Schéma de refactoring de diagrammes de classes basé sur la notion de délégation

Boulbaba Ben Ammar, Mohamed Tahar Bhiri, Jeanine Souquières

To cite this version:

Boulbaba Ben Ammar, Mohamed Tahar Bhiri, Jeanine Souquières. Schéma de refactoring de diagrammes de classes basé sur la notion de délégation. ERTSI (INFORSID 2008), May 2008, Fontainebleau, France. 12 p. hal-00310955

HAL Id: hal-00310955

https://hal.archives-ouvertes.fr/hal-00310955

Submitted on 12 Aug 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Schéma de refactoring de diagrammes de classes basé sur la notion de délégation

Boulbaba Ben Ammar†‡, Mohamed Tahar Bhiri‡et Jeanine Souquières†

†LORIA - Nancy University
615 rue du Jardin Botanique
F-54602 Villers-lès-Nancy, France
‡MIRACL Laboratory - Sfax University
B. P. 802 - 3018 Sfax, Tunisia

{Boulbaba.Ben-Ammar, Jeanine.Souquieres}@loria.fr
tahar_bhiri@yahoo.fr

RÉSUMÉ. *L'activité de refactoring consiste à restructurer un modèle en vue d'améliorer certains facteurs de qualité, tout en préservant la cohérence de ce modèle. Dans cet article, nous proposons un schéma de refactoring de diagrammes de classes basé sur la notion de délégation. L'idée consiste à redistribuer le contenu d'une classe d'un diagramme de classes par déplacement dans une nouvelle classe d'un ensemble d'attributs et de méthodes associées à un concept, au sens type abstrait de données. La vérification de la cohérence est à la fois interne au diagramme de classes et entre les différents diagrammes du modèle en cours de développement. Nous illustrons notre propos sur une étude de cas simplifiée, une application bancaire.*

ABSTRACT. *The refactoring activity consists in restructuring a model in order to improve its quality, preserving the consistency of this model. In this paper, we propose a refactoring pattern based on the delegation concept. The main idea is to redistribute the contents of classe of a class diagram by moving a set of attributes and methods associated to a given concept in a new class, in the abstract data type meaning. The verification of the consistency of the refactoring concerns both the internal consistency of the class diagram as well as the consistency between the different existing diagrams. We illustrate our purposes on simplified case study, a banking application.*

MOTS-CLÉS : *refactoring, restructuration, délégation, diagramme de classes, vérification, cohérence*

KEYWORDS: *refactoring, restructuration, delegation, class diagram, verification, consistency*

1. Introduction

Les activités de conception logicielle ne se limitent pas à la création de nouvelles applications ex nihilo. D'une part, le concepteur est souvent amené à modifier et faire évoluer une application existante. D'autre part, une application ne se construit pas en une seule étape, et un développement en cours nécessite des réorganisations et des remises en cause. L'activité de refactoring ou de restructuration est bien connue en génie logiciel. Elle vise à améliorer certains facteurs de qualité, dont la lisibilité, la compréhension, l'adaptabilité, l'efficacité du modèle. Elle peut aussi être utilisée dans un soucis de simplification. Cett activité a tout d'abord porté sur la restructuration de code par modification de sa structure interne sans changement de son comportement externe (Opdyke, 1992). Un catalogue de refactoring de code Java a été proposé par Fowler (Fowler *et al.,* 1999), basé sur des transformations de base. Plus récemment, la définition de stratégies a conduit à la définition de patrons de refactoring, correspondant à un enchaînement de transformations de base (Kerievsky, 2004).

Dans l'ingénierie logicielle dirigée par les modèles, les techniques de refactoring sont peu nombreuses (Allem *et al.,* 2007). (Hacene *et al.,* 2007) propose une analyse formelle des données relationnelles pour restructurer des modèles UML. Cette analyse formelle s'effectue sur les méta-modèles UML à l'aide de treillis permettant de dériver une hiérarchie d'abstractions à partir d'un ensemble d'entités. Markovic et Baar (Marković *et al.,* 2008) proposent quelques règles de refactoring de base pour des diagrammes de classes en tenant compte des contraintes OCL (OMG, 2005a), inspirées des règles de refactoring proposées dans les langages orientés objet (Refactoring Community, 2005), (Mens *et al.,* 2004).

