ScrashLocator, application de CrashLocator pour la détection d'erreur sur le code source Eclipse en Scala

BAILLEUL Quentin, DOUYLLIEZ, PHILIPPON Romain

2 décembre 2014

Table des matières

1 Introduction

Lors de l'utilisation d'un logiciel, il est possible que celui-ci crashe, c'est-à-dire qu'un événement apparaît empêchant son bon fonctionnement. À ce moment, une stack-trace peut être fournie. Celle-ci est un rapport d'erreur qui est généré par l'application contenant la suite d'appels aux fonctions ou méthodes menant au crash.

Cette stack-trace est envoyée sur un serveur, avec l'accord de l'utilisateur, où celle-ci est classée selon certains critères dans des buckets. Un bucket regroupe l'ensemble des stack-traces provenant à priori d'un bug commun. Le but de cette opération est de faciliter le déboggage pour les développeurs.

Dans l'industrie, Windows utilise ce système avec un ensemble de plus de cinq cents critères pour classer les stack-traces 11 provenant des utilisateurs. Il existe encore actuellement de nombreuses recherches sur ce sujet pour classer soit plus efficacement, soit plus rapidement. D'autres tentent d'améliorer la localisation de l'erreur incriminée.

C'est dans ce contexte que CrashLocator apparaît. Le principe est d'attribuer un score aux fonctions ou méthodes pour déterminer quelles sont celles qui provoquent des crashes dans les applications de Mozilla. Nous tenterons d'implémenter en langage Scala l'algorithme de CrashLocator appliqué sur des stack-traces provenant de l'IDE Eclipse. L'enjeu est de vérifier si CrashLocator reste efficace sur un autre type de stack-trace.

2 Approche

2.1 Dataset Crash Eclipse

Pour ce projet nous avons considéré un dataset regroupant 21 912 rapports de bugs de l'IDE Eclipse. Nous considérons le bucketing déjà réalisé manuellement par les développeurs : un bucket est désigné par un identifiant appelé *"groupId"*. Il contient un ou plusieurs rapports ayant son identifiant dénommé *"bugId"* qui est associé à un *"groupId"*.

Pour ce dataset, nous avons 18 623 buckets au total. Cependant, la plupart d'entre eux sont composés d'une seule stacktrace. Or CrashLocator produit des statistiques sous forme de scores à partir de plusieurs stacktraces, ce sont donc des buckets inutiles. En les enlevants, il reste 1944 buckets regroupant en moyenne 2.69 traces chacun. De manière plus générale l'ensemble de ces buckets regroupes 5233 traces.

2.2 Code source d'Eclipse

Pour que l'algorithme fonctionne, il est nécessaire de rechercher les longueurs de chaque méthode présente dans la stacktrace. Pour cela nous avons récupéré le code source de quelques modules principaux d'Eclipse [1] :

`eclipse.platform` : 713 fichiers
`eclipse.platform.releng` : 141 fichiers
`eclipse.platform.releng.buildtools` : 78 fichiers
`eclipse.paltform.runtime` : 575 fichiers
`eclipse.paltform.swt` : 1 821 fichiers
`eclipse.paltform.team` : 1 534 fichiers
`eclipse.paltform.text` : 1 041 fichiers
`eclipse.paltform.ua` : 941 fichiers

Au total, nous avons 59 031 méthodes contenant en moyenne 9.7 lignes chacunes.

2.3 CrashLocator

Selon le papier, il existe pour chaque fonction un score qui dépend de quatre facteurs :

1. **Fréquence de la fonction (FF) :** le nombre d'appel d'une fonction dans un bucket sur le nombre de bucket

   $FF(f, B) = \frac{N_{f,B}}{N_B}$

   f : méthode étudiée
   B : bucket
   $N_{f,B}$: nombre d'appel de f dans B
   N_B : nombre de stack trace dans B

2. **Fréquence Inverse des buckets (IBF) :** le nombre de bucket divisé par le nombre d'appel d'une fonction sur l'ensemble des buckets

   $IBF(f) = \log\left(\frac{\#B}{\#B_f} + 1\right)$

   f : méthode étudiée
   $\#B$: nombre de buckets
   $\#B_f$: nombre de buckets contenant un appel de f

3. **L'inverse de la distance moyenne au point de crash :** la distance de la fonction du point de crash ajoutée au nombre d'appel de la fonction appelée

1. Consulter la section *platform* à cette adresse

$$IAD(f,B) = \frac{N_{f,B}}{1 + \sum_{j=1}^{n} CallDepth_j(f)}$$

f : méthode étudiée
B : bucket
$N_{f,B}$: nombre d'appel de f dans B
N_B : nombre de stack trace dans B
$CallDepth_j(f)$: nombre d'appel de méthodes pour arriver à f

4. **Lignes de codes d'une fonction (FLOC) :** le nombre de lignes de code la fonction étudiée

$$FLOC(f) = \log\left(LOC(f) + 1\right)$$

f : méthode étudiée

Une fois ces quatre facteurs déterminés, la formule suivante est calculée $Score(f) = FF(f,B) * IBF(f) * IAD(f,B) * FLOC(f)$ et son résultat est sauvegardée en mémoire. Quand toutes les méthodes sont évaluées, l'algorithme renvoie la liste des cinq méthodes les plus suspectes.

