Vers une nouvelle approche d'extraction de la logique métier d'une application orientée objet

Ismaïl Khriss*, Gino Chénard**

* Université du Québec à Rimouski
300, allée des Ursulines, C.P. 3300 Rimouski (Québec) G5L 3A1
ismail_khriss@uqar.qc.ca
** Université du Québec à Montréal
201, avenue du Président-Kennedy, Montréal (Québec) H2X 3Y7
chenard.gino@courrier.uqam.ca

Résumé. Les compagnies font face à d'énormes coûts pour maintenir leurs applications informatiques. Ces applications contiennent des connaissances corporatives importantes qui deviennent difficiles à récupérer après plusieurs années d'opération et d'évolution. Plusieurs approches ont été proposées afin d'extraire du code source des abstractions pour aider les développeurs à assimiler ces connaissances. Cependant, l'abstraction extraite par la plupart des approches combine la logique métier de l'application et son architecture. Nous proposons une nouvelle approche pour extraire le modèle d'une application orientée objet. Ce modèle est donné comme un diagramme de classes UML présentant les classes métier de l'application et leurs interrelations. Cette approche a été validée sur plusieurs systèmes écrits en Java et donne de bons résultats pour les systèmes bien structurés avec un bon style de programmation.

1 Introduction

Au cours de leur vie, les logiciels ont à évoluer. Cette évolution peut être nécessaire afin de corriger des erreurs ou pour l'adaptation à de nouvelles exigences ou technologies. Dans un contexte idéal, une application est bien documentée ce qui rend aisé sa compréhension. Malheureusement, la documentation est souvent incomplète, insuffisante ou non à jour. En l'absence d'une documentation adéquate, la maintenance logicielle devient difficile. Pour combler le manque de documentation, les développeurs passent alors un temps non négligeable à tenter de comprendre le code source du logiciel.

Une solution pour simplifier la tâche de compréhension du logiciel est de faciliter la navigation dans le code source. Des environnements de développement modernes tels que Visual Studio.Net et Eclipse offrent de telles fonctionnalités. Malheureusement, cette pratique présente des limites pour les logiciels de grande taille, car le développeur peut être enseveli sous la quantité d'information affichée. Il faut donc chercher à simplifier et à mieux cibler l'information affichée. Une abstraction intéressante pouvant être obtenue représente les règles métier d'une application. Peu d'approches de rétro-ingénierie tentent d'extraire cette abstraction. Il est difficile de différencier le code relié à la logique métier de celui relié à l'infrastructure de l'application.

Dans cet article, nous présentons une nouvelle approche qui permet d'isoler un modèle métier d'une application en retirant le code relié à l'infrastructure. L'extraction est rendue possible en analysant comment de nouvelles spécifications d'architecture, telles que J2EE, sont mises en application dans des applications orientées objet modernes. Ce modèle prend la forme d'un diagramme de classes UML représentant ses classes métier et leurs relations. Cette approche a été validée sur plusieurs systèmes écrits en Java. Elle donne de bons résultats pour ceux bien structurés avec un bon style de programmation.

Cet article est organisé comme suit. La section 2 détaille notre approche. La section 3 discute de quelques travaux reliés. À la section 4, nous donnons les résultats de la validation de notre approche et des directions que nous allons suivre comme suite à ce travail. Finalement, la section 5 fournit des remarques en guise de conclusion.

2 Étapes de l'approche

Notre approche est constituée de trois étapes principales. La première étape consiste à faire une analyse du code source d'une application afin d'obtenir un arbre syntaxique abstrait (AST). Cet arbre contient en fait les différentes classes de l'application.

La deuxième étape consiste à extraire les classes métier à partir de l'arbre syntaxique abstrait. Cette extraction se base sur un ensemble de règles. La troisième étape consiste à générer un document XML décrivant un diagramme UML montrant les classes métier et leurs relations. Ce document XML suit la norme XMI (pour XML Model Interchange) (Grose *et al.* 2002). Le but ici est de permettre d'utiliser n'importe quel outil de modélisation UML supportant la norme XMI pour permettre une visualisation graphique du modèle extrait.

La première étape et la dernière étape sont basées sur des principes standards. Notre principale contribution est la deuxième étape et nous allons la décrire en détail au travers de cette section. Pour illustrer notre approche, nous allons utiliser une application « jouet ». L'application contient vingt-huit classes et interfaces (voir Tab. 1). Le nom des classes nous informe que nous sommes devant une application bancaire. La figure 1 donne le résultat final de notre approche. Nous voyons que seulement sa logique métier est extraite.

2.1 Définitions

Nous commençons par clarifier quelques termes que nous emploierons dans la description de nos règles.

Le *vocabulaire d'une application* est l'ensemble des mots contenus dans le code d'une application (commentaires exclus). Ce vocabulaire (dénoté par V) est constitué de : V_{BL} : le vocabulaire introduit par la logique métier, V_A : le vocabulaire introduit par une architecture, V_{PL} : le vocabulaire introduit par les langages de programmation. Nous avons $V = V_{BL} \cup V_A \cup V_{PL}$. Notre but est d'identifier V_{BL}. Il est facile de trouver V_{PL} et si nous pouvons trouver V_A, nous pouvons aussi identifier V_{BL} car : $V_{BL} = (V \setminus V_{PL}) \setminus V_A$.