Dans cet article, nous proposons un schéma de refactoring de diagrammes de classes basé sur la notion de délégation (OMG, 2005b), permettant de restructurer et de faire évoluer un diagramme de classes tout en préservant sa cohérence et son comportement. L'idée consiste à redistribuer le contenu d'une classe d'un diagramme de classes par déplacement dans une nouvelle classe d'un ensemble d'attributs et de méthodes associées à un concept, au sens type abstrait de données. Un type abstrait consiste à abstraire des objets ayant des propriétés communes. Les données manipulées peuvent être caractérisées par un invariant. Celui-ci s'exprime en UML par des contraintes OCL. Le schéma de refactoring est complété par des règles de restructuration des contraintes OCL. La vérification de la cohérence est à la fois interne au diagramme de classes et intra-modèles.

L'article est constitué comme suit. La section 2 décrit le schéma de refactoring proposé avec les différentes étapes du processus. Dans la section 3, nous illustrons l'application du schéma de refactoring sur une étude de cas simplifiée, celle d'une application bancaire. En section 4, nous abordons le problème de la cohérence intra-modèles liée au cadre de spécification utilisé, celui de UML. La section 5 décrit une préservation possible du comportement du modèle, par composition d'opérations de refactoring de base. La section 6 conclut et propose une discussion sur les schémas de refactoring et les travaux futurs.

2. Schéma de refactoring d'un diagramme de classes

La construction de diagrammes de classes se fait pas à pas et conduit, dans bon nombre de cas au développement de classes complexes. Une classe dite complexe englobe un ensemble de caractéristiques statiques et dynamiques, certaines relevant du concept principal et d'autres relevant de concepts encapsulés. Ce manque de structuration pose des problèmes pour l'évolution et la réutilisation de tels diagrammes. C'est le cas notamment lorsque des spécialisations sont nécessaires ou lors de l'introduction de nouveaux composants via la relation d'héritage.

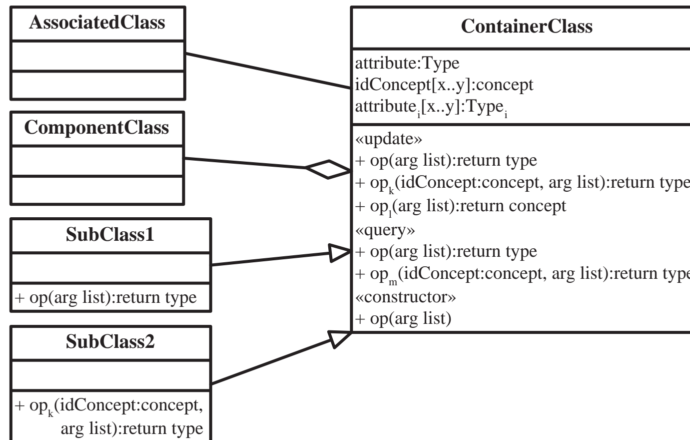

Figure 1. *Diagramme de classes initial*

Un concept est vu comme un type abstrait de données permettant de regrouper un ensemble de propriétés. Il est manipulé à l'aide d'un certain nombre d'opérations, les constructeurs et les opérations d'interrogation. Dans l'approche objet, un type abstrait est modélisé par une classe (Meyer, 2000). Nous proposons d'effectuer une partition entre les différents concepts encapsulés dans la classe identifiée comme classe complexe. Chaque concept indépendant sera extrait de cette classe complexe et modélisé dans une nouvelle classe reliée par une relation de composition avec la classe complexe.

Soit un diagramme de classes, voir Figure 1, constitué de la manière suivante :

– `ContainerClass` représente la classe complexe qui encapsule différents concepts,

– `AssociatedClass` représente une classe liée à `ContainerClass` par une relation d'association,

– `ConponentClass` représente une classe composante de `ContainerClass`,

– `SubClass1` représente une sous-classe de `ContainerClass` incluant des propriétés dynamiques,

– `SubClass2` représente une sous-classe de `ContainerClass` redéfinissant des propriétés dynamiques.

Le schéma de refactoring proposé est basé sur la notion de délégation, utilisation particulière de la communication entre objets. L'idée consiste à redistribuer le contenu d'une classe d'un diagramme de classes par déplacement dans une nouvelle classe d'un ensemble d'attributs et de méthodes associées à un concept, au sens type abstrait de données. Le processus associé s'effectue en cinq étapes :

1) Identification des propriétés à extraire à partir de la classe `ContainerClass` d'un diagramme de classes donné.

2) Création d'une nouvelle classe dans laquelle les propriétés identifiées dans l'étape précédente seront déplacées.

