Rapport du module V&V :

Code coverage

Par : DESCHAMPS Mathieu & ESNAULT Jeremy

Table des matières

I. Introduction :

Dans le cadre du module validation et vérification de notre dernière année de master nous devions réaliser un projet mêlant exécution de tests ,analyse dynamique et introspection de programme.

Nous avons choisis d'implémenter un outil pour de couverture de code, notre programme permettra à son utilisateur d'obtenir la séquence des méthodes appeler lors de l'exécution des tests d'un projet maven ainsi que l'affichage des valeurs des paramètres passé dans ces fonctions (uniquement pour les types primitifs).

Afin d'apporter une bonne expérience à l'utilisateur nous avons décider de créer une IHM à l'aide de JavaFx et d'afficher les résultats en graphe (notre solutions génère des fichier .dot).

Vous pourrez trouver le code source sur notre repository github à l'adresse suivante :

https://github.com/MathieuDeschamps/V-V

II. Solution :

II.1. Fonctionnement :

Pour réaliser les différentes fonctionnalités mentionnées précédemment, nous avons procédé en deux étapes. Une première étape qui va consiste à modifier le code pour récupérer les informations nécessaires à lister les appels de méthode. Une seconde étape qui va consister à construire le graphe des appels de méthodes à partir des informations récupérer à l'étape précédente.

Pour représenter les différents appels de fonctions nous avons opté pour une structure en graphe, dont les nœuds sont les différentes méthodes appelées et les arcs les différents appels réaliser entre les méthodes. De plus, les méthodes et leurs paramètres ont leur propre classe, Enfin à chaque étape correspond une classe TraceFunctionCall pour la première étape et ParseFunctionCall pour la seconde.

II.1.A. Première étape

La première étape consiste à parcourir les différents fichiers source fournissent par l'utilisateur. Lors de ce parcours nous ajoutons du code avant l'appel d'une méthode qui permettra d'enregistrer dans un fichier les noms de la méthode appelée avec leurs paramètres. De plus, nous ajoutons du code à la fin de la méthode pour marquer la sortie de la méthode.

En parallèles de cette étape nous récupérons une liste correspondant à tous les appels de méthodes réalisées dans les fichiers. Ainsi pour un appel de méthode nous construisons un arc avec pour premier nœud la méthode appelante et en second nœud la méthode appelée. Cette liste sera utilisée lors de la seconde étape.

Une fois cette première étape réalisée à la fois dans la suite de test et dans le code testé on exécute la suite de test. L'exécution du code précédemment injecté crée un fichier, dont voici un extrait ci-dessous.

```
#*samples.RecursiveOpTest+positivFact
()
-samples.RecursiveOp+factorielle
(int:3)
-samples.RecursiveOp+factorielle
(int:2)
-samples.RecursiveOp+factorielle
(int:1)
<
<
<
<
```

Illustration 1:Fichier produit par l'exécution d'un test unitaire

II.1.B. Seconde étape

La seconde étape va alors consister à formater le fichier suivant afin de créer plusieurs graphiques un pour chaque test unitaire. Ce graphe sera composé des différents appels réalisé lors de l'exécution d'un test. Nous commençons par séparer le fichier en plusieurs parties une pour chaque test unitaire. Ensuite, en fonction du premier caractère de la ligne une action différente sera réalise:

« * » Création d'un graphe dont le titre est le nom du test unitaire.

« - » Récupération des informations de la méthode appel et ajout de la méthode

à la pile des appels et au graphe précédemment crée.

« (» Récupération des paramètres de la méthode en sommet de pile.

« < » Dépilage de la méthode en sommet de pile.

Enfin nous vérifions que le graphe construit est correcte à l'aide de la liste des appels de méthodes élaborée dans la première partie. Tous les arcs constituant le graphe doivent être dans cette liste. Le graphe alors obtenu est converti en format .dot pour être visualisé par l'IHM.

```
digraph positivFact{
"positivFact()" -> "factorielle(int:3)"[ label="1 "];
"factorielle(int:3)" -> "factorielle(int:2)"[ label="2 "];
"factorielle(int:2)" -> "factorielle(int:1)"[ label="3 "];
}
```

Illustration 2: Résultat en DOT de l'illustration 1

II.2. Manuel d'utilisation :

Voici l'aperçu de notre IHM :

