étudiant d'avoir un retour sur ses éventuelles erreurs commises.

1. Département Ingénierie Informatique de l'Université Catholique de Louvain
2. `https://inginious.info.ucl.ac.be`
3. Université Catholique de Louvain

2.2 Fonctionnement d'INGInious

Cette plateforme *INGInious* a été implémentée en langage de programmation *Python* (version 2) [2]. Elle fonctionne avec deux logiciels libres. Le premier programme se nomme *Docker* [4]. Celui-ce apporte un environnement d'exécution pour faire tourner des applications grâce à des containers virtuels. Le second logiciel, sous licence AGPL, s'appelle *MongoDB* [5] et est utilisé comme base de donnée pour garder trace des soumissions des étudiants.

Un cours sur *INGInious* contient un ensemble de tâches comme représenté schématiquement sur la figure 2.1. Chaque tâche possède un ensemble de fichiers. Parmi ceux-ci, nous retrouvons obligatoirement le fichier `run`.
Lorsqu'un étudiant soumet du code source, un nouveau container *Docker* est lancé avec tous

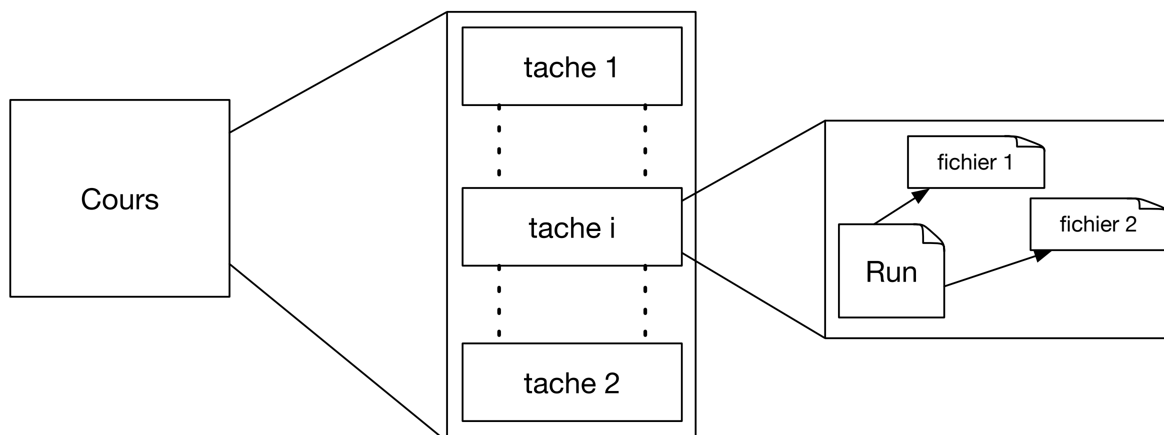


Figure 2.1 – Structure d'un cours dans INGInious

les outils nécessaires pour exécuter la tâche. Le type de container mis en place est spécifié par la tâche. À l'intérieur de ce container, le fichier script `run` sera exécuté permettant ainsi de lancer des commandes pour évaluer le code de l'étudiant et lui produire un feedback intelligent.
Le container "*default*" [6] mis à disposition par la plateforme, dispose déjà d'un ensemble d'outils utiles pour évaluer la soumission d'un étudiant. Parmi ces outils, nous pouvons notamment retrouver :

— `parsetemplate` permettant de parcourir un fichier et de lui injecter le code de l'étudiant où on le désire.

— `feedback` qui nous donne l'occasion d'afficher un feedback à l'étudiant.

D'avantages informations et d'autres outils sont disponibles sur la *documentation d'INGInious* [7].

4. `https://www.docker.com`
5. `http://www.mongodb.com`
6. `https://github.com/UCL-INGI/INGInious-containers`
7. `http://inginious.info.ucl.ac.be/static/doc/teacher_doc/run_file.html#run-file`

Au niveau de l'architecture, *INGInious* est divisé en deux parties distinctes : un "backend" et un "frontend" [2].
La partie "backend" est chargée de faire des vérifications sur la soumission de l'étudiant et envoie le code de l'étudiant dans un container *Docker*, avant d'exécuter la tâche.
La partie "frontend" permet aux étudiants de travailler sur une interface web et fournit aux titulaires de cours des statistiques et de outils pour contrôler leur page.

Chapitre 3

Énoncé du problème

Lorsqu'un code source de l'étudiant est soumis sur la plateforme *INGInious*, il est dans un premier temps inclus dans un ensemble de code. Cela permet de tester et évaluer ce que l'étudiant a produit.
Ensuite, cet ensemble de code doit être compilé avant de l'exécuter. Lors de ces deux étapes, il se peut que des messages soient générés à cause des erreurs. Celles-ci peuvent provenir du code encadrant ou du travail remis par l'apprenant. Si nous supposons que les erreurs sont provoquées par l'étudiant, nous devrions lui retourner un feedback contenant les messages d'erreur le concernant.
Toutefois, il se peut que des informations non-pertinentes ou imprécises soient contenues dans ces messages. Plus particulièrement, nous pouvons distinguer deux types de problèmes liés aux messages d'erreur de compilation et d'exécution.

3.1 Problème de correspondance de ligne

Le premier problème concerne les mauvaises références des lignes où les erreurs se localisent. En effet, nous pouvons rencontrer dans les messages d'erreur des numéros de lignes pointant vers le code d'analyse qui a servi à compiler ou a été exécuté. Cependant ces lignes d'erreur ne concordent aucunement avec les lignes du code de l'étudiant.
Dès lors, ces numéros de lignes contenu dans les messages d'erreur seraient incompris par l'étudiant.

Regardons un exemple de cette situation sur le code 3.1 soumis par un étudiant [1]. Le but de l'exercice était d'écrire un programme affichant le nombre de diviseurs propres d'un entier `n` [2].

1. code soumis par rgoudelouf le 03/11/2014 à 10h18 dans le cadre de la question de bilan final 2 du cours FSAB 1401 : Informatique 1 à l'UCL
2. c'est-à-dire le nombre de diviseurs entiers différents de n

```
1  int nombreDivs = 0;
2  for(int i = 0; i<=n; i++){
3      for(int j = 0; j<=i, j++){
4              if(i%j==0){
5                      nombreDivs++
6              }
7      }
8      System.out.println(i+":"+nombreDivs)
9      nombreDivs = 0;
10 }
```

Code 3.1 – Exemple d'un code soumis par un étudiant

Lors de la soumission, ce code sera inclus dans un ensemble de code qui sera lui-même compilé. Cependant, nous obtenons des erreurs reprises à la figures 3.1.
Nous constatons que les lignes d'erreur (17,19 et 22) ne correspondent pas aux lignes d'erreurs du code 3.1. Les lignes coïncidentes sont la 3 pour la 17, la 5 pour la 19 et la 8 pour la 22 ème ligne.
Si ces lignes correspondantes étaient retournées à l'étudiant, ces informations le guideraient certainement mieux à comprendre d'où provient ses fautes. Ainsi, le premier challenge de ce mémoire sera d'établir une correspondance des numéros de lignes entre ceux se trouvant dans les erreurs de compilation ou d'exécution et ceux de l'étudiant.

```
Q1.java:17: error: ';' expected
                for(int j = 0; j<=i, j++){
                                   ^
Q1.java:17: error: ')' expected
                for(int j = 0; j<=i, j++){
                                    ^
Q1.java:17: error: illegal start of expression
                for(int j = 0; j<=i, j++){
                                        ^
Q1.java:19: error: ';' expected
                           nombreDivs++
                                       ^
Q1.java:22: error: ';' expected
                System.out.println(i+":"+nombreDivs)
                                                    ^
5 errors
```

FIGURE 3.1 – Erreur de compilation issue du code 3.1

3.2 Problème de messages incompréhensibles

Le deuxième problème avec les erreurs de compilation ou d'exécution concerne la précision de l'information retournée.
En effet, nous pouvons le voir à travers les erreurs de compilation sur la figure 3.1. Trois erreurs différentes sont rapportées à la ligne 17 qui correspond à la ligne 3 du code 3.1. Comme nous le constatons facilement, la véritable erreur provient de l'utilisation d'une `","` au lieu d'un `";"` dans la condition de la boucle `for` pour séparer le deuxième et le troisième champ.

Pour éviter que ce genre de situation, il faudra ainsi implémenter un outil capable de réformer les messages d'erreur afin de fournir une information plus adaptée et pertinente pour l'étudiant. En d'autres mots, l'outil construit devra être en mesure de filtrer et transformer les messages d'erreurs dans le but de rapporter un meilleur feedback à l'étudiant.

Chapitre 4

Description de la solution

Pour analyser et transformer les messages d'erreur de compilation ou d'exécution et afficher le résultat à l'étudiant, il est nécessaire de travailler, d'une part, sur le côté "backend" pour l'analyse et la transformation des messages, et d'autre part, sur le "frontend" pour l'affichage du feedback amélioré. Pour ce faire, comme illustre la figure 4.1, nous allons développer trois principaux modules.

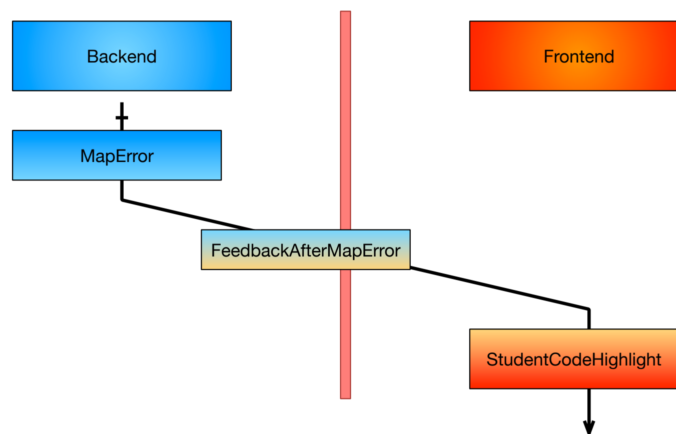


FIGURE 4.1 – Structure du mémoire

Le plus important des trois est `maperror`. Ce module solutionnera les deux problèmes posés dans le cadre de ce mémoire. C'est-à-dire, il établira une correspondance entre les numéros des lignes des messages d'erreurs avec ceux des lignes du code remis par l'étudiant. Par ailleurs, elle donnera la possibilité de faire évoluer les messages d'erreurs à travers une série de règles de transformation.
Le deuxième outil développé, `feedbackaftermaperror`, permettra d'afficher un feedback intelligent à l'étudiant en se basant sur l'analyse faite par `maperror`. Ce deuxième module travaille essentiellement du coté "backend". Cependant, elle permet également d'appelé le script `studentCodeHighlight`(troisième module) comme nous le verrons à la section 4.2. Ces deux premiers outils sont notamment disponibles dans le dossier `container/maperror/bin/`

du CD-ROM contenant les réalisations informatiques réalisées dans le cadre de ce mémoire. Enfin, pour la partie "frontend", le script `studentCodeHighlight` sera conçu pour surligner et annoter dans le code de l'étudiant dans le but de mettre en évidence les lignes où les erreurs ont lieu.

Dans la suite de cette section, nous allons nous attacher à la description de chaque module implémenté. Nous commencerons par décrire le module principal, `maperror`, avant d'expliquer l'implémentation de `feedbackaftermaperror`. Enfin, nous terminerons ce chapitre par comprendre le fonctionnement du script `studentCodeHighlight`.

4.1 Maperror

`Maperror` est le module principal de ce mémoire qui a deux objectifs à remplir. En premier lieu, il doit résoudre le problème de correspondance entre les lignes des messages d'erreurs et le code de l'étudiant. Son second but est de nous donner les moyens de faire évoluer les messages d'erreurs à travers des règles de transformations.

4.1.1 Sources d'information nécessaires

Analysons en premier lieu les informations que nous avons besoin pour répondre aux objectifs de la fonction. Ces informations d'entrée obligatoires sont représentées schématiquement et entourées en bleu sur la figure 4.2. Nous avons besoin comme source d'information :

— **Langage** : le type de langage analysé dans les messages d'erreurs.
Par exemple, *Java_ Compilation* pour analyser les erreurs de compilation du langage Java.

— **Student Code** : le fichier contenant les codes soumis par l'étudiant. Le module ira directement chercher cette information dans les dossiers d'*INGInious*. Le fichier contient un dictionnaire *json* où les clés sont les identifiants des exercices ("problem id) et les valeurs contiennent les entrées de code correspondantes.