3) Intégration de cette nouvelle classe dans le diagramme de classes de départ.

4) Mise à jour de la classe `ContainerClass` par suppression des propriétés qui ont été intégrées dans la nouvelle classe.

5) Évolution de l'invariant du modèle prenant en compte la refactorisation.

Etape 1 : Identification des propriétés à extraire

Cette étape consiste à identifier les propriétés statiques et dynamiques du concept à extraire de la classe complexe `ContainerClass`. Elle est guidée par la recherche de l'identificateur du concept, par exemple un mot clé, l'examen de la multiplicité des différents attributs de la classe et comparaison avec celle de l'identificateur du concept.

Aspects statiques. Les attributs relatifs au concept à extraire ont le profil suivant :

– `idConcept[x..y] : concept`. Cet attribut dénote l'attribut représentant l'identificateur du concept à extraire. Sa multiplicité `[x..y]` est utilisée comme guide pour l'identification des autres attributs reliés au concept à extraire

– $\texttt{attribute}_i$`[x..y] :` $\texttt{type}_i$ avec $i \in 1..n$

Aspects dynamiques. Les méthodes relatives au concept à extraire sont de type modification et consultation. Leur identification est guidée par la présence de l'identificateur du concept en paramètre de ces méthodes. Les méthodes identifiées sont de la forme :

```
«update» + op_k(idConcept : concept, arg list) : return type
«update» + op_l(arg list) : return concept
«query» + op_m(idConcept : concept, arg list) : return type
```

Etape 2 : Création d'une nouvelle classe

Une nouvelle classe, appelée `ExtractedClass`, modélisant le concept à extraire est créée. Cette nouvelle classe hérite, avec modification, de l'ensemble des propriétés identifiées dans l'étape précédente.

Aspects statiques. Ses attributs proviennent de la classe `ContainerClass`, leur type est inchangé et leur multiplicité est égale à `1`, la multiplicité des attributs identifiés sera prise en compte dans la relation entre classes (voir Etape 3).

– idConcept : concept

– $\texttt{attribute}_i$: $\texttt{type}_i$

Aspects dynamiques. Ils sont décrits à partir des méthodes de modification et d'interrogation identifiées dans l'étape précédente dans la classe ContainerClass. Ils donnent naissance à trois types de méthodes :

– *Consultation*. Le profil de ces méthodes est obtenu par recopie du profil des méthodes d'interrogation avec suppression de l'identificateur du concept, idConcept, de la liste des paramètres. Leur profil est de la forme :
«query» + $\texttt{op}_m$(arg list) : return type

– *Modification*. Le profil de ces méthodes est obtenu par recopie du profil des méthodes de modification ayant l'identificateur du concept dans la liste des paramètres, et suppression de cet identificateur :
«update» + $\texttt{op}_k$(arg list) : return type

– *Création*. Le profil de ces méthodes est obtenu par recopie du profil des méthodes de modification ayant l'identificateur du concept comme paramètre résultat dans leur signature, en remplaçant ce paramètre résultat par un paramètre d'entrée :
«constructor» + $\texttt{op}_l$(arg list, idConcept :concept)

Etape 3 : Intégration de la nouvelle classe dans le diagramme de classes

Les différentes relations entre la nouvelle classe ExtractedClass et le diagramme de classes point de départ de la restructuration sont établies, voir Figure 2 :

– Une relation de composition est établie entre ExtractedClass et ContainerClass avec ExtractedClass comme classe composante et ContainerClass comme classe agrégat. Le sens de navigation va de la classe ContainerClass vers la classe ExtractedClass. La cardinalité est égale à x..y, celle de la multiplicité du concept dans la spécification initiale, au niveau de la classe ExtractedClass.

– Les classes AssociatedClass, ComponentClass et SubClass1 n'ont pas de relations explicites avec ExtractedClass.

– SubClass2 redéfinit la méthode $\texttt{op}_k$ de la classe ContainerClass. Cette méthode a été déléguée par ContainerClass à ExtractedClass. Pour satisfaire cette délégation, les restructurations suivantes sont nécessaires :

- Création d'une nouvelle sous-classe, appelée SubExtractedClass qui redéfinit la méthode $\texttt{op}_k$ de la classe ExtractedClass.

- Introduction d'une relation d'héritage entre SubExtractedClass et ExtractedClass.

- Introduction d'une relation de composition entre SubExtractedClass et SubClass2, avec SubExtractedClass comme classe composante et SubClass2 comme classe agrégat.

