Génération automatique d'algorithmes de détection des défauts de conception

Naouel Moha*, Foutse Khomh*, Yann-Gaël Guéhéneuc*
Laurence Duchien**, Anne-Françoise Le Meur**

*Équipe PTIDEJ– GEODES
Département d'informatique et de recherche opérationnelle, Université de Montréal
CP 6128 succ. Centre Ville, Montréal, Québec, H3C 3J7, Canada
{mohanaou, foutsekh, guehene}@iro.umontreal.ca
** Équipe Adam – INRIA Futurs
Université des Sciences et Technologies de Lille – LIFL - UMR CNRS 8022
Cité Scientifique, 59655 Villeneuve d'Ascq Cedex, France
{Laurence.Duchien, Anne-Françoise.LeMeur}@lifl.fr

Résumé. Les défauts de conception sont des problèmes récurrents de conception qui diminuent la qualité des programmes. Plusieurs approches outillées de détection des défauts ont été proposées dans la littérature mais, à notre connaissance, elles utilisent toutes des algorithmes de détection ad-hoc, ce qui rend difficile leur généralisation à d'autres défauts. De plus, elles sont basées principalement sur des métriques, qui ne rendent pas compte de certaines caractéristiques importantes des systèmes analysés, telles que leur architecture. Dans cet article, nous développons notre approche basée sur un méta-modèle des défauts de conception en présentant une génération automatique des algorithmes de détection à partir de gabarits. Nous présentons aussi les performances de la génération et évaluons les algorithmes générés en terme de précision et de rappel. Nous fournissons ainsi des moyens concrets pour automatiser la génération des algorithmes de détection et donc détecter de nouveaux défauts tout en prenant en compte toutes les caractéristiques des systèmes.

1 Introduction

Avec l'omniprésence des systèmes logiciels dans nos sociétés, la qualité est devenue vitale aussi bien pour assurer le fonctionnement adéquat des systèmes que pour diminuer leurs coûts de développement et de maintenance. Celle-ci est évaluée et améliorée principalement pendant les revues techniques et formelles, dont l'objectif est de détecter au plus tôt les erreurs ou les défauts de conception, avant que ces erreurs et défauts ne puissent être transmis à la prochaine étape du développement ou de la maintenance ou, pire, au client (Travassos et al., 1999).

Les défauts de conception sont de mauvaises solutions à des problèmes récurrents de conception, dont l'origine sont de mauvaises pratiques de conception (Perry et Wolf, 1992). Ils sont à l'opposé des patrons de conception (Gamma et al., 1994) et aussi différents des

"défauts" introduits par Halstead et Fenton, lesquels sont des "déviations de spécifications ou d'exigences qui peuvent entraîner des défaillances dans les opérations" (Fenton et Neil, 1999; Halstead, 1977). Ils englobent des problèmes à différents niveaux de granularité : problèmes architecturaux tels que les anti-patrons (Brown et al., 1998) et problèmes de bas niveau tels que les mauvaises odeurs (Fowler, 1999), qui sont généralement des symptômes d'anti-patrons.

Un exemple de défaut de conception est l'anti-patron Spaghetti Code[1] (cf. Brown et al., 1998, page 119), qui caractérise l'utilisation de la programmation procédurale dans un système par objets. Le Spaghetti Code se révèle par des classes qui n'ont aucune structure, déclarant de longues méthodes sans paramètres utilisant des variables globales. Les noms des classes peuvent suggérer une programmation procédurale. Il n'exploite pas et empêche l'utilisation de mécanismes de la programmation par objets, tels que le polymorphisme et l'héritage.

Les défauts ont des conséquences négatives sur la qualité des systèmes orientés objet et rendent difficiles l'ajout, le débogage et l'évolution de systèmes. C'est pourquoi leur détection et leur correction tôt dans le processus de développement peuvent considérablement réduire les coûts de développement et de maintenance (Pressman, 2001). Mais leur détection nécessite d'importantes ressources en temps et en personnel et est sujette à beaucoup d'erreurs (Travassos et al., 1999) car les défauts impliquent généralement plusieurs classes et méthodes.

Plusieurs approches ont été proposées pour détecter les défauts de conception, par exemple (Marinescu, 2004; Munro, 2005; Alikacem et Sahraoui, 2006). Elles utilisent toutes des algorithmes **ad-hoc**, construits à partir d'un sous-ensemble particulier de défauts connus. Ces algorithmes sont **contraints par les services** des plates-formes de détection sous-jacentes (ou le *manque* de services). Ils sont implantés comme des **boîtes noires** parce que les règles de détection sont "codées en dur" et inexplicites et parce que le processus de génération des algorithmes est inexistant. De plus, ils **utilisent principalement des métriques** pour détecter les défauts, ignorant d'autres caractéristiques importantes des systèmes, telle leur architecture. Ainsi, ils **s'adaptent difficilement** à différents contextes.

Dans cet article, nous proposons une approche et un processus pour la génération automatique des algorithmes de détection basée sur des gabarits, à partir de règles spécifiées conformément à un langage de spécification des défauts de conception. Nous montrons ainsi qu'il est possible de dépasser les limites des approches existantes en proposant des algorithmes de détection basés sur **un langage** et **des services appropriés** construits à partir d'une étude exhaustive des défauts. Ce langage et ces services permettent la génération automatique des algorithmes et assurent la **transparence entre les spécifications des règles et les algorithmes**. Les algorithmes peuvent prendre en compte **toutes les caractéristiques des systèmes** lors de la détection et **s'adapter à différents contextes**. Nous validons la génération en terme de performance et les algorithmes générés en terme de précision et rappel, présentant ainsi la première étude de la littérature sur l'exactitude de la détection. Nous utilisons le logiciel libre XERCES v2.7.0 pour valider les algorithmes générés automatiquement.