— **Template** : le fichier dans lequel le code de l'étudiant est inclus. Ce fichier contient des `@préfixe@problèmeid@suffixe@` pour insérer les entrées de code aux bons endroits.

— **rules_file** : le fichier contenant l'ensemble de règle permettant la transformation des messages d'erreurs. Ces règles doivent respecter une syntaxe bien précise comme il sera expliqué à la section 4.1.6.1.

— **error_file** : le fichier où se trouve les messages d'erreurs de compilation ou d'exécution qui seront analysés par l'outil. Ce fichier doit correspondre au **langage** entré.

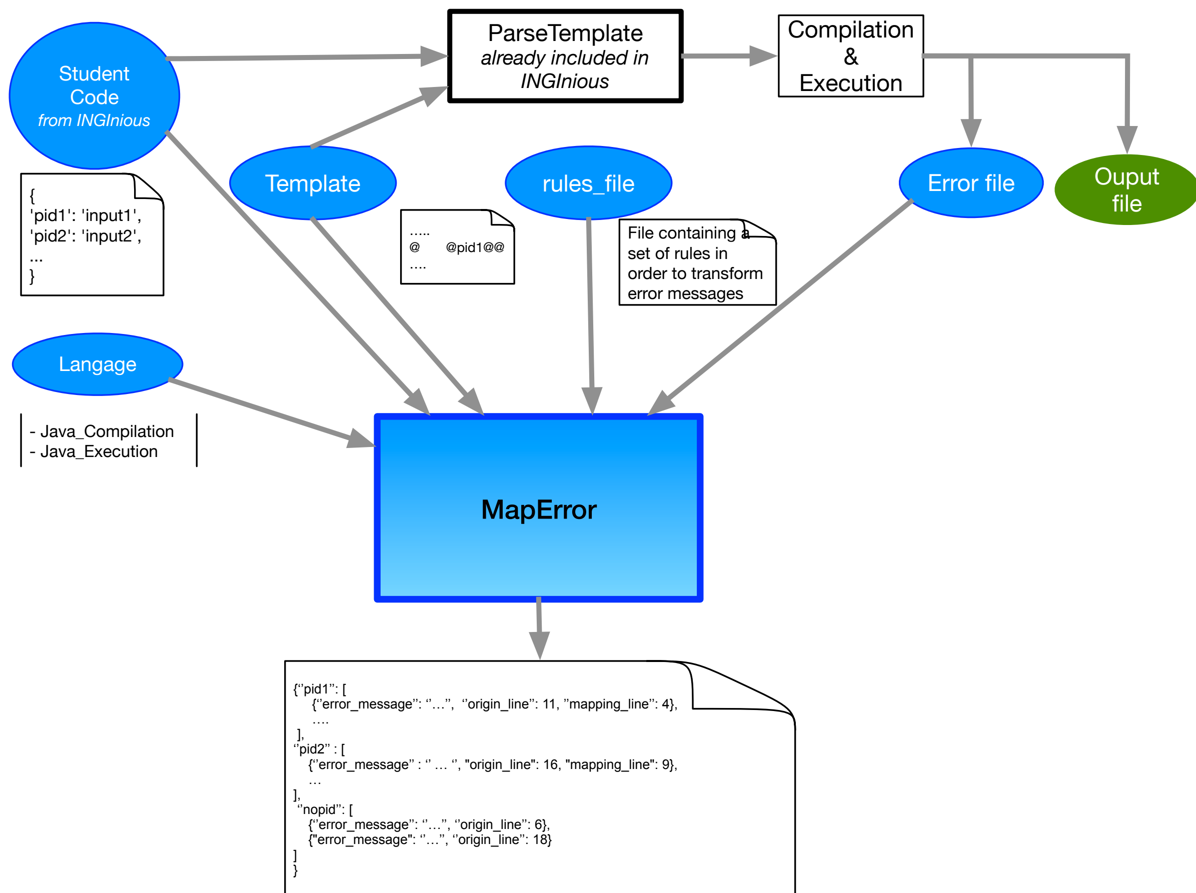


FIGURE 4.2 – Schéma général de l'outil `maperror`

4.1.2 Output de `maperror`

Après avoir utilisé `maperror`, nous devons décider du format de la sortie qui sera produite. Le format choisi est un fichier *json* pour sa capacité à stocker facilement des données et pour sa polyvalence d'être représenté dans la majorité des langages de programmation. Nous pouvons voir un exemple de ce fichier de sortie en bas de la figure 4.2.
Ce fichier contiendra les messages d'erreurs après avoir été traités. Ceux-ci seront accompagnés par leur numéro de ligne d'erreur d'origine dans le code compilé ou exécuté, et complétés par leur éventuelle ligne correspondante dans code de l'étudiant. Ces erreurs sont regroupées par identifiant de problème afin de distinguer facilement les erreurs à chaque exercice.

4.1.3 Comment s'utilise `maperror` ?

Nous allons maintenant expliquer comment notre outil `maperror` s'emploierait.
Lors de la configuration du fichier `run` d'une *tâche* dans *INGInious*, le code de l'étudiant est typiquement inclus dans un ensemble de code. Pour ce faire, nous utilisons l'outil `parsetemplate` disponible dans le container *default* d'*INGInious*. Comme illustre le haut de la figure 4.2, le code d'évaluation sera ensuite compilé et exécuté.

A l'issue de ces étapes, un fichier reprenant les erreurs et un fichier contenant la sortie standard sont généralement produits. Le premier fichier est celui qui nous intéresse car il contient les informations nécessaires pour la compréhension des erreurs de l'étudiant. Cependant, ce fichier d'erreurs présente des messages ayant besoin d'être adaptés ou transformés avant de les afficher à l'étudiant. C'est à cet effet que nous utiliserons notre outil `maperror` après la compilation et l'exécution pour rendre les messages d'erreur plus compréhensibles.

Pour y voir plus clair, un exemple concret de fichier `run` est fourni à l'appendice C, permettant ainsi de voir une façon de procéder avec les outils `maperror`.
Dans cet exemple de code `run`, nous distinguons clairement que l'outil `maperror` est utilisé une première fois à la ligne 22 et une deuxième fois à la ligne 27. Dans la première situation, les erreurs issues de la compilation *Java* seront analysées et transformées, alors que dans le deuxième cas, ce seront les messages d'erreurs d'exécution du langage *Java* qui seront traités.

4.1.4 Architecture de `maperror`

Avant de se plonger dans l'implémentation du module `maperror`, il a été nécessaire de définir une architecture afin de structurer et organiser les différentes parties implémentées.
L'architecture complète du module `maperror` se trouve à l'appendix A. Nous remarquerons qu'une source d'entrée (optionnel) a été rajoutée au module par rapport à la figure 4.2, à savoir, le nombre d'itérations que le fichier de règles de transformations sera parcouru. En effet, il est parfois utile d'évoluer un message d'erreur à travers une succession de règles à appliquer suivant plusieurs itérations.

Nous pouvons voir que l'architecture est divisée en deux parties distinctes. Ce choix d'architecture est motivé afin que chaque partie puisse répondre à chaque objectif défini à la section 3. D'une part, la partie "mapping" permet d'établir une correspondance entre les lignes d'erreur et les lignes du code de l'étudiant. Par ailleurs, cette partie essaie d'identifier l'exercice auquel l'erreur se rapporte.
Quant à l'autre partie, "Transform error messages" est chargée de transformer et faire évoluer les messages d'erreurs à travers une série de règles afin que ceux-ci soient plus compréhensibles pour l'étudiant.

4.1.5 Correspondance de ligne

Concentrons sur la premier partie de `maperror` où l'objectif est, à partir d'un numéro de ligne d'erreur, de trouver son numéro correspondant de ligne dans le code de l'étudiant et de chercher l'exercice auquel l'erreur se rapporte. L'architecture de la partie `mapping` est montrée à la figure 4.3.

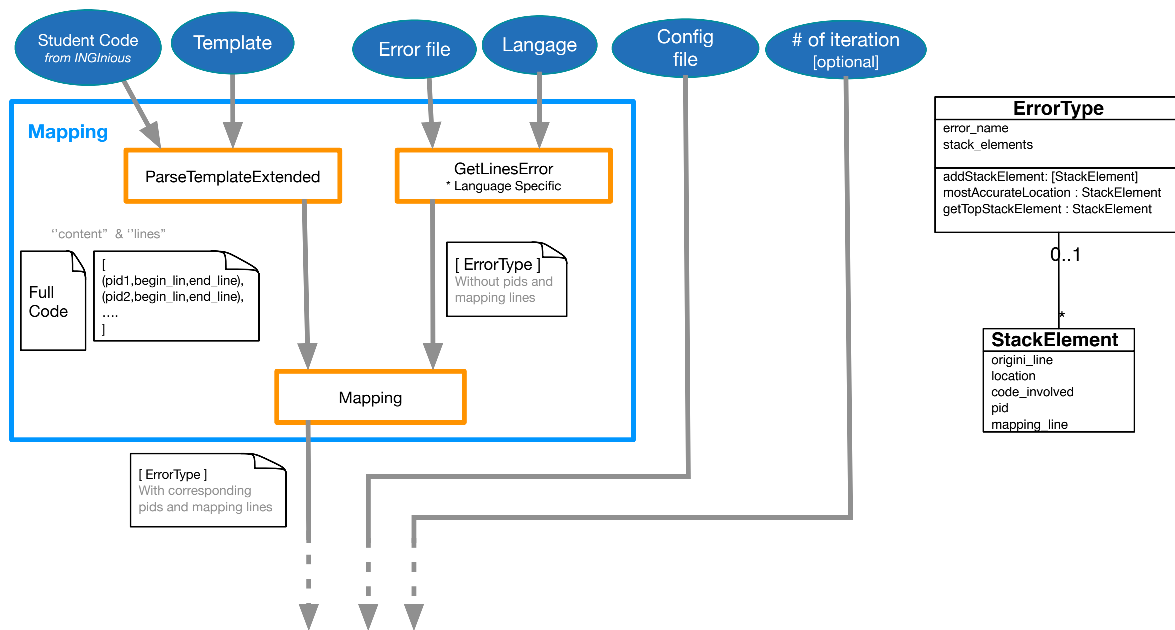


FIGURE 4.3 – Architecture de la première partie "Mapping"

4.1.5.1 `ErrorType`

Avant de décrire chaque fonction implémentée dans cette première partie, nous allons décrire l'utilisation d'une structure de données `ErrorType`.
Lors de sa conception, nous avons dû réfléchir à concevoir une structure générique et adaptable à n'importe quel type de message d'erreurs. Ces messages d'erreurs sont effectivement variables suivant le type de langage. Ainsi, nous devons définir clairement les caractéristiques et les contours des messages d'erreur, de manière à faciliter leur analyse indépendamment du type de langage traité.

Par conséquent, que contient un message d'erreur ? Nous savons qu'une erreur comporte généralement un nom (`error_name`) et une pile d'informations (`stack_elements`) permettant de retracer et remonter à la source de l'erreur.
Chaque élément de cette pile contient habituellement :

— `location` : le nom de méthode ou classe où l'erreur a lieu.
— `code_involved` : le fragment de code ou la ligne entière provoquant l'erreur.
— `origin_line` : le numéro de la ligne où l'erreur s'est produite.

Par ailleurs, nous devons rajouter l'information concernant l'éventuelle correspondance de ligne le code de l'étudiant (`mapping_line`) et l'hypothétique identifiant de l'exercice dans lequel l'étudiant a fait une faute (`pid`). Nous pouvons ainsi définir la structure de `ErrorType` que nous représentons à la droite de la figure 4.3

À cela, nous rajoutons trois méthodes utiles :

— `addStackElement` : rajoute un élément dans `stack_elements`

— `mostAccurateLocation` : cherche et retourne le premier élément de `stack_elements` possédant une correspondance de ligne `mapping_line` et un exercice de problème `pid` non nulles.