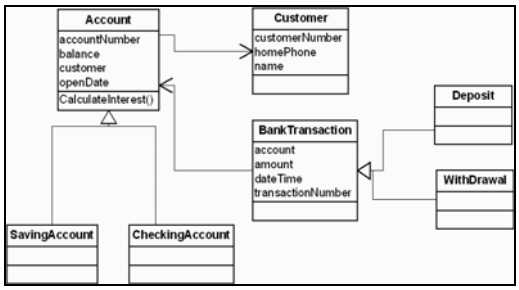

FIG. 1 – Diagramme de classe d'UML résultant de notre approche

Account	BankTransactionWebFacade	CustomerHome	SavingAccountBean
AccountBean	CheckingAccount	CustomerWebFacade	SavingAccountHome
AccountHome	CheckingAccountBean	Deposit	SavingAccountWebFacade
AccountWebFacade	CheckingAccountHome	DepositBean	Withdrawal
BankTransaction	CheckingAccountWebFacade	DepositHome	WithdrawalBean
BankTransactionBean	Customer	DepositWebFacade	WithdrawalHome
BankTransactionHome	CustomerBean	SavingAccount	WithdrawalWebFacade

TAB. 1 – *Nom des classes*

Un *patron d'un identificateur* est un mot (ou une combinaison de mots) qui est une sous chaîne de l'identificateur. Nous obtenons les patrons en divisant les identificateurs par des séparateurs tels que les majuscules ou le symbole de soulignement. Par exemple, l'identificateur `AccountBean` contient trois patrons : `Account`, `Bean` et `AccountBean`.

Un *groupe relié à un patron* est l'ensemble de tous les identificateurs de classes contenant ce patron. Par exemple, pour notre application le groupe {`Deposit`, `DepositBean`, `DepositHome`, `DepositWebFacade`} est le groupe relié au patron `Deposit`. Évidemment, un identificateur peut se retrouver dans plusieurs groupes.

Un *sous-groupe relié à un patron* est obtenu en partitionnant, suivant un critère, le groupe relié à ce patron. Par exemple, le critère peut être de quelle classe il hérite ou quelle interface il implémente. Évidemment, une classe peut se retrouver dans différents sous-groupes d'un groupe relié à un patron. Par exemple, les classes `AccountBean` et `TransactionBean` qui appartiennent au groupe du patron `Bean` appartiennent à des sous-groupes différents suivant le critère d'implémentation d'interface. `AccountBean` implémente l'interface `EntityBean` alors que `TransactionBean` implémente l'interface `SessionBean`.

Une *méthode principale* est une méthode dont le nom n'a pas en commun de patrons avec le nom de la classe ou de ses attributs. Ainsi, les constructeurs et les « setters » ne sont pas des méthodes principales. Dans notre exemple la méthode `setSessionContext` n'est pas une méthode principale de la classe `TransactionBean` car elle possède l'attribut `sessionContext`. Par contre, la méthode `ejbCreate` de la même classe en est une, car son nom n'a pas de patron en commun avec le nom de la classe ou aucun de ses attributs.

2.2 L'étape d'extraction des règles métier

L'étape d'extraction des règles métier est basée sur l'analyse des identificateurs trouvés dans le code source d'une application. Cela est fait à partir d'une élimination par filtrage du

code relié à l'infrastructure. Cette étape est constituée de six sous-étapes : extraire les patrons, extraire les mots architecturaux, extraire les classes métiers, extraire les attributs des classes, extraire les opérations de classes et extraire les relations entre les classes.

2.2.1 Sous-étape d'extraction des patrons

La première sous-étape consiste à extraire les patrons présents dans le nom de toutes les classes d'une application. Nous appellerons L_P l'ensemble des patrons obtenus. Des exemples des éléments de L_P sont : `Account`, `AccountBean`, `Bank`, `Home` et `Bean`.

2.2.2 Sous-étape d'extraction des mots architecturaux

La deuxième sous-étape consiste à extraire les mots architecturaux de l'ensemble de patrons L_P trouvés dans la sous-étape précédente. Cette sous-étape est basée sur trois règles (les règles R1, R2, R3) que nous allons donner dans ce qui va suivre.

Règle 1 (R1). Si un patron isolé n'est pas le nom d'une classe alors ce patron est un mot architectural candidat. Formellement, si nous représentons par L_{AC} l'ensemble de tous les mots architecturaux candidats et par L_C l'ensemble des noms de toutes les classes d'une application, nous devrons avoir :

$$L_{AC} = L_P \backslash L_C \qquad (1)$$

Cette règle fait une première sélection et se base sur le fait qu'un mot architectural ne devrait jamais être choisi comme nom d'une classe. Dans notre exemple, les mots suivants sont sélectionnés comme mots architecturaux candidats : `Bean`, `Checking`, `Home`, `Web`, etc. Par contre, `Account` n'est pas sélectionné, car le code source contient une classe ayant ce nom.

Règle 2 (R2). Un mot architectural candidat est sélectionné pour faire partie du vocabulaire d'architecture si au moins deux classes font partie exclusivement de son groupe. Formellement, pour un mot architectural X , G_x dénotant son groupe, nous aurons :

$$\begin{gathered}\left(X \in L_{AC}\right) \wedge \left(\left|G_x \backslash \bigcup_{Y \in L_{AC} \wedge Y \neq X} G_y\right| \geq 2\right) \\ \Rightarrow X \in V_A\end{gathered} \qquad (2)$$

Intuitivement, nous devons trouver dans le code source de l'application au moins deux classes[1] qui implantent exclusivement le mécanisme architectural représenté par le mot architectural candidat. Dans notre exemple, cette règle ne conserve que les mots architecturaux candidats `Bean`, `Home` et `WebFacade`. Les autres ne seront donc pas des mots architecturaux.

