Rapport de stage

Instantiation de modèles Java réalistes par mesure de métriques

Florian Galinier - florian.galinier@etu.umontpellier.fr

Maître de stage : Clémentine Nébut
Encadrant : Adel Ferdjoukh

7 août 2015

Résumé

L'objectif de ce stage était l'amélioration de l'outil Grimm de génération de modèle en intégrant une génération des relations dirigée par des probabilités obtenues via des mesures réelles.

En effet, l'outil Grimm permettait à partir d'un meta-modèle, par exemple celui de Java, la génération de modèles conformes à celui-ci. Cependant, les modèles ainsi générés étaient relativement uniformes et peu réalistes. Nous avons donc traité le cas de Java et de la génération de modèles et de code de ce langage. Nous avons relevé différentes métriques de programmation orientée objet sur des projets existants, nous permettant d'obtenir des statistiques que nous avons utilisées pour générer des modèles plus réalistes.

Remerciements

Je tiens tout d'abord à remercier Clémentine Nébut pour m'avoir proposé ce stage qui m'a permis de découvrir le monde de la recherche.

Je remercie également énormément Adel Ferdjoukh. Merci à lui d'avoir su me guider et me débloquer, et d'avoir répondu à mes questions (pas toujours en rapport avec le stage). J'ai appris énormément à ses côtés et ai pris vraiment beaucoup de plaisir à effectuer ce stage.

Merci également à Clément pour le covoiturage, d'avoir réussi à me supporter durant ce stage et d'avoir financé une (grosse) partie de mes cafés. Merci également à lui pour la relecture, ainsi qu'à Marjorie.

Table des matières

1 Introduction **4**

2 Statistiques sur les métriques **5**
2.1 Choix des métriques 5
2.1.1 État de l'art 5
2.1.2 Choix des outils 6
2.2 Calcul des métriques 6
2.2.1 Mise en œuvre 6
2.2.2 Données 6

3 Génération de code **12**
3.1 Analyse de l'existant 12
3.1.1 Introspection 12
3.1.2 Bibliothèques 12
3.1.3 Autres solutions 12
3.2 Bibliothèque Source Code Generator 14
3.2.1 Principes 14
3.2.2 Génération des fichiers 14
3.2.3 Problèmes rencontrés lors de la génération 14

4 Résultats **19**
4.1 Protocole des expérimentations 19
4.2 Expérimentations 19
4.2.1 Expérimentations de test 19
4.2.2 Expérimentations retenues 20
4.2.3 Résultats retenus 21
4.3 Résultats métriques 22
4.3.1 Visibilités 22
4.3.2 Répartitions des propriétés dans les classes 23
4.3.3 Autres métriques 23

5 Conclusion **29**

A Méta-modèle Java **33**

B Métriques utilisées **34**

C Projets utilisés **36**
C.1 Liste des projets GitHub 36
C.2 Liste des projets du Qualitas Corpus 38

D Résultats des expérimentations **41**
D.1 Résultats de la première expérimentation 41
D.2 Résultats de la deuxième expérimentation 43
D.3 Résultats de la troisième expérimentation 45
D.4 Résultats de la quatrième expérimentation 48
D.5 Résultats de la cinquième expérimentation 51
D.6 Résultats de la sixième expérimentation 54
D.7 Résultats retenus 56

Bibliographie **62**

Chapitre 1

Introduction

Les méta-modèles sont des abstractions de modèles (des modèles de modèles) utilisés dans le secteur de l'IDM (Ingénierie Dirigée par les Modèles). Ils permettent la description des modèles en associant concepts et relations entre ces concepts ; ils sont ainsi une représentation de la syntaxe des modèles et peuvent être comparés aux grammaires des langages.

L'instantiation d'un méta-modèle a pour objectif la génération d'un (ou plusieurs) modèle(s) conforme(s) au-dit méta-modèle. Dans [Ferdjoukh et al., 2015], une approche de la génération des méta-modèles via l'utilisation de `CSP` (*Constraint Satisfaction Problem*) est proposée. Un des problèmes soulevés dans l'introduction de [Ferdjoukh et al., 2015] est celui de la diversité, et bien qu'il parvienne en partie à y répondre avec l'outil Grimm, les modèles générés restent assez peu réalistes. En effet, pour un méta-modèle Java par exemple, les modèles générés sont très peu semblables à ce qu'un développeur aurait pu écrire.

L'objectif de ce stage était l'amélioration de cet outil grâce à l'utilisation de statistiques sur des métriques de code orienté objet. En compulsant un certain nombre de projets réels, de qualités professionnelles ou non, nous avons pu extraire des lois statistiques que nous pourrions réutiliser pour la génération de modèle réaliste. Nous avons choisi de traiter le cas de projets Java, ce langage étant aujourd'hui parmi les plus répandus et utilisés, ce qui nous permettait d'avoir un corpus potentiel important, mais également un certain nombre d'outils à disposition conséquent.

Le méta-modèle que nous avons utilisé (cf. A.1) est un sous-ensemble du méta-modèle Java standard, ceci nous permettant de nous limiter aux métriques que nous pourrions extraire grâce aux logiciels choisis.

Chapitre 2

Statistiques sur les métriques

2.1 Choix des métriques

2.1.1 État de l'art

La bibliographie fait apparaître un grand nombre de métriques pour le code et notamment pour la programmation orientée objet. Ces métriques ont pour objectif de fournir des données quantitatives permettant l'analyse de la qualité du code. On peut ainsi améliorer un programme en tentant d'obtenir de meilleures métriques sur des points du code à problèmes. Par exemple, des fonctions contenant trop de lignes de codes peuvent traduire un manque de factorisation.

Dans le cadre de notre démarche, ce sont les métriques s'attachant non pas à la forme du code en lui-même qui nous intéressent, mais celles en rapport avec sa structure. Nous avons donc orienté nos recherche exclusivement vers les métriques de la programmation orientée objet, les métriques du type nombre de lignes de codes ne nous apportant rien sur la structure des programmes Java.

L'article de [Chidamber and Kemerer, 1994] décrit six de ces métriques, qui sont :

WMC (Weigthed Method Per Class) : le nombre pondéré par le poids des méthodes par classe. Avec un poids de 1, on obtient ainsi le nombre de méthodes par classe.

DIT (Depth of Inheritance Tree) : la profondeur maximale entre une classe et la racine de son arbre d'héritage.

NOC (Number Of Children) : le nombre de classes qui hérite de la classe mesurée.

CBO (Coupling Between Object classes) : le nombre de classes avec lesquelles la classe mesurée est en relation.

RFC (Response For a Class) : le nombre de méthodes/constructeurs qui peuvent être appelés directement ou indirectement dans une classe.

LCOM (Lack of Cohesion Of Methods) : cette métrique exprime la cohésion d'une classe, i.e. si la classe devrait être découpée en plusieurs classes. Cette métrique a fait l'objet de nombreux débats, notamment sur la façon de la calculer (ref. [Henderson-Sellers, 1995]).

Sur les six métriques proposées par [Chidamber and Kemerer, 1994], celles que nous avons retenues sont `WMC` (avec un poids de 1), `DIT`, `NOC` et `CBO`.

La famille des métriques `MOOD` (`MOOD` et `MOOD2`) proposée dans [Abreu and Carapuça, 1994] contient également un certain nombre de métriques utilisées dans la mesure de programme orienté objet, dont notamment les métriques `MHF` et `AHF` qui sont le pourcentage de méthodes/attributs qui ne sont pas publics. Nous avons retenu uniquement ces deux métriques, les autres métriques n'étant pas assez liées à la structure de Java.

2.1.2 Choix des outils

Notre choix s'est dans un premier temps assez vite tourné vers le plugin *Metrics* pour *Eclipse*[1]. Ce dernier permettait entre autres le calcul des métriques `WMC`, `DIT`, `NOC` (nommé `Number of Sub-Classes (NSC)`), mais aussi des métriques comme le nombre de paramètres par méthodes (`PAR`), le pourcentage de classes abstraites/interfaces (`RMA`). Cet outil nous donnait ainsi accès à un grand nombre de métriques intéressantes. Il nécessitait cependant la compilation d'un projet pour l'analyser (ce qui n'était pas toujours possible étant donné que certains projets étaient fournis avec des bibliothèques manquantes) et possédait peu de documentation pour l'exécution en ligne de commande via les tâches *Ant*.

C'est pour ces raisons que nous l'avons finalement écarté au profit du plugin *CodePro AnalytiX*[2]. Celui-ci, bien que ne fournissant pas l'ensemble des métriques que nous désirions, ne nécessitait pas la compilation des projets pour l'analyse et possède de plus une documentation très détaillée. Ce dernier permet entre autres le calcul des métriques `WMC`, `DIT` et `NOC` moyennes. On retrouve également des métriques qui pourraient être rapprochées à celles proposées par [Abreu and Carapuça, 1994], comme le nombre de méthodes/attributs en fonction de leur visibilité, ainsi que des métriques du style `LOC`, avec des mesures du nombre de lignes, du nombre de caractères, du nombre de commentaires, etc.

2.2 Calcul des métriques

2.2.1 Mise en œuvre

La première partie du stage a été consacrée à l'automatisation du calcul de métriques sur les corpus que nous avions choisis. En effet, nous avons fait le choix de relever les métriques sur deux grandes catégories de projets :

— des projets professionnels ;

— des projets de qualités plus variables.

Nous avons pour le premier cas choisi d'utiliser le *Qualitas Corpus* (ref. [Tempero et al., 2010]) qui contient 112 projets de qualité « professionnelle » d'assez grandes tailles.

Pour notre second corpus, nous avons choisi d'effectuer les mesures sur un ensemble de projets tirés de le forge *GitHub*, qui met à disposition une API[3] permettant le parcours et le téléchargement de projets. L'utilisation de la bibliothèque Java `org.kohsuke.github-api`[4] basée sur cette API nous a permis l'automatisation du téléchargement de projets publics, indépendamment de leur qualité, contenu et taille.

Nous avons utilisé les tâches *Ant* proposées par *CodePro AnalytiX* nous permettant de déclencher le calcul de métrique sur un projet donné. Pour chaque projet, nous pouvions ainsi générer un fichier au format `CSV` (*Comma-Separated Values*) contenant les différentes métriques choisies.

L'automatisation du calcul des métriques sur le Qualitas Corpus a quant à elle était faite via des scripts *Bash*.

2.2.2 Données

Sur les 112 projets contenus dans le Qualitas Corpus, certains ont été écartés en raison de contraintes techniques (lorsque les fichiers sources n'étaient pas inclus notamment), et nous avons au final exécuté le calcul sur 98 projets (cf. C.2).

Nous avons également exécuté le calcul de métriques sur une liste de 100 projets tirés de GitHub (cf. C.1).

1. http ://metrics.sourceforge.net/
2. https ://developers.google.com/java-dev-tools/codepro
3. https ://developer.github.com/
4. http ://github-api.kohsuke.org/

Nous avons pu extraire des données recueillies un certain nombre d'informations intéressantes. Elles laissent tout d'abord apparaître trois grandes catégories :

- les métriques suivants une loi normale (cf. Fig. 2.1) ;
- les métriques suivants une loi exponentielle inverse (cf. Fig. 2.2) ;
- les autres métriques, ne suivants aucune loi.

Ainsi, si les valeurs numériques ne sont pas les mêmes sur les deux corpus, on observe cependant que globalement les lois qui ressortent sont identiques. En effet, par exemple, le nombre de classe par package suit une loi exponentielle dans le cas du Qualitas Corpus et du Corpus GitHub (cf. Fig. 2.3).

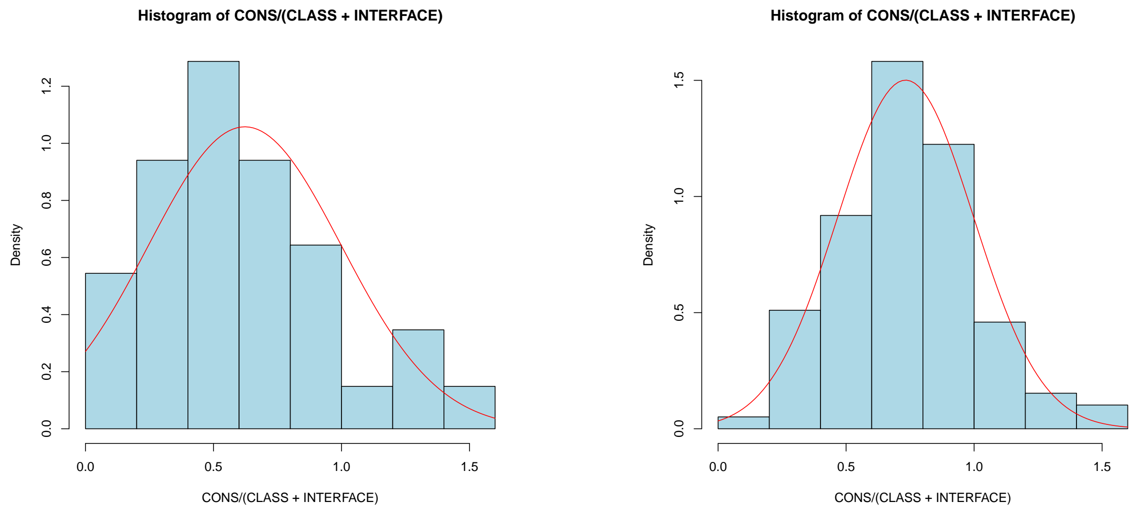


FIGURE 2.1 – Nombre moyen de constructeurs par classe (Git | QCorpus). En rouge, la loi probabiliste théorique.