— `getTopStackElement` : récupère le premier élément de `stack_elements`

Voyons maintenant un exemple d'utilisation avec le message d'erreur suivant :

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Main.method1(Main.java:19)
        at Main.main(Main.java:10)
```

Dans cet exemple, l'erreur provient de la division par 0 et est causée par la méthode `method1` qui est appelée par la `main`. Après le traitement de cette première partie *mapping*, nous obtenons une structure de données équivalente à celle représentée à la figure 4.4.
Sur cette figure, nous apercevons une instance d'`ErrorType` créée et que la liste `stack_elements`

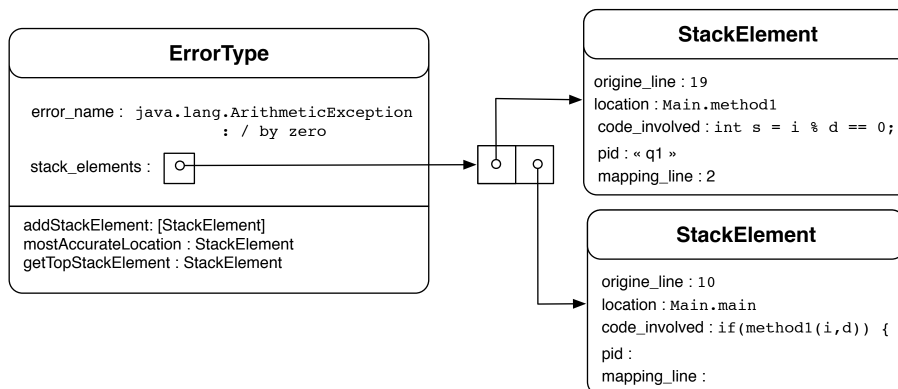


FIGURE 4.4 – Exemple d'une structure de données `ErrorType`

possède deux éléments `StackElement`. Cette liste peut être vu comme une pile d'information retraçant le propagation de l'erreur. Puisque nous retrouvons l'origine de l'erreur en haut de la pile, nous avons la méthode `method1` en premier élément et la méthode `main` en second élément.

4.1.5.2 `parseTemplateExtended`

Lorsque nous voulons faire une correspondance entre les lignes d'erreurs et celles du code d'un étudiant, nous avons besoin d'une part le code de l'étudiant (*Student Code*), et d'autre part, le code source dans lequel le code de l'étudiant a été inclus.
Le code complet dans lequel le code de l'étudiant est injecté dans le modèle `Template`, peut s'obtenir avec la commande `parsetemplate` proposée dans *INGInious*. Cependant, cet outil ne

donne pas la possibilité de récupérer les lignes où le code a été injecté. Ces lignes nous permettraient ainsi d'établir la correspondance voulue.

Par conséquent, nous implémentons la fonction `parseTemplateExtended` ayant pour objectif de reprendre les fonctions de `parsetemplate` et de nous procurer, en plus, les lignes où le code de l'étudiant a été inclus.
Comme nous le voyons sur la figure 4.3, la sortie de `parseTemplateExtended` nous retournera un dictionnaire dont le contenu de la clé `"content"` est le code complet, et la clé `"lines"` contient une liste de triplet `(pid, begin_line, end_line)`. Le premier élément est l'identifiant du problème auquel le code de l'étudiant répond, le deuxième est le numéro de ligne où le code commence et le dernier permet d'avoir le numéro de ligne où le code remis s'arrête.

Prenons un exemple d'un template (code 4.1) et le code d'un étudiant (code 4.2). Nous devrons ainsi après exécution de `parseTemplateExtended`, avoir le code 4.3 et une liste `[("q1",6,8)]`

```
1  public class Main{
2    public static void main (String args[])
3    {
4       System.out.println("Result : " + Main.method1(4,0));
5    }
6 @     @q1@@
7 }
```

Code 4.1 – Exemple de Template

```
1  public static boolean method1(int i, int d){
2       return i % d == 0;
3    }
```

Code 4.2 – Exemple d'un code d'étudiant

```
1  public class Main{
2    public static void main (String args[])
3    {
4       System.out.println("Result : " + Main.method1(4,0));
5    }
6    public static boolean method1(int i, int d){
7       return i % d == 0;
8    }
9 }
```

Code 4.3 – Exemple de code après avoir inclus le code de l'étudiant

Tout le détail de l'implémentation de la fonction `parseTemplateExtended` peut être trouvé dans `container/inginious/templateparserextended.py` .
Cette fonction prend deux paramètres :

- `template` : le contenu du template dans lequel le code de l'étudiant sera inséré
- `data` : un dictionnaire dont le champ `input` contient le code de l'étudiant.

L'analyse du template se fait ligne par ligne afin de pouvoir avoir un compteur de ligne. Celui-ci permettra de savoir où le code de l'étudiant sera inséré.
Il est aussi important à noter que la boucle `while` est préférée à la place d'une boucle `for` afin que les changements dans le template soient pris en compte à la ligne suivante.
Un autre détail d'implémentation à mentionner est l'utilisation de la librairie `regex` de python [1].
Il nous permet notamment d'utiliser des expressions régulières pour faciliter la recherche de chaines de caractères spécifiques.
Le reste des détails de l'implémentation peut être trivialement compris à travers le code.

4.1.5.3 `getlineserror`

Après avoir obtenu le code complet et les lignes où le code de l'étudiant commence et s'arrête, nous devons récupérer les numéros de lignes d'erreur. Comme décrit à la section 4.1.5.1, nous stockons les messages d'erreurs dans des `ErrorType` permettant de structurer et faciliter l'analyse.
Nous implémentons, ainsi, la fonction `getlineserror` dans le but de transformer les messages d'erreurs en `ErrorType`.

Il est évident que les messages d'erreurs varient d'un langage programmation à l'autre. Certains langages différencient aussi la structure de leur messages suivant s'ils sont issues de la compilation ou de l'exécution (par exemple *Java*).
Dans le cadre de ce mémoire, nous allons nous concentrer sur le langage de programmation *Java*.
Deux fonctions différentes doivent être implémentées suivant si ce sont des messages d'erreur de compilation ou d'exécution qui sont analysés. Pour le premier cas, nous appelons la fonction `getCompJavaErrorLines` et l'autre cas, `getExecJavaErrorLines`.
Dans le cadre d'une extension du module à d'autres types de langage, il sera nécessaire de créer de nouvelles fonctions adaptées à leur type de message d'erreur.

Tout d'abord, mettons en évidence les caractéristiques des messages d'erreur pour la compilation *Java*. Les messages d'erreurs commencent souvent par `nameFile.java:line:` où `namefile` est le nom de fichier et `line` est le numéro de ligne où se produit l'erreur de compilation. L'exemple suivant nous permet de le confirmer.

1. `https://docs.python.org/2/library/re.html`

```
Q1.java:17: error: ';' expected
                    for(int j = 0; j<=i, j++){
                                       ^
Q1.java:17: error: ')' expected
                    for(int j = 0; j<=i, j++){
                                        ^
Q1.java:17: error: illegal start of expression
                    for(int j = 0; j<=i, j++){
                                            ^
Q1.java:19: error: ';' expected
                               nombreDivs++
                                           ^
Q1.java:22: error: ';' expected
                    System.out.println(i+":"+nombreDivs)
```

Il suffit ainsi de chercher ces en-têtes dont nous pouvons déjà retirer la ligne d'erreur (`origin_line`). Nous pouvons ensuite avoir le nom de l'erreur (`error_name`) qui suit l'en-tête. Enfin, le fragment de code impliqué dans l'erreur (`code_invoved`) se trouve à la ligne suivant la ligne de l'en-tête. À défaut d'avoir une localisation plus explicite de l'erreur, nous pouvons prenons comme `location` tout ce qui trouve dans l'en-tête.
Nous notons aussi que les messages d'erreur de compilation en *Java* ne contiennent aucune pile retraçant l'exécution de l'erreur. En effet, une erreur de compilation a lieu sur une ligne bien précise et ne se propage pas. Au lieu, de créer deux types de structures séparés pour les erreurs de compilation et d'exécution, nous allons caractériser la pile de `ErrorType` comme une pile d'un élément unique contenant toutes les informations nécessaires.

Procédons de même pour les erreurs d'exécution en *Java*. Chaque erreur d'exécution commence par `Exception in thread`. Le nom de l'erreur suit par ailleurs cet en-tête comme le prouve cet exemple suivant.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Main.method1(Main.java:19)
        at Main.main(Main.java:10)
```

Chaque ligne suivant la ligne de l'en-tête est un élément permettant de localiser la propagation de l'erreur. Ces lignes sont reconnaissables car elles commencent par un `"at"`. Ainsi, chacune de ces lignes sera encodée dans un `StackElement`. Ensuite sur ces lignes `"at"`, suit la localisation (`location`) entre parenthèses. Et en fin de ligne, nous pouvons récupérer la ligne d'origine.

L'implémentation des fonctions `getCompJavaErrorLines` et `getExecJavaErrorLines` utilise

essentiellement la libraire `regex` de python pour facilement aller récupérer les chaines de caractères respectant les expressions régulières recherchées. Le détail de l'implémentation peut être retrouvé dans `container/inginious/getlineserror.py` .

4.1.5.4 La fonction `mapping`

Avec les lignes où les codes de l'étudiant commencent et terminent et avec les numéros de lignes des erreurs, nous avons maintenant toutes les données nécessaires pour effectuer le calcul de la correspondance entre les lignes d'erreur et celles du code de l'étudiant. Telle est la mission de la fonction `mapping`.

La fonction `mapping` (`container/inginious/maperror.py`) prend deux paramètres :

- `content_with_lines` : la sortie de `parseTemplateExtended`
- `list_errors` : la liste de `ErrorType` sortant de `getLinesError`

Pour chaque erreur trouvée dans `list_errors`, la fonction va déterminer la correspondance de ligne et chercher le "pid" au quel l'erreur se rapporte. Pour ce faire, nous devons d'abord vérifier si la ligne d'origine de l'erreur est comprise dans le code de l'étudiant. Cela revient à checker si la `origin_line` est entre la "`begin_line`" et la "`end_line`" d'un "`pid`".
Si c'est le cas, la ligne de correspondance est calculée comme suit :

```
mapping_line = origin_line - begin_line +1
```

En effet, si nous revenons au code 4.3 et que la ligne d'origine de l'erreur est à la 7ème ligne, alors :

```
mapping_line = 7 - 6 + 1 = 2.
```

Nous constatons effectivement dans le code 4.2 que l'erreur vient de la deuxième ligne.

`mapping` retourne ainsi la liste de `ErrorType` (`list_errors`) en rajoutant les éventuelles informations concernant la correspondance de ligne et l'exercice auquel l'erreur appartient.

4.1.6 Transformation des messages d'erreur

Après avoir analysé la première partie de `maperror`, nous allons nous focaliser maintenant sur la seconde partie. Cette partie a pour objectif de transformer les messages d'erreurs afin qu'ils soient plus compréhensibles pour l'étudiant. Cela va se réaliser en prenant l'information contenue dans les messages d'erreur et en la faisant évoluer à travers l'application d'une série de règles. L'architecture de cette seconde partie est visible sur la figure 4.5.

