Refactoring des applications à objets pour un meilleur découplage et une non-anticipation des instanciations

Soumia Zellagui and Joffray Braga

LIRMM,CNRS, Université de Montpellier, France
zellagui@lirmm.fr
joffraybraga@gmail.com

Résumé

La modularité des applications à objets est une préoccupation majeure depuis les premières années des langages de programmation. Cet intérêt résulte de la volonté de faire face aux difficultés de compréhension, maintenance et réutilisation causées par les nombreuses dépendances entre les éléments de ces applications. Par ailleurs, les applications à base de composants logiciels sont caractérisées par la forte cohésion et le couplage faible entre leurs différentes entités permettant d'améliorer la réutilisation et la compréhension du code. Par conséquent, plusieurs travaux ont été proposés pour la migration des applications à objets vers des applications à base de composants dans lesquelles chaque composant est constitué d'un ensemble de classes. Contrairement à ces travaux, notre objectif consiste à améliorer le niveau de modularité d'une manière à ce que chaque classe représente un descripteur de composant. Ce papier propose une approche pour l'amélioration de la modularité des applications à objets. Cette approche identifie un ensemble de "Défauts" dits "de modularité" et propose des algorithmes pour leur détection dans le code. Ces algorithmes permettront par la suite d'éliminer ces "Défauts" en appliquant de façon automatique des opérations de refactoring. Cette étape de refactoring est brièvement mentionnée en proposant les premières idées que nous envisageons d'appliquer. Une implémentation des algorithmes de détection a été validée sur cinq programmes libres.

1 Introduction

En génie logiciel, la maintenance du logiciel désigne [1] les modifications apportées à un produit logiciel après sa mise en oeuvre pour corriger ses fautes, améliorer son efficacité ou toute autre caractéristique extra-fonctionnelle, ou encore adapter celui-ci à un environnement modifié. Elle exige beaucoup d'effort pour les développeurs. Cet effort est principalement déterminé par la difficulté de compréhension du code source [2] ; cette difficulté étant liée aux dépendances nombreuses et complexes entre les différents éléments de ce code. Pour minimiser cet effort et améliorer la réutilisation, ces logiciels doivent être les plus modulaires possible.

La modularité est un principe important du génie logiciel. Elle est considérée comme un attribut de qualité interne qui a une influence sur les attributs de qualité externes [11] tels que la maintenabilité et la réutilisabilité. C'est une propriété d'un système décomposé en plusieurs entités indépendantes appelées modules. Elle permet le développement collaboratif de différentes parties (modules) d'un même système par différentes personnes, de substituer ou réparer les parties défectueuses d'un système sans affecter les autres parties et de réutiliser les parties existantes dans différents contextes. Les modules sont reliés les uns aux autres. la mesure de la relation entre modules est connue sous le nom de couplage. L'un des objectifs de la modularité exige d'avoir des modules avec un faible couplage et une forte cohésion [23]. La cohésion signifie que chaque module se focalise sur une tâche unique. Parmi les entités de première classe en programmation, qui reflètent la notion de module, il y a les composants, les

aspects ou encore les classes dans les applications à objets, qui respectent un certain nombre de bonnes pratiques de développement.

La restructuration des applications à objets en applications à base de composants leur permet de bénéficier des avantages de la programmation par composants, en particulier, le découplage et la non-anticipation[1] [10], et ainsi améliorer leur modularité. Les travaux proposés [4, 3] pour cette restructuration suggèrent que l'unité de modularité, donc de réutilisation, est un groupe de classes. Afin d'améliorer le niveau de réutilisation, nous défendons l'idée de refactoriser les classes individuellement pour les rendre plus modulaires.

Nos travaux de recherche s'articulent autour de la restructuration des applications à objets de façon à ce que chaque classe pourra être considérée comme un descripteur de composant. Le processus général permettant d'aboutir à ce but est composé de deux étapes. La première étape, sur laquelle nous nous concentrons dans ce papier, consiste en la détection de "Défauts" de modularité, des parties du code source qui violent les deux principes de découplage et de non-anticipation. Nous avons identifié deux types de "Défauts" : l'inexistence ou l'incomplétude des interfaces fournies (I2PI[2])/requises(I2RI[3]) et l'existence des instanciations anticipées(EAI[4]). La deuxième étape permet la suppression de ces "Défauts" en appliquant de façon automatique une composition d'opérations de refactoring de code.

Ce papier présente en premier un état de l'art sur les approches proposées pour l'amélioration de la modularité (section 2). Il détaille ensuite les étapes du processus proposé (section 3), l'étape de détection en particulier. La section 4 présente un exemple illustrant notre processus. La section 5 décrit l'implémentation et l'expérimentation de la phase de détection sur un ensemble de cinq programmes Java libres. Enfin, la section 6 conclut le papier et discute les travaux futurs que nous envisageons de mener.

2 Etat de l'art

L'amélioration de la modularité des systèmes existants est une préoccupation majeure depuis des années. La modularité peut prendre plusieurs formes en termes d'entités de première classe en programmation: modules/composants et classes bien structurées. Nous avons organisé l'état de l'art autour de ces entités. La suite de cette section présente, dans un premier temps, les travaux autour du refactoring d'applications à objets visant à les transformer en applications à modules ou à composants. Ensuite, expose les travaux sur le refactoring d'applications à objets visant à améliorer leur structure en termes de couplage.