Règle 3 (R3). Un mot architectural candidat est sélectionné pour faire partie du vocabulaire d'architecture si un des sous-groupes qui lui sont reliés (en relation avec l'implantation d'une interface ou de l'héritage d'une classe abstraite) satisfait les trois conditions suivantes :

1. il contient au moins deux classes
2. une de ses classes possède au moins une méthode principale
3. toutes ses classes ont les mêmes méthodes principales.

[1] L'implantation d'un mécanisme architectural doit se répéter au moins une fois (d'où le nombre 2 de la règle) pour qu'il soit considéré.

Formellement, si pour un mot architectural X, nous dénotons par G_x un de ses sous-groupes et par P_x l'ensemble des méthodes principales des classes appartenant à G_x, nous devrons avoir :

$$\begin{gathered}(X \in L_{AC}) \wedge (|G_x| \geq 2) \wedge (P_x \neq \varnothing) \wedge \\ (\forall C \in G_x, \forall m \in P_x, m \in C) \\ \Rightarrow X \in V_A\end{gathered} \quad (3)$$

Intuitivement, le mécanisme architectural derrière un mot architectural devrait être implanté uniformément. Dans notre exemple, `Bean` est conservé dans V_A puisqu'un de ses sous-groupes (celui implantant l'interface `EntityBean`) satisfait les conditions des règles.

2.2.3 Sous-étape d'extraction des classes métier

Une fois les mots architecturaux extraits, nous pouvons trouver dans l'application analysée, les mots du code source susceptibles de représenter des noms de classes métier. Cela est fait par la règle suivante.

Règle 4 (R4). Pour chaque classe du code source, le nom, obtenu en enlevant toutes les occurrences des mots architecturaux le contenant, devient le nom d'une classe métier. Formellement, si pour une classe C, $strip(V_A, C)$ est une fonction qui enlève dans le nom de C les occurrences de tous les mots du vocabulaire d'architecture. Nous devrons avoir :

$$\begin{gathered}C \in L_c \\ \Rightarrow strip(V_A, C) \in V_{BL}\end{gathered} \quad (4)$$

Pour notre exemple, nous obtenons les classes métier suivantes : `Account`, `BankTransaction`, `CheckingAccount`, `Customer`, `Deposit`, `SavingAccount` et `Withdrawal`.

2.2.4 Sous-étape d'extraction des attributs

Le but de cette sous-étape de l'étape d'extraction des règles métier est d'identifier pour chaque classe métier découverte dans la troisième étape, les mots du code source qui peuvent représenter ses attributs. Cette sous-étape est basée sur la règle R5.

Règle 5 (R5). Un attribut appartenant à une classe d'un groupe relié à une classe métier est considéré comme un attribut de cette classe si cet attribut n'appartient pas à toutes les classes d'aucun des sous-groupes d'un mot architectural en lien avec le critère d'implantation d'une interface ou d'héritage d'une classe abstraite. Un sous-groupe doit contenir au moins deux classes. Formellement, si nous dénotons par G_C un sous-groupe relié à une classe métier C, par $G_{A,C}$ l'ensemble des attributs appartenant à G_C, par a un de ces attributs et par G_x un groupe relié à un mot architectural X, nous devrons avoir :

$$\begin{gathered}(C \in V_{BL}) \wedge (a \in G_{A,C}) \wedge \\ (\nexists X \in V_A, (|G_x| \geq 2) \wedge (\forall C1 \in G_x, a \in C1)) \\ \Rightarrow a \in C\end{gathered} \quad (5)$$

Dans notre exemple, pour la classe métier `Account` et l'attribut `accountNumber`, il n'y a aucun groupe architectural où toutes les classes possèdent cet attribut. L'attribut est alors

certainement un attribut métier de la classe métier. Cette règle discerne également que l'attribut `entityContext` n'est pas un attribut métier de la classe `Account`.

2.2.5 Sous-étape d'extraction des opérations métier

L'objectif de cette sous-étape est d'identifier pour chaque classe métier découverte par la troisième sous-étape, les mots du code source susceptibles de représenter ses opérations métier. Cette sous-étape est décrite par les règles R6 et R7.

Règle 6 (R6). Une méthode appartenant à une classe d'un groupe relié à une classe métier est considérée comme une opération de cette classe métier si cette méthode n'appartient pas à toutes les classes appartenant à un sous-groupe d'un mot architectural en lien avec le critère d'implantation d'une interface ou d'héritage d'une classe abstraite. Formellement, si nous dénotons par G_C un sous-groupe relié à une classe métier *C*, par $G_{M,C}$ l'ensemble contenant les méthodes de toutes les classes appartenant à G_C, par m une de ces méthodes et par G_x un groupe relié à un mot architectural X, nous devrons avoir :

$$(C \in V_{\mathrm{BL}}) \wedge (m \in G_{M,C}) \wedge (\nexists X \in V_{\mathrm{A}}, (|G_x| \geq 2) \wedge (\forall C1 \in G_X, m \in C1)) \Rightarrow m \in C \quad (6)$$

Dans notre exemple, puisque toutes les classes du groupe relié à `Bean` possèdent la méthode `ejbActivate`, cet attribut n'est pas une opération métier de la classe `Account`.