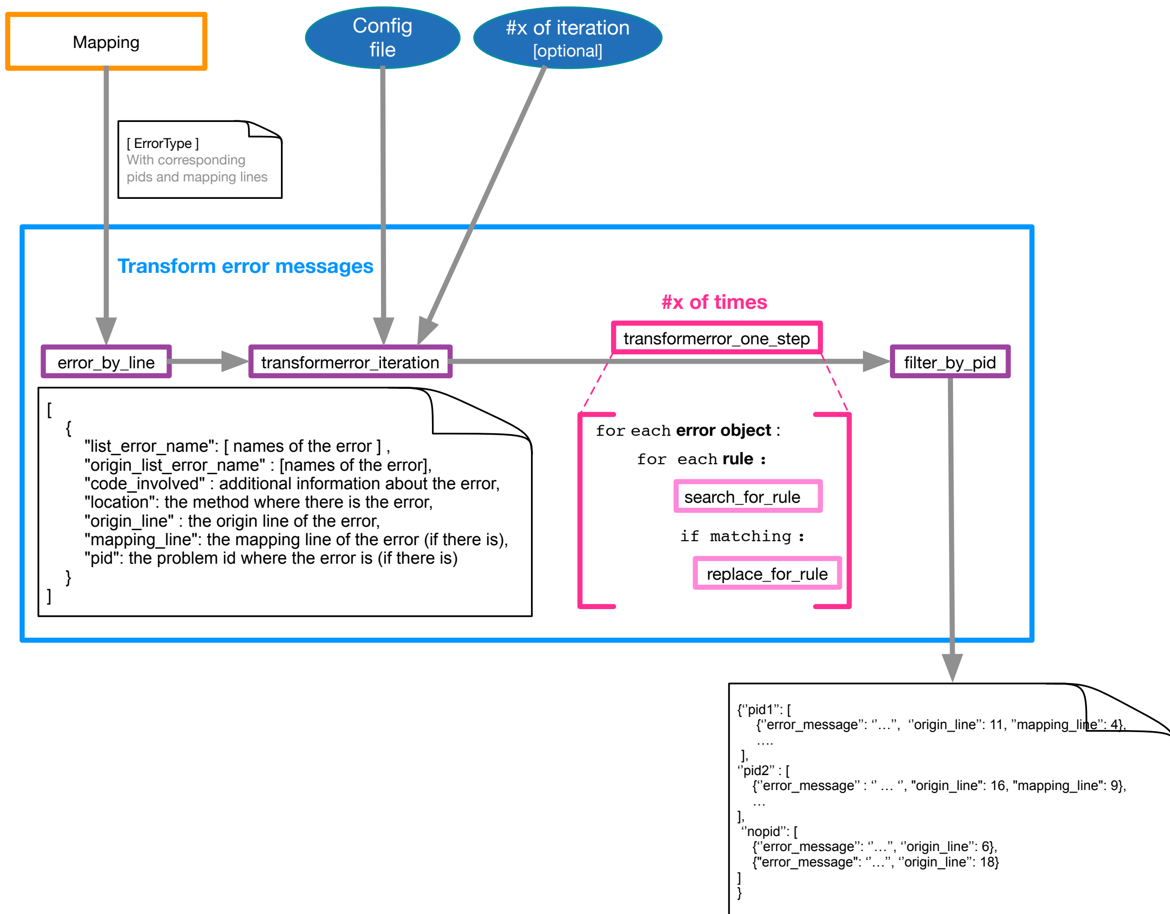


FIGURE 4.5 – Architecture de `transform error messages`, la deuxième partie de `Maperror`

4.1.6.1 Le fichier de règles de transformation

Pour faire évoluer les messages d'erreurs, il est nécessaire d'indiquer comment les transformer. Pour ce faire, nous pouvons renseigner cette transformation suivant des règles.
La syntaxe de ces règles suivent la grammaire définit à l'appendice D.1. Nous pouvons voir que la grammaire est difficile à comprendre. Ainsi, un exemple est montré à l'appendice D.2 pour permettre de mieux cerner la syntaxe.
Une explication de la création d'une règle sera fournie dans le chapitre "Mise en Pratique"à la section 6.1.2. Essayons ici de comprendre la structure de ces règles mise en place.

Chaque règle dispose tout d'abord d'un nom (name) pour facilement la reconnaitre. Ensuite, il faut spécifier pour la règle la recherche à faire et la transformation à appliquer. La recherche est défini à travers l'objet search qui peut combiner plusieurs règles logiques afin d'optimiser la correspondance.

Quant à la transformation, elle est spécifiée grâce à l'objet replace. Deux possibilités sont offertes. La première est de remplacer complètement un champ (field et repl). Le second choix est de choisir une transformation en se basant sur l'ancien message (field, pattern et repl).
Ce deuxième mode nous permet d'utiliser la fonction re.sub [2] de la librairie *regex* python afin de manipuler des patterns et de réutiliser des données de la chaine de caractères trouvées. Pour ce cela, Python utilise un système de groupe [1].
Les groupes fonctionnent d'abord en mettant des parenthèses (...) dans le champ "pattern" comme par exemple la ligne 67 de l'appendice D.2.
Ensuite, dans le champ repl, la correspondance trouvée est référencée par \g<1> ou \1 pour le premier groupe, \g<2> ou \2 pour le second groupe, etc... Le groupe 0 constitue le match complet du pattern.

Prenons un exemple pour mieux comprendre le système de groupe[1].
Soit un texte : Isaac Newton, physicist.
Dans ce texte, recherchons le pattern "(\w+) (\w+)" où "\w" désigne un caractère alphanumérique (a-z, A-Z ou 0-9) et "+" désigne 1 ou plusieurs fois le regex précédent le +.
Le match et le groupe 0 trouvés seront ainsi Isaac Newton.
Le groupe 1 sera Isaac et le groupe 2 sera Newton.

4.1.6.2 Structure de données de transformation

Lorsque nous jetons un coup d'œil à l'architecture de la deuxième partie (figure 4.5), une seconde structure de données est utilisée permettant de grouper les erreurs par ligne. Ce type de structure est construite par la fonction error_by_line et évolue après chaque application de la fonction transformerror_iteration. Elle consiste en une liste d'objets dont les champs sont :

- list_error_name : la liste de nom erreurs qui vont évolués à travers les séries de règles de transformation.
- origin_list_error_name : la liste de nom d'erreurs d'origine avant le processus de transformation.
- code_involved : le fragment de code qui génère les erreurs.
- location : le lieu où l'erreur se produit
- origin_line : la ligne d'origine d'où provient les erreurs
- mapping_line : la ligne de correspondance avec le code de l'étudiant

2. regex package de python : https://docs.python.org/2/library/re.html#re.sub

— `pid` : l'identifiant de problème associé où les erreurs ont lieu.

4.1.6.3 `error_by_line`

Bien qu'il soit possible qu'une même erreur concerne plusieurs lignes, nous avons décidé de regrouper les erreurs par leur numéro de ligne d'origine. En effet, les messages d'erreurs de compilation et d'exécution sont souvent concernés par une seule ligne.
Ainsi, nous implémentons la fonction `error_by_line` pour regrouper les erreurs par ligne et produire la structure de donnée décrite au point 4.1.6.2. L'entièreté de l'implémentation de cette fonction peut être trouvée dans `container/maperror/inginious/transformerror.py`.

Cette fonction va prendre en argument la liste d'`ErrorType` retournée par la fonction `mapping` de la première partie du module `maperror`.
Ensuite, la méthode traite chaque élément de cette liste et tente de les caser dans un dictionnaire avec des numéro de ligne comme champ de clé. Ce dictionnaire permet d'aisément repérer si une erreur existe déjà sur une ligne.
Après, il suffit de retourner les valeurs de ce dictionnaire dans une liste afin de former la "structure de données de transformation".

Cependant, il se peut que pour un `ErrorType`, il existe plusieurs `StackElements` pour retracer l'erreur. Comment choisir, dès lors, quelle est la location la plus précise et pertinente pour le code de l'étudiant ?
À cet effet, nous utilisons la fonction `mostAccuranteLocation()` de l'`ErrorType`. Si aucune correspondance n'est trouvée avec le code de l'étudiant, alors nous prenons, par défaut, la location de l'erreur au début de la pile avec la fonction `getTopStackElement()`.

4.1.6.4 `transformerror_iteration`

Une fois que les erreurs ont été triées par ligne, nous pouvons commencer notre travail de transformation des messages. Nous créons, à cet égard, la fonction `transforerror_iteration` permettant d'itérer un certain nombre de fois les règles de transformation sur la structure de donnée de transformation.

La fonction `transforerror_iteration` va appeler `transformerror_one_step` le nombre de fois que nous voulons itérer. Comme son nom l'indique, cette sous-fonction applique un seul passage des règles de transformation sur notre structure de transformation.
Nous pouvons retrouver sur la figure 4.5 une description de l' algorithme de la fonction `transformerror_one_step`. Une première boucle va parcourir chaque élément de la structure de données de transformation (`error_object`) et une seconde boucle traite individuellement chaque règle de transformation (`rule`) afin de trouver une correspondance (`search_for_rule`) et d'appliquer la transformation (`replace_for_rule`) si un `"matching"` est trouvé.

`search_for_rule` est une autre sous-fonction qui va permettre de dire si une "objet erreur" respecte le champ "search" d'une règle. Par exemple, voici une règle :

```
{
    "name": "rule2",
    "search":  { "and": [
        {"code_involved":"for"},
        {"or":[
            {"error_name":"';' expected"},
            {"error_name": "illegal start of expression"},
            {"error_name": "not a statement"},
            {"error_name": "'\\)       ' expected"}
        ]}
    ]},
    "replace": {"field":"error_name", "repl":"Erreur de syntaxe dans la boucle 'for'"}
}
```

et une structure de données de transformation contenant un seul "objet erreur" :

```
[
    {
        "list_error_name" : [ "';' expected", "')' expected",  "illegal start of expression" ]
        "origin_list_error_name" : [ "';' expected", "')' expected",  "illegal start of expression" ]
        "code_involved" : "for(int j = 0; j<=i, j++){"
        "location" : "Q1.java:17"
        "origin_line": 17
        "mapping\_line" : 3
        "pid" : "q1"
    }
]
```

Nous constatons que l'"objet erreur" rentre bien dans les critères de la "search" de la règle de transformation. En effet, il contient un "for" dans le code impliqué et trois des quatre messages d'erreur dans le "or".

Une fois qu'il y a une correspondance entre une règle et un "objet erreur", la sous-fonction `replace_for_rule` sera appelée. Cette méthode commence à rechercher tous les champs rentrant dans les critères du "search" et applique la transformation des champs indiqués dans "replace".

Si nous reprenons notre exemple, nous obtiendrons :

```
1  [
2    {
3      "list_error_name" : [ "Erreur de syntaxe dans la boucle 'for
       '" ]
4      "origin_list_error_name" : [ "';' expected", "')' expected",
         "illegal start of expression" ]
5      "code_involved" : "for(int j = 0; j<=i, j++){"
6      "location" : "Q1.java:17"
7      "origin_line": 17
8      "mapping\_line" : 3
9      "pid" : "q1"
10   }
11 ]
```

Toute l'implémentation de la fonction `transformerror_iteration` et toutes les fonctions auxiliaires appelées se trouvent dans `container/maperror/inginious/transformerror.py`.

4.1.6.5 `filter_by_pid`

Après que la transformation des messages d'erreur soit accomplie, il nous reste à concevoir le fichier de sortie de `maperror`, décrit à la section 4.1.2. Nous construisons la fonction `filter_by_pid` pour remplir ce rôle.
Pour réaliser cela, la structure de données de transformation sortie de `transform_error` est nécessaire comme entrée. Avec celle-ci, la fonction va filtrer les "objets erreur" par identifiant de problème ("pid"). Les erreurs n'appartenant à aucun "pid", se retrouveront sous la clé "**nopid**". Notons également que l'étiquette "**general**" sera utilisée pour indiquer un problème impliquant tous les exercices.

L'implémentation complète de cette fonction est, de nouveau, disponible dans `container/maperror/inginious/transformerror.py`.

4.2 Feedbackaftermaperror

Après avoir développé l'outil `maperror` dans le cadre de ce mémoire, un deuxième module a été produit et nous l'appellerons `feedbackaftermaperror`. Celui-ci a été créé dans le but d'afficher la sortie du premier module `maperror` sur l'interface web où l'étudiant attend son joli feedback.

4.2.1 Source d'entrée de `feedbackaftermaperror`

Une représentation schématique du fonctionnement du module `feedbackaftermaperror` est illustrée à la figure 4.6. Nous pouvons souligner que cette fonction prend comme un premier argument le fichier *json* produit par le module `maperror`. Le deuxième argument est optionnel et permet d'afficher un message d'erreur spécifique, dans le cas où aucune erreur n'est trouvée dans le fichier *json*.
Après application de `feedbackaftermaperror`, les erreurs analysées seront affichées comme un feedback sur l'interface 'frontend'.