Cet article est organisé comme suit : la section 2 présente succinctement le langage pour la détection des défauts de conception en utilisant l'exemple du Spaghetti Code. La section 3 détaille la génération des algorithmes de détection en utilisant le Spaghetti Code comme exemple illustratif. La section 4 décrit la validation de notre approche avec la spécification et la détection de quatre défauts de conception : le Blob, le Functional Decomposition, le Spaghetti Code, et le Swiss Army Knife, sur le système libre XERCES v2.7.0. La section 5 présente une étude de l'état de l'art. La section 6 conclut et présente les travaux futurs.

[1] Ce défaut, tout comme ceux présentés plus tard, se situe vraiment entre la conception et l'implantation.

2 Rappel sur le langage de spécification des défauts

Dans (Moha et al., 2006a), nous avons présenté un langage pour spécifier les défauts de conception en vue de leur détection, basé sur un méta-modèle (Thomas, 2002; Moha et al., 2006b). Nous nommons ce langage SADSL (pour *Software Architectural Defect Specification Language*). Nous décrivons succinctement ce langage dans un souci de compréhension et de complétude. Ce langage consiste à spécifier un ensemble de règles en des fiches de règles. Chaque règle décrit l'une des quatre propriétés des constituants d'un système. Les constituants d'un système sont ses classes, interfaces, méthodes, variables d'instances et de classes, relations, etc. Les propriétés de ces constituants peuvent être mesurables, lexicales, structurelles ou relationnelles. Les propriétés mesurables permettent d'exprimer des mesures des attributs internes des constituants d'un système. Les propriétés lexicales permettent d'identifier les noms des constituants. Les propriétés et les relations structurelles portent sur les structures des constituants et leurs relations, par exemple l'existence de paramètres pour une méthode ou la présence d'une relation d'agrégation entre deux classes (Guéhéneuc et Albin-Amiot, 2004). Le langage permet la combinaison de règles en utilisant des opérateurs ensemblistes.

```
1  RULE_CARD:SpaghettiCode {
2    RULE:SpaghettiCode
           {INTER LongMethod NoParameter NoInheritance NoPolymorphism ProceduralName UseGlobalVariable};
3    RULE:LongMethod       {METRIC LOC_METHOD VERY_HIGH 10.0};
4    RULE:NoParameter      {METRIC NMNOPARAM VERY_HIGH 5.0};
5    RULE:NoInheritance    {METRIC DIT 1 0.0};
6    RULE:NoPolymorphism   {STRUCT NO_POLYMORPHISM};
7    RULE:ProceduralName   {LEXIC CLASS_NAME (Make, Create, Exec...)};
8    RULE:UseGlobalVariable {STRUCT USE_GLOBAL_VARIABLE};
9  };
```

FIG. 1 – *Fiche de règle du Spaghetti Code.*

La figure 1 présente la fiche de règle du Spaghetti Code. Cette fiche caractérise les classes comme Spaghetti Code utilisant l'intersection de plusieurs règles (ligne 2). Une classe est un Spaghetti Code si elle déclare des méthodes avec un très grand nombre de lignes de code (propriété mesurable, ligne 3), sans paramètres (propriété mesurable, ligne 4) ; si elle n'utilise pas d'héritage (propriété mesurable, ligne 5), et de polymorphisme (propriété structurelle, ligne 6), et a un nom qui rappelle les noms de type procédural (propriété lexicale, ligne 7), et déclarant/utilisant des variables globales (propriété structurelle, ligne 8). Plus de détails sur la syntaxe et la grammaire des fiches de règles sont disponibles dans (Moha et al., 2006a,b). Les règles sont spécifiées par des experts en qualité, des concepteurs ou des développeurs. Elles peuvent être affinées par ceux-ci soit en ajoutant de nouvelles règles soit en les modifiant en prenant en compte les caractéristiques des systèmes analysés. Actuellement, nous offrons uniquement la possibilité de détecter des défauts liés à la structure des programmes mais des travaux en cours (Janice Ka-Yee Ng et Guéhéneuc, 2007), nous permettrons également de détecter des défauts liés au comportement.

Différences par rapport aux travaux précédents

Les aspects novateurs présentés dans cet article par rapport à nos travaux précédents (Moha et al., 2006a,b) sont double. Premièrement, cet article propose une méthode pour la génération automatique d'algorithmes de détection des défauts de conception en utilisant des gabarits. Cette première contribution se base sur un métamodèle et un langage de spécification des défauts tels que présentés dans la section 2. Deuxièmement, cet article présente la validation de cette méthode incluant un rapport sur à la fois la précision et le rappel des algorithmes de détection sur le logiciel libre XERCES.

3 Génération des algorithmes de détection

L'implantation du générateur d'algorithmes n'est pas en soi un réel défi car la génération de code est un problème bien maîtrisé. Par contre, notre réel apport se situe dans le passage automatique des spécifications des défauts de conception aux algorithmes de détection pour éviter la construction manuelle de ces algorithmes qui est coûteuse et peu réutilisable et pour assurer la traçabilité entre les spécifications et les occurrences des défauts détectés tout en permettant d'utiliser toutes les propriétés des constituants des systèmes.