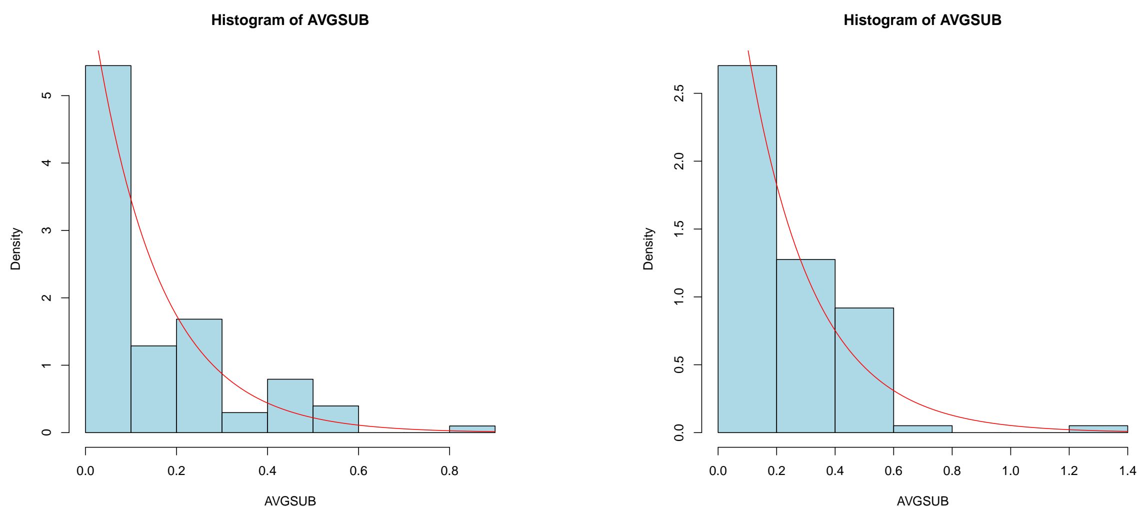


FIGURE 2.2 – Nombre moyen de sous-classes par classe (Git | QCorpus). En rouge, la loi probabiliste théorique.

En outre, les deux corpus ont des pourcentages de répartition de visibilité des attributs assez proches (moins de 5 points d'écart pour chaque visibilité ; cf. Fig. 2.4) et des répartitions de visibilité des méthodes quasi-similaires (moins de 1.5 points d'écart ; cf. Fig. 2.5).

Les deux corpus diffèrent cependant sur deux points :

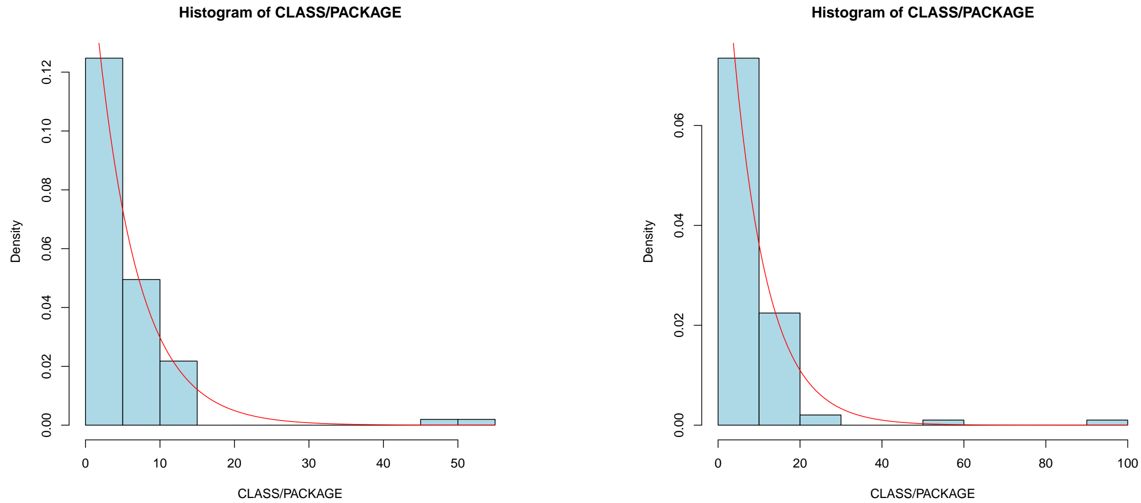


FIGURE 2.3 – Nombre moyen de classes par package (Git | QCorpus). En rouge, la loi probabiliste théorique.

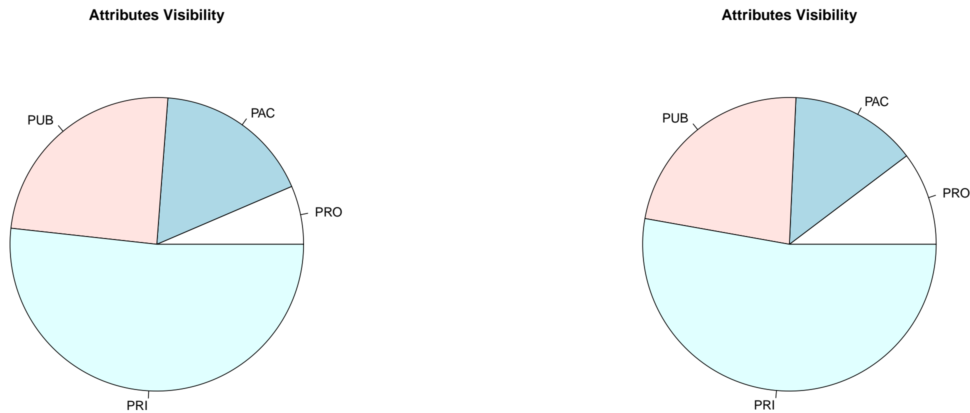


FIGURE 2.4 – Répartition moyenne des visibilités des attributs (Git | QCorpus)

— les attributs par classes (cf. Fig. 2.6) ;
— et les méthodes par classes (cf Fig. 2.7).

En effet, dans les deux cas, ces statistiques suivent une loi normale sur le Qualitas Corpus et une loi exponentielle sur le Corpus GitHub.

Les données métriques ainsi récupérées lors de la première partie de ce stage nous ont permis d'extraire les lois observées sur les modèles Java. L'application Grimm a ainsi été modifiée pour pouvoir générer des relations suivant les deux lois probabilistes observées (ainsi que la génération de manière uniforme) par l'ajout de contraintes globales au CSP. La résolution de ce CSP devait ainsi permettre la génération de modèle respectant les observations statistiques.

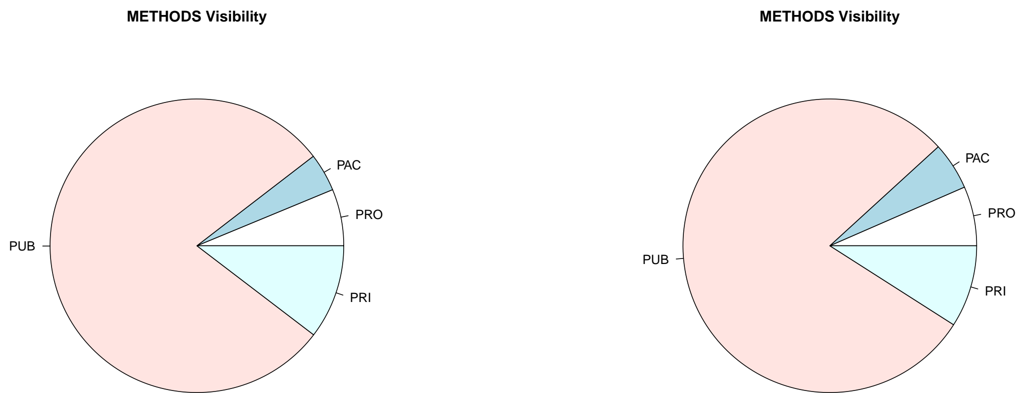


FIGURE 2.5 – Répartition moyenne des visibilités des méthodes (Git | QCorpus)

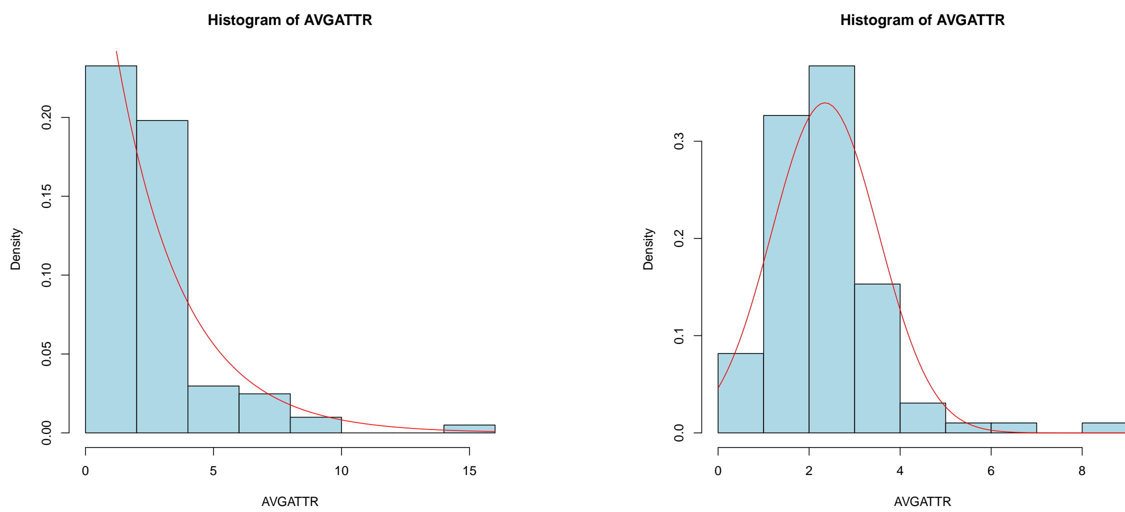
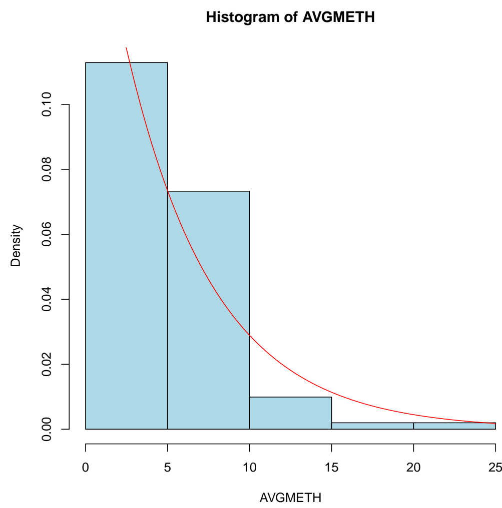


FIGURE 2.6 – Nombre moyen d'attributs par classe (Git | QCorpus). En rouge, la loi probabiliste théorique.



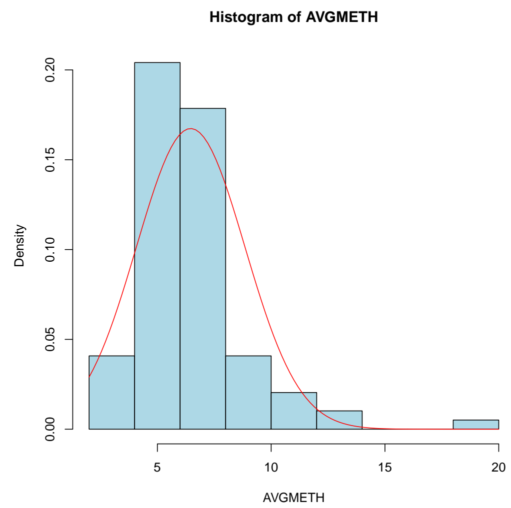
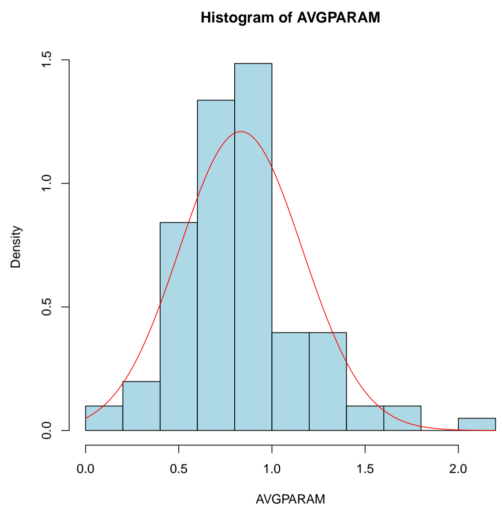


Figure 2.7 – Nombre moyen de méthodes par classe (Git | QCorpus). En rouge, la loi probabiliste théorique.