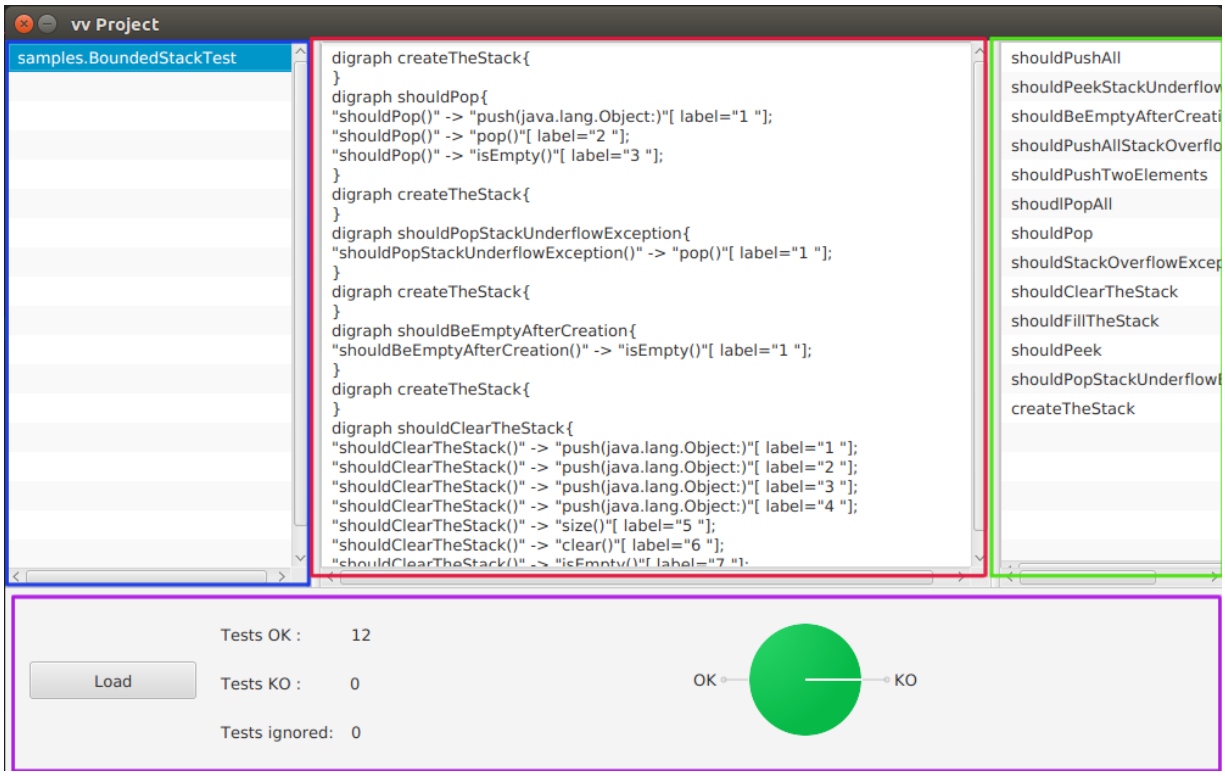

Illustration 3: IHM

Elle est divisé en quatre zone :

-La violette en dessous contient le bouton qui permet de charger un projet maven et affiche le récapitulatif des tests de la classe sélectionnée via un graphique piechart.

-La bleu à gauche permet de sélectionner une classe du projet chargé.

-La rouge au milieu affiche la trace d'exécution par cas de test. Cette trace d'exécution est au format .dot

-La verte à droite permet de sélectionner un cas de test précis dans la classe en cours, à la sélection une nouvelle fenêtre est ouverte avec l'image du graphe d'exécution.

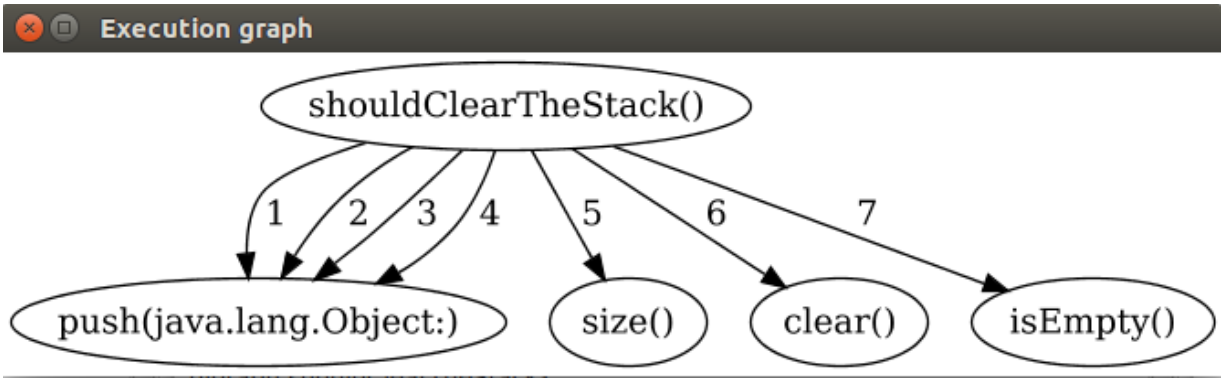

Illustration 4: Graphe d'exécution

Nous avons choisis de numéroter les arcs afin que l'utilisateur comprenne bien la succession des méthodes appelées, nous affichons aussi le type des paramètres des foncions (et leurs valeurs pour les types primitifs).

III. Conclusion :

Nous sommes plutôt satisfait du travail accompli, nous avons réaliser une application fonctionnelle et ergonomique qui correspond au cahier des charges et objectifs que nous nous étions fixé au début du projet.

Néanmoins il y a bien sur un tas d'évolutions possible comme par exemple compter le pourcentage de code qui à été testé.