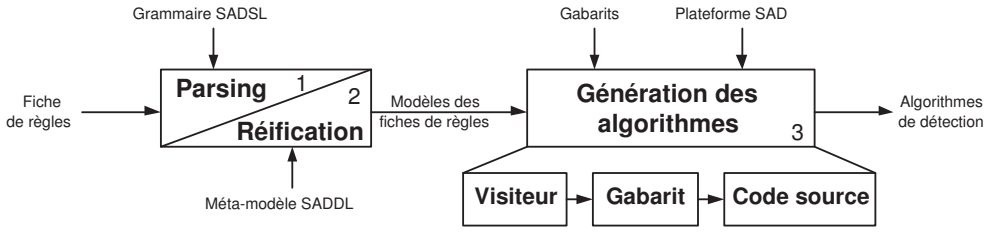

FIG. 2 – *Processus de génération.*

Le processus de génération que nous proposons se décompose en trois étapes automatiques similaires aux étapes traditionnelles de la compilation des langages, passant des fiches de règles à leur réification à la génération des algorithmes de détection, comme le montre la figure 2 :

1. **Analyse syntaxique.** La première étape consiste à analyser la syntaxe d'une fiche de règles. Un analyseur syntaxique construit avec JFLEX et JAVACUP[2] à partir de la grammaire SADSL des fiches et augmenté avec des actions sémantiques appropriées correspondant à des instructions de code pour construire les modèles des fiches de règles.
2. **Réification.** Les actions sémantiques produisent, lors de l'analyse d'une fiche, un modèle de la fiche, instance du méta-modèle SADDL (*Software Architectural Defects Definition Language*) (Moha et al., 2006b). Le méta-modèle SADDL est utilisé pour réifier les fiches. Il définit les constituants nécessaires pour représenter les fiches, règles, opérateurs ensemblistes et les propriétés.
3. **Génération des algorithmes.** Le modèle d'une fiche est enfin visité pour générer le code source associé à chaque constituant du modèle sur la base de gabarits pré-définis.

[2]Plus d'information sur JFlex et JavaCUP à `http://www2.cs.tum.edu/projects/cup/`.

Les gabarits et le code source généré se basent sur notre plate-forme pour la détection des défauts de conception SAD (*Software Architectural Defects*), qui fournit les services d'accès aux constituants d'un système et à leurs propriétés.

3.1 Caractéristiques principales de la génération

Notre solution au problème de la génération automatique des algorithmes de détection possèdes trois caractéristiques principales qui la distingue des travaux précédents.

- La génération des algorithmes de détection est implantée sous forme d'un visiteur sur les modèles des fiches de règles. Ce choix est similaire au choix habituel d'un compilateur de 'visiter' les noeuds de l'arbre de syntaxe abstraite pour produire du code. Ainsi, l'unique source de données pour la génération est la fiche de règles, assurant ainsi la traçabilité des algorithmes générés et des occurrences détectées avec la spécification du défaut.
- Afin de factoriser un maximum de code et de simplifier le visiteur, la génération utilise des gabarits de code pré-définis pour chaque type de propriétés. Ces gabarits contiennent des balises qui sont remplacées par le code spécifique à une fiche de règles lors de la génération.
- Le code source des gabarits utilise les services offerts par la plate-forme SAD qui permettent un accès facile aux constituants d'un système (dans lequel détecter les défauts) et à leurs propriétés.

3.2 Détails sur la génération

SAD. La génération utilise les services de la plate-forme SAD et est basée sur des patrons. Ces services implantent des opérations sur des relations, des opérateurs, des propriétés, et des valeurs ordinales comme trouvés dans les fiches. La plate-forme fournit aussi des services pour construire, accéder et analyser des systèmes logiciels. Ainsi, nous pouvons calculer des métriques, analyser des relations structurelles, appliquer des analyses lexicales et structurelles sur des classes et appliquer les règles. La plate-forme SAD définit aussi les services que les algorithmes de détection doivent fournir via les interfaces `IAntiPatternDetection` et `ICodeSmellDetection`.

Gabarits. Les gabarits sont des extraits de code source Java implantant une partie des interfaces `ICodeSmellDetection` et `IAntiPatternDetection`, avec des balises bien définis à remplacer par du code généré. Pendant la visite du modèle d'une fiche tel qu'indiqué dans la figure 2, nous remplaçons les balises des gabarits avec les données et les valeurs appropriées venant des fiches. Nous utilisons des gabarits car nos travaux précédents (Moha et al., 2006b,a) ont montré que les algorithmes de détection ont des structures récurrentes. Ainsi, nous regroupons naturellement toutes les structures communes des algorithmes de détection au sein de gabarits, par opposition aux approches existantes, où les algorithmes sont implantés manuellement et avec peu de réutilisation. Le code source généré pour une fiche est l'algorithme de détection du défaut de conception correspondant et ce code est directement exécutable sans aucune intervention manuelle.

Code généré. Les algorithmes de détection générés sont par construction déterministes. Nous n'avons pas besoin de réviser manuellement le code car le processus de génération assure

l'adéquation du code source par rapport aux fiches. Le code généré tend parfois lui-même vers un Spaghetti Code car il est généré automatiquement. Cependant, il n'est pas destiné à être lu par les développeurs. Il est prévu de l'améliorer et l'optimiser dans les travaux futurs. Les algorithmes de détection générés sont en Java. Par contre, il est possible d'appliquer ces algorithmes sur des programmes avec des langages autres que Java pour autant qu'il existe un analyseur syntaxique pour ces langages.

3.3 Exemple de génération

La figure 3 présente un exemple de génération d'une propriété mesurable et d'un opérateur ensembliste pour illustrer l'utilisation du visiteur, des gabarits et des services de la plate-forme SAD. Nous détaillons maintenant chaque caractéristique en nous appuyant sur les exemples donnés sur la figure. Nous ne présentons pas d'exemple de génération des propriétés lexicales, structurelles et relationnelles car elles sont similaires à celles illustrées.