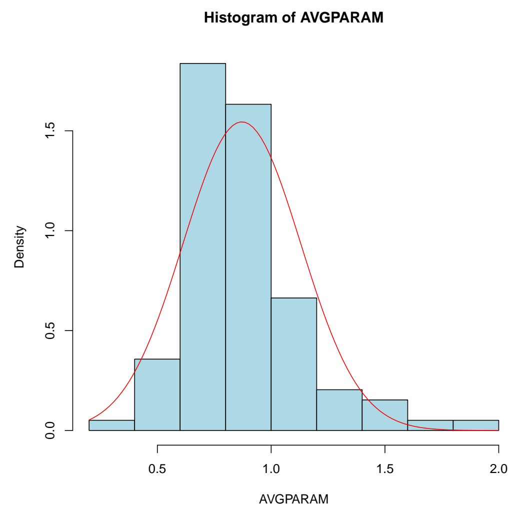
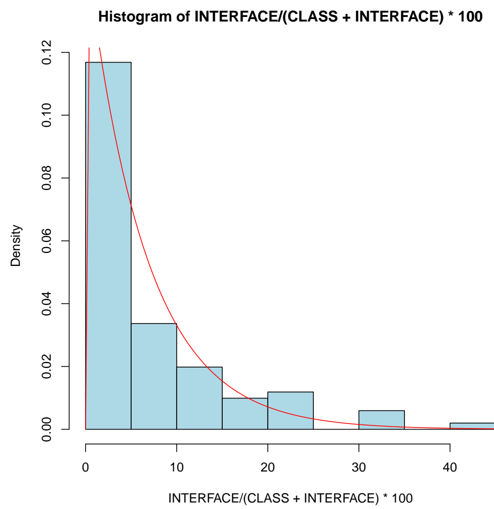


Figure 2.8 – Nombre moyen de paramètres par méthode (Git | QCorpus). En rouge, la loi probabiliste théorique.



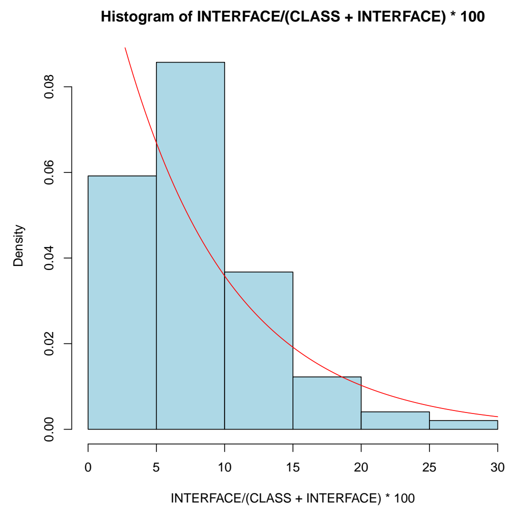


FIGURE 2.9 – Nombre moyen d'interfaces par package (Git | QCorpus). En rouge, la loi probabiliste théorique.

Chapitre 3

Génération de code

3.1 Analyse de l'existant

3.1.1 Introspection

L'objectif de cette partie du stage était de générer, à partir des modèles créés par Grimm, du code source Java, compilable si possible, permettant de fournir un exemple plus réel et tangible de la génération.

La première piste fut celle de l'introspection (ref. [Oracle, 2014]). Il est cependant apparu assez vite que l'introspection et la reflexion de Java ne permettent de travailler que sur du code existant, et ne permettent ainsi pas la génération dynamique de code, comme le permet C# par exemple [1].

3.1.2 Bibliothèques

La seconde piste a été de regarder du côté des bibliothèques existantes permettant la génération de code à l'exécution. Il existe ainsi un grand nombre de bibliothèques permettant la génération et l'exécution de code durant l'exécution d'un programme. C'est ainsi ce que permet la bibliothèque *Jumbo* proposée par [Kamin et al., 2003]. Elle ne permet cependant pas la génération de fichier source Java, et c'est pour cette raison que nous l'avons écartée.

La seconde bibliothèque étudiée fut la *Byte Code Engineering Library* (ref. [Apache, 2009]) qui permettait de générer des fichiers. Cependant, comme son nom l'indique, c'est en manipulant du byte code Java que cette bibliothèque travaille et elle permet donc uniquement la génération de fichiers `.class`. La difficulté d'utilisation, vu l'obligation de passer par du byte code et le fait que nous aurions dû passer par une phase lourde de décompilation, nous a amené à écarter également l'utilisation de cette bibliothèque.

3.1.3 Autres solutions

C'est dans la littérature que nous avons finalement trouvé une dernière piste. En effet, dans [Forman et al., 2004], le chapitre 7 propose une solution pour la génération de code et l'utilisation de l'API *reflect* de Java pour générer les `.class` et tester la validité du code généré. Le lien vers les sources étant morts, nous avons pris la décision de créer une bibliothèque semblable mais plus adaptée à nos besoins.

Figure 3.1 – Diagramme UML de la bibliothèque SCGenerator

3.2 Bibliothèque Source Code Generator

3.2.1 Principes

La principale idée de la bibliothèque *Source Code Generator* (abrégée par la suite en *SC-Generator*) était de créer en Java un méta-modèle de Java. En effet, chaque élément de Java qui nous intéressait est représenté dans la bibliothèque par une classe et les relations entre ces éléments par des relations entre les classes de la bibliothèque. Nous obtenons ainsi quelque chose de très proche du méta-modèle Java que nous avions choisi d'utiliser (cf. 3.1).

Chaque classe devait ensuite pouvoir être transformée en code source. Ceci a été fait par la surcharge de la méthode `toString()` qui, pour chaque élément de la bibliothèque, renvoie une chaîne de caractères correspondant à son écriture en code source Java. Par ailleurs, les éléments contenus par d'autres éléments sont automatiquement ajoutés lors de l'écriture des éléments conteneurs (par exemple, la méthode `toString()` de la classe `SCClass` renverra une chaîne de caractères contenant la définition de cette classe mais également des méthodes et attributs contenus). Cependant, contrairement à ce qui est habituel, ce sont les classes et les interfaces qui contiennent le package dans lequel elles se situent et non pas l'inverse.

3.2.2 Génération des fichiers

Le point d'entrée de la bibliothèque est la classe `SCGenerator`. C'est elle qui permettra la création d'éléments (les constructeurs des différents éléments étant protégés) et qui les stockera dans une liste. Elle permet également de récupérer un élément déjà existant via son identifiant et l'écriture des objets complexes (`SCClass` et `SCInterface`) dans des fichiers `.java`.

L'appel à la méthode `generate()` de `SCGenerator` aura également pour conséquence de générer des fichiers pour tous les types complexes suivis par l'instance de `SCGenerator` et de les placer dans des dossiers correspondants aux packages de la classes ; i.e. une classe ayant pour package `a.b.(...).c` sera générée dans le dossier `a/b/.../c`.

3.2.3 Problèmes rencontrés lors de la génération

Un des principaux problèmes rencontrés lors de la conception de la bibliothèque était l'obtention de code compilable. En effet, nous avons dû, à plusieurs reprises, modifier la façon dont nous générions le code à cause de code non compilable.

Les constructeurs

Un de ces problèmes, sans doute le plus notable, était celui des constructeurs ; il s'est agit en réalité de deux problèmes. Le premier problème fut celui des constructeurs avec même nombre de paramètres et de mêmes types. Le code généré (voir exemple List. 3.1) n'était ainsi pas compilable en raison de la duplication des constructeurs. La solution que nous avons choisi a consisté à générer aléatoirement le type des paramètres, diminuant ainsi grandement les chances d'obtenir deux constructeurs avec une signature identique.

```
public class C8 {
  public C8 (int P1,int P2,int P3) {}
  public C8 (int P4,int P5,int P6) {}
}
```

Listing 3.1 – Constructeurs à même signature générés

1. https ://msdn.microsoft.com/en-us/library/650ax5cx%28v=vs.110%29.aspx

Le second problème des constructeurs fut celui des constructeurs sans paramètres. Si le problème de plusieurs constructeurs avec n paramètres $\mid n > 1$ était résolu par la méthode expliquée précédement, nous avions cependant encore des classes générées non compilables telles que dans l'exemple List. 3.2. En effet, les signatures des constructeurs encore une fois étaient identiques mais ne pouvaient être différenciées par le type de leurs paramètres. Nous avons choisi de résoudre ce problème en supprimant simplement la génération de constructeurs vides sans paramètres.

```java
public class C8 {
  public C8 () {}
  public C8 () {}
  public C8 (int P1, boolean P2, byte P3) {}
  public C8 (char P4, char P5, int P6) {}
}
```

Listing 3.2 – Génération de plusieurs constructeurs vides

Cycle dans l'héritage

Une autre erreur que nous avons rencontrée lors de l'ajout de la bibliothèque dans Grimm fut celle des cycles d'héritage. En effet, dès les premiers tests de génération, nous avons eu affaire à du code non compilable en raison de la présence de classes héritant d'elles-mêmes (cf. Fig. 3.2). Plutôt que de modifier la bibliothèque (le problème étant plus lié au méta-modèle qu'à la bibliothèque elle-même), nous avons fait le choix d'ajouter une contrainte `OCL` (*Object Constraint Language*) empêchant une classe de s'étendre elle-même, ce qui a fait disparaitre ce premier problème mais en a fait apparaître un second.

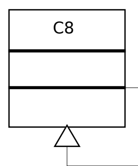


FIGURE 3.2 – Classe héritant d'elle-même

En effet, bien que nous n'avions plus le problème d'une classe héritant d'elle-même, nous avions cependant des modèles générés possédant des cycles dans la hiérarchie d'héritage et, par conséquent, étant incompilables (cf. Fig. 3.3). C'est par l'ajout de contraintes `OCL` une fois encore que fut résolu ce problème.

Problème de transtypage implicite

Lors de la génération d'une méthode, le type de retour est analysé par la bibliothèque et une valeur par défaut est retournée pour ne pas avoir d'erreur de compilation (cf. List. 3.3).

Si les transtypages implicites ne sont absolument pas un problème en Java dans ces cas-là, nous avons eu des erreurs au niveau des constructeurs. En effet, lorsqu'une classe hérite d'une classe possédant (au moins) un constructeur, son (ou ses) constructeur(s) doi(ven)t faire appel à au moins un des constructeurs de la super-classe. C'est pour cette raison que la bibliothèque génère automatiquement un appel à `super()` dans les constructeurs en utilisant les valeurs par défaut pour les types des paramètres (cf. List. 3.4).

Les transtypages implicites ne sont cependant pas admis dans ces cas-là, ce qui entraînait des erreurs lors de la compilation du code généré. La solution retenue pour contourner ce problème

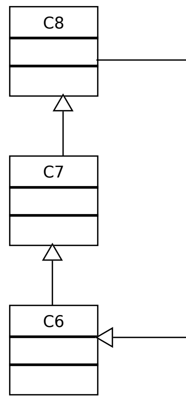


FIGURE 3.3 – Cycle dans l'héritage

```
public boolean M13 (byte P34) {
  return false;
}

public void M14 () {
  return ;
}

public float M15 () {
  return 0;
}
```

Listing 3.3 – Exemples de méthodes générées

```
public class C10 {
  public C10 (boolean P45,byte P46) {}
}

public class C9 extends C10 {
  public C9 (byte P47) {
    super(false ,0);
  }
}
```

Listing 3.4 – Exemple d'appel non fonctionnel à `super()` généré

a été de modifier la valeur par défaut des types primitifs pour que le transtypage soit explicite (cf. List. 3.5).

Cette correction est devenue obsolète après la correction du problème de l'absence de redéfinition des constructeurs hérités (cf. 3.2.3).

Redéfinition des constructeurs

Le dernier problème a avoir été corrigé (durant le début de la phase d'expérimentation) fut celui de la redéfinition des constructeurs. En effet, dans le cas où une classe héritant d'une autre classe possédant un constructeur avec paramètres ne possède elle-même pas de constructeur (exemple List. 3.6), la compilation est impossible. L'absence de surcharge ou de redéfinition de

```java
public class C10 {
  public C10 (boolean P45,byte P46) {}
}

public class C9 extends C10 {
  public C9 (byte P47) {
    super(false ,(byte) 0);
  }
}
```

Listing 3.5 – Exemple d’appel fonctionnel à `super()` généré

constructeurs faisant appel à `super()` entraîne ici une erreur.

```java
public class C9 {
  public C9 (int P1,boolean P2) {}
}

public class C8 extends C9 {
  // Erreur lors de la compilation :
  // C8.java:3: error: constructor C9 in class C9 cannot be applied
      to given types;
}
```

Listing 3.6 – Code non compilable dû à l’absence de constructeurs dans C8

Nous avons dans un premier temps tenté de résoudre ce problème en augmentant simplement le nombre d’instances de constructeur générées, mais il est apparu qu’il faudrait un nombre beaucoup trop important de constructeurs pour que le rendu final soit réaliste. La solution choisie a été de modifier la génération des classes pour que chacune d’entre elles ait toujours un constructeur sans paramètres par défaut faisant appel à `super()` (cf. List. 3.7).

```java
public class C9 {
  public C9 () {
    super();
  }
  public C9 (int P1,boolean P2) {
    super();
  }
}

public class C8 extends C9 {
  public C8 () {
    super();
  }
}
```

Listing 3.7 – Génération de constructeurs vides pour chaque classe

Ainsi, les classes auxquelles nous n’ajoutions pas de constructeurs étaient tout de même pourvues d’au moins un constructeur : le constructeur sans paramètre. Par ailleurs, toutes les classes possédant désormais un constructeur sans paramètre, la génération des constructeurs a

été simplifiée pour faire appel à `super()` sans paramètre dans tous les cas (les classes n'héritant pas d'autres classes faisant ainsi appel au super-constructeur sans paramètre de `Object`).

Chapitre 4

Résultats

4.1 Protocole des expérimentations

Afin de tester si la nouvelle application permettait une génération de modèles plus réalistes, nous avons fait le choix de lancer la génération de modèles de diverses tailles avec 3 versions de Grimm :

— l'ancienne version avec seulement la génération de fichier ajoutée (nommée simplement *Grimm* par la suite) ;

— une version permettant la génération de fichier et contenant des contraintes `OCL` en plus (le cassage des cycles par exemple) (nommée *Grimm avec optimisations*) ;

— ainsi qu'une version possédant les contraintes de la version précédente et avec prise en compte des statistiques sur les métriques (nommée *Grimm avec statistiques*).