2.1 Refactoring de classes en composants

Les composants sont des entités logicielles réutilisables dotées d'un ensemble d'interfaces permettant de les utiliser : appeler leurs fonctions et les assembler avec d'autres composants. Une interface déclare des services fournis ou réquis par un composant. Deux composants peuvent être connectés si l'un fournit un service requis par l'autre. Cependant, La programmation par objet est de nos jours probablement le paradigme de programmation le plus populaire et est la principale méthode pour la conception et l'implémentation des systèmes logiciels. Pour améliorer la modularité, et ainsi la maintenabilité et la réutilisabilité, des systèmes existants,

[1] Le programmeur d'un ne doit pas intégrer un code spécifique dans le code d'un composant afin de permettre son assemblage

[2] Inexistence or Incompleteness of Provided Interfaces

[3] Inexistence or Incompleteness of Required Interfaces

[4] Existence of Anticipated Instanciations

un certain nombre de travaux se sont intéressés à la restructuration de ces systèmes en systèmes à base de composants.

L'identification de composants à partir du code source d'une application à objets est un problème traité dans plusieurs travaux[21, 8, 3, 4]. Une distinction s'est dessinée entre deux techniques réalisant ce but : l'extraction de composants réutilisables (sans connexions avec d'autres) et la transformation vers des systèmes à composants (interconnectés).

2.1.1 Extraction de composants réutilisables

Washizaki et al [21] proposent une technique pour l'extraction automatique de parties réutilisables depuis les programmes Java. Cette tâche d'extraction commence par la sélection d'un critère d'extraction qui représente une fonctionnalité devant être réutilisée. Les parties de l'application, qui sont identifiées, sont transformées en composants JavaBeans par la suite.

Constantinou et al [8] proposent une autre approche qui aide les développeurs à démêler les structures complexes d'applications à objets. Le processus consiste en l'identification et la transformation des classes responsables de la complexité structurelle. Cette approche diffère de la précédente dans le fait qu'elle se focalise sur la réduction de la taille des composants condidats.

Dans ces travaux, les composants identifiés peuvent partager un certain nombre (qui peut être parfois assez important) de classes. Si l'on construit une nouvelle application avec plusieurs composants ainsi obtenus, nous nous retrouverons alors avec des classes dupliquées. Nous défendons dans notre travail, l'idée de traiter la modularité à une granularité plus fine. Ainsi nous rendons les classes de vrais descripteurs de composants, modulaires et réutilisables.

2.1.2 Migration vers des applications à composants

De manière générale, ce processus de migration passe par deux étapes : i) la description de l'architecture à composants, qui consiste en un ensemble de clusters (groupes de classes) et des relations entre ces clusters ; et ii) la transformation du code à objets en code à composants. La première étape a été largement étudiée dans la littérature [9, 7, 14].

Allier et al [3] proposent une méthode de migration qui passe par les deux étapes. Ces auteurs suggérent une approche pour automatiser le processus de transformation d'une application à objets en une application à composants et illustrent cette méthode sur des applications réelles implémentées en Java qui sont transformées en applications OSGi. Le processus général est composé de trois étapes : identification des composants de base, identification des interfaces (fournies et requises) et la description de la manière de communication entre les interfaces d'un composant et ses classes. Cette méthode a l'avantage de rendre les interfaces des composants opérationnelles par l'utilisation des deux patrons de conception Adapter et Façade. Par contre, elle introduit une duplication du code, car pour chaque connexion (interface fournie-interface requise) entre deux composants, deux classes (Façade et Adapter) et une interface sont ajoutées. Donc pour qu'un composant puisse utiliser un service (invoquer une méthode M) d'un autre composant, il doit invoquer la méthode M de la classe Façade qui invoque la méthode M de la classe Adapter et à la fin cette méthode va invoquer la méthode M de la classe qui fournit le service. Toutes ces transitions ont forcément un impact sur le temps d'exécution.

Dans un autre travail, Alshara et al [4] proposent une autre approche pour transformer automatiquement des applications à objets implémentées en Java en applications à composants OSGi. Cette approche prend en entrée une application à objets et la description de l'architecture à composants correspondante (obtenue par analyse automatique du code source, en utilisant

une technique proposée par ces mêmes auteurs dans des travaux antérieurs). Ensuite, la transformation du code consiste à remplacer les dépendances entre classes qui appartiennent à des clusters (composants) différents par des interactions via des interfaces. Pour cela le problème de violation explicite et implicite d'encapsulation est résolu. La violation explicite apparait lorsqu'une classe dans un composant crée une instance d'une autre classe dans un autre composant (problème d'instanciation). La violation implicite est liée aux dépendances implicites entre composants telles que l'héritage. Le problème se pose lorsque la classe mère et sa classe fille ne sont pas dans le même composant (cluster de classes).

Dans ces deux travaux, nous remarquons que l'unité de modularité, et donc de réutilisation, est un groupe de classes (un cluster). Si un utilisateur veut développer une nouvelle application en utilisant une classe indépendante ou un sous-ensemble de classes dans un cluster, il est amené à réutiliser le composant entier. Afin d'optimiser le niveau de réutilisation, nous défendons l'idée de refactoriser les classes individuellement pour les rendre plus modulaires, ressemblant ainsi à des descripteurs de composants.

La programmation par aspects est un paradigme ayant pour but d'améliorer la modularité du logiciel en localisant les préoccupations entrelacées dans des modules nommés aspects. La motivation qui pousse les développeurs à utiliser ce paradigme de programmation est le fait que les fonctionnalités transversales sont réparties dans le code et emmêlées avec les fonctionnalités de base ce qui a un impact négatif sur les activités d'évolution et de maintenance. La séparation entre les préoccupations va permettre aux deux types de code (métiers et aspects) à être facilement compréhensibles, testables et maintenables.