Etape 4 : Mise à jour de la classe `ContainerClass`

Cette étape consiste à faire les mises à jour nécessaires de la classe `ContainerClass`, après extraction du concept :

– La suppression des aspects statiques identifiés dans la section 2, ces aspects ayant été reportés dans sa composante `ExtractedClass`.

– Le profil des méthodes déléguées reste aussi dans la classe `ContainerClass`, leur définition est reportée au niveau de la classe `ExtractedClass` (voir Etape 5).

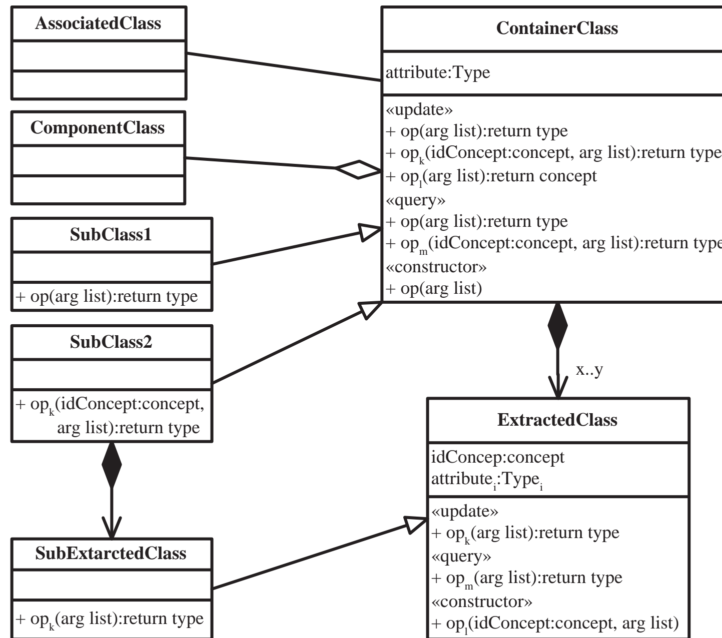

Figure 2. *Diagramme de classes restructuré*

Etape 5 : Évolution de l'invariant du modèle

Un diagramme de classes peut être accompagné d'invariants exprimés en termes de contraintes OCL. La restructuration du diagramme nécessite de faire évoluer l'invariant s'il porte sur la partie refactorisée (Marković *et al.,* 2007).

Afin d'avoir des expressions OCL cohérentes avec le nouveau diagramme, les expressions contenant des propriétés statiques ou dynamiques identifiées dans la section 2 seront modifiées de la façon suivante :

– Elles sont réécrites dans la classe `ContainerClass` en utilisant les règles de réécriture présentées dans le tableau 1.

Expressions OCL initiales	Expressions OCL restructurées
Contexte de ContainerClass	
self.attribute$_i$->forAll (attr : type$_i$ | condition$_i$(attr))	self.extractedClass->forAll (c : ExtractedClass | condition$_i$(c.attribute$_i$))
op(arg list)	op(arg list)

Tableau 1. *Règles de réécriture des contraintes dans le contexte de ContainerClass*

– Elles sont transférées vers la classe `ExtractedClass` en utilisant les règles présentées dans le tableau 2.

Expressions OCL initiales	Expressions OCL restructurées
Contexte de ContainerClass	Contexte de ExtractedClass
self.attribute$_i$->forAll (attr : type$_i$ | condition$_i$(attr))	condition$_i$(self.attribute$_i$)
op(idConcept, arg list)	op(arg list)

Tableau 2. *Règles de réécriture des contraintes dans le contexte de ExtractedClass*

3. Étude de cas : application bancaire simplifiée

Soit un système simplifié constitué de deux acteurs, un ensemble de banques et un ensemble de clients. Un client peut être client de plusieurs banques et une banque peut avoir plusieurs clients. Une banque offre un ensemble de services à ses clients : ouverture d'un compte ; retrait d'une somme sur un compte identifié ; dépôt d'une somme sur un compte identifié ; consultation du solde d'un compte identifié ; fermeture d'un compte identifié. De plus, une banque permet d'effectuer des conversions d'argent en utilisant les services offerts par un convertisseur (en euros, en dollars, ...). Il est à noter l'existence d'un type de banque particulier appelé Dépôt, qui offre un service permettant à tout instant de connaître le montant global disponible.