Règle 7 (R7). Une méthode appartenant à une classe d'un groupe relié à une classe métier est considérée comme une opération de cette classe si le nom de cette méthode ne contient pas le nom d'une classe ou le nom d'un attribut d'une classe de ce groupe. Formellement, si nous dénotons par G_C le groupe relié à une classe métier `C`, par $G_{M,C}$ (respectivement $G_{A,C}$) l'ensemble de toutes les méthodes (respectivement les attributs) des classes appartenant à G_C, par m (respectivement a) une de ces méthodes (respectivement attributs), par G_x un groupe relié à un mot architectural X et par $sub(p1, p2)$ une fonction qui retourne les booléens *vrai* si $p1$ est une sous-chaîne de $p2$ et *faux* dans le cas contraire, nous devrons avoir :

$$(C \in V_{\mathrm{BL}}) \wedge (m \in G_{M,C}) \wedge (\forall C1 \in G_C, \forall a \in G_{A,C}, sub(C1, m) = false \wedge sub(a, m) = false) \Rightarrow m \in C \quad (7)$$

Cette règle permet de détecter les méthodes utilitaires comme les « getters ». Par exemple, la méthode `getAccountNumber` ne sera pas considérée comme une opération métier de la classe `Account` puisque le nom de la méthode contient le nom de l'attribut `accountNumber`.

2.2.6 Sous-étape d'extraction des relations entre les classes

La dernière sous-étape de l'extraction des règles métier consiste à extraire du code, les relations qui existent entre les classes métier. Cette sous-étape est basée sur les règles R8 et R9.

Règle 8 (R8). Si le type d'un attribut d'une classe métier $C1$ est une classe métier $C2$ alors cet attribut n'est plus considéré comme un attribut de la classe $C1$ mais plutôt une associa-

tion de la classe $C1$ vers la classe $C2$. Le nom de l'extrémité $C2$ de cette association portera le nom de cet attribut. Formellement, si pour deux classes métier $C1$ et $C2$, a un attribut de $C1$, $type(e)$ une fonction qui retourne le type d'un élément e, AS l'ensemble des relations de type *association*, et $end1(R)$ (respectivement $end2(R)$) une fonction qui retourne l'extrémité gauche (respectivement droite) d'une association R, nous devrons avoir :

$$\begin{gathered}(C1 \in V_{BL}) \wedge (C2 \in V_{BL}) \wedge (a \in C1) \wedge (type(a) = C2) \\ \Rightarrow (C1 = C1 \setminus \{a\}) \wedge (R \in AS) \wedge \\ (type(end1(R) := C1)) \wedge (type(end2(R) := C2)) \wedge \\ (name(end2(R)) := a)\end{gathered} \quad (8)$$

Par exemple, comme l'attribut `customer` de la classe métier `Account` est du type de la classe métier `Customer`, une association entre les deux classes sera créée et l'attribut `customer` ne sera plus un attribut de la classe `Account`.

Règle 9 (R9). Une relation d'héritage où le descendant est la classe métier $C1$ et l'ancêtre est la classe métier $C2$ (ou une classe de son groupe G_{C1}), il y a dans le modèle métier résultant une relation d'héritage entre $C1$ et $C2$. Formellement, si nous dénotons par IH l'ensemble des relations d'héritage, $sub(R)$ (respectivement $\sup(R)$) une fonction qui retourne la sous-classe (respectivement la super classe) de la relation d'héritage R et par $inherits(p1, p2)$ qui est une fonction qui retourne le booléen *vrai* si $p1$ hérite de $p2$ et *faux* dans le cas contraire. Nous devrons avoir :

$$\begin{gathered}(C1 \in V_L) \wedge (C2 \in V_L) \wedge \\ (\exists C3 \in G_{c1}, \exists C4 \in G_{c2} / (inherits(C3, C4)) \\ \Rightarrow R \in IH \wedge sub(R) := C1 \wedge \sup(R) := C2\end{gathered} \quad (9)$$

Dans notre exemple, une relation d'héritage est créée entre la classe métier `CheckingAccount`, et la classe métier `Account` dans le modèle métier parce que le premier hérite d'une classe (`CheckingAccountWebFacade`) du groupe de la seconde.

3 Travaux reliés

Dans cette section, nous discutons de quelques approches reliées.

Hung et Zoo (Hung et Zou 2005) ont le même objectif que nous. Ils proposent une approche pour extraire la logique métier d'une application. Par contre, leur approche ne supporte que les architectures trois tiers. Ils identifient la couche de logique métier en examinant le flux d'information entre les couches. Nous considérons que l'analyse de flux d'information peut donner de bons résultats pour nos objectifs. Nous allons donc étudier cet aspect pour voir comment nous pouvons employer ce genre d'analyse.

Plusieurs approches ont été proposées pour extraire un diagramme de classe d'UML du code source d'une application (Jackson et Waingold 2001 ; Gogolla et Kollman 2000 ; Kollmann et Gogolla 2001 ; Barowski et Cross Ii 2002 ; Guéhéneuc et Albin-Amiot 2004 ; Keschenau 2004 ; Sutton et Maletic 2007). Aucune de ces approches n'effectue une sélection parmi les classes à importer dans le modèle UML. Cependant, bon nombre d'entre elles font un bon travail dans l'extraction de caractéristiques UML pour lesquelles la correspondance n'est pas un à un. Des exemples de ces caractéristiques sont les relations de composition et

d'agrégation, qui peuvent être implantées de la même façon (Kollman *et al.* 2002). À l'opposé, notre travail fait seulement une analyse simple d'extraction des associations entre les classes métier (voir notre discussion dans la prochaine section).