Afin de migrer les applications existantes vers des applications à aspects, plusieurs méthodes ont été proposées. Celles-ci consistent en deux phases : la fouille d'aspects (aspect mining) qui consiste en l'identification du code qui représente des préoccupations existantes transverses, et la transformation de ce code en aspects (refactoring). La première étape a été largement étudiée dans la littérature [6, 15, 19, 20].

Binkly et al [5] proposent une approche pour le refactoring d'une application Java en une application AspectJ. Ils supposent que l'étape d'identification est déja faite (le code est annoté). Les parties du code censées subir un refactoring sont les appels des méthodes pour cela ils proposent six types de refactoring selon la position de l'appel (début/fin d'une méthode, avant/aprés un appel de méthode, par exemple). Le processus (itératif) se divise en quatre étapes. La première étape est la découverte qui consiste à analyser le code marqué et le comparer avec les six cas possibles de refactoring. la sortie de cette découverte est la liste des types de refactoring applicables, rangée par ordre décroissant (le meilleur en haut). Cet ordre est défini selon une échelle de priorités entre les six types de refactoring. La deuxième étape est la sélection du type de refactoring à appliquer par l'utilisateur qui a le choix de suivre l'ordre de l'étape précédente (choisir le meilleur) ou l'annuler. Si l'utilisateur trouve que la liste des types de refactoring ne lui convient pas et qu'il a besoin d'un type précis, il peut faire des transformations sur le fragment du code actuel d'une manière que le type de réusinage voulu soit applicable sur le code. Sinon, il choisit un type de la liste pour la dernière étape du processus. Dans cette étape, le code marqué est retiré de la classe et déplacé vers un aspect. Le processus s'arrêtera lorsqu'il n'existe plus de code marqué.

Compartivement à nos travaux, la modularité visée par ces travaux ne porte pas sur les mêmes éléments des programmes. En effet, dans ces travaux, le refactoring vise à séparer les parties du code relevant de la logique métier (fonctionnalités) des parties du code transverses, qui sont de nature extra-fonctionnelle (authentification et cryptage par exemple, pour l'attribué de qualité sécurité) ou techniques (accès à une base de données, par exemple). Dans nos travaux, nous visons à rendre les applications à objets modulaires en isolant les parties du code qui

relèvent de la description de l'architecture (instanciations, connexions, déclarations d'interfaces requises et fournies, entre autres) des parties du code traitant la logique métier. Les deux catégories de travaux sont parfaitement complémentaires.

2.2 Classes ou programmes mieux découplés

Toutes les apprcohes proposées dans la catégorie précédente visent à améliorer la modularité en passant d'un paradigme de programmation (programmation par objets) à un autre (programmation par composants ou par aspects) connu pour offrir plus de modularité. Cependant, il est possible que l'amélioration de la modularité soit faite en restant dans le même paradigme (en maintenant le même langage de programmation).

Dans ce contexte, Shah et al [18] proposent un algorithme qui utilise différentes techniques de refactoring et permet de retirer automatiquement les dépendances indésirables dans les programmes Java sans affecter leurs fonctionnalités. Cet algorithme vise à éliminer les défauts de modularité représentés par quatre types d'anti-patrons : dépendances circulaires entre les packages, connaissance de sous-types, abstraction sans découplage et héritage dégénéré. Pour cela, ils ont classifié les dépendances entre classes en quatre grandes catégories et pour chaque catégorie ils ont précisé le type de refactoring adéquat.

Wong et al [22] proposent une autre approche. Cette approche nommée Clio permet la détection de la perte de modularité causée par le passage d'une version d'un logiciel à une autre. Clio détecte et localise ces violations. L'idée de base dans ce travail consiste à considérer le fait que si le changement d'un module, pour répondre aux demandes de modifications, implique le changement d'un autre module alors que ces deux modules sont supposés être indépendants, donc une violation de modularité est détectée. Le processus est composé de trois étapes : calcul du couplage structurel, calcul du changement du couplage et identification des violations de modularité en comparant les résultats de la première et la deuxième étape.

Dans les travaux précédents, nous avons cité des méthodes d'amélioration de la modularité couvrant les langages de programmation à objets (Java en particulier). Il existe aussi un ensemble considérable d'approches pour d'autres langages et paradigmes de programmation (voir travaux connexes dans [17]). Li et al [13] proposent une méthode ciblant le langage fonctionnel Erlang. Un programme Erlang consiste en un ensemble de modules et chaque module définit un ensemble de fonctions. Cet article propose un outil de refactoring qui permet d'améliorer la modularité des programmes en detectant des "bad smells " et les supprimer en utilisant une variété de techniques de refactoring. Ces "bad smells" sont au nombre de quatre : mauvaises dépendances inter-modules, dépendances cycliques entre modules, modules servant des objectifs multiples et modules trés grands.

Tous ces travaux partagent le même objectif qui est l'amélioration de la modularité d'une application mais ils diffèrent dans le choix de ce qui est un "bad smell" de modularité. Notre méthode a le même objectif mais avec une exigence en plus qui est d'avoir à la fin du processus une classe conforme à un descripteur de composant, dans lequel le découplage est plus fort, grâce aux dépendances déclarées comme types abstraits et la non-anticipation des instanciations.