2.4 Implémentation

Le langage de développement utilisé est Scala pour des questions d'optimisation et de mises à l'échelle. Pour cela, l'emploi des futures permet de calculer simultanément le score de chaque fonction appelée dans le bucket :

```
// barrier intialized with the number of method to compute the ranking
    score
val latch = new CDL(methodSet size)

methodSet foreach { methodName =>
       Future {
              val futureComputeFF: Future[Double] = Future {
                     functionFrequenceScore(bucketId, methodName)
              }
              val futureComputeIBF: Future[Double] = Future {
                     functionInverseBucketFrequency(bucketId,
                         methodName)
              }
              val futureComputeIAD: Future[Double] = Future {
                     functionInverseAverageDistanceCrashPoint(bucketId,
                         methodName)
              }
              val futureComputeFLOC: Future[Double] = Future {
                     functionLineOfCode(methodName)
              }

              for {
                     ff <- futureComputeFF
```

```
                        ibf <- futureComputeIBF
                        iad <- futureComputeIAD
                        floc <- futureComputeFLOC
                } yield {
                        ranking.put(methodName, ff * ibf * iad * floc)
                        latch countDown
                }
        }
}

latch await // returns the result when all computations are over
```

Le code rassemble 431 lignes de codes et se structure autour de cette hiérarchie :

- `idl.bdp.ccrashlocator.processing`
- `idl.bdp.ccrashlocator.bucketing`
- `idl.bdp.ccrashlocator.spoon`

Chaque package permet comme son nom l'indique d'effectuer soit le bucketing, soit l'analyse du code source d'Eclipse soit le calcul du score de suspicion des méthodes contenues dans un bucket donné.

3 Résultats

Il n'est pas possible de vérifier les résultats donnés par ScrashLocator et ce pour plusieurs raisons. La première concerne le calcul du score. En réalité, il existe un autre paramètre à prendre en compte lors du calcul du score IAD : c'est le décalage de pile (ou offset) entre la fonction appelée et le haut de la pile. Or, c'est une notion inexistante en JAVA parce que la stacktrace affiche l'enchainement des appels de méthodes qui ont mené au crash. De ce fait, seule la taille d'une méthode est considérée. Cette valeur passée au logarithme donne à son tour une autre valeur très faible. Cela se répercute alors sur le score de suspicion global qui s'en retrouve drastiquement réduit. En effet, les plus gros score obtenus ne dépassaient pas les 10%.

En outres, la vérification des résultats est impossible avec la méthodologie employée par les développeurs d'Eclipse. Il n'est pas possible de retrouver à partir d'un rapport de bugs [2] le fix qui a été opéré. Il n'existe ni de lien redirigeant vers un commit, ni un détail quelconque concernant la modification effectuée. De plus, la consultation des dépôts Git n'a pas été fructueuse : le soucis est de rechercher le module auquel la méthode appartient sachant que ce module peut appartenir à Eclipse, comme il peut appartenir à un plugin maintenu par le projet Eclipse. Une fois le dépôt trouvé, il faut vérifier qu'un des commits possède bien un message contenant l'ID du bug afin de dire si le bucket a bien été cor-

2. Consulter le site bugs.eclipse.org

rigé. Or pour la plupart des buckets que nous avons analysé, en procédant de cette méthode en recherchant dans les logs Git (aussi bien sur les dépôts propres à Eclipse que sur ses dépôts Github) et cela n'a donné aucun résultat.

4 Conclusion

Le projet est un demi-échec car d'un côté nous avons bien un prototype fonctionnel mais d'un autre, nous ne pouvons pas tester la véracité de ses résultats. Même si nous avons exposé ses défauts, il aurait été intéressant de vérifier déjà dans un premier temps si, parmi les cinq méthodes les plus suspicieuses que renvoit ScrashLocator, au moins l'une d'entre elle est génératrice du crash analysé. Puis pour vérifier de manière plus globale, des calculs de recall et de précision auraient put être envisagés. Grâce à ces résultats, nous aurions pu travailler plus efficacement sur l'amélioration du prototype et rechercher une façon d'exprimer la formule du FLOC pour en trouver une nouvelle plus adaptée au format des stacktraces JAVA.