Le travail proposé par Anquetil et Lethbridge (Anquetil et Lethbridge 1999) prend la même approche que le notre dans le sens qu'il est aussi basé sur les identificateurs trouvés dans le code source. Leur objectif est de partitionner les vieux systèmes en sous-systèmes. Les identificateurs examinés sont le nom des fichiers. Leur analyse est plus générale du fait qu'ils prennent en compte que les noms peuvent contenir des abréviations. Ces abréviations sont ou ne sont pas divisées par des séparateurs qui peuvent même se chevaucher.

Hamou-Lhadj et al. (Hamou-Lhadj *et al.* 2005) introduisent une approche pour filtrer les traces d'exécution d'une application pour en retirer les composantes utilitaires et conserver celles implémentant un concept de haut niveau. Une composante utilitaire est identifiée par le fait que cette dernière sera accédée par plusieurs autres composantes d'une application. L'approche construit un graphe où les nœuds sont des composantes et les arêtes les appels entre composantes.

Plusieurs travaux basés sur la détection de clichés sont disponibles dans la littérature. Dans la plupart des cas, ceux-ci recherchent des patrons de conception. La plupart des approches proposent un langage pour spécifier un patron de conception à rechercher. Certains s'intéressent seulement à la spécification statique d'un patron, comme le travail de Kramer et Prechelt (Kramer et Prechelt 1996). D'autres s'intéressent aussi à la description dynamique des patrons de conception comme le travail de Heuzeroth et al. (Heuzeroth *et al.* 2003). Les travaux basés sur la détection de clichés sont limités par le fait qu'il est difficile sinon impossible de prévoir toutes les implémentations possibles d'un patron de conception. Par exemple, Kim et Benner ont identifié que le patron Observer possède au moins douze implémentations différentes (Kim Jung et Benner Kevin 1996). Notez que c'est pour cette raison que quelques travaux analysent à la place la présence des micro patrons (Kim *et al.* 2006). Les micro patrons sont des patrons de conception à grains plus fins et plus près de code source que les patrons.

Le travail proposé par Ducasse et al. (Ducasse *et al.* 1999) est un exemple d'approche de détection de clones. L'aspect le plus intéressant de leur approche est qu'ils introduisent une méthode visuelle pour la détection de clones. Nous pensons que la détection de clones (pseudo clones particulièrement) pourrait augmenter l'exactitude de notre travail. En effet, elle pourrait nous aider à identifier les parties du code source qui mettent en application un mécanisme architectural, puisque les morceaux de code mettant en application le même mécanisme architectural sont répétitifs ou semi-répétitifs.

4 Validation de l'approche

Afin de valider notre approche, nous avons développé un outil la soutenant. Dans cette section, nous présentons notre cadre de validation. Par la suite nous présentons les résultats obtenus par le processus de validation. Finalement, nous discutons de plusieurs aspects de l'approche proposée et pointons quelques travaux futurs.

4.1 Cadre de validation

Nous avons appliqué notre prototype à plusieurs systèmes de tailles différentes. La première mesure obtenue est la précision. Elle cherche à évaluer si les éléments identifiés comme éléments métier en sont vraiment. Les classes identifiées par l'outil comme classes métier et qui n'en sont pas vraiment sont des « faux positifs ». La seconde mesure est la couverture (ou taux de rappel). Il s'agit d'évaluer quel pourcentage des éléments recherchés est identifié. Nous avons tenté d'obtenir des mesures sur tous les éléments importants de notre approche. Ces éléments sont : les mots architecturaux, les classes métier, les opérations métier, les attributs métier et leurs relations.

Le protocole que nous avons suivi pour obtenir les mesures se résume ainsi :

- Les résultats obtenus par notre prototype sont comparés avec la documentation du système analysé s'il en possède. La documentation que nous recherchons est un document contenant un diagramme de classes représentant les classes métier du système ou tout document permettant d'extraire ce genre d'informations.
- Dans le cas où cette comparaison montrerait que la documentation n'est peut-être pas à jour, une analyse manuelle du code est faite pour vérifier notre hypothèse.
- Si la documentation n'est pas présente, nos résultats sont comparés avec les résultats obtenus suite à une analyse manuelle du code.
- L'analyse manuelle du code est faite quand la taille du système analysé le permet.
- Dans le cas où la taille du système ne permettrait pas une analyse manuelle, cette analyse sera plutôt faite sur un échantillon du système analysé. Dans le cas de l'échantillonnage, nous avons sélectionné en analysant le code source un ensemble de classes métier et de couches architecturales présentes dans le système pour ensuite vérifier leur détection par l'outil.

4.2 Résultats de validation

Le tableau 2 résume les résultats obtenus par notre processus de validation. Le premier test est simplement notre exemple d'illustration conçu spécialement pour mettre en évidence les propriétés de notre approche. Cela explique le résultat parfait réalisé.

4.2.1 Code généré par SourceCafe

Nous avons soumis à notre outil un exemple de code généré avec un générateur de code nommé « SourceCafe » de la compagnie EJD Technologies (EJD 2006). Ce qui est intéressant avec cet exemple, c'est qu'un diagramme représentant les classes métier se trouvant dans cette implantation est disponible. Notre outil a permis de détecter pratiquement tous les mots architecturaux (couverture de 83%). Cependant, la précision de nos règles pour cet aspect n'a été que de 45%. L'approche a identifié de faux mots architecturaux à cause de la règle R3. La raison de ceci est que toutes les classes reliées à un faux mot architectural font partie du groupe d'un vrai mot architectural; c'est pourquoi la règle R3 se trouve satisfaite. Il suffit donc d'ajouter une condition supplémentaire faisant cette vérification pour que le problème se corrige.