Pour chaque modèle généré, nous stockions son temps de résolution ainsi que la compilabilité des fichiers sources générés.

4.2 Expérimentations

4.2.1 Expérimentations de test

Première expérimentation

La première expérimentation a mis en évidence de nombreux problèmes. En effet, si les résolutions se sont passées comme prévu dans les cas de la version standard de Grimm, les versions optimisée et avec statistiques n'ont donné des résultats que pour un nombre d'instances inférieur à 1000 (cf. Tables D.2 et D.3). En effet, le solveur utilisé, *Abscon* (ref. [Merchez et al., 2001]), ne parvenait pas à résoudre le CSP généré dû à un trop grand nombre de contraintes. Une réécriture de ces dernières a permis la résolution de ce problème.

En outre, bien que les modèles générés avec la version standard de Grimm ne soient pas compilables comme prévu (cf. Table D.1), le fait que les quatre seuls modèles des versions avec optimisations et avec statistiques ne le soient pas n'était pas le résultat escompté. Nous avons donc étudié les sources générées pour déterminer la raison de cette absence de compilation. Nous avons trouvé 2 cas :

— un projet de la version avec optimisations ne compilait pas car deux constructeurs d'une même classe avait été générés avec les mêmes nombres de paramètres et les mêmes types dans le même ordre. C'est un résultat aléatoire que nous espérons corriger avec la version avec statistique et il est donc normal de trouver de tels résultats ;

— les deux projets de la version avec statistiques ainsi que l'autre projet de la version avec optimisations ne compilaient pas en raison du problème évoqué en 3.2.3 (non corrigé à ce moment-là).

Les problèmes ont été corrigés pour la deuxième phase d'expérimentation.

Deuxième expérimentation

Lors de cette deuxième phase, nous avons choisi de lancer les tests sur des nombres d'instances plus bas pour vérifier s'il subsistait des problèmes. Nous avons pu observer durant cette expérimentation que le nombre de projets compilables parmi ceux générés par la version avec optimisations a largement augmenté (8 projets sur 10 compilables ; cf. Table D.5).

Bien que le nombre de projets compilables parmi ceux générés par la version avec statistiques ait également augmenté, nous avions cependant un nombre de modèles compilables inférieur à celui de la version avec optimisations (6 contre 8 ; cf. Table D.6).

L'étude des sources générées nous a montré que le problème qui empêchait la compilation était l'absence de constructeurs dans une classe héritant d'une classe possédant un constructeur. Nous avions alors pris la décision d'augmenter le nombre d'instances pour les constructeurs pour la prochaine phase d'expérimentation.

Troisième expérimentation

Lors de la troisième phase d'expérimentation, nous avons lancé les données sur des nombres d'instances plus élevés, ce qui nous a permis de constater que la résolution du problème des contraintes était efficace (cf. Tables D.8 et D.9).

Nous avons également observé que les données pour la version avec statistiques (Table D.9) étaient semblables à celles de la seconde phase. L'observation des sources générées nous a également permis de déterminer que le problème de redéfinition de constructeur était toujours présent. Nous avons alors pris la décision de rajouter dans chaque classe un constructeur sans paramètres faisant appel à `super()` (cf. 3.2.3).

4.2.2 Expérimentations retenues

Quatrième expérimentation

Cette quatrième phase d'expérimentation fut la première à donner des résultats intéressants. En effet, nous obtenons 16 résultats compilables pour la version avec optimisations (cf. Table D.11) et 36 résultats compilables pour la version avec statistiques sur 40 générations (cf. Table D.12).

Les résultats non compilables dans la version avec statistiques l'étaient en raison d'un problème que nous avions rencontré lors de la deuxième phase et que nous pensions avoir résolu ; les projets n'étaient pas compilables car les attributs des classes étaient de type `void`, réservé normalement aux méthodes.

En effet, dans le méta-modèle (cf. Fig. A.1) les attributs (*Variable*) et les méthodes (*Method*) sont tous deux en relation avec les types. Ainsi, les types primitifs étaient générés en prenant en compte le type `void` en raison de la relation avec les méthodes, ce qui entraînait parfois la relation dans le modèle généré entre les instances des attributs et ce type primitif. Nous avons finalement pris la décision de générer de façon aléatoire les types des attributs et des méthodes en ne tenant pas compte des relations générées dans le modèle, étant donné que nous n'avons pas de statistiques sur les types des attributs et des méthodes.

Ce sont les premiers résultats que nous avons décidé de retenir. En effet, la seule « erreur » restante n'a pas influencé les résultats, celle-ci pouvant se retrouver de façon identique dans la version avec statistiques et dans la version avec optimisations (la version standard ne produisant toujours que du code non compilable).

Cinquième expérimentation

Cette phase d'expérimentation nous a permis d'obtenir dans les modèles générés avec la version avec statistiques que peu de projets non compilables (cf. Table D.15) et un peu plus

dans la version avec optimisations (cf. Table D.14). Ces projets n'étaient pas compilables en raison de la présence de deux constructeurs identiques dans certaines classes ; on remarque cependant déjà de façon claire que l'utilisation des statistiques réelles permettent d'améliorer grandement le nombre de projets compilables.

Par ailleurs, les résultats restent semblables à ceux de la phase précédente.

Sixième expérimentation

Cette expérimentation a été lancée sur un échantillon de 20 générations afin d'arrondir le nombre total de projets pour chaque version à 100 (les deux phases précédentes ayant été lancées sur des échantillons de 40 générations). Les résultats sont également ici très semblables aux deux précédents.

4.2.3 Résultats retenus

Les résultats obtenus mettent en évidence que l'utilisation de probabilités permet la génération d'un plus grand nombre de projets compilables (cf. Fig. 4.1). En effet, les projets qui demeurent non compilables le sont du fait qu'il existe deux constructeurs avec mêmes paramètres. Le type des paramètres étant aléatoire, c'est la répartition des-dits paramètres qui permet de rendre un projet compilable ou non (i.e. une classe avec deux constructeurs avec même nombre de paramètres possède une chance de rendre le projet incompilable).

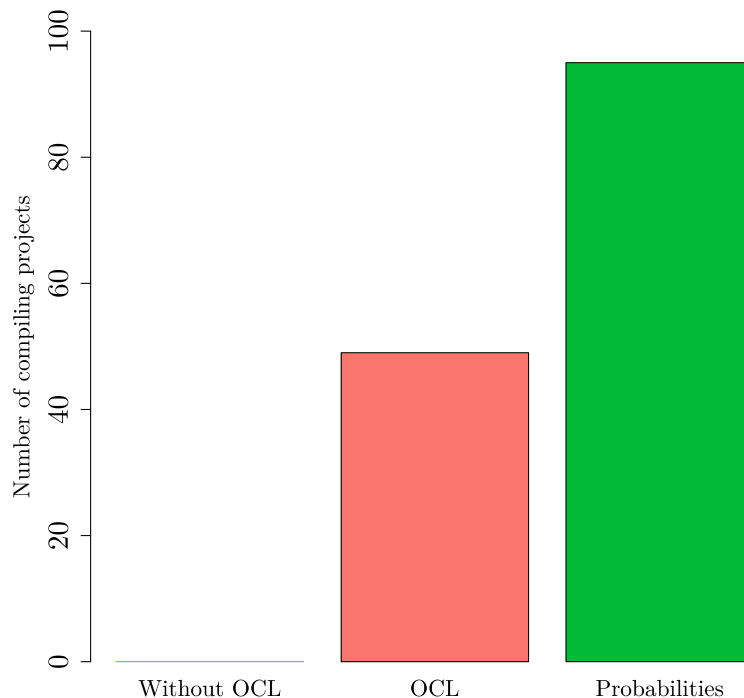


FIGURE 4.1 – Compilabilité des projets générés pour chaque versions

Ainsi, il y a beaucoup plus de chance de se retrouver avec une classe ayant deux constructeurs identiques dans les projets générés par les versions sans statistiques qu'avec celle avec statistiques ; en effet, la répartition des constructeurs y est plus hétérogène et il en est de même pour le nombre des paramètres des-dits constructeurs.

Les projets générés sans optimisations `OCL` et sans statistiques ne sont quant à eux pas compilable, les modèles générés contenant systématiquement un ou plusieurs problèmes corrigés par les contraintes (cycle dans l'héritage, etc.).

On observe cependant (cf. Fig. 4.2) que l'ajout de contraintes `OCL` augmente, de manière prévisible, le temps de résolution du `CSP` généré. Ainsi, si les temps de résolution sont linéaires pour la première version, ils sont exponentiels pour les deux autres versions.

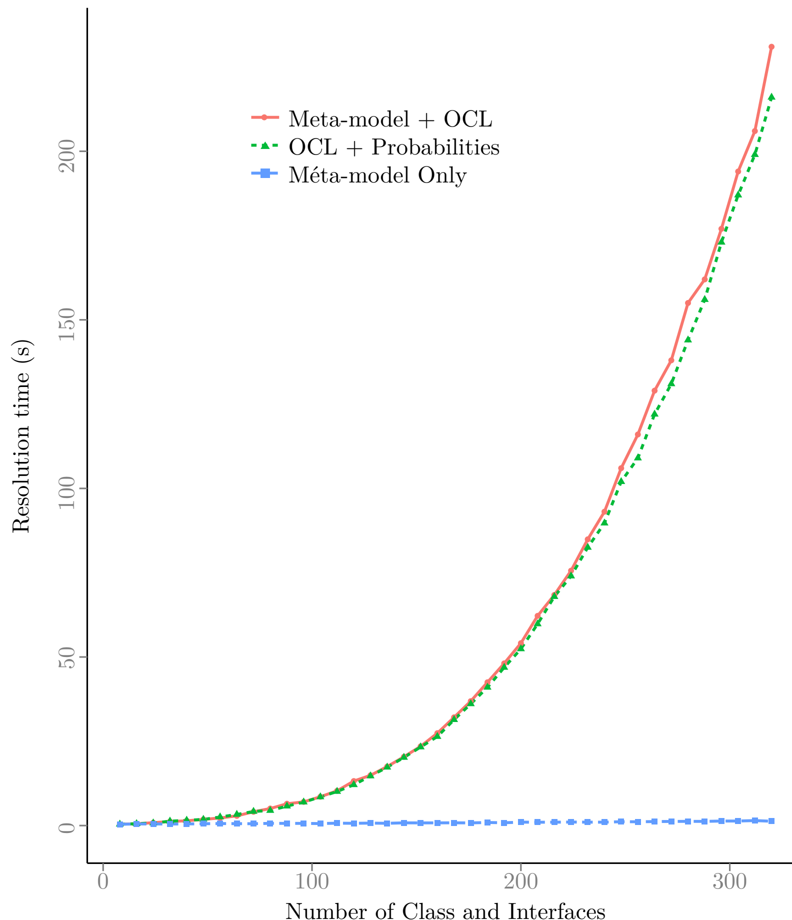


FIGURE 4.2 – Temps de résolutions des `CSP` pour les différentes versions

On remarque par ailleurs que les temps de génération pour la version avec statistiques sont légèrement plus rapides que ceux de la version avec optimisations, sans que l'on ne puisse pour l'instant expliquer pourquoi.

4.3 Résultats métriques

4.3.1 Visibilités

Très tôt dans les expérimentations, la répartition métrique des visibilités obtenues s'est avérée très proche de celles mesurées sur les deux corpus. En effet, on peut observer Fig. 4.3 et Fig. 4.4 que les répartitions des différentes visibilités sont semblables entre les projets réels et les projets générés.

On obtient ainsi un modèle `Java` plus réaliste au niveau des visibilités, d'autant plus que l'on observe que dans les cas de projets générés sans prise en compte des statistiques, le solveur va associer à toutes les propriétés une unique visibilité.