Une première description de ce système sous la forme d'un diagramme de classes est présentée Figure 3. Dans ce diagramme, nous remarquons que la classe Banque est une classe complexe qui encapsule des notions relatives aux banques et des notions relatives aux comptes des clients. Ce diagramme peut être restructuré à l'aide du schéma proposé dans la section 2.

Le processus de refactoring se décompose comme suit :

Etape 1. Identification des propriétés à extraire à partir de la classe `Banque`.
Aspects statiques. Un compte est identifié par `idCompte`. L'examen de la multiplicité de cet attribut et sa comparaison avec les autres attributs de la classe `Banque` guident l'identification des attributs reliés à ce concept, à savoir `solde`, `idClient`, `nom` et `adresse`
Aspects dynamiques. La présence du concept dans les méthodes de la classe `Banque`

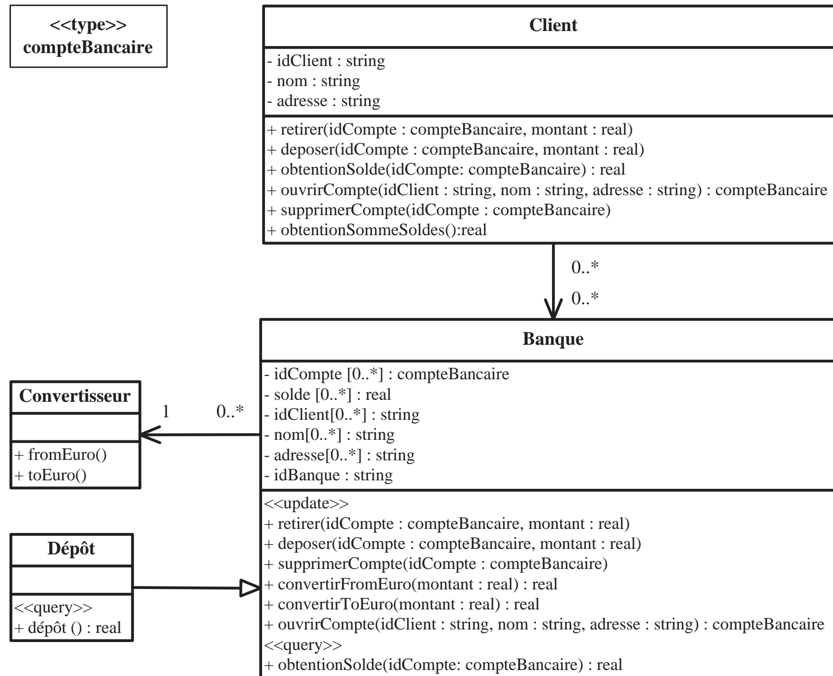

Figure 3. *Diagramme de classes pour l'application bancaire*

nous amène à identifier les méthodes `retirer`, `deposer`, `supprimerCompte`, `ouvrirCompte` et `obtentionSolde`.

Etape 2. Création d'une nouvelle classe, `Compte` modélisant le compte d'un client, avec comme aspects statiques et dynamiques les propriétés identifiées dans l'étape précédente.

Etape 3. Intégration de la classe `Compte` dans le diagramme de classes de départ, comme une composante de la classe `Banque`.

Etape 4. Mise à jour de la classe `Banque` par suppression des propriétés statiques identifiées dans l'étape 1.

L'application des quatre premières étapes du schéma de refactoring du diagramme de classe présenté Figure 3 génère le diagramme de classes présenté Figure 4.

Etape 5. Évolution de l'invariant du modèle.

OCL permet d'exprimer deux types de contraintes sur l'état d'un objet ou d'un ensemble d'objets : les invariants qui doivent être respectés en permanence et la définition en termes de pré et post-conditions des méthodes. L'application de la cinquième étape du processus de refactoring nous conduit aux évolutions suivantes.