	Nbr. de mots architecturaux	Précision en %	Couverture en %	Nbr. de classes métier	Précision en %	Couverture en %	Nbr. d'opérations métier	Précision en %	Couverture en %	Nbr. d'attributs métier	Précision en %	Couverture en %	Nbr. d'héritages	Précision en %	Couverture en %	Nbr. d'associations	Précision en %	Couverture en %
Test1	3	100	100	7	100	100	1	100	100	9	100	100	4	100	100	2	100	100
Generated	6	45	83	4	50	100	0	0	100	26	80	92	0	100	100	3	0	0
PetStore	*18	17	22	*45	20	60	*72	74	69	*177	91	91	*4	0	0	*7	0	0
OFBS	*22	32	27	*33	41	73	*53	100	98	*62	97	97	*2	0	0	*0	0	100
VSM	*12	100	58	*17	22	82	*19	*23	16	*29	100	93	*3	0	0	*5	14	20

** estimé*

TAB. 2 – *Résultats*

Notre outil a trouvé toutes les classes métier (couverture de 100%); quant à la précision, elle a été de 50%. Un seul mot architectural non identifié a été responsable de la majorité des erreurs dans l'identification des classes métier. En effet, le système utilise des noms de classes qui finissent par le mot `Model`. Or, nos règles n'ont pas permis de l'identifier comme faisant partie de l'architecture. En outre, le système contient des classes utilitaires que l'outil a considérées comme des classes métier.

Pour les opérations métier, il n'y en a pas, car le code original ne fait que représenter le modèle dans l'architecture EJB. Les deux opérations qui ont été trouvées viennent des faux mots architecturaux détectés. L'évaluation de la détection des relations d'héritage et d'association est plus complexe. Une des classes métier, ayant 2 des 3 associations, n'a pas été détectée. Dans le tableau, nous avons été sévères : les relations détectées bien que bonnes avec les classes détectées ne représentent pas ce que nous cherchions. Donc précision et couverture de 0%. Aucun héritage n'était recherché.

4.2.2 Java Pet Store

Java Pet Store est un exemple classique, fourni par Sun, illustrant l'application de la technologie J2EE (Singh *et al.* 2002). Notre approche donne une précision et une couverture de respectivement 17% et 22% pour l'identification des mots architecturaux. La raison principale de ceci est le fait qu'un mécanisme architectural représentant le mot architectural n'est pas toujours implanté de façon uniforme partout dans le code. Ainsi, les classes implantant ce mécanisme ne possèdent pas les mêmes méthodes principales (voir la règle R3). Nous avons effectué un autre essai pour voir l'impact de donner un seuil de tolérance à la règle. L'essai démontre qu'à 60% (de méthodes au lieu de regarder si toutes les classes ont les mêmes méthodes principales; voir la condition (3) de la règle R3) nous identifions le maximum des mots architecturaux.

Pour la détection des classes métier, l'outil donne une précision et une couverture respectivement 20% et 60%. Comme l'identification des mots architecturaux a une incidence sur

l'extraction des classes métier, augmenter les résultats sur le premier contribuera sûrement à augmenter les résultats du second. Au total, 27 classes métier ont été correctement détectées. Comme il y a plusieurs mots architecturaux et classes métier détectées faussement et plusieurs autres qui ne le sont pas, il est difficile de mesurer avec fiabilité la détection des attributs et des méthodes. La méthode choisie a consisté – pour chacune des classes métier estimées et effectivement détectées – à déterminer leurs attributs et méthodes. Les résultats pour la précision et la couverture des attributs et des classes métier détectées sont relativement bons, car en examinant les classes métier nous constatons que la majorité de ces éléments se retrouvent dans la classe de base (celle qui n'implante pas l'architecture) et qui est celle généralement détectée. Les résultats sont un peu moins bons pour les méthodes, car nous constatons que les méthodes métier sont un peu plus souvent réparties dans les classes implantant un mot architectural et aussi qu'une méthode en lien avec un attribut métier non détecté ne va pas être conservée. Une fois les mots architecturaux correctement détectés, il est évident que la précision augmentera.

En ce qui concerne les associations et les héritages, nous avons décidé d'y aller sévèrement comme avec le système précédent. Nous avons comparé directement ce qui a été trouvé avec ce que nous estimions devoir y être. Pour ce faire, nous nous sommes basés sur les classes métier estimées. Les résultats ne sont pas très convaincants. Il y a beaucoup trop d'associations détectées, car il y a trop de classes métier trouvées, ce qui implique la possibilité de retrouver plus de liens entre elles. Les héritages, quant à eux, sont aussi mal détectés, car il s'agit d'héritages entre classes métier non détectées pour au moins l'une des deux. Encore ici, une meilleure détection des classes métier augmenterait cette performance.

4.2.3 OTN Financial Brokerage Service (OFBS)

OFBS est un logiciel qui simule un service de courtage en ligne (Oracle 2006a). Notre approche donne une précision (respectivement couverture) de 32% (respectivement 27%) dans l'identification des mots architecturaux. Pour les classes métier, elle donne une précision (respectivement couverture) de 41% (respectivement 73%).

Les explications données pour le système précédent au sujet des résultats obtenus lors de l'identification des mots architecturaux restent aussi valables pour cet exemple. La seule différence est que cette identification n'a pas eu beaucoup d'incidences sur les résultats de l'identification des classes métier. Les remarques pour les attributs, les méthodes, les héritages et les relations sont aussi semblables.