3 Processus de refactoring

Les travaux énoncés dans la section 2.1.2 visent à atteindre des objectifs de maintenabilité et de réutilisabilité en améliorant la modularité des applications à objets existantes. Cette amélioration est faite par le groupement des classes dans des clusters et l'élimination des dépendances existantes entre clusters. Mais les dépendances à l'interieur d'un cluster restent toujours. Le fait de grouper des classes dans un cluster est un choix judicieux du point de vue de la cohésion

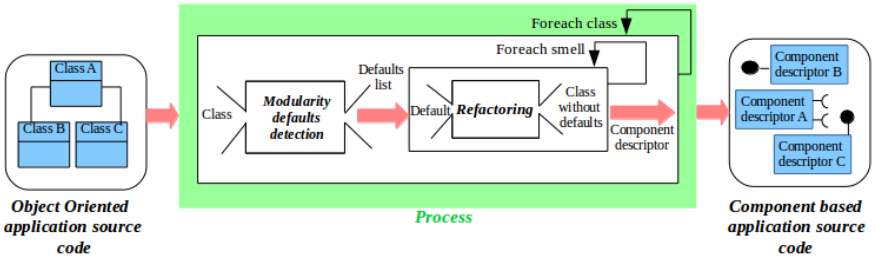

Figure 1: Processus de refactoring

(les classes coopèrent pour former une fonctionnalité précise). Cependant, dans notre travail nous mettons l'accent sur le couplage minimal pour avoir une modularité plus fine et ainsi une meilleure réutilisabilité.

Pour atteindre notre objectif, nous proposons une méthode qui permet de rendre chaque classe de l'application à objets indépendante des autres classes. Donc le descripteur de composant de notre point de vue est une seule classe et non pas un ensemble de classes. Pour cela, il faudra respecter deux principes fondateurs de la programmation par composants : le découplage et la non-anticipation [10]. Partant de ce point de vue, nous avons spécifié deux types de "Défauts"de modularité , I2PI/I2RI et EAI, qui présentent une violation de ces deux principes et qui doivent être éliminé du code.

Contrairement aux approches précédentes, la notre ne nécessite pas une étape de groupement des classes dans des clusters. Le processus que nous allons suivre consiste en deux étapes présentées dans la figure 1.

La première étape consiste à détecter les "Défauts" de modularité. Cette détection est faite en appliquant une analyse statique sur le code source de l'application à objets. Pour effectuer cette analyse, nous recourons à un outil externe : Spoon[1]. À la fin de cette étape, une liste de "Défauts" de modularité existants dans une classe est obtenue(pour le moment nous avons déterminé deux types).

Une fois un "Défaut" est détecté, il faut savoir comment l'éliminer. Ceci est fait dans la deuxième étape qui consiste a restructurer le code de chaque classe d'une manière à ce que tous les "Défauts" seront éliminés. Cette restructuration est achevée par le biais de l'application automatique d'une composition d'opérations de refactoring.

Dans cette section, nous allons définir les deux types de "Défauts" et nous donnons une représentation formelle, donc non-ambiguë, de chacun. Pour simplifier cette représentation, nous allons utiliser un méta-modèle (figure 2), construit à partir de la documentation JDT [2], qui groupe les éléments prinicipaux du code source d'une application à objets. Ce méta-modèle ne contient que les éléments dont nous avons besoin pour la représentation formelle de chaque "Défaut" par une contrainte OCL.

3.1 Inexistence ou incompletude des interfaces fournies (I2PI)/requises (I2RI)

La plupart des applications à objets existantes possèdent plusieurs dépendances entre leurs différentes entités pour l'accomplissement coopératif d'un traitement. Il est donc possible, par

[1] http://spoon.gforge.inria.fr/
[2] http://help.eclipse.org/mars/index.jsp

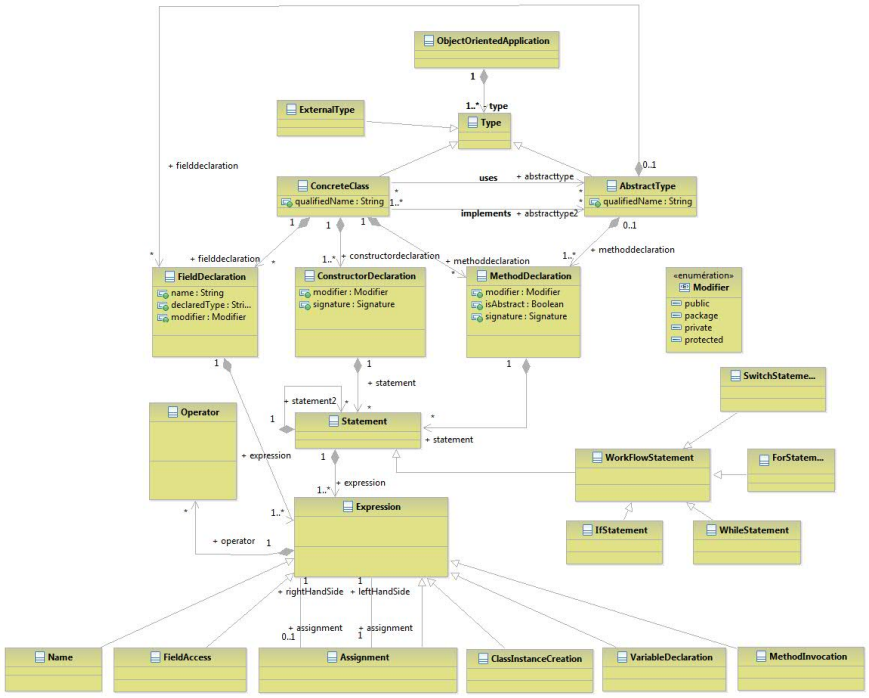

Figure 2: Méta-Modèle extrait de la documentation JDT

exemple, qu'un attribut soit typé par une classe concète de l'application. Cette utilisation de types concrets est justifiée par le fait qu'il arrive parfois qu'une classe n'implémente aucun type abstrait ou qu'une méthode publique définie dans une classe n'est déclarée dans aucun type abstrait réalisé par cette classe alors que cette méthode est accessible depuis n'importe où. Ces types abstraits sont dit "incomplets".