Propriétés mesurables. Une propriété mesurable définie une valeur numérique ou ordinale pour une métrique spécifique. Les valeurs ordinales sont définies avec une échelle de Likert à 5 points : très élevé (very high), élevé (high), moyen (medium), faible (low), très faible (very low) (cf. Figure 1, lignes 3-5 et Figure 3(a)). Les valeurs numériques sont utilisées pour définir des valeurs relatives à toutes les classes du système sous analyse (cf. Figure 1, ligne 5). Les métriques peuvent être sommées ou soustraites. Le degré de logique floue, qui est la valeur numérique qui suit la valeur numérique ou ordinale de la métrique, correspond à la marge acceptable autour de ces dernières valeurs.

Nous utilisons principalement les métriques décrites par Chidamber et Kemerer (1994) : profondeur d'héritage `DIT`, lignes de code dans une classe `LOC_CLASS`, lignes de code dans une méthode `LOC_METHOD`, nombre d'attributs déclarés dans une classe `NAD`, nombre de méthodes déclarées dans une classe `NMD`, manque de cohésion entre méthodes `LCOM`, nombre de méthodes sans paramètres `NMNOPARAM`.

Les métriques peuvent être comparées soit à des valeurs numériques soit à des valeurs ordinales. Nous définissons les valeurs ordinales avec la technique de la boîte à moustaches pour lier les valeurs ordinales à des valeurs concrètes de métriques. Une boîte à moustaches (Chambers et al., 1983) partage un ensemble de valeurs numériques en quatre quarts ou quartiles. Le premier quartile (ou quartile inférieur) et le troisième quartile (ou quartile supérieur) de la distribution de valeurs représentent les limites de la boîte à moustaches. L'interquartile qui correspond à la longueur de la boîte est utilisé pour calculer les valeurs extrêmes ou queues. Les queues sont des limites théoriques où toutes les valeurs doivent tomber si la distribution était normale. Les valeurs des queues supérieure et inférieure sont calculées respectivement comme $Q3 + 1.5 \times IQ$ and $Q1 - 1.5 \times IQ$.

Nous associons des valeurs ordinales avec les quartiles comme suit : très faible (respectivement très élevé) correspond à des valeurs de métriques en dessous de la queue inférieure (respectivement au-dessus de la queue supérieure) ; faible (respectivement élevé) correspond à des valeurs entre la queue inférieure et le quartile inférieure (respectivement entre le quartile supérieur et la queue supérieure) ; moyen correspond aux valeurs entre les quartiles inférieur et supérieur.

La boîte à moustaches permet de comparer des valeurs de métriques entre elles, sans avoir à fixer des seuils artificiels. Cependant, la boîte à moustaches ne résout pas complètement le

problème lié aux valeurs seuils. En effet, si nous inférons que les larges classes sont des classes avec un nombre d'attributs et de méthodes plus élevé que 40, une classe avec 39 attributs et méthodes ne sera pas considérée comme large. Un degré de logique floue permet de pallier à ce problème en spécifiant une marge autour des valeurs seuils identifiées (Alikacem et Sahraoui, 2006) afin d'inclure des valeurs proches d'un seuil. Par exemple, si la distribution des tailles des classes est entre 1 et 50 et que nous fixons le degré de logique floue à 10%, nous obtenons une marge de $10\% \times 50 = 5$ et, ainsi, la classe avec 39 attributs et méthodes sera alors considérée comme large avec 10% comme degré de logique floue.

Le gabarit donné sur la figure 3(e) est une classe appelée `<CODESMELL>Detection` qui étend la classe `CodeSmellDetection` et implémente `ICodeSmellDetection`. Il déclare la méthode `performDetection()`, qui consiste à calculer la métrique spécifiée sur chaque classe du système. Toutes les valeurs des métriques sont comparées entre elles avec la boîte à moustaches (classe `BoxPlot`) pour identifier les valeurs ordinales, c'est-à-dire des valeurs extrêmes ou des valeurs normales. Ensuite, la boîte à moustaches retourne seulement les classes avec des valeurs de métriques qui vérifient les valeurs ordinales.

La figure 3(c) présente le processus de génération du code pour vérifier une propriété mesurable définie dans la fiche de règles du Spaghetti Code sur un ensemble de constituants. Lorsque la fiche est visitée sur le modèle de la fiche de règles, nous remplaçons le tag `<CODESMELL>` par le nom de la règle *`LongMethod`*, le tag `<METRIC>` par le nom de la métrique `LOC_METHOD`, le tag `<FUZZINESS>` par la valeur 10.0 correspondant au degré de logique floue, et le tag `<ORDINAL_VALUES>` par la méthode associée avec la valeur ordinale `VERY_HIGH`.

La figure 3(g) présente le code source généré correspondant au résultat de la génération à partir de la règle donnée sur la figure 3(a).

Opérateurs. Les règles peuvent être combinées en utilisant des opérateurs ensemblistes tels que l'intersection, l'union ou la différence (Figure 3(b)). La génération du code pour les opérateurs entre règles dans une fiche est légèrement différente de la génération pour les propriétés : le gabarit présenté dans la figure 3(f) contient également une classe appelée `<CODESMELL>Detection` qui étend la classe `CodeSmellDetection` et implémente `ICodeSmellDetection`. Mais, la méthode `performDetection()` consiste à combiner avec un opérateur ensembliste la liste des classes de chaque opérande de la règle donnée à la figure 3(b) et à retourner les classes qui satisfont cette combinaison. La figure 3(d) présente le processus relié à la génération de code pour les opérateurs ensemblistes dans la fiche du Spaghetti Code. Quand un opérateur est visité dans le modèle de la fiche, les balises associées aux opérandes de la fiche, `operand1 : LongMethod, operand2 : NoParameter` et la balise `<OPERATION>` sont remplacées par le type d'opérateur ensembliste spécifié dans la fiche, c'est-à-dire intersection. Les opérandes correspondent aux algorithmes de détection générés (cf. Figure 3(h)).