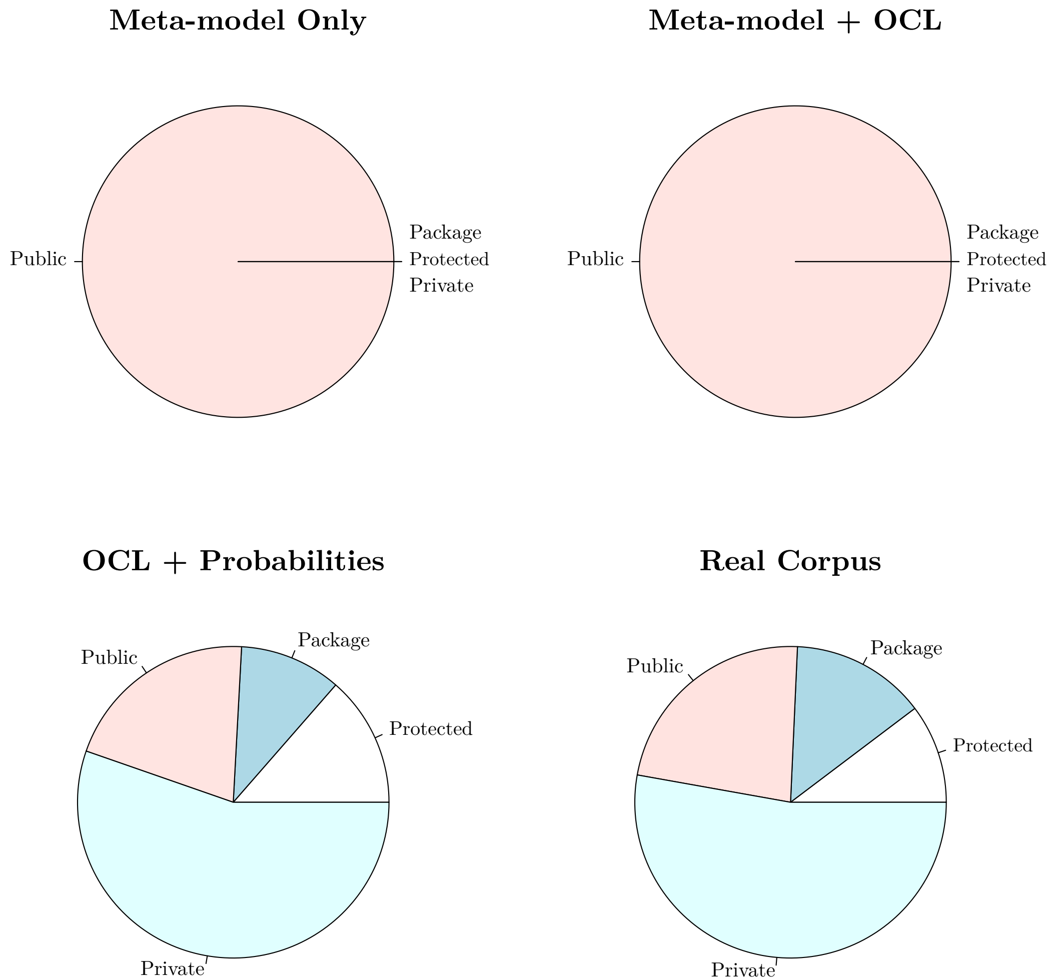


FIGURE 4.3 – Répartitions des visibilités des attributs dans les différentes versions

4.3.2 Répartitions des propriétés dans les classes

Dans les figures 4.5 à 4.7, on peut observer que, dans les projets générés par les versions sans statistiques, les répartitions des différentes propriétés des classes (attributs, méthodes, constructeurs) sont faites de façon uniforme ou peu réaliste.

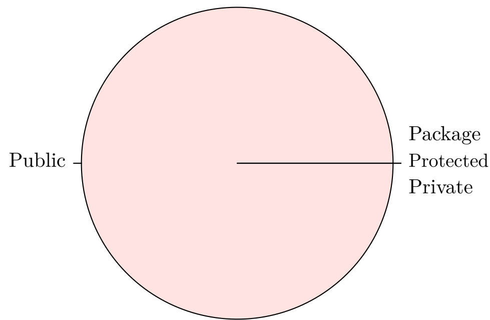
En effet, on observe par exemple Fig. 4.5 que la version avec optimisations génère des classes contenant toutes le même nombre d'attributs, de façon totalement uniforme, ce qui est extrêmement peu réaliste. A contrario, sur la même figure on peut observer que la version avec statistiques produit des projets ayant une répartition suivant une loi normale, de manière semblable à la répartition réelle.

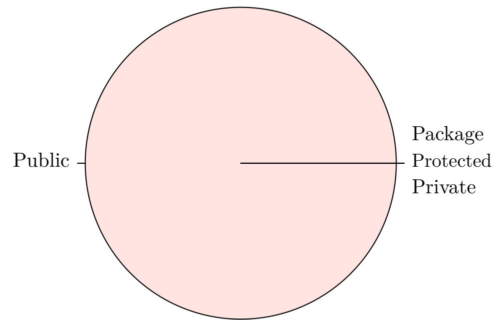
On peut donc estimer que l'objectif de génération réaliste est ici atteint.

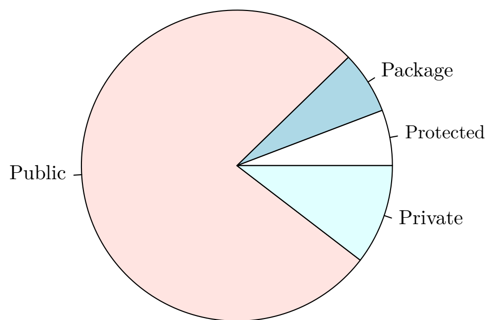
4.3.3 Autres métriques

La répartition des paramètres par méthodes (cf. Fig. 4.8) amène des conclusions similaires aux conclusions de la partie 4.3.2. Il est cependant intéressant de noter que c'est en raison de la génération de manière uniforme des paramètres par classes des versions sans statistiques que l'on obtient des projets non compilables.

En effet, c'est cette répartition qui entraîne l'apparition de plusieurs constructeurs avec un







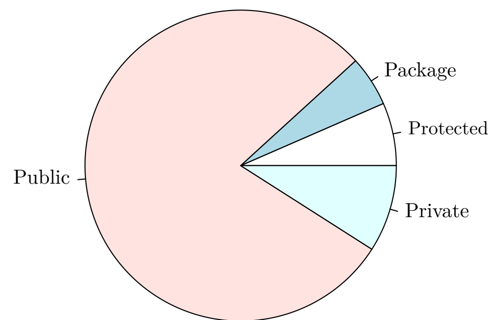


FIGURE 4.4 – Répartitions des visibilités des méthodes dans les différentes versions

nombre de paramètres identiques dans les projets générés. Il « suffit » alors que la génération aléatoire des types des-dits paramètres les dotent de types identiques et dans le même ordre pour que le projet ainsi généré ne soit plus compilable.

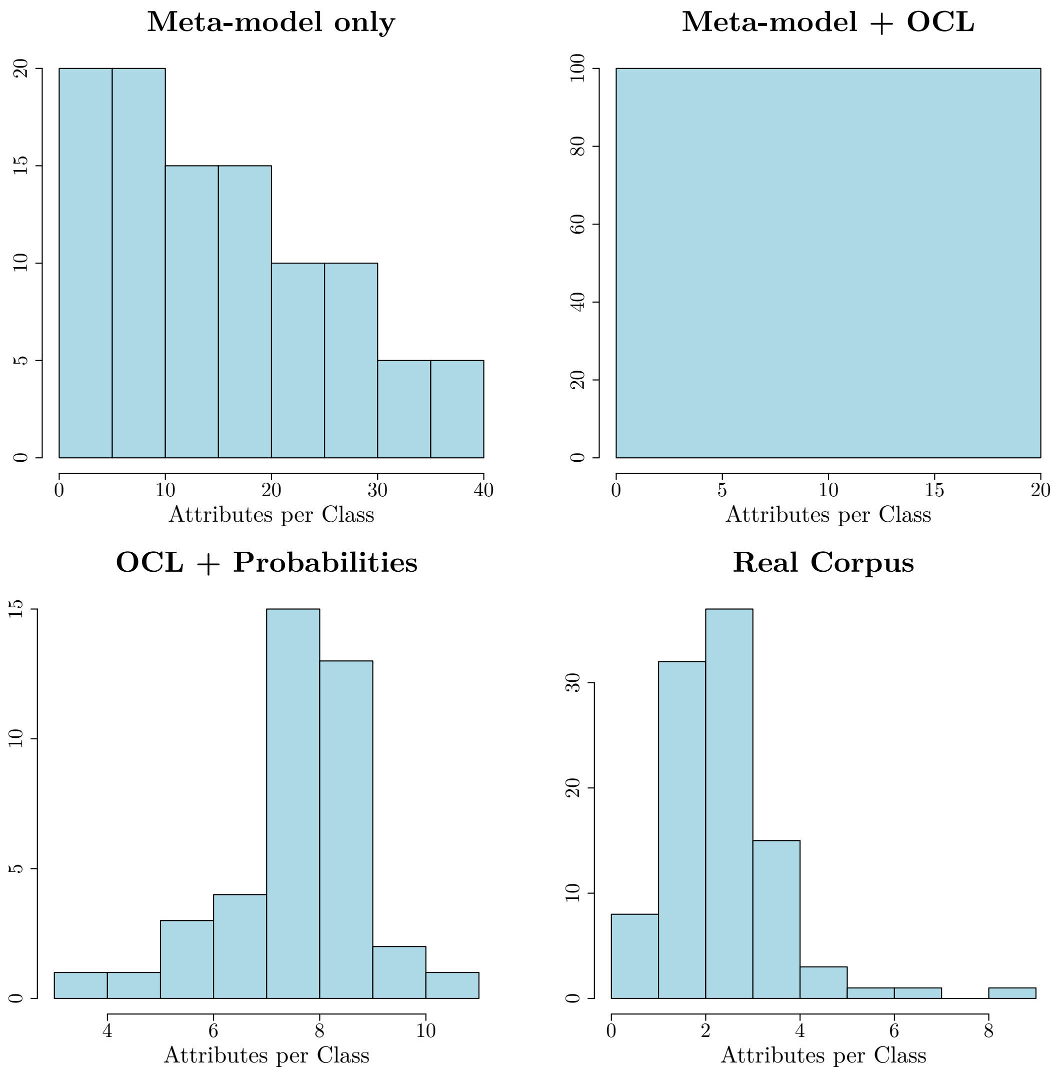


FIGURE 4.5 – Répartitions des attributs par classe dans les différentes versions.

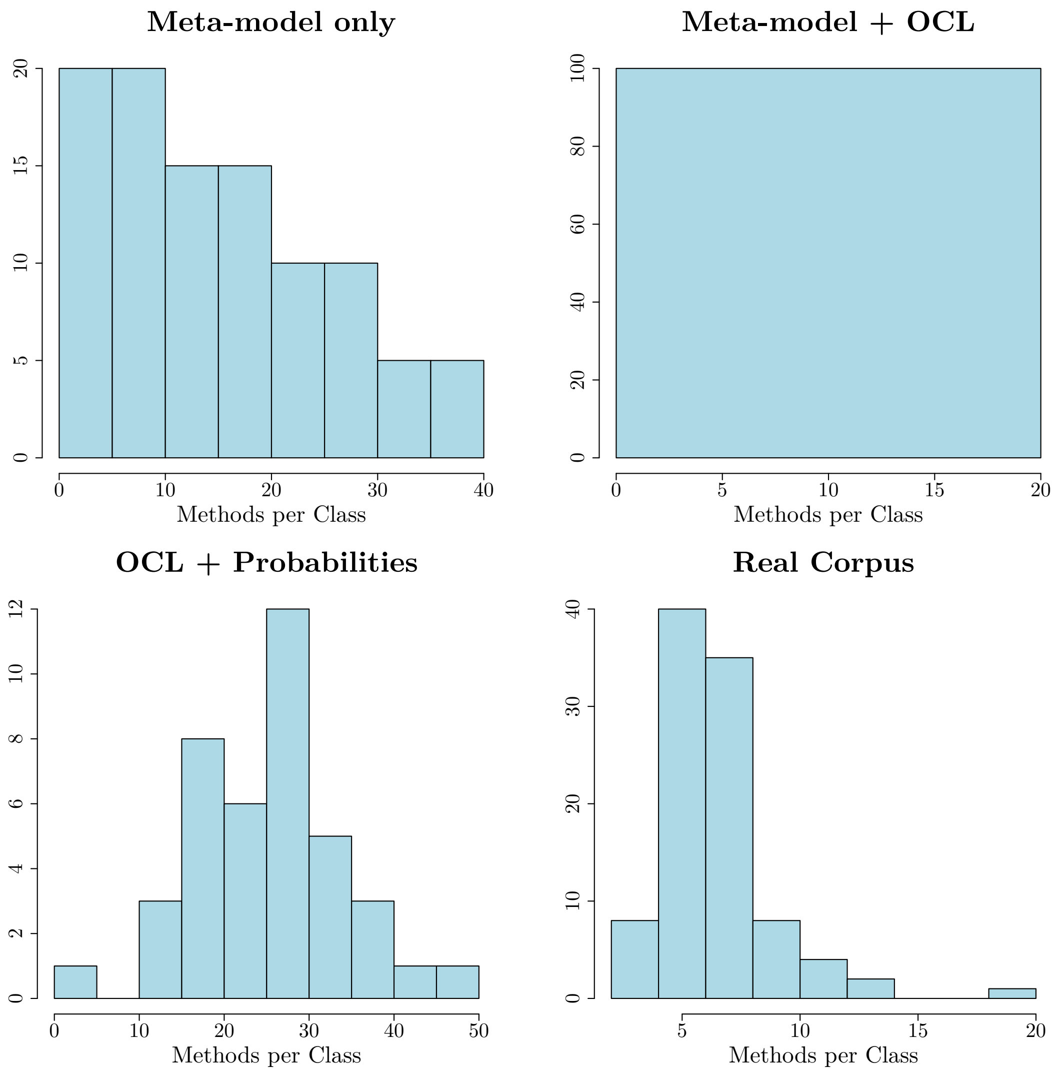


FIGURE 4.6 – Répartitions des méthodes par classe dans les différentes versions.