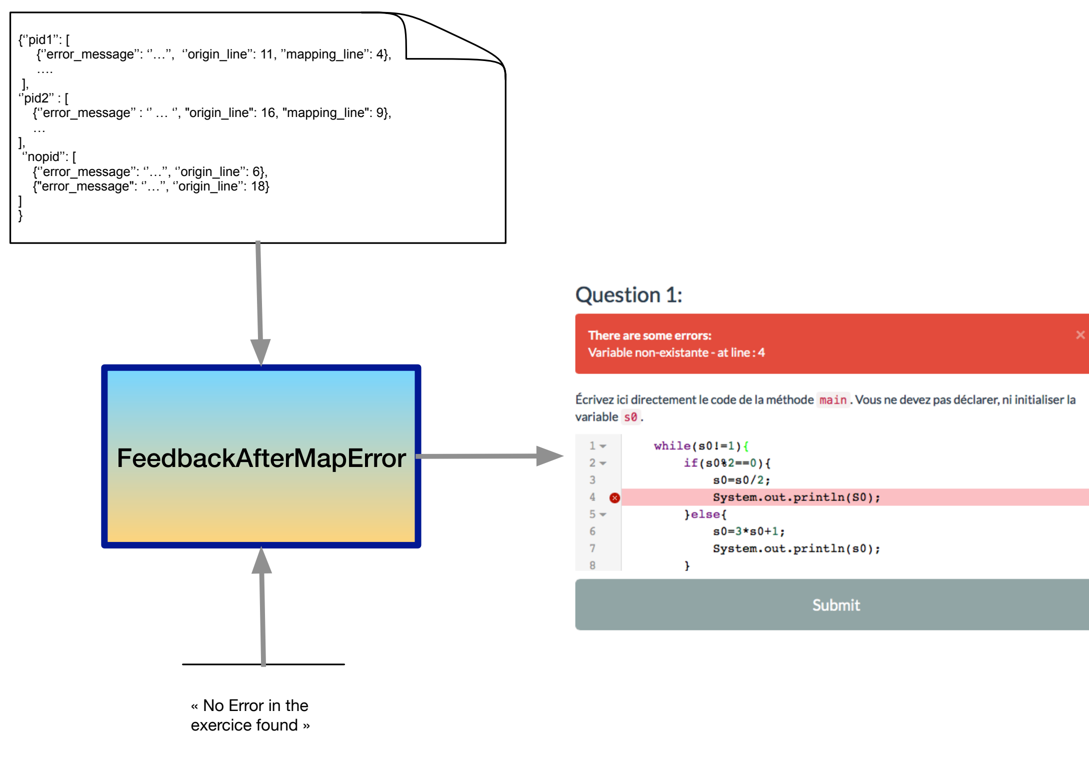


FIGURE 4.6 – Schéma général de l'outil `feedbackaftermaperror`

4.2.2 Implémentation du Feedbackaftermaperror

Pour implémenter la fonction *feedbackaftermaperror*, nous profitons des fonctions déjà disponibles dans le container "*default*" d'`INGInious`. Nous utiliserons les fonctions :

— `set_global_feedback(message)` : affiche un certain message général à l'utilisateur.
— `set_problem_feedback(msg,pid)` : affiche un certain message à l'utilisateur pour un problème précis.
— `set_result('failed'/'success')` : montre un feedback positif ou négatif

Pour utiliser ces fonctions, nous avons droit à une chaine de caractères par "pid" et/ou un message général. Ainsi, pour chaque "problème id", nous allons former un seul "string" en concaténant chaque erreur sous un "string" :

```
"$ error message :" + message d erreur + "@error line : " + la ligne d erreur
correspondante
```

Les caractères spéciaux `$` et `@` sont utilisés pour permettre au javascript `studentCodeHighlight` de facilement séparer les messages d'erreurs et les lignes d'erreur.

Par ailleurs, dans les messages uniques de chaque problème, `feedbackaftermaperror` insère un lien vers une fonction du *javascript* qui s'exécutera sur la page lors l'affichage du message. Pour injecter ce script, il suffit de rajouter à chaque message la chaine de caractères suivant :

```
"<script type="text/javascript"> highlightErrors(); </script>"
```

Ceci permettra d'appeler la fonction `highlightErrors()` de ce fichier javascript lorsque le feedback est retourné à l'étudiant.

Tout le détail de l'implémentation `feedbackaftermaperror` peut être retrouvé dans `container/maperror/bin/feedbackaftermaperror`.

4.3 Script studentCodeHighlight

Comme troisième module, nous voulons surligner les lignes d'erreurs et les accompagner par des annotations directement dans le code de l'étudiant. Ces indications contiendront les messages d'erreurs dans le but d'aider et guider l'étudiant à comprendre ses erreurs.
Pour ce faire, nous implémenterons une fonction javascript `highlightErrors` agissant sur l'interface "frontend" d'*INGInious*. Cette fonction et toutes les autres sous-fonctions se trouvent dans un fichier `js/studentCodeHighlight.js`.
Par ailleurs, cette fonction du javascript s'exécutera automatiquement lorsque le professeur utilisera le module `feedbackaftermaperror` dans son fichier `run` d'une tâche. Ceci se fait grâce à l'appel de la fonction inséré de manière invisible dans le feedback de l'étudiant.

A la figure 4.7, nous pouvons observer un exemple de résultat produit par l'appel de la fonction `highlightErrors` sur le code d'un étudiant. La ligne 4 est surlignée en rouge et une annotation est mise sur cette même ligne pour indiquer qu'une variable sur cette ligne n'existe pas (`S0`).

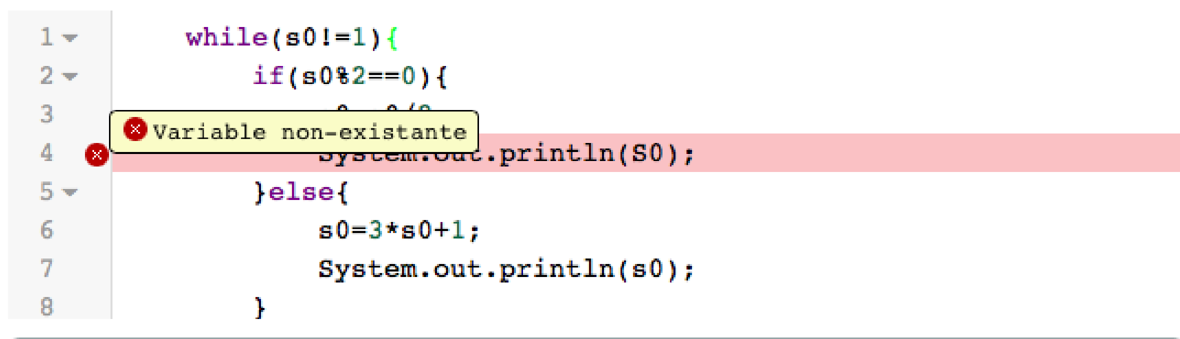


FIGURE 4.7 – Illustration du résultat produit par la fonction `feedbackaftermaperror` sur le code de l'étudiant

Plus concrètement, pour arriver à ce résultat, il est tout d'abord nécessaire de récupérer les lignes d'erreurs et les messages d'erreurs produits par `feedbackaftermaperror`. Comme nous l'avons mentionné à la section 4.2.2, chaque "pid" va afficher un message concaténant plusieurs "string" ressemblant à :

```
"$ error message :" + message d'erreur + "@error line : " + la ligne d erreur
correspondante
```

Nous devons séparer chaque message d'erreur avec sa ligne d'erreur pour chaque "pid". La fonc-

tion `extractErrorLines(problemId)` sera chargée de cette tâche pour un "pid" donné. Cette fonction nous retournera un dictionnaire avec les lignes comme clés et les messages d'erreur comme valeurs.

Ensuite, après avoir extrait les numéros de lignes et les messages d'erreurs, nous pouvons surligner les lignes et les annoter par leur(s) message(s) d'erreurs.
Nous déléguons cette tâche à la fonction `highlightErrorsForDico(pid, dico)`. Le premier argument est le "pid" concerné et le deuxième paramètre est le dictionnaire contenant les lignes et les erreurs retournées par `extractErrorLines`.

Avant de pouvoir surligner et annoter dans le code de l'étudiant, il faut d'abord récupérer l'éditeur de code ("editor") dans lequel l'étudiant a inséré son code. Le script d'*INGInious* permet déjà cela avec la fonction `getEditorForProblemId`. Un exemple d'"editor" est montré à la figure 4.7.
Nous pouvons ensuite surligner une ligne avec la ligne de code suivante :

```
editor.doc.addLineClass(line-1, 'background', 'line-error');
```

où `line` est le numéro de ligne que nous voulons surligner.
Pour l'annotation, nous devons écrire la ligne suivante :

```
editor.setOption("lint", function(){return annotations});
```

où `annotations` est une liste d'objet de forme :

```
{
   from: CodeMirror.Pos(line-1),
   to: CodeMirror.Pos(line-1),
   message: error_message ,
   severity: "error"
}
```

avec `error_message` étant le message d'annotation et `line` la ligne d'erreur.

Enfin, si nous ne voulons pas avoir un message affiché comme à la figure 4.8, nous devons "nettoyer" le message du feedback. Ainsi la fonction `showCleanMessage` se charge d'enlever les caractères "$" et "@" pour rendre l'affichage du feedback plus lisible.

Question 1:

There are some errors in your answer:
$ error message : Variable non-existante@ error line : 4

Écrivez ici directement le code de la méthode `main`. Vous ne devez pas déclarer, ni initialiser la variable `s0`.

```
1   while(s0!=1){
2       if(s0%2==0){
3           s0=s0/2;
4           System.out.println(S0);
5       }else{
6           s0=3*s0+1;
7           System.out.println(s0);
8       }
```

Submit

FIGURE 4.8 – Exemple d'un message feedback sans avoir appliqué la fonction `showCleanMessage`

Chapitre 5

Tests, évaluations et résultats

Les objectifs de départ de ce mémoire étaient de concevoir un outil permettant de faire évoluer les messages d'erreurs et de préciser la ligne d'erreur dans le code de l'étudiant sur la plateforme *INGInious*.
Nous voulons voir maintenant si notre module `maperror` répond à ces objectifs en évaluant ses résultats produits.

Pour réaliser ces tests, nous allons reprendre quelques exercices du cours "*LFSAB 1401 : Informatique 1*" enseigné à l'Université catholique de Louvain et se servir des codes soumis par les étudiants.
Deux expériences seront réalisées. En premier lieu, nous vérifierons le fonctionnement général de l'entièreté du module `maperror` avec `feedbackaftermaperror` et le javascript en frontend.
Ensuite, nous allons évaluer la qualité des feedbacks produits en comparant avec ceux donnés actuellement aux étudiants dans le cadre de ce cours.

L'ensemble des tests ont été réalisés en reprenant les énoncés de trois exercices (QBF1, QBF2 et QBF3 [1]). Pour chacune de ces tâches, nous redéfinissons le fichier *run* afin d'appeler nos outils, `maperror` et `feedbackaftermaperror` [2]. Un exemple de ce fichier est inclus à l'appendice C.

5.1 Test global des outils

5.1.1 Méthode

Pour tester le fonctionnement général des outils conçus, nous devons nous assurer qu'ils sont disponibles et installés aux bons endroits. L'installation des outils et leur utilisation seront documentées dans le chapitre 6.

1. Questions de bilan final 1, 2 et 3
2. Rappel : le script s'exécute dynamiquement après l'affichage du feedback

Après avoir défini correctement la tâche et les fichiers nécessaires pour l'évaluation d'une soumission de code, nous pouvons simuler la réalisation d'une tâche en lançant localement le serveur *INGInious* sur notre machine [3].
Nous pouvons ainsi reproduire les actions des étudiants et imiter leur soumissions de code source.

5.1.2 Résultat

Jusqu'à présent, les tests réalisés n'ont relevés aucun mauvais fonctionnement majeur. Toutefois, en réalité, nous ne sommes pas à l'abri d'une soumission d'un étudiant qui fasse planter l'évaluation et l'analyse du code.
Nous pouvons voir sur la figure 5.1 un exemple de résultat obtenu après une simulation d'une soumission d'un code d'un étudiant. Après avoir écrit le code répondant à la question QBF1, le bouton "Submit" a été cliqué. Nous obtenons comme prévu un surlignement de la ligne 11 où l'erreur se produit, une annotation sur la ligne et un feedback clair.

FIGURE 5.1 – Exemple de résultat d'une simulation des outils

3. L'installation de la plateforme *INGInious* sur notre machine est expliqué sur `http://inginious.info.ucl.ac.be/static/doc/installation.html#installation-of-inginious`

5.2 Test de la pertinence du feedback

Après s'être assuré du bon fonctionnement des outils implémentés, nous pouvons également évaluer la plus-value apportée par notre feedback à l'étudiant.

5.2.1 Méthode

Ce test peut se faire en comparant nos résultats produits avec ce qui est produit par les exercices d'origine utilisés dans le cadre du cours LFSAB 1401.

Les exercices, QBF1, QBF2 et QBF3, fonctionnent actuellement avec le container *Pythia*[4] contenant des outils pour réaliser des tests unitaires et pour fournir un feedback intelligent.