4.2.4 Virtual Shopping Mall (VSM)

VSM est une application implantant un centre commercial virtuel (Oracle 2006b). L'analyse de cette application a été facilitée par le fait qu'Oracle fournit sur son site un diagramme de classes. Toutefois, nous avons remarqué que l'implantation ne représente pas tout à fait le modèle et que donc notre approche isole bien entendu ce qui a été implanté.

Notre approche donne une précision (respectivement couverture) de 100% (respectivement 58%) dans l'identification des mots architecturaux. Il y a deux catégories de mots architecturaux qui ne sont pas identifiées. La raison de la non-identification de la première catégorie est la même dont nous avons déjà parlé, à savoir la non-uniformisation de l'implantation de ces mécanismes. Pour les mots architecturaux de la deuxième catégorie, la raison est ailleurs. En effet, nous avons trouvé que chacun de ces mécanismes a été implanté

seulement une fois (dans une seule classe). Or, nos règles supposent qu’un mot architectural doit se retrouver au moins en deux endroits.

Pour les classes métier, l’outil donne une précision de 22% et une couverture de 82%. Les classes utilitaires (le système en contient plusieurs) sont responsables en grande partie de la baisse de précision. Les remarques pour les attributs, les méthodes, les héritages et les relations sont aussi semblables aux deux systèmes précédents.

4.3 Discussion et travaux futurs

4.3.1 Extraction d'associations

Nous avons déjà mentionné que notre travail fait seulement une analyse simple pour extraire des associations entre les classes métier. Sur cet aspect, plus d'information peut être extraite du code d'une application, telle que la multiplicité d'une association. Nous pourrions également explorer d'autres genres de relations comme l’agrégation et la composition. Nous n'avons pas donné beaucoup d'attention à cet enjeu. Notre approche, qui est basée sur les identificateurs n'est pas suffisante pour répondre à cet enjeu. En fait, comme l’indique (Kollman *et al.* 2002), les informations précises au sujet de la multiplicité d'une association, par exemple, exigent idéalement l'utilisation des techniques d’analyse dynamique.

4.3.2 La flexibilité dans l'identification de patrons

Une autre limitation de notre approche vient de la rigidité de l'algorithme utilisé pour extraire les identificateurs (première étape). Même si elles sont bien connues, les normes établies pour les identificateurs ne sont pas toujours respectées et elles peuvent différer selon les systèmes. Ici aussi, nous ne nous sommes pas concentrés sur cette problématique. Plusieurs approches l'ont déjà étudié. Par exemple, le travail proposé par Anquetil et Lethbridge (Anquetil et Lethbridge 1999), discuté dans la section 3, pourrait être appliqué avant notre approche si nous voulons supporter des applications qui ne suivent pas un bon style de programmation.

4.3.3 Sémantique des patrons

Les patrons n'ont pas toujours la même fonction. Les patrons des identificateurs peuvent représenter plusieurs choses comme : une action, un nom ou un état. Parfois la fonction d’un patron donné change d'un identificateur à l'autre. Il est difficile d’analyser cet aspect. Cependant, il peut nous aider à différencier une composante architecturale d’une métier. Par exemple, dans le système Eclipse, le mot `classifies` représente un mot architectural dans un contexte, tandis que dans un autre, il représente le nom d'une classe métier. D'ailleurs, il y a quelques noms de classe métier détectés qui emploient (ou contiennent) un verbe, ce qui est inusuel. Par exemple dans le système NetBeans de Sun, nous détectons des noms de classes métier comme `Add` et `Change`. Probablement que ces classes sont de faux positif.

Quelques mots architecturaux sont seulement le même patron écrit différemment, par exemple, `adapter` et `adaptor`. Quelques patrons dérivent du même mot, par exemple : `Export` et `Exported`. Par conséquent, si nous pouvions discerner le type et l’utilité des patrons, il serait possible d’améliorer notre approche. Ce n'est pas une tâche facile. Nous étudions

actuellement le travail proposé par Caprile et Tonella (Caprile et Tonella 1999), qui fournit une approche pour découvrir des synonymes parmi des marques.

4.3.4 Homonymes

Les classes ayant le même nom sont fréquentes dans certains systèmes. Plusieurs classes peuvent avoir le même nom, mais ne pas représenter la même classe métier. Dans ce cas, elles ne sont pas liées aux mêmes classes. Une approche complémentaire est nécessaire pour différencier ces classes et leurs classes reliées.

4.3.5 Présence des mots architecturaux multiples

Les mots architecturaux contenus dans les classes de base peuvent indiquer quelques cas intéressants. Fréquemment, il y a des noms de classe contenant plus de deux mots architecturaux. Il y a aussi des noms de classes composés seulement des mots architecturaux (dans le système Eclipse, 757 classes disparaissent à cause de cela). Ici encore, une approche supplémentaire est nécessaire afin d'aborder cette question.

5 Conclusion

Dans cet article, nous présentons une nouvelle approche pour isoler un modèle de logique métier du code source d'une application en y retirant le code infrastructure connexe. L'extraction est rendue possible par l'analyse de la manière dont de nouvelles caractéristiques d'architecture sont mises en application dans des applications orientées objets modernes. Le modèle de logique métier est donné dans un diagramme de classes d'UML montrant ses classes métier et leurs interrelations. L'approche a été validée sur plusieurs systèmes de différentes tailles et donne de bons résultats pour ceux ayant une architecture implantée uniformément. Étant basée sur les identificateurs dans le code source, notre approche supporte mieux les systèmes développés d'après de bonnes règles d'appellation. Le processus de validation nous a permis d'identifier différentes avenues pour améliorer notre approche.