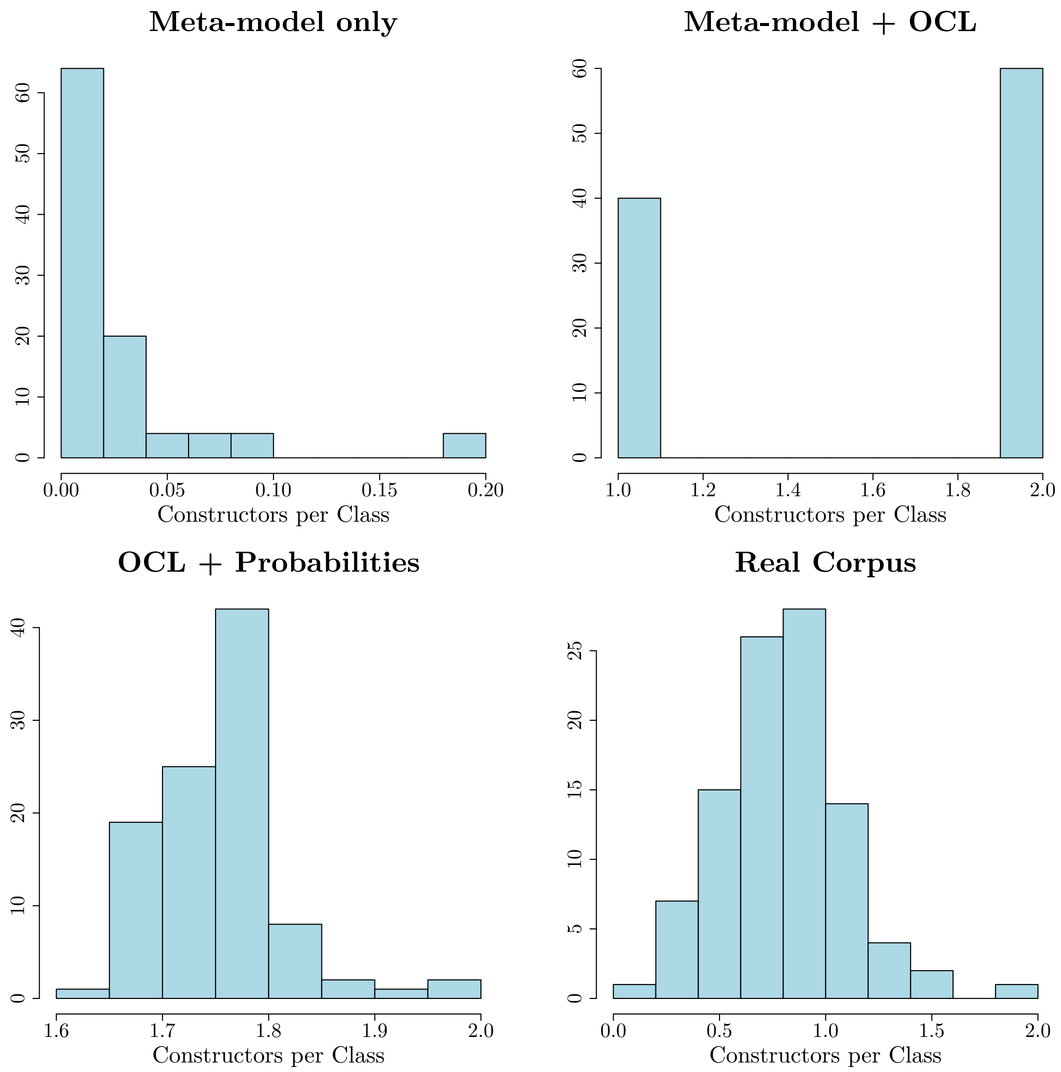


FIGURE 4.7 – Répartitions des constructeurs par classe dans les différentes versions.

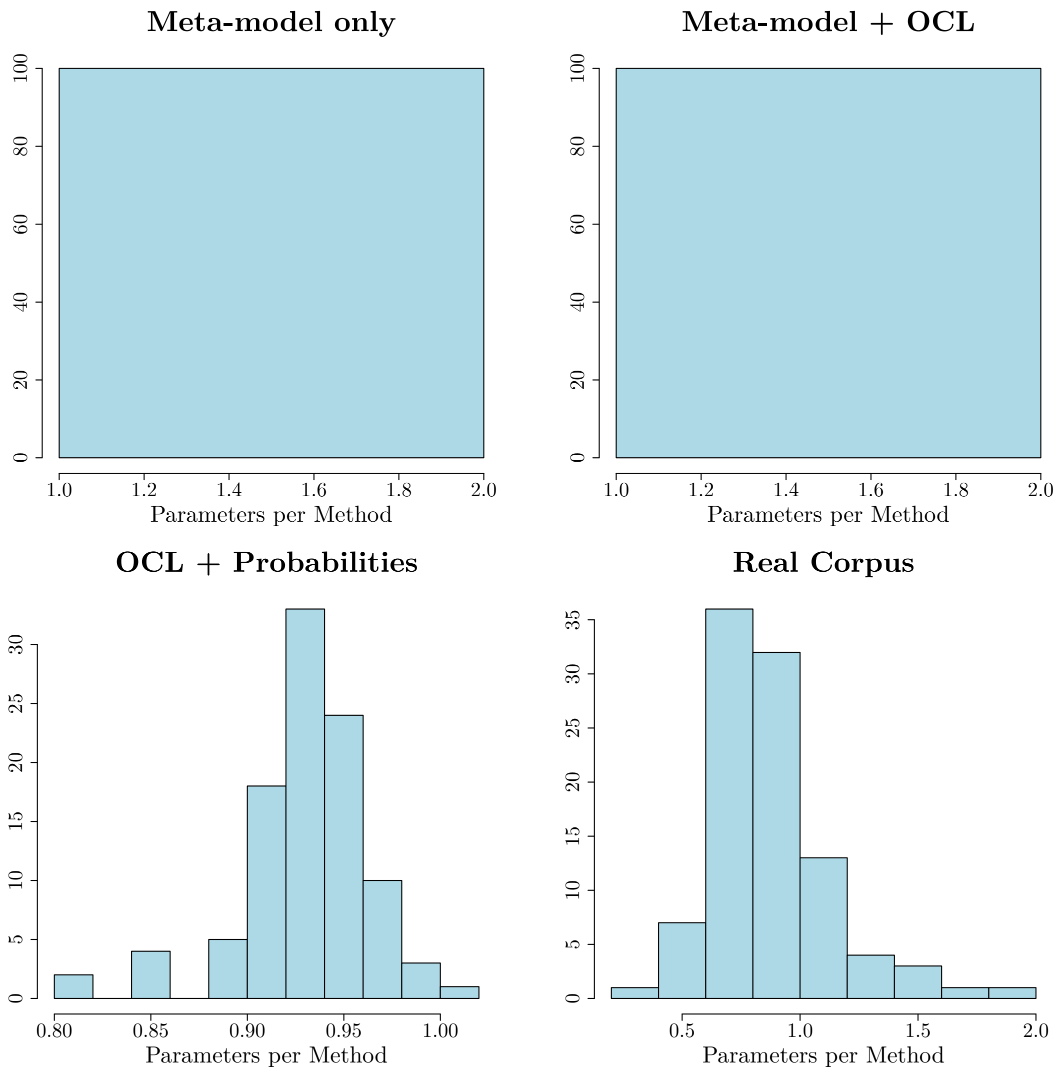


FIGURE 4.8 – Répartitions des paramètres par méthode dans les différentes versions.

Chapitre 5

Conclusion

L'objectif de réalisme du stage a semble-t-il été (tout du moins en partie) atteint. En effet, l'instantiation du méta-modèle `Java` avec l'ancienne version de Grimm ne générait pas des modèles proches des modèles réels (voir partie 4.3), mais de plus elle générait des projets qui, à cause de ce manque de réalisme, n'étaient pas compilables.

De plus, ce projet nous a permis de nous rendre compte que la compilabilité d'un modèle généré peut ne tenir qu'à peu de choses ; sans les contraintes `OCL`, quasiment tous les projets générés comportent des cycles dans leurs héritages.

La génération de code compilable et réaliste ouvre la voie à une possible utilisation dans le cadre des tests de compilateur. On peut en effet imaginer générer un corpus de projets plus ou moins important pour obtenir des jeux de tests proches des projets réels. Il n'y aurait ainsi plus besoin de recourir à une longue et souvent fastidieuse recherche de projets (bien que certains projets aillent dans ce sens, comme vu dans la partie 2.2.1 de ce rapport). Il faudrait cependant pour cela encore améliorer l'outil en prenant en compte d'autres métriques auxquelles nous n'avons pas eu accès, faute d'outils adéquats (et peut-être développer nous-même l'outil adéquat). On peut encore imaginer améliorer le projet en ajoutant le support d'un méta-modèle plus complet dans le but d'en obtenir un plus proche du méta-modèle `Java` réel (en incluant les énumérations par exemple).

Par ailleurs, d'un point de vue personnel, ce stage m'aura permis de m'ouvrir au monde de la recherche et d'en découvrir des aspects plus concrets. J'ai ainsi pu travailler avec beaucoup de plaisir avec des personnes passionnées et apprendre énormément. Celui-ci m'a par ailleurs conforté dans mes choix d'orientations et m'a apporté un réel goût pour la recherche scientifique.

Table des figures

2.1 Nombre moyen de constructeurs par classe (Git | QCorpus). En rouge, la loi probabiliste théorique. 7
2.2 Nombre moyen de sous-classes par classe (Git | QCorpus). En rouge, la loi probabiliste théorique. 7
2.3 Nombre moyen de classes par package (Git | QCorpus). En rouge, la loi probabiliste théorique. 8
2.4 Répartition moyenne des visibilités des attributs (Git | QCorpus) 8
2.5 Répartition moyenne des visibilités des méthodes (Git | QCorpus) 9
2.6 Nombre moyen d'attributs par classe (Git | QCorpus). En rouge, la loi probabiliste théorique. 9
2.7 Nombre moyen de méthodes par classe (Git | QCorpus). En rouge, la loi probabiliste théorique. 10
2.8 Nombre moyen de paramètres par méthode (Git | QCorpus). En rouge, la loi probabiliste théorique. 10
2.9 Nombre moyen d'interfaces par package (Git | QCorpus). En rouge, la loi probabiliste théorique. 11

3.1 Diagramme UML de la bibliothèque SCGenerator 13
3.2 Classe héritant d'elle-même 15
3.3 Cycle dans l'héritage 16

4.1 Compilabilité des projets générés pour chaque versions 21
4.2 Temps de résolutions des `CSP` pour les différentes versions 22
4.3 Répartitions des visibilités des attributs dans les différentes versions 23
4.4 Répartitions des visibilités des méthodes dans les différentes versions 24
4.5 Répartitions des attributs par classe dans les différentes versions. 25
4.6 Répartitions des méthodes par classe dans les différentes versions. 26
4.7 Répartitions des constructeurs par classe dans les différentes versions. 27
4.8 Répartitions des paramètres par méthode dans les différentes versions. 28

A.1 Méta-modèle Java utilisé 33

Liste des tableaux

D.1 Résultats du 28/07 : Grimm 41
D.2 Résultats du 28/07 : Grimm avec optimisations 41
D.3 Résultats du 28/07 : Grimm avec statistiques 42
D.4 Résultats du 29/07 : Grimm 43
D.5 Résultats du 29/07 : Grimm avec optimisations 43
D.6 Résultats du 29/07 : Grimm avec statistiques 44
D.7 Résultats du 30/07 : Grimm 45
D.8 Résultats du 30/07 : Grimm avec optimisations 46
D.9 Résultats du 30/07 : Grimm avec statistiques 47
D.10 Résultats du 31/07 : Grimm 48
D.11 Résultats du 31/07 : Grimm avec optimisations 49
D.12 Résultats du 31/07 : Grimm avec statistiques 50
D.13 Seconds résultats du 31/07 : Grimm 51
D.14 Seconds résultats du 31/07 : Grimm avec optimisations 52
D.15 Seconds résultats du 31/07 : Grimm avec statistiques 53
D.16 Résultats du 04/08 : Grimm 54
D.17 Résultats du 04/08 : Grimm avec optimisations 55
D.18 Résultats du 04/08 : Grimm avec statistiques 55
D.19 Résultats retenus : Grimm 58
D.20 Résultats retenus : Grimm avec optimisations 60
D.21 Résultats retenus : Grimm avec statistiques 62

Listings

3.1 Constructeurs à même signature générés 14
3.2 Génération de plusieurs constructeurs vides 15
3.3 Exemples de méthodes générées 16
3.4 Exemple d'appel non fonctionnel à `super()` généré 16
3.5 Exemple d'appel fonctionnel à `super()` généré 17
3.6 Code non compilable dû à l'absence de constructeurs dans C8 17
3.7 Génération de constructeurs vides pour chaque classe 17