L'utilisation de types abstraits, considérés comme interfaces fournies et requises pour les classes, constitue un des principes de base de la modularité dans les applications à composants. Pour respecter ce principe, chaque classe doit exposer toutes ses méthodes publiques dans des types abstraits pour que les autres classes qui utilisent ces méthodes puissent dépendre de ces types abstraits et non pas d'une classe concrète.

3.1.1 Méthodes exportées des types abstraits et types des attributs

Nous considérons comme "Défaut" de modularité de type I2PI les deux situations suivantes :

- Une classe implémente une méthode publique et la signature de cette méthode n'existe pas dans les types abstraits réalisés par cette classe ou lorsqu'une classe ne réalise aucun type abstrait alors qu'elle implémente des méthodes publiques. Ce "Défaut" est représenté par un triplet (I2PI.M,classe,signaturesManquantes).

- Une classe déclare des attributs publiques. Ce "Défaut" est représenté par un triplet (I2PI.F,classe,NomsDesAttributs).

Nous considérons comme "Défaut" de type I2RI le cas où une classe déclare des attributs qui ont comme types, des classes concrètes et non pas des types abstraits. Ce "Défaut" est

représenté par un triplet (I2RI,classe,signatureDesMéthodesPubliques[1]).

Pour le "Défaut" de type I2PI, nous l'exprimons en OCL par :

```
context ConcreteClass inv :
self.fieldDeclaration->select(f:FieldDeclaration |
f.modifier = Modifier::public)->notEmpty()
or
not ((self.implements.methodDeclarations
->select(m:MethodDeclaration | m.isAbstract))->collect(signature)
->includesAll((self.methodDeclaration->select(m:MethodDeclaration |
m.modifier = Modifier::public))->collect(signature)))
```

Listing 1: Contrainte OCL pour formaliser le bad smell 1

et le "Défaut" de type I2RI est exprimé en OCL de la façon suivante :

```
context ConcreteClass inv:
self.fieldDeclaration.declaredType -> exists(t:Type | t.oclAsType(
    ConcreteClass))
```

Listing 2: Contrainte OCL pour formaliser le bad smell 2

3.1.2 Algorithme de détection de I2PI

L'algorithme 1 explique la manière d'identification de "Défauts" de type I2PI.

3.1.3 Algorithme de détection de I2RI

La façon d'identifier des "Défauts" de type I2RI est exprimée par l'algorithme 2.

3.1.4 Une proposition de refactoring

Le "Défaut" de modularité de type I2PI.M présenté dans la section 3.1.1 peut être éliminé en ajoutant chaque signature manquante d'une méthode publique implémentée dans une classe a l'un des types abstraits réalisés par cette classe. Le choix du type abstrait est fait en comparant la cohésion entre les méthodes dont les signatures existent déjà dans ce type abstrait avec la méthode que nous voulons ajouter. Si elles ont une forte cohésion, la signature de cette méthode est ajoutée, sinon un autre type abstrait est choisi. Une signature peut être ajoutée à un type abstrait à condition qu'il ne soit pas implémenté par une autre classe (différente de celle qui implémente la méthode publique). Si cette condition n'est pas vérifiée, il faut créer un nouveau type abstrait.

Le deuxième "Défaut" de type I2PI.F nécessite le changement du modificateur de l'attribut et le rendre privé et l'ajout de ses accesseurs à la classe qui le déclare.

Le "Défaut" de type I2RI peut être éliminé en remplaçant tous les types (des attributs) qui représentent une référence à une classe concrète par une référence à un type abstrait. Ces types abstraits sont crées en analysant les méthodes externes invoquées dans la classe et en récupérant les signatures de ces méthodes et les ajouter à ce type abstrait.

3.2 Existence d'instanciations anticipées EAI

Dans les classes, les variables typés par des types abstraits (interfaces requises), sont parfois affectés de références vers des objets instanciés à l'intérieur de ces même classes. Nous avons ici ce que l'on appelle une "Instanciation anticipée". Cette instanciation peut se faire dans différents

[1] Ce sont les signatures des méthodes externes invoquées dans la classe

Algorithm 1 detecting inexistance or incompleteness of provided interfaces

Input: The set of classes and abstract types in the source code
Output: All the defaults of type I2PI

```
defaults ← ∅
for all c ∈ C do
    M_c ← getMethodsDeclarations(c)
    F_c ← getFieldsDeclarations(c)
    FN ← ∅                          % A set of field names
    SD ← ∅                          % A set of signatures of public methods defined in c
    for all m ∈ M_c do
        if modifier(m)="public" then
            SD ← SD + getSignature(m)
        end if
    end for
    for all f ∈ F_c do
        if modifier(f) = "public" then
            FN ← FN + fieldName(f)
        end if
    end for
    SR ← getSignatures(getImplInterfaces(c))
    if SD ≠ SR then
        defaults ← defaults + ("I2PI", c, SD − SR)
    end if
    if FD ≠ ∅ then
        defaults ← defaults + ("I2PI", c, FD)
    end if
end for
return defaults
```

niveaux dans le code par exemple l'objet créé peut être stocké dans un attribut d'une classe, dans les variables locales des méthodes ou encore dans les paramètres des méthodes. Dans le présent article, nous nous intéréssons aux objets stockées dans un attribut d'une classe (initialisation des attributs par des instanciations, et instanciations dans les constructeurs) et qui ne sont pas dans une structure de contrôle (pour les instanciations à l'interieur des constructeurs).