Les résultats des exercices d'origine sont stockés dans des fichiers `.test` dont un exemple est montré à la figure E.1 à l'appendice E. Ces données ont été obtenues via le titulaire du cours LFSAB 1401.
Dans chacun de ces fichiers, nous retrouvons notamment le code soumis par l'étudiant, l'identifiant de l'étudiant, le résultat après évaluation du code et éventuellement un feedback remis.

Grâce à l'outil `testtask` fourni par *INGInious* pour tester des tâches, nous pouvons comparer les résultats repris par ces fichiers `.test` et ceux générés par nos nouveaux modules.
Effectivement, cet outil simulera notre tâche redéfinie en reprenant le code de l'étudiant dans le fichier test et fournira une comparaison des résultats.
Cependant, l'outil ne nous permet de tester qu'un seul fichier `.test` à la fois. Ainsi, nous devons étendre cet outil afin de simuler un ensemble de soumission. Nous pouvons retrouver cet outil sous le nom `mytesttask` dans le dossier `Result`.

5.2.2 Résultats bruts

Les résultats bruts sont repris dans les fichiers `resultQBF1Test`, `resultQBF2Test`, `resultQBF3Test` dans le dossier `Result`. Ces fichiers comportent les résultats des simulations de exercices QBF1, QBF2 et QBF3 respectivement.
Chacun de ces fichiers contiennent un ensemble de résultats produits pour chaque soumission d'un code d'étudiant. Un exemple de résultat pour une soumission d'élève peut être aussi vu à la figure E.2 de l'appendice E. Ce résultat est produit par la simulation du fichier à la figure E.1.

Dans les résultats, nous avons d'abord les informations sur le fichier test simulé et l'identifiant de l'étudiant.

4. La plateforme Pythia est un ancien projet développé par les chercheurs du département informatique de l'UCL (INGI).

Ensuite, un dictionnaire est fourni pour nous renseigner sur les résultats de la simulation de notre tâche redéfinie. Les champs important sont :

— `stdout` : la sortie standard produit par le fichier *run*
— `problems` : le feedback spécifique à chaque problème retourné à l'étudiant.
— `result` : le résultat de l'évaluation de l'exercice
— `text` : le feedback général retourné à l'étudiant

Par après, nous avons une comparaison entre le feedback issue de l'exercice d'origine (se trouvant dans le fichier test) et le feedback que nous avons produit avec les outils `maperror` et `feedbackaftermaperror`.

Par exemple, le résultat à la figure E.2 nous indique que le feedback généré par nos outils cible mieux l'erreur que le feedback donné précédemment.
En effet, nous observons que l'erreur ne vient non pas de la ligne 12 mais la ligne 1 dans le code de l'étudiant. Notre module `maperror` nous permet cette correspondance de ligne.
Par ailleurs, dans le feedback d'origine, 4 erreurs sont produites et celle-ci ne permettent pas vraiment de comprendre la véritable erreur (ou bien il faut lire entre les lignes). Notre outil retraite ces erreurs et cible l'erreur vers laquelle l'étudiant ne doit qu'écrire le contenu de la `"Main"`.

5.2.3 Filtrer les résultats bruts

Nous avons à notre disposition 90 soumissions d'étudiant pour la QBF1, 78 pour la QBF2 et 52 pour la QBF3. Ce qui représente un total 220 résultats bruts à analyser.
Pour faciliter l'évaluation des résultats, un script [5] a été écrit en python permettant d'analyser les résultats et filtrer les informations pertinentes. Par ailleurs, ce script va catégoriser chaque résultat en trois catégories :

— *success* : l'analyse d'origine et celle générée par les nouveaux outils ne décèlent aucune erreur dans le code de l'étudiant et l'évalue comme un code juste.
— *failed* : Une erreur est repérée par l'analyse d'origine et la nouvelle analyse. Toutefois, le feedback apporté peut être différent.
— *neither* : l'analyse d'origine produit un résultat différent de notre nouvelle analyse.

La première catégorie *success* nous apporte rien car aucun feedback n'est apporté. Les deux analyses jugent que le code est juste et aucune erreur n'est à signaler.
La deuxième catégorie *failed* nous permet d'évaluer l'apport de nouveaux outils dans le feedback généré à l'étudiant.
Enfin, nous devons plus prêter attention à la troisième catégorie *neither*. Effectivement ce résultat nous dit qu'un des deux analyses prétend que le code n'a aucune faute et est juste, tandis que l'autre feedback dit le contraire. En d'autres termes, soit l'analyse d'origine reporte un code juste et que notre nouvelle analyse reporte une erreur ; ou soit nous avons la situation contraire.

5. Fichier `filterBadResult.py` dans le dossier `Result`

Ainsi, cela peut signifier deux choses. Soit notre module améliore l'évaluation d'un code ou bien cela peut également dire que nos nouveaux outils se plantent dans l'évaluation.

Nous pouvons voir la situation de la répartition des résultats dans les 3 catégories dans le tableau 5.1.
Pour le premier exercice, nous avons un bon nombre de résultats permettant de comparer le feedback d'origine avec celui dont nous tentons d'améliorer. Nous apercevons aussi qu'il y a moins d'erreurs dans le code de l'étudiant pour les deux autres exercices. Cela reflète probablement l'évolution de la qualité de soumission des étudiants.

Test	# de soumissions	Success	Failed	NEITHER
QBF1	90	44 (49%)	44 (49%)	2 (2%)
QBF2	78	52 (67%)	26 (33%)	0 (0%)
QBF3	52	36 (69%)	14 (27%)	2 (4%)

Table 5.1 – Résumé des résultats répertoriés dans les 3 catégories

5.2.4 Analyse et interprétations des résultats

Après avoir filtré et catégorisé les résultats, nous devons les analyser et les interpréter. Nous avons rassemblé dans un fichier excel `ResultatResume.xlsx` tous les résultats de tests réalisés avec leur interprétation.

Les interprétations possibles peuvent se distinguer en 4 groupes :

— **Meilleur feedback** rassemble les résultats dont le feedback nous donne une meilleure information pour comprendre l'erreur de l'étudiant par rapport au feedback d'origine.
Ce meilleur feedback est rendu possible grâce à l'évolution des messages d'erreurs et une précise location des erreurs de lignes.

— **Aucun apport** regroupe tous les résultats "success" et des résultats "failed" dont le feedback n'apporte rien en plus au feedback d'origine.
En d'autres termes, le résultat produit n'apporte ainsi aucune valeur rajoutée au feedback pré-existant.

— **Feedback moins précis** réunit les résultats qui génèrent un feedback moins pertinent que celui d'origine.

— **Mauvais apport** sont les résultats provenant de la catégorie *neither* et indique une différence de résultat final entre l'analyse initial et notre analyse.

Nous pouvons résumer les résultats et leur interprétations dans le tableau 5.2. Nous remarquons que pour les trois exercices, notre outil ne bouleverse pas les feedback produits précédemment car les résultats n'apportant aucun apport sont majoritaires. En effet, pour pourvoir comparer l'apport net dans le feedback apporté par notre outil, nous devons enlever les résultats "success". Ceux-ci n'apportent pas vraiment un vrai message à l'étudiant pour l'aider à comprendre son erreur puisqu'il a réussi l'exercice.

Test	**# de soumissions**	**Meilleur feedback**	**Aucun apport**	**Feedback moins précis**	**Mauvais apport**
QBF1	90	23 (26%)	50 (56%)	15 (17%)	2 (2%)
QBF2	78	13 (17%)	54 (69%)	11 (14%)	0 (0%)
QBF3	52	9 (17%)	41 (79%)	0 (0%)	2 (4%)

TABLE 5.2 – Résumé des résultats de comparaison de feedback d'origine et celui fourni après traitement par `maperror`

Le même tableau est reproduit au tableau 5.3 en enlevant les résultats où l'étudiant a réussi l'exercice. Nous constatons dès lors que `maperror` apporte une plus-value et améliore le feedback d'origine. Ceci peut s'expliquer par le fait que les messages d'erreurs peut être transformés afin de contribuer positivement au feedback.

Test	**# de soumissions raté par l'étudiant**	**Meilleur feedback**	**Aucun apport**	**Feedback moins précis**	**Mauvais apport**
QBF1	46	23 (50%)	6 (13%)	15 (33%)	2 (4%)
QBF2	26	13 (50%)	2 (8%)	11 (42%)	0 (0%)
QBF3	16	9 (56%)	5 (31%)	0 (0%)	2 (13%)

TABLE 5.3 – Résumé des résultats en comparant le feedback d'origine et celui fourni après traitement par `maperror` en ayant enlevé les résultats "success".

5.2.5 Facteurs influençant les résultats

Nous constatons qu'il y aussi des résultats négatifs reportés par le tableau 5.3.
Tout d'abord, dans les deux premiers exercices, nous avons un taux élevé de feedback moins précis que celui d'origine. Pour expliquer ces chiffres, nous devons regarder le feedback produit lorsque le résultat du code de l'étudiant n'est pas bon.
En effet, contrairement à l'analyse d'origine effectuée avec *Pythia*, nous reportons aucune indication sur le test unitaire réalisé lorsqu'il échoue.
Toutefois, nous constatons que cet effet n'est pas lieu dans les soumissions du troisième exercice. Ceci peut être sans doute expliqué par un type différent d'exercice.

Ensuite, dans les résultats de ce même tableau 5.3, nous pouvons également voir quelques cas de "mauvais apport". C'est-à-dire, nous avons à faire à des résultats d'évaluation différents pour une soumission d'un étudiant.
Ces quatre cas de "false positives" sont dûs à notre évaluation trop basique du code de l'étudiant. Cela ne se serait pas produit si nous avions fait des tests plus poussés afin de mieux évaluer le code source de l'apprenant.

Enfin, les résultats obtenus dépendent évidemment du fichier de règles de transformations. Jusqu'à présent, ces règles ont été construites au fur et à mesure de la découverte de nouvelles erreurs. Nous ne sommes pas à l'abri de la trouvaille de messages d'erreur ne s'appliquant à aucune des règles déjà rencontrés. Dès lors, nous devons nous efforcer à mettre à jour ce fichier de règles afin prendre en considération le maximum de cas possibles.

5.3 Conclusion des tests réalisés

En conclusion des tests réalisés sur les soumissions des étudiants sur trois exercices, nous pouvons dire que l'utilisation de `maperror` contribue de manière générale à l'amélioration du feedback grâce à la possibilité de faire évoluer les messages d'erreur à travers des règles et à un meilleur pointage de la ligne d'erreur.

Néanmoins, deux facteurs peuvent altérer ces bons résultats. En premier lieu, il faut que l'analyse et l'évaluation du code source de l'étudiant soient assez poussées afin qu'un code d'un étudiant ne répondant pas à l'objectif fixé, soit jugé comme un code juste.
Et deuxièmement, il est nécessaire que le fichier de règles de transformation soit régulièrement modifié afin de prendre en compte le maximum de situations d'erreur différentes.

Chapitre 6

Mise en pratique

Nous allons dans ce chapitre décrire comment mettre en œuvre les outils réalisés dans le cadre de ce mémoire.

6.1 `maperror`

6.1.1 Utilisation de la commande

Nous pouvons retrouver l'API de `maperror` en annexe B.1. La commande `maperror` prend 4 paramètres obligatoires et 2 optionnels.

```
maperror   [-h] [-o OUTFILE] [-i ITERATION]
           language template_file error_file rules_file
```

Concernant les arguments nécessaires, nous avons :

— `language` : spécifie le langage et le type de message d'erreur à analyser. Exemple : *Java_ Compilation*, *Java_ Execution*

— `template_file` : indique le nom de fichier du "Template" dans lequel le code de l'étudiant sera inclus. Ce fichier contient un/des endroit(s) : `@@pid@@`

— `error_file` : indique le nom de fichier qui contient les messages d'erreurs issus de la compilation ou de l'exécution d'un programme

— `rules_file` : indique le nom de fichier contenant les règles de transformations des messages d'erreur.

Par ailleurs, en plus de l'option `-h` permettant d'afficher la documentation, il existe deux options :

— `-o OUTFILE` : spécifie le nom de la sortie de fichier.

— `-i ITERATION` : précise le nombre de fois qu'il faut parcourir les règles de transformation

6.1.2 Fichier de règles de transformation

La fonction `maperror` a besoin du fichier de règles de transformation (`error_file`) afin de faire évoluer les messages d'erreurs à analyser.
Ce fichier contient un unique objet principal avec un tag `"rules"` contenant la liste des règles de transformations.

```
{
   "rules": [
      Objet de regle 1,
      Objet de regle 2,
      ....
   ]
}
```

Pour ajouter une règle, il suffit d'étendre la liste en respectant la syntaxe spécifier par la grammaire se trouvant à l'appendice D.