Références

N. Anquetil et Lethbridge, T. C. (1999). Recovering software architecture from the names of source files. *Journal of Software Maintenance: Research and Practice*, 11(3): 201-221.

L. A. Barowski et Cross Ii, J. H. (2002). Extraction and Use of Class Dependency Information for Java. dans *Proceedings of the Ninth Working Conference on Reverse Engineering*. Richmond, VA, USA, IEEE Computer Society: pp. 309- 315.

B. Caprile et Tonella, P. (1999). Nomen Est Omen: Analyzing the Language of Function Identifiers. dans *Proceedings of the Sixth Working Conference on Reverse Engineering*. Atlanta, GA, USA, IEEE Computer Society Press: pp. 112-122.

S. Ducasse, Rieger, M. et Demeyer, S. (1999). A Language Independent Approach for Detecting Duplicated Code. dans *Proceedings of the IEEE International Conference on Software Maintenance*. Oxford, UK, IEEE Computer Society Press: pp. 109-118.

T. EJD. 2006. «SourceCafe». EJD Technologies En ligne. <http://www.sourcecafe.com> Consulté le 15 août 2007.

M. Gogolla et Kollman, R. (2000). Re-documentation of Java with UML class diagrams. dans *Proceedings of 7th Reengineering Forum, Reengineering Week*. Zurich, Switzerland: pp. 41-48.

T. J. Grose, Doney, G. C. et Brodsky, S. A. (2002). *Mastering XMI: Java Programming with XMI, XML, and UML*. Wiley

Y. Guéhéneuc et Albin-Amiot, H. (2004). Recovering binary class relationships: Putting icing on the UML cake. *SIGPLAN Not*, 39(10): 301-314.

A. Hamou-Lhadj, Braun, E., Amyot, D. et Lethbridge, T. C. (2005). Recovering Behavioral Design Models from Execution Traces. dans *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*. Manchester, UK, IEEE Computer Society Press: pp. 112-121.

D. Heuzeroth, Holl, T., Högström, G. et Löwe, W. (2003). Automatic Design Pattern Detection. dans *Proceedings of the 11th IEEE International Workshop on Program Comprehension*. Portland, OR, USA, IEEE Computer Society: pp. 94-103.

M. Hung et Zou, Y. (2005). Extracting Business Processes from Three-Tier Architecture Systems. dans *Proceedings of International Workshop on Reverse Engineering To Requirements*. Pittsburgh, PA, USA: pp. 24-28.

D. Jackson et Waingold, A. (2001). Lightweight extraction of object models from bytecode. *IEEE Trans. Softw Eng.*, 27(2): 156-169.

M. Keschenau (2004). Reverse engineering of UML specifications from java programs. dans *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. Vancouver, British Columbia, Canada, ACM Press: pp. 326 - 327

J. Kim Jung et Benner Kevin, M. (1996). Implementation patterns for the observer pattern. *dans Pattern languages of program design*. Addison-Wesley Longman Publishing Co. Inc.: 75-86.

S. Kim, Pan, K. et Whitehead, J. E. J. (2006). Micro pattern evolution. dans *Proceedings of the 2006 international workshop on Mining software repositories*. Shanghai, China, ACM Press: pp. 40-46

R. Kollman, Selonen, P., Stroulia, E., Systä, T. et Zundorf, A. (2002). A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering. dans *Proceedings of the Ninth Working Conference on Reverse Engineering*. Richmond, VA, USA, IEEE Computer Society Press: pp. 22-32.

R. Kollmann et Gogolla, M. (2001). Application of UML Associations and Their Adornments in Design Recovery. dans *Proceedings of the Eighth Working Conference on Reverse Engineering*. Stuttgart, Germany, IEEE Computer Society: pp. 81–90.

C. Kramer et Prechelt, L. (1996). Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. dans *Proceedings of the Third Working Conference on Reverse Engineering*. Monterey, CA, USA, IEEE Computer Society Press: pp. 208-215.

Oracle. 2006a. «OTN Financial Brokerage Service 10g». Oracle. En ligne. <www.oracle.com/technology/sample_code/tech/java/j2ee/fbs10g/index.html> Consulté le 15 août 2007.

Oracle. 2006b. «Virtual Shopping Mall 1.3 ». Oracle. En ligne. <http://www.oracle.com/technology/sample_code/tutorials/vsm1.3/over/design.htm> Consulté le 15 août 2007.

I. Singh, Stearns, B. et Johnson, M. (2002). *Designing enterprise applications with the J2EE platform*. Addison-Wesley Longman Publishing Co., Inc.

A. Sutton et Maletic, J. I. (2007). Recovering UML class models from C++: A detailed explanation. *Information and Software Technology*, 49(3): 212-229.

Summary

Companies face huge costs to support their software applications. These applications contain important corporate knowledge, which becomes difficult to recover after several years of operation and evolution. Several approaches have been proposed for extracting source code abstractions in order to help developers recover this knowledge. However, abstractions extracted by most approaches combine the business logic of an application and its architecture (or its infrastructure). In this paper, we propose a new approach to extract the business logic model of an object-oriented application. This model is given as a UML class diagram showing the business classes of the application and their interrelations. This approach was validated on several systems written in Java and gives good results for well-structured systems with good programming style.