L'inconvenient de cette instanciation est que chaque objet référencé devient une dépendance forte car l'objet appelant doit connaitre chacun des objets qu'il va utiliser avant de les instancier et nous pouvons remarquer aussi que l'objet appelant prend la responsabilité de créer les instances des objets auxquels il va déléguer tout ou partie de ses traitements.

3.2.1 Instanciation anticipée

Une instanciation anticipée est une instruction de type expression d'affectation, dans laquelle la partie gauche consiste en une expression de type accès à un attribut (`FieldAccess` dans le méta-modèle) et la partie droite consiste en une expression d'instanciation (`ClassInstanceCreation` dans le méta-modèle). Cette instanciation peut être directe et explicite, en utilisant le constructeur d'une classe (avec *new*) ou indirecte et implicite, suite à l'invocation d'une méthode. L'initialisation d'un attribut par une création d'une instance d'une classe est aussi considérée comme instanciation anticipée (la partie gauche consiste en une ex-

Algorithm 2 detecting inexistance or incompleteness of required interfaces

Input: The set of classes and abstract types in the source code
Output: All the defaults of type I2RI

```
defaults ← ∅
for all c ∈ C do
  S ← ∅
  F_c ← getFieldsDeclarations(c)
  Mi_c ← getMethodsInvocations(c)
  for all f ∈ F_c do
    t ← getDeclaredType(f)
    if t ∈ C then
      for all m ∈ Mi_c do
        if receiver(m)==t then
          S ← S ∪ getSignature(m)
        end if
      end for
    end if
  end for
  if S ≠ ∅ then
    defaults ← defaults + ("I2RI", c, S)
  end if
end for
return defaults
```

pression de déclaration de variable `VariableDeclaration` dans le méta-modèle). Ce "Défaut" est représenté par un triplet (EAI,classe,(affectation,ligneDeAffectation)).

La contrainte suivante formalise ce "Défaut" :

```
context Concreteclass inv :
(self.constructorDeclaration.statement-> select(not(oclIsTypeOf(
    WorkflowStatement))
and expression.assignment.leftHandSide -> oclIsTypeOf(FieldAccess)
and expression.assignment.rightHandSide -> oclIsTypeOf(ClassInstanceCreation))
    )
-> notEmpty()
or
(self.fieldDeclaration.expression.assignment -> select(leftHandSide
-> oclIsTypeOf(VariableDeclaration)
and rightHandSide -> oclIsTypeOf(ClassInstanceCreation)))
-> notEmpty();
```

Listing 3: Contrainte OCL pour formaliser le bad smell 3

3.2.2 Algorithme de détection de EAI

La manière de détection des EAI est exprimée par l'algorithme 3.

3.2.3 Une proposition de refactoring

Comme point de départ, nous nous limitons aux instanciations qui ne sont pas à l'intérieur d'une structure de contrôle. Donc, ce "Défaut" peut être éliminé en supprimant les instanciations

Algorithm 3 detecting existance of early instanciation EAI

Input: The set of classes and abstract types in the source code
Output: All the defaults of type EAI

```
defaults ← ∅
for all c ∈ C do
  CO_c ← getConstructorsDeclarations(c)
  F_c ← getFieldsDeclarations(c)
  for all f ∈ F_c do
    if type(f) = "assignement" ∧ rightHandSide(f) = "classInstanceCreation" then
      defaults ← defaults + ("EAI", c, (f, getLine(f)))
    end if
  end for
  for all co ∈ CO_c do
    Statements ← getStatements(co)
    for all st ∈ Statements do
      if type(st) = "assignement" ∧ rightHandSide(st) = "classInstanceCreation" then
        defaults ← defaults + ("EAI", c, (st, getLine(st)))
      end if
    end for
  end for
end for
return defaults
```

du code et les déléguer à une entité externe qui se charge de les créer pour l'objet appelant. Cela peut se faire en utilisant l'injection de dépendances (elle est intégrée dans beaucoup de frameworks tels que : Spring [1], Google Guice [2] et PicoContainer [3]), si l'instanciation est explicite (avec new).

4 Exemple

Nous prenons l'exemple présenté dans la figure 3 pour expliquer les opérations de refactoring effectuées sur quelques cas de "Défauts" de modularité. Dans cet exemple nous avons deux classes, A et B, et une interface PiA implémentée par la classe A. Nous commençons par les "Défauts" contenus dans la classe A. Cette dernière déclare deux attributs publiques (f1 et f2) et deux méthodes publiques (m1 et m2). Le type de l'attribut f1 est une classe concrète B. La méthode m1 contient une invocation d'une méthode externe m3 dont le receveur est l'attribut f1. La méthode m2 est définie dans la classe A mais la déclaration de sa signature n'existe pas dans l'interface PiA. Donc cette classe contient les "Défauts" représentés par les rectangles colorés suivants :