4 Validation

Nous montrons maintenant l'intérêt de la génération des algorithmes de détection en spécifiant et détectant quatre défauts de conception de haut niveau constitués d'une quinzaine de défauts de bas niveau ou mauvaises odeurs. Par manque de place, nous ne présentons pas les défauts et invitons le lecteur à consulter le livre de Brown et al. (1998) : les quatre défauts Spaghetti Code (cf. page 119), Blob (cf. page 73), Functional Decomposition (cf. page 97), et

```
4  RULE:LongMethod {METRIC LOC_METHOD
                    VERY_HIGH 10.0};
```

(a) Extrait du Spaghetti Code.

```
3  RULE:Inter1
       {INTER LongMethod NoParameter};
```

(b) Extrait du Spaghetti Code.

```
public void visit(IMetric aMetric) {
  replaceTAG("<CODESMELL>", aRule.getName());
  replaceTAG("<METRIC>", aMetric.getName());
  replaceTAG(<FUZZINESS>,
    aMetric.getFuzziness());
  replaceTAG(<ORDINAL_VALUE>,
    aMetric.getOrdinalValue());
}
private String getOrdinalValue(int value) {
  switch (value) {
    case VERY_HIGH :
      "getHighOutliers";
    case HIGH :
      "getHighValues";
    case MEDIUM :
      "getNormalValues";
    ...
  }
```

(c) Visiteur.

```
public void visit(IOperator anOperator) {
  replaceTAG("<OPERAND1>",
    anOperator.getOperand1());
  replaceTAG("<OPERAND2>",
    anOperator.getOperand2());
  switch (anOperator.getOperatorType()) {
    case OPERATOR_UNION :
      operator = "union";
    case OPERATOR_INTER :
      operator = "intersection";
    ...
  }
  replaceTAG("<OPERATION>", operator);
```

(d) Visiteur.

```
public class <CODESMELL>Detection
  extends CodeSmellDetection
  implements ICodeSmellDetection {
  public Set performDetection() {
    IClass c = iteratorOnClasses.next();
    LOCofSetOfClasses.add(
      Metrics.compute(<METRIC>, c));
    ...
    BoxPlot boxPlot = new BoxPlot(
      <METRIC>ofSetOfClasses, <FUZZINESS>);
    Map setOfOutliers =
      boxPlot.<ORDINAL_VALUE>();
    ...
    suspiciousCodeSmells.add( new CodeSmell(
      <CODESMELL>, setOfOutliers));
    ...
    return suspiciousCodeSmells;
  }
```

(e) Gabarit.

```
public class <CODESMELL>Detection
  extends CodeSmellDetection
  implements ICodeSmellDetection {
  public void performDetection() {
    ICodeSmellDetection cs<OPERAND1> =
      new <OPERAND1>Detection();
    op1.performDetection();
    Set set<OPERAND1> =
      cs<OPERAND1>.listOfCodeSmells();
    ICodeSmellDetection cs<OPERAND2> =
      new <OPERAND2>Detection();
    op2.performDetection();
    Set set<OPERAND2> =
      cs<OPERAND2>.listOfCodeSmells();
    Set setOperation = Operators.getInstance().
      <OPERATION>(set<OPERAND1>, set<OPERAND2>);
    this.setSetOfSmells(setOperation);
  }
```

(f) Gabarit.

```
public class LongMethodDetection
  extends CodeSmellDetection
  implements ICodeSmellDetection {
  public Set performDetection() {
    IClass c = iteratorOnClasses.next();
    LOCofSetOfClasses.add(
    Metrics.compute("LOC_METHOD", c));
    ...
    BoxPlot boxPlot = new BoxPlot(
      LOC_METHODofSetOfClasses, 10.0);
    Map setOfOutliers =
      boxPlot.getHighOutliers();
    ...
    suspiciousCodeSmells.add( new CodeSmell(
      LongMethod, setOfOutliers));
    ...
    return suspiciousCodeSmells;
  }
```

(g) Code généré.

```
public class Inter1
  extends CodeSmellDetection
  implements ICodeSmellDetection {
  public void performDetection() {
    ICodeSmellDetection csLongMethod =
      new LongMethodDetection();
    csLongMethod.performDetection();
    Set setLongMethod =
      csLongMethod.listOfCodeSmells();
    ICodeSmellDetection csNoParameter =
      new NoParameterDetection();
    csNoParameter.performDetection();
    Set setNoParameter =
      csNoParameter.listOfCodeSmells();
    Set setOperation = Operators.getInstance().
      intersection(setLongMethod, setNoParameter);
    this.setSetOfSmells(setOperation);
  }
```

(h) Code généré.

FIG. 3 – *Génération pour les propriétés mesurables (à gauche) et les opérateurs ensemblistes (à droite).*

Swiss Army Knife (cf. page 197) y sont décrits. Nous reportons également la précision et le rappel pour les algorithmes de détection des défauts de conception. Nous n'avons pas connaissance d'autres travaux reportant à la fois la précision et le rappel. Nous validons les résultats de la détection du défaut de conception Spaghetti Code dans XERCES v2.7.0, qui permet de construire des analyseurs syntaxiques XML en Java et contient 71,217 lignes de code, 513 classes et 162 interfaces. Dans cette validation, nous cherchons à obtenir un rappel de 100% car nous devons identifier tous les défauts de conception pour améliorer la qualité et faciliter les revues techniques.