6.1.2.1 Objet de règle

Plus concrètement, il faut d'abord créer l'objet de la règle de forme :

```
{
   "name" : nom de la regle,
   "search" : l'objet de recherche
   "replace": l'objet de transformation
}
```

6.1.2.2 Objet de recherche

L'objet de recherche indique ce qu'il sera recherché dans le message d'erreur. Celui-ci peut être un simple champ de recherche de forme :

```
{"champ":  pattern, "occurence": l'occurence}
```

où le `champ` est à choisir parmi : error_name, code_involved, location, pid, origin_line ou mapping_line.

Nous pouvons aussi choisir de combiner les champs de recherche à l'aide de connecteur logiques (and, or, not). Par exemple, pour une série de disjonction, nous pouvons avoir :

```
{"or":[
   {"error_name":"'\\S+' expected"},
   {"error_name": "illegal start of expression"},
   {"error_name": "not a statement"},
   {"error_name": "'\\)' expected"}
]}
```

Pour les patterns, nous pouvons utiliser tous les modèles utilisés par le module`regex` de Python [1].
Il est important de noter que pour indiquer un "\" dans json, il est nécessaire de le précéder d'un autre "\". Ainsi pour indiquer une référence à un caractère alpha-numérique, il faut indiquer "\\w".

6.1.2.3 Objet de transformation

Après avoir spécifié la recherche, il faut spécifier la transformation. Deux types d'objets sont possibles.
La première est la forme simple où nous voulons seulement remplacer un champ. L'objet sera de format :

```
{
   "field" : champ,
   "repl" : pattern
}
```

où le `champ` est à choisir parmi : error_name, code_involved, location, pid, origin_line ou mapping_line.

La deuxième possibilité nous permet de récupérer une information dans le champ à remplacer. Cela se fait grâce aux regex de python. Cela donne un format

```
{
   "field" : champ,
   "pattern" : pattern a chercher,
   "repl" : pattern a replacer
}
```

1. `https://docs.python.org/2/library/re.html`

où le pattern à chercher contient un/des groupe(s) de parenthèses du type : (...). Cela permet d'inclure ce groupe dans le pattern à remplacer via "\\g<numero_du_groupe>".
Par exemple :

```
{
    "field" : "error_name",
    "pattern" : "'(\\S+)' expected",
    "repl" : "Manque de \\g<1>"
}
```

Par ailleurs, comme pour la recherche, il est possible de combiner plusieurs transformations. Dans ce cas, l'objet de conjonction sera plus souvent utilisé.

```
{
    "and" : [
        {"field": champ, "repl":pattern},
        {"field": champ, "pattern": pattern a chercher, "repl" :
        pattern a remplacer},
        ...
    ]
}
```

6.1.3 Étendre pour d'autres langages de programmation

maperror a été conçu de manière générique afin qu'il soit extensible à d'autres langages de programmation.
Pour permettre une analyse de messages d'erreur d'un autre langage, il est nécessaire de faire certaines modifications : définir un nouveau "parsing" et faire référence à cette fonction d'analyse.

6.1.3.1 Définir un nouveau parsing

Toute analyse d'un nouveau type d'erreur doit avoir une nouvelle fonction qui va analyser ces messages d'erreur. Les fonctions de "parsing" existantes se trouvent actuellement dans le fichier getlineserror.py dans le dossier container/maperror/iniginious/.
Il est possible d'étendre ce fichier en s'inspirant de ce qui existe déjà. La complexité n'est pas très élevée grâce à l'utilisation des regex de python.
Cette fonction doit récupérer les données des messages d'erreurs, les enregistrer dans des objets ErrorType (voir section 4.1.5.1) et retourner une liste d'ErrorType.

6.1.3.2 Etendre `maperror`

Après avoir écrit notre fonction de "parsing" pour notre nouvel langage, il faut évidemment dire à `maperror` d'y faire référence.
Pour cela, il est nécessaire d'ouvrir le fichier `conainer/maperror/bin/maperror` et de rajouter :

```
elif args.language == "nouveau_langage" :
        error_list_without_mapping = inginious.getlineserror.nv_parsing(
        error_content)
```

où `nouveau_langage` est le titre qu'on veut donner à notre type langage à analyser. Celui-ci sera utilisé comme premier paramètre par `maperror` pour spécifier le type d'analyse.
Et `nv_parsing` est le nom de notre nouvelle fonction pour analyser les messages d'erreurs se trouvant dans le fichier `getlineserror.py`

Après toutes ces extensions, il faut évidemment mettre à jour le container *maperror* afin de voir les modifications apportées dans notre environnement d'exécution.

6.2 `feedbackaftermaperror`

L'API de `feedbackaftermaperror` se trouve en annexe B.2. La commande `feedbackaftermaperror` prend 1 paramètre obligatoire et 1 optionnel.

```
feedbackaftermaperror  [-h] [-f feedback] errorshandled
```

Le paramètre nécessaire est `errorshandled` qui permet de spécifier le nom de fichier de sortie de `maperror`.
En ce qui concerne l'option, il est possible avec `-f feedback` de préciser le message de feedback à afficher dans le cas où aucune erreur n'est constaté dans le fichier `errorhandled`.

6.3 Changement nécessaire sur *INGInious*

Pour rendre disponible les outils `maperror` et `feedbackaftermaperror` comme une commande sur la plateforme *INGInious*, il est nécessaire de faire quelques manipulations.

6.3.1 Inclure le container `maperror`

Avant d'utiliser les outils `maperror` et `feedbackaftermaperror`, il est nécessaire de télécharger et s'assurer la présence du container *maperror* dans les dockers d'*INGInious*. Ce container permet d'inclure les outils et de les utiliser sur la plateforme *INGInisous* comme une commande à partir d'un fichier *run*.
Pour l'instant le dossier du container est disponible sous le dossier `container/maperror`. A

terme, il serait mieux de le rendre disponible depuis le "git" contenant tous les containers fournis par *INGInious*.

Il est également possible d'étendre le container *maperror* afin de rajouter tous les outils qu'on a besoin. Pour cela, il faudrait rajouter la configuration de son docker (Dockerfile) la ligne :

```
FROM maperror
```

En outre, il est bon à savoir que le container *maperror* étend le container *default* d'*INGInious*, permettant ainsi d'utiliser aussi les outils de base fournis par la plateforme *INGInious*.

6.3.2 Javascript `studentCodeHighlight.js`

Pour utiliser le javascript `studentCodeHighlight.js` en tant que telle, il n'y rien strictement rien à faire. En effet, c'est `feedbackaftermaperror` qui se charge d'inclure, de manière transparent, un appel au script dans les messages de feedback. Une fois que le message de feedback est affiché sur la page, le script s'exécutera automatiquement.

Cependant, il est nécessaire d'installer correctement le javascript. Pour cela, il faut d'abord inclure le plugin (`plugins/plugin_studentCodeHighlight.py`) dans les plugins d'*INGInious*[2]. Ce module permet d'étendre *INGInious* en injectant les fonctions du javascript sur la page.
Ensuite, après installé le plugin, il est indispensable d'inclure le fichier javascript (`js/studentCodeHighlight.js`) dans les scripts d'*INGInious*[3].

6.3.3 Fichier de configuration d'*INGInious*

Une fois que le container et le javascript sont en place, il faut faire deux petits rajouts dans le fichier de configuration d'*INGInious*[4].
La première est de rajouter une référence au nom du container Docker *maperror*. La configuration des containers devient ainsi :

```
1 "containers": {
2     "default": "ingi/inginious-c-default",
3     "maperror": "maperror"
4 }
```

où la troisième ligne a été rajoutée.

Le deuxième rajout nécessaire est de rajouter le plugin dans la configuration. La configuration des plugins devient donc :

2. dans le dossier `frontend/plugins/`
3. dans le dossier `static/js/`
4. Pour plus d'informations sur la configuration d'*INGInious*, il est possible d'en trouver sur `http://inginious.info.ucl.ac.be/static/doc/installation.html#configuring-inginious`

```
1  "plugins": [
2      {
3          "plugin_module": "frontend.plugins.git_repo",
4          "repo_directory": "./repo_submissions"
5      },
6      {
7          "plugin_module": "frontend.plugins.auth.demo_auth",
8          "users": {"test":"test"}
9      },
10     {
11         "plugin_module": "frontend.plugins.task_file_managers.
           json_manager"
12     },
13     {
14         "plugin_module": "frontend.plugins.
           plugin_studentCodeHighlight"
15     }
16 ]
```

où les lignes 13 à 15 ont été rajoutées.

Chapitre 7

Conclusion

L'objectif général du travail fixé par ce mémoire a été de concevoir un module pouvant améliorer la qualité des feedbacks sur la plateforme *INGInious* en se concentrant sur les messages d'erreurs.
Pour répondre à ce but fixé, l'outil `maperror` a été mis en œuvre. Deux axes distincts ont été suivis pour fournir un feedback plus enrichi.
Le premier axe a été de fournir une correspondance entre les lignes d'erreur sortant du compilateur ou issue de l'exécution et les lignes dans le code de l'étudiant. La première partie de `maperror`, *mapping*, a été implémentée pour répondre à ce premier objectif.
La seconde direction fut de créer un outil permettant de faire évoluer les messages d'erreurs à travers une série de règles de transformations. La partie *transform error messages* permet de s'occuper de ce second objectif.

Par ailleurs, avec l'outil `feedbackaftermaperror` et le javascript, un travail a été réalisé sur un meilleur affichage du feedback, guidant ainsi l'apprenant à comprendre les erreurs. Grâce à ces implémentations, l'étudiant verra ainsi les erreurs être surlignés et annotés d'informations précieuses.

De plus, nous avons vu à travers les tests réalisés que les fonctions implémentées apportaient une plus-value au feedback. Cependant, la qualité des feedback dépend d'une part de la profondeur de l'évaluation du code réalisée pour la tâche, et d'autre part de la qualité des règles de transformation.

En outre, il est à noter que les outils réalisés dans ce mémoire ont été implémentés et testés pour le langage de programmation Java. Cependant, les modules ont été conçus indépendamment du langage de programme qui sera analysé. Il est donc envisageable d'étendre le module `maperror` à d'autres langages de programmation.

Enfin, d'autres futurs travaux sur l'amélioration des feedbacks pourraient compléter ce travail en exploitant d'autres pistes que les messages d'erreurs. Notamment, une recherche sur une meilleure évaluation d'un code soumis et générique pour n'importe quel langage pourrait être recherché.