Annexe A

Méta-modèle Java

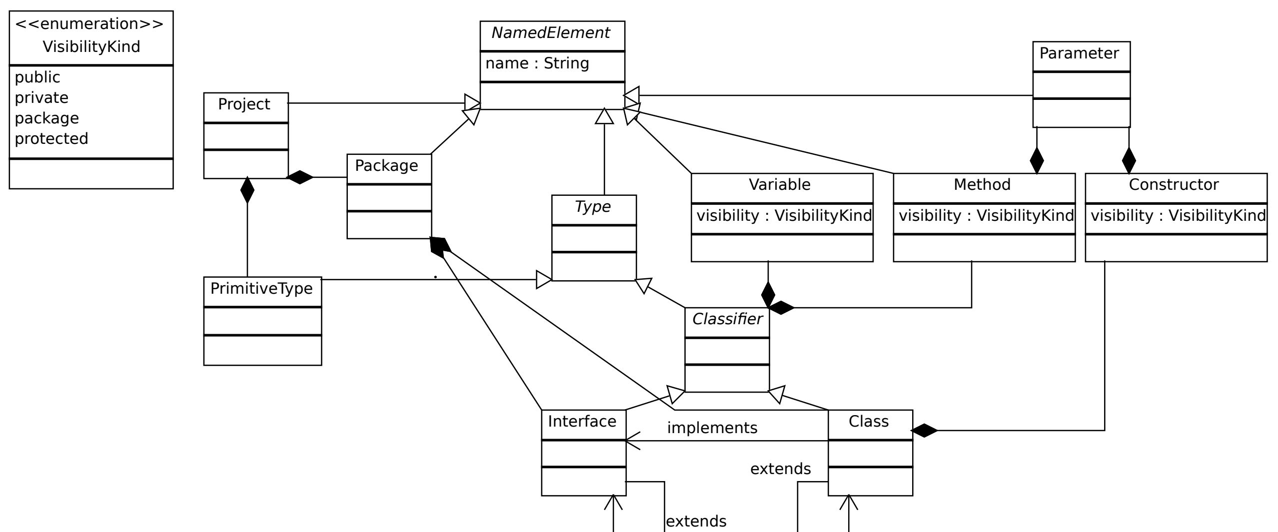


Figure A.1 – Méta-modèle Java utilisé

Annexe B

Métriques utilisées

Les différentes métriques proposées par *CodePro AnalytiX* choisies sont les suivantes :

- Abstractness
- Average Depth of Inheritance Hierarchy
- Average Number of Contructors Per Type
- Average Number of Fiels Per Type
- Average Number of Methods Per Type
- Average Number of Parameters
- Average Number of Subtypes
- Numbers of Constructors
  - public
  - protected
  - package
  - private
- Number of Fields
  - instance
    - public
    - protected
    - package
    - private
  - static
    - public
    - protected
    - package
    - private
- Number of Methods
  - instance
    - public
    - protected
    - package
    - private
  - static
    - public
    - protected
    - package
    - private
- Number of Packages
- Number of Types
  - interface

  - public
  - protected
  - package
  - private
- class
  - public
  - protected
  - package
  - private

Annexe C

Projets utilisés

C.1 Liste des projets GitHub

- Afton/whiteboard
- agentcoops/gridsweeper
- alx/reprap-host-software
- ashanan/cosmo
- astubbs/wicket-contrib-groovy
- azambuja/pagerankmapreduce
- boboroshi/rsm
- booleanman/rackinterfacefilter
- britt/github-java
- britt/hivedb
- britt/hivedb-blobject
- bsimpson/halcyon
- chrisvest/nanopool
- codeprimate/password-vault
- credmp/maven-emacs-plugin
- cyberfox/jbidwatcher
- dalen/familytree
- davidB/scala-maven-plugin
- davidyang/yappr
- davie/jettybuilder
- davie/old-git4idea
- dpc/dpcgoban
- dustin/diggwatch
- dustin/java-digg
- dustin/java-memcached-client
- dustin/spyjar
- dustin/web-thermometer
- easytiger/jquoteticker
- ellen/cosmo
- eml/java-mogilefs
- endisd/penrose-server
- fbrunel/twitterdroid
- Fudge/gitidea
- gingerhendrix/monkeytest
- hamish/easystart
- hamish/easyweb

— henrosoft/henrosoftbrickles
— henrosoft/nehsics
— identityxx/penrose-server
— identityxx/penrose-studio
— jbfeldis/upshot-smash-uploader
— jboner/remoteproxy
— jicksta/stomptomanagerbridge
— jkriss/monomic
— joshsmoore/androidmoney
— jrray/gotefarm
— jrudolph/jcosmos
— jrudolph/scolorz
— klauern/callingruby
— koke/amlights
— LukeHoersten/nonblocking
— maborg/weshowthemoney-com
— mbadran/yarc
— mheath/adbcj
— mircea/tigus
— mojodna/searchable
— monojohnny/jdbc-tester
— mreid/acrp
— mreid/geovex
— mreid/siroc
— mth/yeti
— mtodd/halcyon-clients
— mudge/collapsing-puzzle
— mvdetsen/mvdetsen
— myabc/mediateca
— myabc/nbgit
— myabc/nbmerb
— myabc/roller-defensio
— n0ha/yui-compressor-ant-task
— nam/ltt
— nitenichiryu/nbgit
— nuance/java-nlp-utils
— olabini/jvyamlb
— parabuzzle/toobs
— pdorrell/tenblocks
— pdxrod/clearcut
— pgm/ittyflow
— Pluto/aipluto
— raykrueger/hibernate-memcached
— rictic/gunit
— roddotnet/clearcut
— roes/familytree
— scharris/jDBMD
— simonpk/fireeagle-updater-midlet
— simonpk/j2me-oauth
— sl4mmy/ant-skeleton
— slagyr/limelight

— slangevi/bruse
— soemirno/timesheets
— spoike/doodle
— takai/jruby-dsl-example
— timshadel/jboss-rules-presentation-may-2007
— timshadel/logging-datastore
— travis/cosmo
— tristan/django-servlet
— uggedal/halcyon
— victorr/jsqueak
— w4x/boolangstudio
— wwwjscom/ir--top-k-graphs
— ymirpl/nsaper

C.2 Liste des projets du Qualitas Corpus

— Ant
— ANTLR
— aoi
— Axion
— Batik SVG Toolkit
— C-JDBC
— castor
— Apache Cayenne
— Checkstyle
— Cobertura
— Colt
— Columba
— Apache Commons Collections
— Compiere
— Display Tag Library
— drawswf
— DrJava
— EMMA : a free Java code coverage tool
— Exo Platform
— Find Bugs
— Fitjava
— FitLibrary
— freecol
— FreeCS
— FreeMind
— Galleon
— Gantt Project
— GeoTools
— Hadoop Common
— Heritrix
— Hibernate
— HyperSQL
— HtmlUnit
— Informa
— iText PDF

— ivata op
— JAG
— james
— Java Assembling Language
— JasperReports
— javacc
— jboss
— jchempaint
— jEdit
— Jena
— jext
— jFin Date Math
— JFreeChart
— JGraph
— JGraphPad
— JGraphT
— JGroups
— JMeter
— JMoney
— JOggPlayer
— JParse
— JPF
— JREFactory
— Java powered Ruby implementation
— JSP Wiki
— jstock
— JjsXe
— JTOpen
— JUnit
— Log4j
— Lucene
— Marauroa
— MegaMek
— mvnForum
— Naked Objects
— NekoHTML
— OpenJMS
— OSCache
— PicoContainer
— PMD
— POI
— Pooka
— ProGuard
— Quartz
— QuickServer
— Quilt
— The Roller Weblogger
— SableCC
— sandmark
— Java Runtime Analysis Toolkit
— SQuirrel SQL

— Struts
— sunflow
— Tomcat
— Trove
— Velocity Engine
— Vuze
— Web Curator Tool
— WebMail
— Weka
— Xalan
— Xerces
— XMOJO

Annexe D

Résultats des expérimentations

Dans tous les tableaux ci-dessous, un 0 dans la colonne `Compilable ?` indique que le modèle généré n'était pas compilable tandis qu'un 1 indique que la compilation a réussie sans erreur.

D.1 Résultats de la première expérimentation

Nombre d'instances	Temps de résolution (s)	Compilable ?
470	2.96	0
4700	1953	0
940	15.0	0
1410	50.1	0
1880	120.	0
2350	235.	0
2820	407.	0
3290	671.	0
3760	978.	0
4230	1426	0

Table D.1 – Résultats du 28/07 : Grimm

Nombre d'instances	Temps de résolution (s)	Compilable ?
470	95.9	0
4700	0	
940	1816	0
1410	0	
1880	0	
2350	0	
2820	0	
3290	0	
3760	0	
4230	0	

Table D.2 – Résultats du 28/07 : Grimm avec optimisations

Nombre d'instances	Temps de résolution (s)	Compilable ?
470	96.0	0
4700	0	
940	2819	0
1410	0	
1880	0	
2350	0	
2820	0	
3290	0	
3760	0	
4230	0	

Table D.3 – Résultats du 28/07 : Grimm avec statistiques

D.2 Résultats de la deuxième expérimentation

Nombre d'instances	Temps de résolution (s)	Compilable ?
49	0.35	0
490	0.59	0
98	0.38	0
147	0.42	0
196	0.47	0
245	0.52	0
294	0.48	0
343	0.51	0
392	0.53	0
441	0.53	0

Table D.4 – Résultats du 29/07 : Grimm

Nombre d'instances	Temps de résolution (s)	Compilable ?
49	0.40	1
490	5.02	1
98	0.54	1
147	0.79	1
196	1.20	1
245	1.63	1
294	1.89	0
343	2.43	0
392	2.92	1
441	3.77	1

Table D.5 – Résultats du 29/07 : Grimm avec optimisations

Nombre d'instances	Temps de résolution (s)	Compilable ?
49	0.39	1
490	5.04	0
98	0.57	1
147	0.85	1
196	1.11	1
245	1.48	1
294	1.72	0
343	2.07	1
392	2.94	0
441	3.49	0

Table D.6 – Résultats du 29/07 : Grimm avec statistiques

D.3 Résultats de la troisième expérimentation

Nombre d'instances	Temps de résolution (s)	Compilable ?
49	0.34	0
490	0.65	0
539	0.55	0
588	0.67	0
637	0.69	0
686	0.71	0
735	0.65	0
784	0.68	0
833	0.69	0
882	0.67	0
931	0.98	0
98	0.36	0
980	0.95	0
1029	0.74	0
1078	0.72	0
1127	0.77	0
1176	0.77	0
1225	0.77	0
1274	0.80	0
1323	0.92	0
1372	0.90	0
1421	1.03	0
147	0.47	0
1470	1.03	0
196	0.42	0
245	0.45	0
294	0.51	0
343	0.50	0
392	0.51	0
441	0.65	0

Table D.7 – Résultats du 30/07 : Grimm

Nombre d'instances	Temps de résolution (s)	Compilable ?
49	0.39	1
490	5.04	0
539	6.52	1
588	7.87	1
637	9.75	1
686	11.4	1
735	12.8	0
784	16.7	1
833	17.9	1
882	21.6	1
931	27.6	0
98	0.58	1
980	33.0	1
1029	31.9	0
1078	36.7	0
1127	42.1	0
1176	47.8	0
1225	52.8	0
1274	59.6	1
1323	70.6	1
1372	76.0	0
1421	84.1	1
147	0.96	1
1470	93.1	0
196	1.09	0
245	1.63	1
294	2.23	0
343	2.77	0
392	3.44	1
441	4.36	1

Table D.8 – Résultats du 30/07 : Grimm avec optimisations

Nombre d'instances	Temps de résolution (s)	Compilable ?
49	0.41	1
490	4.86	0
539	6.05	0
588	8.24	1
637	9.04	1
686	10.5	0
735	12.9	0
784	15.1	0
833	18.3	0
882	23.7	1
931	26.7	0
98	0.64	1
980	29.2	0
1029	31.4	1
1078	36.0	0
1127	41.7	0
1176	47.3	0
1225	53.1	0
1274	60.0	0
1323	65.6	0
1372	73.7	0
1421	81.6	0
147	0.87	1
1470	90.4	0
196	1.40	1
245	1.68	1
294	1.91	0
343	2.49	1
392	3.81	0
441	4.13	0

Table D.9 – Résultats du 30/07 : Grimm avec statistiques

D.4 Résultats de la quatrième expérimentation

Nombre d'instances	Temps de résolution (s)	Compilable ?
49	0.34	0
490	0.57	0
539	0.57	0
588	0.55	0
637	0.58	0
686	0.64	0
735	0.64	0
784	0.69	0
833	0.66	0
882	0.67	0
931	0.71	0
98	0.38	0
980	0.68	0
1029	0.72	0
1078	0.72	0
1127	0.74	0
1176	0.74	0
1225	0.85	0
1274	0.93	0
1323	0.89	0
1372	0.94	0
1421	0.99	0
147	0.39	0
1470	0.94	0
1519	1.03	0
1568	1.00	0
1617	1.12	0
1666	1.08	0
1715	1.08	0
1764	1.13	0
1813	1.17	0
1862	1.18	0
1911	1.21	0
196	0.43	0
1960	1.25	0
245	0.46	0
294	0.46	0
343	0.48	0
392	0.51	0
441	0.54	0

TABLE D.10: Résultats du 31/07 : Grimm

Nombre d'instances	Temps de résolution (s)	Compilable ?
49	0.39	1
490	5.05	1

539	6.47	1
588	7.02	1
637	8.55	0
686	10.3	0
735	13.2	0
784	14.9	1
833	17.5	0
882	20.4	1
931	23.5	0
98	0.55	1
980	27.4	1
1029	32.1	0
1078	36.9	1
1127	42.5	0
1176	48.1	0
1225	54.1	0
1274	62.2	0
1323	68.4	0
1372	75.6	1
1421	84.9	1
147	0.85	1
1470	93.1	0
1519	106.	1
1568	116.	0
1617	129.	0
1666	138.	0
1715	155.	0
1764	162.	0
1813	177.	0
1862	194.	0
1911	206.	0
196	1.20	1
1960	231.	0
245	1.42	1
294	1.83	0
343	2.25	1
392	2.88	1
441	4.09	0

TABLE D.11: Résultats du 31/07 : Grimm avec optimisations

Nombre d'instances	Temps de résolution (s)	Compilable ?
49	0.40	1
490	4.46	1
539	5.76	1
588	6.96	1
637	8.49	1
686	10.1	1
735	12.1	0