4.1 Application des algorithmes de détection générés

Nous appliquons les algorithmes de détection générés sur un modèle du système XERCES et obtenons toutes les classes suspectes qui ont potentiellement des défauts de conception. L'appel aux algorithmes de détection générés est automatique en utilisant les services fournis par la plate-forme SAD.

En suivant l'exemple du défaut de conception Spaghetti Code et de XERCES, nous obtenons d'abord un modèle du système XERCES en utilisant les constituants du méta-modèle PADL, présenté dans (Albin-Amiot et al., 2002). Ce modèle est obtenu par rétro-conception du code source de XERCES. Ensuite, nous appliquons l'algorithme de détection généré pour le Spaghetti Code sur le modèle du système à détecter pour obtenir la liste des classes suspectes.

Nous validons les résultats des algorithmes de détection en analysant les classes suspectes dans le contexte du modèle complet du système et de son environnement. Cette validation consiste à identifier les classes suspectes considérées comme de vraies positives par rapport au contexte du système et les fausses négatives, c'est-à-dire les classes ayant des défauts mais qui ne sont pas reportées par les algorithmes.

Ainsi, nous projetons la validation dans le domaine de l'extraction d'information et utilisons les mesures de précision et de rappel, où la précision évalue le nombre de réels défauts identifiés et le rappel évalue le nombre de réels défauts manqués par les algorithmes, avec les définitions suivantes (Frakes et Baeza-Yates, 1992) :

$$\text{précision} = \frac{\left|\{\text{défauts existants}\} \cap \{\text{défauts détectés}\}\right|}{\left|\{\text{défauts détectés}\}\right|}$$

$$\text{rappel} = \frac{\left|\{\text{défauts existants}\} \cap \{\text{défauts détectés}\}\right|}{\left|\{\text{défauts existants}\}\right|}$$

La validation est réalisée en utilisant des résultats indépendants obtenus manuellement car seuls des développeurs peuvent évaluer si une classe suspecte est *réellement* un défaut ou une fausse positive selon la fiche de règle, le contexte et les caractéristiques du système. Nous avons demandé à trois étudiants de maîtrise (équivalent à un Master 2 Recherche) et deux développeurs indépendants d'analyser manuellement XERCES en utilisant seulement les livres de Brown et Fowler et leur propre compréhension pour identifier les défauts de conception et de calculer la précision et le rappel sur les classes suspectes. Chaque fois qu'il y a eu un doute sur une classe candidate, ils ont pris les livres comme référence pour décider par consensus si oui ou non la classe était réellement un défaut de conception. Ils ont réalisé une étude minutieuse de XERCES et produit un fichier XML contenant les classes suspectes pour les quatre défauts de conception. Cette tâche étant fastidieuse, certains défauts de conception ont été manqués par

erreur, c'est pourquoi nous avons demandé à d'autres développeurs de réaliser la même tâche à nouveau sur XERCES pour confirmer les résultats et sur d'autres systèmes pour augmenter notre base de données.

4.2 Résultats

Nous reportons les temps de génération et de détection des algorithmes, le nombre de classes suspectes, et les précisions et rappels. Nous avons réalisé tous les calculs sur un Intel Dual Core à 1.67GHz avec 1Gb de RAM. Les temps de calcul n'incluent pas la construction des modèles du système, qui est de 31 secondes, mais incluent les accès pour calculer les métriques et pour vérifier les relations structurelles et les propriétés lexicales et structurelles.

Défauts	Nombre de classes	Nombres de vraies positives		Nombres de défauts détectés		Précision	Rappel	Temps de détection
Blob	513	39	(7,6%)	44	(8,6%)	88,6%	100,00%	2,45s
Functional Decomp.		15	(2,9%)	29	(5,6%)	51.7%	100,00%	0,91s
Spaghetti Code		46	(9,0%)	76	(14,8%)	60,5%	100,00%	0,23s
Swiss Army Knife		23	(4,5%)	56	(10,9%)	41,1%	100,00%	0,08s

TAB. 1 – *Précision et rappel dans* XERCES *v2.7.0. (En parenthèses, le pourcentage des classes affectées par un défaut de conception.)*

La table 1 montre la précision et le rappel de la détection de quatre défauts de conception dans XERCES v2.7.0. Les rappels de nos algorithmes de détection sont 100% pour chaque défaut de conception. Les précisions se situent entre 41.07% et plus de 80%, représentant entre 5.65% et 14.81% du nombre total de classes, ce qui est raisonnable pour un développeur d'analyser à la main, par rapport à analyser le système en entier—513 classes—manuellement.

Pour le Spaghetti Code, nous avons trouvé 76 classes suspectes. Sur les 76 classes suspectes, 46 sont en fait des Spaghetti Code précédemment identifiés dans XERCES manuellement par des développeurs indépendants des auteurs, ce qui mène à une précision de 60,53% et un rappel de 100,00% (cf. la troisième ligne dans la table 1). Le fichier résultat contient toutes les classes suspectes, incluant la classe `org.apache.xerces.impl.xpath.regex.RegularExpression` déclarant 57 méthodes. Parmi ces 57 méthodes, la méthode `matchCharArray(Context, Op, int, int, int)` est typique d'un Spaghetti Code, en particulier car cette méthode n'utilise pas l'héritage et le polymorphisme, elle utilise 18 variables globales, et compte 1, 246 LOC, alors que la longueur maximale d'une méthode en utilisant la boîte à moustaches est 254,5 LOC.