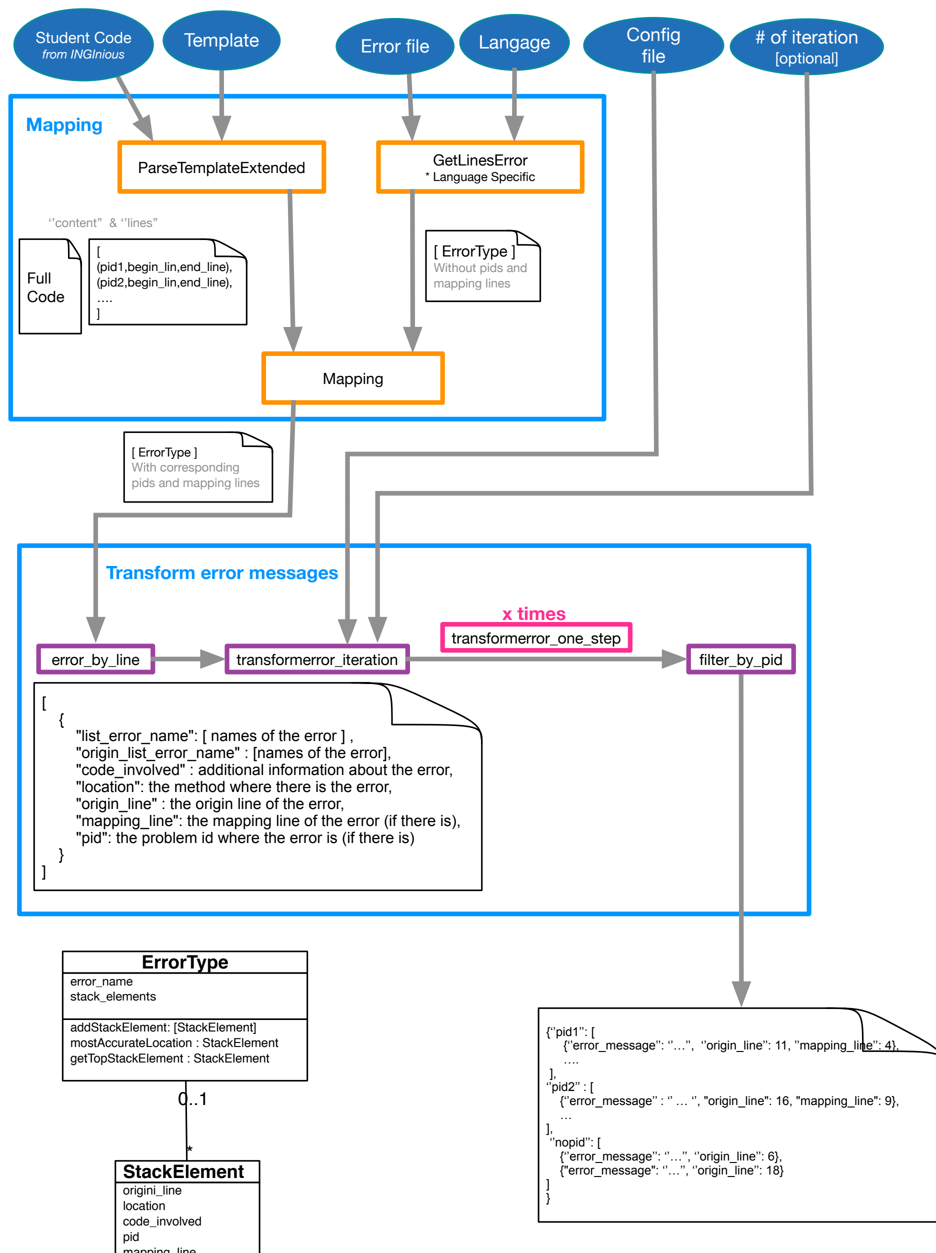
Bibliographie

[1] Python Software Foundation. re — regular expression operations. https://docs.python.org/2/library/re.html#module-re.

[2] INGInious Team. What is inginious ? http://inginious.info.ucl.ac.be/static/doc/teacher_doc/what_is_inginious.html.

Annexe A

Architecture de maperror



Annexe B

API des outils

B.1 API de maperror

```
usage: maperror                              [-h] [-o OUTFILE] [-i ITERATION]
                                             language template_file error_file rules_file

Map the error lines between a code program and a part of this code
and transform the error messages through the rules

positional arguments:
 language                                    The language and the type of errors to analyze :
                                             Java_Compilation, Java_Execution
 template_file                               The template file with @@ @@
 error_file                                  The file containing errors
 rules_file                                  The rules file contains a set of rules allowing
                                             to transform errors

optional arguments:
 -h, -help                                   show this help message and exit
 -o OUTFILE, -outfile OUTFILE                The output file where the error lines will
                                             be written
 -i ITERATION, -iteration ITERATION          The number of iteration to perform on the
                                             transformation of errors
```

B.2 API de `feedbackaftermaperror`

```
usage: feedbackaftermaperror      [-h] [-f feedback] errorshandled

Feedback the json mapping errors file output by maperror

positional arguments:
 errorshandled                    The input json file containing the mapping output
                                  by maperror

optional arguments:
 -h, -help                        show this help message and exit
 -f feedback, -feedback feedback  Message in case of success
```

Annexe C

Exemple d'un code Run

```
1   #!/bin/bash
2   # Inclure le code de l etudiant
3   parsetemplate --output Q1.java Template.java
4
5   # Compiler et Executer le code test
6   javac Q1Check.java 2> q1CheckComp.err
7   java Q1Check 0 1 9 10 283 1> q1.out 2> q1CheckExec.err
8
9   # Verification si bon resultat produit et aucune erreur compilation
10  if [ ! -z $(grep "true" q1.out) ] && ! [ -s q1CheckComp.err ]
11  then
12     feedback -r "success" -f "Vous avez reussi la question" -i 'q1'
13     feedback -r "success" -f "Vous avez reussi la question" -i 'subproblems'
14  # Verification si faux resultat produit
15  elif [ ! -z $(grep "false" q1.out) ]
16  then
17     feedback -r "failed" -f "Le resultat final n'est pas bon" -i 'q1'
18     feedback -r "success" -f "Les sous-methodes compilent" -i 'subproblems'
19  # Verification si Erreur de compilation
20  elif [ -s q1CheckComp.err ]
21  then
22     maperror Java_Compilation Template.java q1CheckComp.err ErrorConfigFile.json
        -o result.json
23     feedbackaftermaperror result.json -f "Pas d'erreur trouvee :-D"
24  # Verification si Erreur d'execution
25  elif [ -s q1CheckExec.err ]
26  then
27     maperror Java_Execution Template.java q1CheckExec.err ErrorConfigFile.json -
       o result.json
28     feedbackaftermaperror result.json -f "Pas d'erreur trouvee :-D"
29  # Alors pas bon resultat
30  else
31     feedback -r "failed" -f "Pas bon Resultat" -i 'q1'
32  fi
```

Annexe D

Fichier de transformation de règles

D.1 Syntaxe du fichier de transformation de règles

```
<configuration>            : :=   ’{’ "rules" ’ :’ <rules_list> ’}’
<rules_list>               : :=   ’[’ < rule > (’,’  < rule >)⋆ ]’
<rule>                     : :=   ’{’
                                  "names" ’ :’ <string> ’,’
                                  "search" ’ :’ <search_object> ’,’
                                  "replace" ’ :’ <replace_object>
                                  ’}’
<search_object>            : :=   <field_object>
                             |    ’{’ "not" ’ :’ <field_object> ’}’
                             |    ’{’ "and" ’ :’ <search_object> ’}’
                             |    ’{’ "or" ’ :’ <search_object> ’}’
<replace_object>           : :=   <replace_object_simple>
                             |    <replace_object_pattern>
<replace_object_simple>    : :=   ’{’
                                  "field" ’ :’ <field> ’,’
                                  "repl" ’ :’ <string>
                                  ’}’
<replace_object_pattern>   : :=   ’{’
                                  "field" ’ :’ <string_field> ’,’
                                  "pattern" : <pattern> ’,’
                                  "repl" : <repl>
                                  ’}’
<field_object>             : :=   ’{’
                                  <string_field> ’ :’ <string> | <int_field> : <integer>
                                  ( ’,’ "occurence" : <integer> ) *
                                  ’}’
<string_field>             : :=   "error_name"
                             |    "code_involved"
                             |    "location"
                             |    "pid"
<int_field>                : :=   "origin_line"
                             |    "mapping_line"
<pattern>                  : :=   voir paramètre de re.sub [1]
<repl>                     : :=   voir paramètre de re.sub [2]
```

D.2 Exemple de fichier de règles de transformation

```
{"rules":[
   {
      "name" : "begin by main",
      "search" : { "and":[
         {"error_name": "illegal start of expression", "occurence": 2},
         {"error_name": "';' expected", "occurence":2},
         {"code_involved": "main"}
      ]},
      "replace": {"field":"error_name", "repl":"Ecriver seulement le contenu de
       la Main"}
   },
   {
      "name" : "begin by class",
      "search" : { "and":[
         {"error_name": "illegal start of expression"},
         {"code_involved": "class"}
      ]},
      "replace": {"field":"error_name", "repl":"N'ecriver pas la class ni la
      Main mais seulement le contenu de la Main"}
   },
   {
      "name": "rule1",
      "search" : {"and":[
         {"error_name":"ArithmeticException"},
         {"error_name": "/ by zero"}
      ]},
      "replace": {"field": "error_name", "repl": "Division par 0 a un moment
      donne"}
   },
   {
      "name": "rule2",
      "search": {"and":[
         {"code_involved":"for"},
         {"or":[
            {"error_name":"';' expected"},
            {"error_name": "illegal start of expression"},
            {"error_name": "not a statement"},
            {"error_name": "'\\)' expected"}
         ]}
      ]},
      "replace": {"field":"error_name", "repl":"Erreur de syntaxe dans la
      boucle 'for'"}
   },
   {
      "name": "rule3",
```

```
42         "search" : {"and":[
43             {"error_name":"cannot find symbol"},
44             {"or":[
45                 {"code_involved":"symbol:"},
46                 {"code_involved":"variable"}
47             ]}
48         ]},
49         "replace": {"field": "error_name", "repl": "Variable non-existante"}
50     },
51     {
52         "name": "rule3bis",
53         "search" : {"and":[
54             {"error_name":"cannot find symbol"}
55         ]},
56         "replace": {"field": "error_name", "repl": "Variable non-existante"}
57     },
58     {
59         "name": "rule4",
60         "search": {"error_name":"'\\S' expected"},
61         "replace": {"field":"error_name", "pattern":"'(\\S)' expected", "repl":"
           Manque de \\g<1>"}
62     },
63     {
64         "name" : "oubli de ferme un block",
65         "search": {"error_name": "reached end of file (\\S+) parsing"},
66         "replace" : {"and":[
67             {"field":"error_name", "pattern":"reached end of file (\\S+) parsing",
               "repl":"Manque d'un '}' fermant la '\\g<1>'"},
68             {"field":"pid", "repl":"general"}
69         ]}
70     }
71 ]}
```

Annexe E

Exemple de test

```
...
courseid: LFSAB1401
grade: 0.0
input:
    q1: |-
        public static void main(){
          while(s0!=1){
            if(s0%2==0){s0=S0/2;
                System.out.println(s0);
            }
            else{s0=3*s0+1;
                System.out.println(s0);
            }
          }
        }
problems:
    q1: |-
        <p>
          Erreur de compilation (votre programme est donc mal écrit) :
        </p>
        <pre>
        Syracuse.java:12: illegal start of expression
                        public static void main(){
                        ^
        Syracuse.java:12: illegal start of expression
                        public static void main(){
                               ^
        Syracuse.java:12: ’;’ expected
                        public static void main(){
                                     ^
        Syracuse.java:12: ’;’ expected
                        public static void main(){
                                                 ^
        4 errors
        </pre>
result: failed
...
username: ahalbardier
...
```


Figure E.1 – Exemple de fichier `.test`

```
[5/90] Testing input file : 5495e4dfaff41404e9ada6cb.test
ahalbardier
{"tests": {}, "stdout": "",
"problems": {"q1": "$ error message : Ecriver seulement le contenu de la Main
@ error line : 1\n<script type=\"text/javascript\"> highlightErrors(); </script>"},
"result": "failed", "stderr": "", "archive": "H4sICApzVFUC/2FyY2hpdmUudGd6AO3R
zWoCMRAH8DzKEHpoodRN2HXBWunFY99AkKijrsasTbK2RXz3ZrGWUhBPFgr/H4SB+YAJ0xHXlyVlWa
aoMq31j/hNKF3oIitVrlJeqbLMBBV/sJtoQjSeSLzVfs3+fN+l+ukjp/hPjMdz5tnETNcPq1C7q92/
m+dn76+7xa/7511dCMpw/6vby62vJ5Y3QfZoL19VCvKG2Pva04ZDMAumHg2nvtqxp8BN6mUXyTJNax
fZNTRjsoZeTOWevwZt5dopNXL9kCa3keLHlp9GMvJ77KzMzhzTIzmgZbVY2vTisBONt3eP1O8cywN5
uCfpOTQ2tnvNTWV5Jg8CAAAAAAAAAAAAAAAAAAAABan25CU9AAKAAA"}

-> Feedback for problem id q1 doesn't match :
         Expected result : <p>
  Erreur de compilation (votre programme est donc mal écrit) :
</p>
<pre>
Syracuse.java:12: illegal start of expression
                public static void main(){
                ^
Syracuse.java:12: illegal start of expression
                public static void main(){
                       ^
Syracuse.java:12: ';' expected
                public static void main(){
                             ^
Syracuse.java:12: ';' expected
                public static void main(){
                                         ^
4 errors
</pre>
         Actual result :
         $ error message : Ecriver seulement le contenu de la Main
         @ error line : 1
        <script type="text/javascript"> highlightErrors(); </script>
```

FIGURE E.2 – Résultat issue de la simulation du fichier à la figure E.1