784	14.7	1
833	17.3	1
882	20.2	1
931	23.3	1
98	0.55	1
980	26.4	1
1029	31.4	1
1078	36.1	1
1127	41.0	1
1176	46.9	0
1225	52.4	1
1274	59.8	1
1323	67.9	1
1372	74.0	1
1421	82.5	1
147	0.75	1
1470	89.7	1
1519	102.	1
1568	109.	1
1617	122.	0
1666	131.	1
1715	144.	1
1764	156.	1
1813	173.	1
1862	187.	1
1911	199.	1
196	1.30	1
1960	216.	1
245	1.56	1
294	1.82	0
343	2.54	1
392	3.29	1
441	4.18	1

Table D.12: Résultats du 31/07 : Grimm avec statistiques

D.5 Résultats de la cinquième expérimentation

Nombre d'instances	Temps de résolution (s)	Compilable ?
49	0.38	0
490	0.66	0
539	0.65	0
588	0.65	0
637	0.59	0
686	0.71	0
735	0.67	0
784	0.73	0
833	0.67	0
882	0.81	0
931	0.79	0
98	0.48	0
980	0.8	0
1029	0.82	0
1078	0.79	0
1127	0.91	0
1176	0.83	0
1225	1.01	0
1274	0.99	0
1323	1.00	0
1372	1.04	0
1421	1.01	0
147	0.48	0
1470	1.00	0
1519	1.15	0
1568	1.12	0
1617	1.19	0
1666	1.22	0
1715	1.25	0
1764	1.24	0
1813	1.34	0
1862	1.35	0
1911	1.48	0
196	0.46	0
1960	1.28	0
245	0.49	0
294	0.54	0
343	0.60	0
392	0.57	0
441	0.55	0

TABLE D.13: Seconds résultats du 31/07 : Grimm

Nombre d'instances	Temps de résolution (s)	Compilable ?
49	0.39	1
490	5.81	0

539	7.12	0
588	7.84	1
637	9.33	1
686	10.8	0
735	13.7	0
784	15.3	1
833	17.8	1
882	20.4	1
931	23.9	1
98	0.61	1
980	27.9	0
1029	32.6	0
1078	37.5	0
1127	45.9	0
1176	48.8	0
1225	55.0	1
1274	66.6	0
1323	69.3	1
1372	77.6	0
1421	85.8	1
147	0.92	1
1470	95.3	0
1519	106.	0
1568	115.	0
1617	126.	0
1666	142.	0
1715	158.	1
1764	173.	0
1813	188.	0
1862	204.	0
1911	229.	0
196	1.30	1
1960	240.	1
245	1.79	0
294	2.33	0
343	2.67	1
392	3.90	1
441	4.91	1

TABLE D.14: Seconds résultats du 31/07 : Grimm avec optimisations

Nombre d'instances	Temps de résolution (s)	Compilable ?
49	0.40	1
490	4.97	1
539	6.17	1
588	7.26	1
637	9.16	1
686	10.6	0

735	12.6	1
784	15.2	1
833	18.4	1
882	20.8	1
931	24.5	1
98	0.63	1
980	29.8	1
1029	32.1	1
1078	36.9	1
1127	44.0	1
1176	47.4	1
1225	53.2	1
1274	62.5	1
1323	69.0	1
1372	74.3	1
1421	83.3	1
147	0.96	1
1470	92.3	1
1519	104.	1
1568	110.	1
1617	123.	1
1666	136.	1
1715	144.	1
1764	162.	1
1813	180.	1
1862	187.	1
1911	206.	1
196	1.35	1
1960	225.	1
245	1.68	1
294	2.28	1
343	3.1	1
392	3.44	1
441	4.27	1

TABLE D.15: Seconds résultats du 31/07 : Grimm avec statistiques

D.6 Résultats de la sixième expérimentation

Nombre d'instances	Temps de résolution (s)	Compilable ?
49	0.34	0
490	0.55	0
539	0.55	0
588	0.58	0
637	0.61	0
686	0.59	0
735	0.63	0
784	0.65	0
833	0.69	0
882	0.67	0
931	0.69	0
98	0.37	0
980	0.73	0
147	0.40	0
196	0.42	0
245	0.46	0
294	0.47	0
343	0.50	0
392	0.53	0
441	0.54	0

Table D.16 – Résultats du 04/08 : Grimm

Nombre d'instances	Temps de résolution (s)	Compilable ?
49	0.39	1
490	4.68	1
539	5.94	1
588	7.05	1
637	8.66	0
686	10.2	1
735	12.2	0
784	15.5	1
833	17.5	1
882	20.4	0
931	23.8	1
98	0.56	1
980	27.6	0
147	0.75	1
196	1.22	1
245	1.47	1
294	1.97	0
343	2.27	1
392	3.01	1
441	3.63	1

TABLE D.17 – Résultats du 04/08 : Grimm avec optimisations

Nombre d'instances	Temps de résolution (s)	Compilable ?
49	0.40	1
490	5.14	1
539	5.82	1
588	6.97	1
637	8.67	1
686	10.3	1
735	12.0	1
784	15.1	1
833	17.3	1
882	19.4	1
931	23.4	1
98	0.57	1
980	26.8	1
147	0.80	1
196	1.11	1
245	1.46	1
294	1.99	1
343	2.31	1
392	3.35	1
441	3.61	1

TABLE D.18 – Résultats du 04/08 : Grimm avec statistiques

D.7 Résultats retenus

Nombre d'instances	Temps de résolution (s)	Compilable ?
49	0.34	0
490	0.57	0
539	0.57	0
588	0.55	0
637	0.58	0
686	0.64	0
735	0.64	0
784	0.69	0
833	0.66	0
882	0.67	0
931	0.71	0
98	0.38	0
980	0.68	0
1029	0.72	0
1078	0.72	0
1127	0.74	0
1176	0.74	0
1225	0.85	0
1274	0.93	0
1323	0.89	0
1372	0.94	0
1421	0.99	0
147	0.39	0
1470	0.94	0
1519	1.03	0
1568	1.00	0
1617	1.12	0
1666	1.08	0
1715	1.08	0
1764	1.13	0
1813	1.17	0
1862	1.18	0
1911	1.21	0
196	0.43	0
1960	1.25	0
245	0.46	0
294	0.46	0
343	0.48	0
392	0.51	0
441	0.54	0
49	0.38	0
490	0.66	0
539	0.65	0
588	0.65	0
637	0.59	0
686	0.71	0
735	0.67	0

784	0.73	0
833	0.67	0
882	0.81	0
931	0.79	0
98	0.48	0
980	0.8	0
1029	0.82	0
1078	0.79	0
1127	0.91	0
1176	0.83	0
1225	1.01	0
1274	0.99	0
1323	1.00	0
1372	1.04	0
1421	1.01	0
147	0.48	0
1470	1.00	0
1519	1.15	0
1568	1.12	0
1617	1.19	0
1666	1.22	0
1715	1.25	0
1764	1.24	0
1813	1.34	0
1862	1.35	0
1911	1.48	0
196	0.46	0
1960	1.28	0
245	0.49	0
294	0.54	0
343	0.60	0
392	0.57	0
441	0.55	0
49	0.34	0
490	0.55	0
539	0.55	0
588	0.58	0
637	0.61	0
686	0.59	0
735	0.63	0
784	0.65	0
833	0.69	0
882	0.67	0
931	0.69	0
98	0.37	0
980	0.73	0
147	0.40	0
196	0.42	0
245	0.46	0

294	0.47	0
343	0.50	0
392	0.53	0
441	0.54	0
	Total :	0

Table D.19: Résultats retenus : Grimm

Nombre d'instances	Temps de résolution (s)	Compilable ?
490	5.05	1
539	6.47	1
588	7.02	1
637	8.55	0
686	10.3	0
735	13.2	0
784	14.9	1
833	17.5	0
882	20.4	1
931	23.5	0
98	0.55	1
980	27.4	1
1029	32.1	0
1078	36.9	1
1127	42.5	0
1176	48.1	0
1225	54.1	0
1274	62.2	0
1323	68.4	0
1372	75.6	1
1421	84.9	1
147	0.85	1
1470	93.1	0
1519	106.	1
1568	116.	0
1617	129.	0
1666	138.	0
1715	155.	0
1764	162.	0
1813	177.	0
1862	194.	0
1911	206.	0
196	1.20	1
1960	231.	0
245	1.42	1
294	1.83	0
343	2.25	1
392	2.88	1
441	4.09	0
49	0.39	1

490	5.81	0
539	7.12	0
588	7.84	1
637	9.33	1
686	10.8	0
735	13.7	0
784	15.3	1
833	17.8	1
882	20.4	1
931	23.9	1
98	0.61	1
980	27.9	0
1029	32.6	0
1078	37.5	0
1127	45.9	0
1176	48.8	0
1225	55.0	1
1274	66.6	0
1323	69.3	1
1372	77.6	0
1421	85.8	1
147	0.92	1
1470	95.3	0
1519	106.	0
1568	115.	0
1617	126.	0
1666	142.	0
1715	158.	1
1764	173.	0
1813	188.	0
1862	204.	0
1911	229.	0
196	1.30	1
1960	240.	1
245	1.79	0
294	2.33	0
343	2.67	1
392	3.90	1
441	4.91	1
49	0.39	1
490	4.68	1
539	5.94	1
588	7.05	1
637	8.66	0
686	10.2	1
735	12.2	0
784	15.5	1
833	17.5	1
882	20.4	0

931	23.8	1
98	0.56	1
980	27.6	0
147	0.75	1
196	1.22	1
245	1.47	1
294	1.97	0
343	2.27	1
392	3.01	1
441	3.63	1
	Total :	50

Table D.20: Résultats retenus : Grimm avec optimisations

Nombre d'instances	Temps de résolution (s)	Compilable ?
49	0.40	1
490	4.46	1
539	5.76	1
588	6.96	1
637	8.49	1
686	10.1	1
735	12.1	0
784	14.7	1
833	17.3	1
882	20.2	1
931	23.3	1
98	0.55	1
980	26.4	1
1029	31.4	1
1078	36.1	1
1127	41.0	1
1176	46.9	0
1225	52.4	1
1274	59.8	1
1323	67.9	1
1372	74.0	1
1421	82.5	1
147	0.75	1
1470	89.7	1
1519	102.	1
1568	109.	1
1617	122.	0
1666	131.	1
1715	144.	1
1764	156.	1
1813	173.	1
1862	187.	1
1911	199.	1
196	1.30	1

1960	216.	1
245	1.56	1
294	1.82	0
343	2.54	1
392	3.29	1
441	4.18	1
49	0.40	1
490	4.97	1
539	6.17	1
588	7.26	1
637	9.16	1
686	10.6	0
735	12.6	1
784	15.2	1
833	18.4	1
882	20.8	1
931	24.5	1
98	0.63	1
980	29.8	1
1029	32.1	1
1078	36.9	1
1127	44.0	1
1176	47.4	1
1225	53.2	1
1274	62.5	1
1323	69.0	1
1372	74.3	1
1421	83.3	1
147	0.96	1
1470	92.3	1
1519	104.	1
1568	110.	1
1617	123.	1
1666	136.	1
1715	144.	1
1764	162.	1
1813	180.	1
1862	187.	1
1911	206.	1
196	1.35	1
1960	225.	1
245	1.68	1
294	2.28	1
343	3.1	1
392	3.44	1
441	4.27	1
49	0.40	1
490	5.14	1
539	5.82	1

588	6.97	1
637	8.67	1
686	10.3	1
735	12.0	1
784	15.1	1
833	17.3	1
882	19.4	1
931	23.4	1
98	0.57	1
980	26.8	1
147	0.80	1
196	1.11	1
245	1.46	1
294	1.99	1
343	2.31	1
392	3.35	1
441	3.61	1
	Total :	95

Table D.21: Résultats retenus : Grimm avec statistiques

Bibliographie

[Abreu and Carapuça, 1994] Abreu, F. B. and Carapuça, R. (1994). Object-oriented software engineering : Measuring and controlling the development process. In *Proceedings of the 4th international conference on software quality*, volume 186.

[Apache, 2009] Apache, B. (2009). Byte code engineering library, november 2009.

[Chidamber and Kemerer, 1994] Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6) :476–493.

[Ferdjoukh et al., 2015] Ferdjoukh, A., Baert, A.-E., Bourreau, E., Chateau, A., Coletta, R., and Nebut, C. (2015). Instantiation of Meta-models Constrained with OCL : a CSP Approach. In *MODELSWARD, International Conference on Model-Driven Engineering and Software Development*, pages 213–222.

[Forman et al., 2004] Forman, I. R., Forman, N., and Ibm, J. V. (2004). Java reflection in action. pages 143–177.

[Henderson-Sellers, 1995] Henderson-Sellers, B. (1995). *Object-oriented metrics : measures of complexity.* Prentice-Hall, Inc.

[Kamin et al., 2003] Kamin, S., Clausen, L., and Jarvis, A. (2003). Jumbo : Run-time code generation for java and its applications. In *Proceedings of the International Symposium on Code Generation and Optimization : Feedback-directed and Runtime Optimization*, CGO '03, pages 48–56, Washington, DC, USA. IEEE Computer Society.

[Merchez et al., 2001] Merchez, S., Lecoutre, C., and Boussemart, F. (2001). Abscon : A prototype to solve csps with abstraction. In *CP*, pages 730–744.

[Oracle, 2014] Oracle (2014). java.lang.reflect (Java Platform SE 7).

[Tempero et al., 2010] Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., and Noble, J. (2010). Qualitas corpus : A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345.