4.3 Discussion

La validation montre que notre approche est appropriée pour spécifier des défauts de conception et générer des algorithmes de détection. En effet, le langage permet de décrire plusieurs défauts de conception. Nous avons décrit quatre défauts de conception différents composés de 15 mauvaises odeurs. **Les algorithmes de détection générés ont un rappel de 100% et ont une précision moyenne supérieure à 40%.** Les algorithmes de détection reportent moins de $2/3$ des fausses positives par rapport au nombre de vraies positives et rapportent toutes les classes avec des défauts. La table 1 rapporte les valeurs de la précision et du

rappel pour XERCES v2.7.0. **La complexité des algorithmes générés est raisonnable.** Les algorithmes générés ont des temps d'exécution inférieurs à quelques minutes et ont des temps de génération de l'ordre de quelques secondes. Ce temps dépend du nombre de classes dans le système n, du nombre de propriétés à vérifier et des opérateurs pour combiner ces propriétés : $(c + op) \times \mathcal{O}(n)$, où c est le nombre de propriétés et op le nombre d'opérateurs.

Les résultats dépendent des spécifications des défauts de conception. Les spécifications ne doivent être ni trop vagues pour ne pas détecter trop de classes suspectes ni trop contraignantes pour ne pas manquer des défauts de conception. Avec notre approche, les développeurs peuvent raffiner les spécifications systématiquement, selon les classes suspectes détectées et leur connaissance de la conception et l'implantation du système via des raffinements itératifs si nécessaires des fiches.

5 État de l'art

Plusieurs approches et techniques de détection des défauts ont été proposées dans la littérature. Nous présentons ici seulement les approches et techniques directement liées à la détection des défauts de conception.

Plusieurs livres fournissent des vues assez larges sur les pièges (Webster, 1995), les heuristiques (Riel, 1996), les mauvaises odeurs (Fowler, 1999), et les anti-patrons (Brown et al., 1998), ces livres sont destinés à une large audience pour un usage éducatif. Cependant, ils décrivent *textuellement* les défauts de conception, il est donc difficile de construire des algorithmes de détection à partir de descriptions textuelles car ils manquent de précision et sont sujets à mauvaise interprétation. Travassos et al. (1999) ont introduit un processus basé sur des inspections manuelles pour identifier des défauts de conception. Aucune tentative n'a été réalisée pour automatiser ce processus donc il ne s'adapte donc pas facilement à des larges systèmes et il couvre seulement la détection manuelle des défauts et non pas leurs spécifications.

Plusieurs approches semi-automatique pour la détection des défauts existent. Parmi celles-ci, Marinescu (2004) a présenté une approche basée sur les métriques pour détecter des mauvaises odeurs avec des *stratégies de détection* et implantée dans l'outil IPLASMA. Cette approche introduit des stratégies basées sur les métriques capturant des déviations par rapport aux principes de bonne conception. Les stratégies de détection sont une avancée vers des spécifications précises des mauvaises odeurs. Dans notre approche, nous développons cette idée sous la forme de fiches de règles et proposons des algorithmes de génération de code. L'outil IPLASMA a été évalué en termes de précision, le rappel n'a pas été calculé. Munro (2005) a également remarqué les limitations des descriptions textuelles et proposé un patron pour décrire les mauvaises odeurs de manière plus systématique. C'est une avancée vers des spécifications plus précises des mauvaises odeurs mais les mauvaises odeurs restent néanmoins des descriptions textuelles sujettes à mauvaise interprétation. Munro a aussi proposé des heuristiques basées sur les métriques pour détecter des mauvaises odeurs similaires aux stratégies de détection de Marinescu. Alikacem et Sahraoui ont proposé un langage de description de règles de qualité pour détecter des problèmes de bas niveau dans les systèmes orientés objet (Alikacem et Sahraoui, 2006) mais ils ne proposent pas de processus de génération.

De nombreux outils tels que PMD (2002), CHECKSTYLE (2004), et FXCOP (2006) détectent des problèmes liés aux standards de codage, à des patrons de bugs ou à du code inutilisé. Ainsi, ils se focalisent sur des problèmes d'implémentation mais n'adressent pas les défauts de conception de plus haut niveau tels que les anti-patrons.

6 Conclusion et travaux futurs

Dans cet article, nous avons introduit une approche pour générer des algorithmes de détection des défauts de conception et nous l'illustrons en utilisant le défaut Spaghetti Code.

Nous avons spécifié plusieurs défauts de conception, généré automatiquement les algorithmes de détection en utilisant des patrons, et validé les algorithmes de détection générés en termes de précision et de rappel, pour la première fois, sur XERCES v2.7.0, un système logiciel libre. Nous avons montré que les algorithmes de détection sont raisonnablement efficaces et précis et ont un bon rappel.

Dans cet article, nous avons expliqué un processus pour générer des algorithmes de détection des défauts de conception. Il est difficile de faire une comparaison avec les travaux précédents car ceux-ci n'expliquent pas concrètement leurs spécifications et leurs techniques pour la détection des défauts. Avec notre approche et sa validation en termes de précision et de rappel, nous définissons un point de repère pour des futures comparaisons quantitatives.

Dans les travaux futurs, en plus d'utiliser le dictionnaire WORDNET, améliorer le code généré, intégrer les plate-formes existantes, et calculer la précision et le rappel sur plus de systèmes, nous comptons offrir plus de flexibilité dans la génération de code en utilisant les technologies XML, en particulier Clearwater (Swint et al., 2005) qui permet la construction de générateurs de code pour mapper des langages liés au domaine à des plate-formes d'exécution multiples. Nous validerons également notre approche sur plus de défauts et de systèmes.

Remerciements

Nous sommes reconnaissants envers Giuliano Antoniol, Kim Mens, Dave Thomas, et Stéphane Vaucher pour leurs commentaires sur des précédentes versions de cet article. Nous sommes également redevables aux étudiants de master et aux développeurs qui ont réalisés l'analyse manuelle de XERCES et à Duc-Loc Huynh et Pierre Leduc pour leur aide dans la construction de SAD. Les auteurs sont partiellement soutenus par une subvention du CRSNG.