1. Le rectangle rouge représente un "Défaut" de type "I2PI.F". Pour l'éliminer, il faut:changer la visibilité des attributs (f1 et f2) et la rendre privée, ajouter les accesseurs de ces attributs à la classe A, ajouter les declarations des signatures des accesseurs à l'interface PiA (cette interface n'est pas implémentée par d'autres classes, s'il existe d'autres classes qui l'implémentent, une nouvelle interface est créée et les declarations des signatures seront

[1] https://spring.io/
[2] https://github.com/google/guice
[3] http://picocontainer.com/

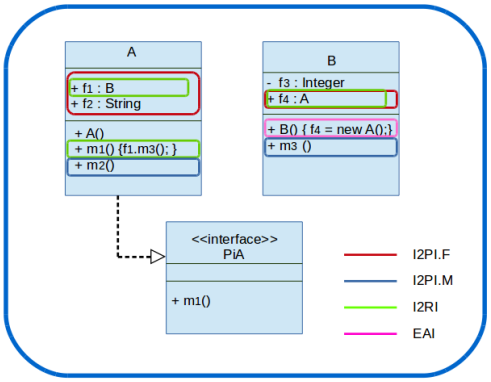

Figure 3: Classes avec "Défauts"

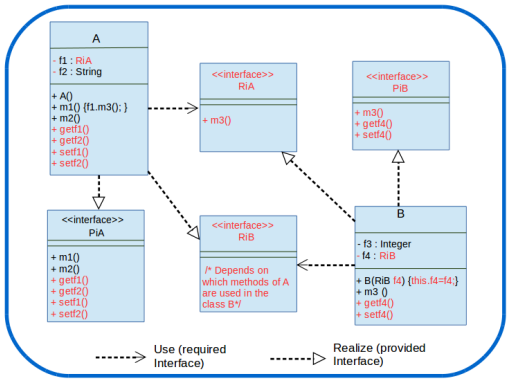

Figure 4: Classes aprés refactoring

ajoutées à cette interface) et modifier toutes les instructions d'accès à ces attributs par l'utilisation des accesseurs.

2. Les deux rectangles vert représentent un "Défaut" de type "I2RI". Pour l'éliminer, il faut:chercher toutes les invocations de méthodes externes dont le receveur est l'attribut B, récupérer les signatures de ces méthodes, créer une nouvelle interface (considérée comme interface requise), ajouter les signatures à cette interface et changer le type de l'attribut qui était une classe concrète et le rendre l'interface nouvellement créée.

3. Le rectangle bleu représente un "Défaut" de type "I2PI.M". Pour l'éliminer il faut juste ajouter la declaration de la signature de m2 à l'interface PiA (s'il existe d'autres classes qui implémentent cette interface , une nouvelle interface est créée et la declaration des signatures sera ajoutée à cette interface).

Pour la deuxième classe B, nous n'allons pas parler de tous les "Défauts". Nous prenons le "Défaut" représenté par le rectangle rose sur la figure 3. Ce "Défaut" est de type EAI, pour l'éliminer il faut remplacer l'instanciation par un passage d'une instance de l'interface, créée pour remplacer le type de l'attribut f4 par un type abstrait, au constructeur de la classe B. Le résultat de l'application de ces opérations de refactoring est représenté dans la figure 4.

5 Implémentation et premières expérimentations

Nous avons implémenté la phase d'identification et nous l'avons utilisé pour évaluer la fréquence des "Défauts" de modularité dans des applications réelles. Cet outil prend en entrée deux répertoires, le premier contenant les sources à analyser et le second les librairies au format ".jar" et construit une structure proche de celle d'un `AST (Abstract Syntax Tree)`. Cette structure sera utilisée comme paramètre pour les algorithmes d'identification des différents "Défauts".

Pour notre étude, nous avons choisi 5 applications, développées en Java, qui sont toutes open source. Nous avons recueilli des applications possèdant différentes tailles (nombre de classes) et développées par différentes équipes pour éviter l'influence des caractéristiques liées aux habitudes de l'équipe sur les résultats. Le tableau 1 fournit une brève description de ces applications.

Application	Description	LOC	nb classes (Tc)	nb interfaces + nb classes abstraites (Ta)	Abstraction (Ta/Tc)
FreeCS1.3	Serveur de discussion instantanée	23012	139	17+6	16,5%
CoCoME	Application commerciale	5779	99	21+0	21,2%
log4j	Un framework de logging	2129	214	20+16	16,8%
JUnit4.11	Un framework pour écrire des tests	7428	1042	37+30	6,42%
Jasml0.10	Outil de visualisation et d'edition de classes Java	5732	50	1+0	2%

Table 1: Les applications choisies

Les tailles des applications varient entre 2129 et 23012 LOC, 50 et 1024 classes concrètes et 1 et 67 types abstraits. Dans cette évaluation, Nous déffinissons dans la dernière colonne du tableau une métrique "Abstraction de l'application" qui représente le rapport entre les types abstraits (interfaces + classes abstraites de l'application) et nombre de classes concrètes. De point de vue abstraction, nous pouvons remarquer que CoCoME possède la plus grande valeur et Jasml0.10 la plus petite.

Le tableau 2 résume les résultats de notre étude expérimentale. Nous pouvons remarquer que les cinq applications contiennent les deux types de "Défauts" (I2PI/I2RI et EAI). Si nous prenons Jasml, premièrement, nous trouvons que toutes les méthodes publiques définies dans les classes de cette application, leurs signatures n'existent pas dans les interfaces. Il était évident que nous obtenons ce résultat car la valeur d'abstraction de l'application était faible. Deuxièment, nous pouvons remarquer aussi que le nombre d'instanciations considérées comme "Défauts" dans notre cas représente 75% du nombre total d'instanciations dans cette application ce qui est un pourcentage élevé.

Cette évaluation de l'échantillon des applications a montré que la prévalence des "Défauts" dans les applications à objets existantes est non négligeable ce qui nécessite un refcatoring pour l'amélioration de leur modularité.

6 Conclusion et perspectives

Ce papier présente une méthode de refactoring que nous proposons pour l'amélioration de la modularité des applications à objets existantes. Cette méthode consiste à détecter les deux types de "Défauts" de modularité que nous avons identifié, I2PI/I2RI et EAI, et les supprimer en utilisant des opérations de refactoring. Seule l'étape de détection a été implémentée, la

Application	Défauts				
-	I2PI		I2RI		EAI
-	Attributs public	Méthodes publiques absentes des interfaces / nb total de méthodes publiques	Attributs de type classe propre à l'application / nb total d'attributs	Invocations de méthodes externes / nb total d'invocations	Instanciations pouvant être extraites / nb total d'instan cia-tions
FreeCs	419	869/1146	144 (41 public)/969	1222/10080	12/79
CoCoME	7	295/383	61 (0 public)/283	119/1570	7/30
log4j	158	1245/1421	192 (48 public)/976	396/5982	26/93
Junit	254	2650/2798	244 (118 public)/819	99/7258	25/58
Jasml	439	84/84	8(1 public)/514	170/1182	6/8

Table 2: Identification des "Défauts" de modularité

seconde étape est en cours d'étude et de réalisation

Nous pouvons énoncer comme perspectives : la conduite des expérimentations sur un nombre important d'applications et la définition d'un protocole de validation de toute l'approche.

References

[1] Ieee standard for software maintenance. *IEEE Std. 1219-1998. The Institute of Electrical and Electronics Engineers*, 1998.

[2] Nouh Talal Alhindawi. *Supporting Source Code Comprehension During Software Evolution and Maintenance*. PhD thesis, Kent State University, 2013.

[3] Simon Allier, Salah Sadou, Houari Sahraoui, and Régis Fleurquin. From object-oriented applications to component-oriented applications via component-oriented architecture. In *Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on*, pages 214–223. IEEE, 2011.

[4] Zakarea Alshara, Abdelhak-Djamel Seriai, Chouki Tibermacine, Hinde Lilia Bouziane, Christophe Dony, and Anas Shatnawi. Migrating large object-oriented applications into component-based ones: instantiation and inheritance transformation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 55–64. ACM, 2015.

[5] David Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Tool-supported refactoring of existing object-oriented code into aspects. *Software Engineering, IEEE Transactions on*, 32(9):698–717, 2006.

[6] Silvia Breu and Jens Krinke. Aspect mining using event traces. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 310–315. IEEE Computer Society, 2004.

[7] Sylvain Chardigny. *Extraction d'une architecture logicielle à base de composants depuis un système orienté objet. Une aproche par exploration.* PhD thesis, Université de Nantes, 2009.

[8] Eleni Constantinou, Athanasios Naskos, George Kakarontzas, and Ioannis Stamelos. Extracting reusable components: a semi-automated approach for complex structures. *Information Processing Letters*, 115(3):414–417, 2015.

[9] Stéphane Ducasse and Damien Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Trans. Software Eng.*, 35(4):573–591, 2009.

[10] Luc Fabresse. *Du découpage à l'assemblage non anticipé de composants: Conception et mise en oeuvre du langage à composants Scl.* PhD thesis, Université Montpellier II-Sciences et Techniques du Languedoc, 2007.

[11] Norman E Fenton and Shari Lawrence Pfleeger. Software metrics: A rigorous and practical approach. 1998.

[12] Martin Fowler. Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland*, 1997.

[13] Huiqing Li and Simon Thompson. Refactoring support for modularity maintenance in erlang. In *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on*, pages 157–166. IEEE, 2010.

[14] Thibaud Lutellier, Devin Chollak, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidovic, and Robert Kroeger. Comparing software architecture recovery techniques using accurate dependencies. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, pages 69–78, 2015.

[15] Marius Marin, Arie Van Deursen, and Leon Moonen. Identifying aspects using fan-in analysis. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 132–141. IEEE, 2004.

[16] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, 36(1):20–36, 2010.

[17] Soroush Radpour, Laurie Hendren, and Max Schäfer. Refactoring matlab. In *Compiler Construction*, pages 224–243. Springer, 2013.

[18] Syed Muhammad Ali Shah, Jens Dietrich, and Catherine McCartin. On the automation of dependency-breaking refactorings in java. In *2013 IEEE International Conference on Software Maintenance*, pages 160–169. IEEE, 2013.

[19] Paolo Tonella and Mariano Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 112–121. IEEE, 2004.

[20] Tom Tourwe and Kim Mens. Mining aspectual views using formal concept analysis. In *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, pages 97–106. IEEE, 2004.

[21] Hironori Washizaki and Yoshiaki Fukazawa. A technique for automatic component extraction from object-oriented programs by refactoring. *Science of Computer programming*, 56(1):99–116, 2005.

[22] Sunny Wong, Yuanfang Cai, Miryung Kim, and Michael Dalton. Detecting software modularity violations. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 411–420. ACM, 2011.

[23] Edward Yourdon and Larry L Constantine. *Structured design: Fundamentals of a discipline of computer program and systems design.* Prentice-Hall, Inc., 1979.