Références

Albin-Amiot, H., P. Cointe, et Y.-G. Guéhéneuc (2002). Un méta-modèle pour coupler application et détection des design patterns. In M. Dao et M. Huchard (Eds.), *Actes du* 8^e *colloque Langages et Modèles à Objets*, Volume 8, numéro 1-2/2002 of *RSTI – L'objet*, pp. 41–58. Hermès Science Publications.

Alikacem, E. et H. Sahraoui (2006). Détection d'anomalies utilisant un langage de description de règle de qualité. In R. Rousseau, C. Urtado, et S. Vauttier (Eds.), *actes du* 12^e *colloque Langages, Modèles, Objets*, pp. 185–200. Hermès Science Publications.

Brown, W. J., R. C. Malveau, W. H. Brown, H. W. M. III, et T. J. Mowbray (1998). *Anti Patterns : Refactoring Software, Architectures, and Projects in Crisis* (1^{st} ed.). John Wiley and Sons.

Chambers, J. M., W. S. Clevelmd, B. Kleiner, et P. A. Tukey (1983). *Graphical methods for data analysis*. Wadsworth International.

CheckStyle (2004). http ://checkstyle.sourceforge.net.

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

Fenton, N. E. et M. Neil (1999). A critique of software defect prediction models. *Software Engineering 25*(5), 675–689.

Fowler, M. (1999). *Refactoring – Improving the Design of Existing Code* (1^{st} ed.). Addison-Wesley.

Frakes, W. B. et R. Baeza-Yates (1992). *Information Retrieval : Data Structures and Algorithms*. Prentice-Hall.

FXCop (2006). http ://www.gotdotnet.com/team/fxcop/.

Gamma, E., R. Helm, R. Johnson, et J. Vlissides (1994). *Design Patterns – Elements of Reusable Object-Oriented Software* (1^{st} ed.). Addison-Wesley.

Guéhéneuc, Y.-G. et H. Albin-Amiot (2004). Recovering binary class relationships : Putting icing on the UML cake. In D. C. Schmidt (Ed.), *Proceedings of the* 19^{th} *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 301–314. ACM Press.

Halstead, M. H. (1977). *Elements of Software Science (Operating and programming systems series)*. New York, NY, USA : Elsevier Science Inc.

Marinescu, R. (2004). Detection strategies : Metrics-based rules for detecting design flaws. In *Proceedings of the* 20^{th} *International Conference on Software Maintenance*, pp. 350–359. IEEE Computer Society Press.

Moha, N., Y.-G. Guéhéneuc, et P. Leduc (2006a). Automatic generation of detection algorithms for design defects. In S. Uchitel et S. Easterbrook (Eds.), *Proceedings of the* 21^{st} *Conference on Automated Software Engineering*. IEEE Computer Society Press. Short paper.

Moha, N., D.-L. Huynh, et Y.-G. Guéhéneuc (2006b). Une taxonomie et un métamodèle pour la détection des défauts de conception. In R. Rousseau (Ed.), *actes du* 12^{e} *colloque Langages et Modèles à Objets*, pp. 201–216. Hermès Science Publications.

Munro, M. J. (2005). Product metrics for automatic identification of “bad smell" design problems in java source-code. In F. Lanubile et C. Seaman (Eds.), *Proceedings of the* 11^{th} *International Software Metrics Symposium*. IEEE Computer Society Press.

Perry, D. E. et A. L. Wolf (1992). Foundations for the study of software architecture. *Software Engineering Notes 17*(4), 40–52.

PMD (2002). http ://pmd.sourceforge.net/.

Pressman, R. S. (2001). *Software Engineering – A Practitioner's Approach* (5^{th} ed.). McGraw-Hill Higher Education.

Riel, A. J. (1996). *Object-Oriented Design Heuristics*. Addison-Wesley.

Janice Ka-Yee Ng et Y.-G. Guéhéneuc (2007). Identification of behavioral and creational design patterns through dynamic analysis. In A. Zaidman, A. Hamou-Lhadj, et O. Greevy (Eds.), *Proceedings of the* 3^{rd} *International Workshop on Program Comprehension through Dynamic Analysis*, pp. 34–42. Delft University of Technology. TUD-SERG-2007-022.

Swint, G. S., C. Pu, G. Jung, W. Yan, Y. Koh, Q. Wu, C. Consel, A. Sahai, et K. Moriyama (2005). Clearwater : extensible, flexible, modular code generation. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pp. 144–153. ACM.

Thomas, D. (2002). Reflective software engineering – From MOPS to AOSD. *Journal of Object Technology 1*(4), 17–26.

Travassos, G., F. Shull, M. Fredericks, et V. R. Basili (1999). Detecting defects in object-oriented designs : using reading techniques to increase software quality. In *Proceedings of the* 14^{th} *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 47–56. ACM Press.

Webster, B. F. (1995). *Pitfalls of Object Oriented Development* (1^{st} ed.). M & T Books.

Summary

Design defects are recurring design problems that decrease the quality of programs and thus hinder their development and maintenance. Consequently, several design defect detection approaches and tools have been proposed in the literature. However, to the best of our knowledge, they all use ad-hoc detection algorithms; this makes difficult their generalisation to other defects, and they are based mainly on metrics, which do not reflect certain important analyzed characteristics, such their architecture. In this paper, we develop our approach based on a metamodel of design defects by presenting an automatic generation of detection algorithms from templates. We present also the generation performances and evaluate the generated algorithms in terms of precision and recall. We thus provide concrete means to automate the generation of detection algorithms and hence to detect new defects by taking into account the characteristics of systems.