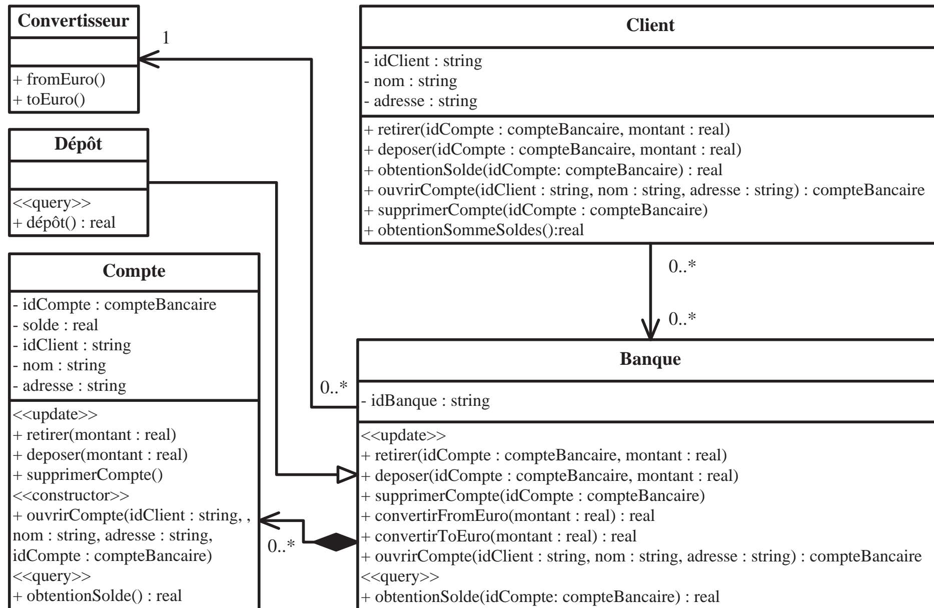

Figure 4. *Diagramme de classes de l'application bancaire restructuré*

Restructuration de l'invariant du modèle. L'expression de l'invariant *"le solde de chaque compte doit rester positif"* est associé à la classe Banque dans la modélisation initiale. Sa formalisation en OCL est la suivante :

```
Context Banque
inv : self.solde->forAll(s : real | s >= 0)
```

Après application des règles de réécriture proposées dans l'étape 5 du processus de refactoring, voir Section 2, la restructuration de cette contrainte OCL génère les deux contraintes suivantes, l'une associée à la classe Banque et l'autre à la classe Compte :

```
Context Banque
inv : self.compte->forAll(c : Compte | c.solde >= 0)
Context Compte
inv : self.solde >= 0
```

Restructuration des méthodes. La définition de la méthode retirer, avant délégation, est définie en OCL de la manière suivante :

```
Context Banque::retirer(idCompte : compteBanquaire, montant : real)
pre : montant  > 0 and montant <= obtentionSolde(idCompte)
post : obtentionSolde(idCompte) = obtentionSolde(idCompte)@pre
                                  - montant
```

D'après les règles de réécriture, la définition de cette méthode n'est pas modifiée dans le contexte de la classe Banque. Elle est transférée vers la classe Compte, avec suppression du paramètre idCompte :

```
Context Compte::retirer(montant : real)
pre : montant  > 0 and montant <= obtentionSolde()
post : obtentionSolde() = obtentionSolde()@pre - montant
```

Les expressions OCL ont été simplifiées, dû au changement de contexte d'utilisation : une collection de comptes est gérée dans le contexte Banque, alors que seul un compte (self) est concerné dans le contexte Compte.

4. Cohérence intra-modèles

Un modèle UML est généralement constitué de plusieurs types de diagrammes, un diagramme statique décrivant la structure du système, et un ou plusieurs diagrammes exprimant sa dynamique. Ces diagrammes n'étant pas indépendants, la refactorisation du diagramme de classes nécessite d'examiner la propagation des modifications engendrées afin de maintenir la cohérence globale du système en cours de développement (Simmonds *et al.,* 2004), (Mens *et al.,* 2004).

Un diagramme d'activités est une représentation graphique possible du comportement d'une méthode. Elle peut accompagner un diagramme de classes pour présenter la dynamique du système modélisé. A titre d'exemple, le diagramme d'activités de la méthode retirer, dont la définition OCL avant refactoring est donnée Section 3, est décrit Figure 5.

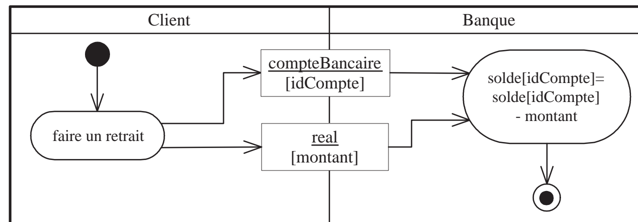

Figure 5. *Diagramme d'activités de la méthode retirer avant refactoring*

Après application du schéma de refactoring de diagrammes de classes, la méthode retirer est déléguée à un Compte. Son comportement n'est pas modifié, voir sa définition OCL Section 3 ; le diagramme d'activités correspondant est proposé Figure 6.

5. Préservation du comportement

Dans l'ingénierie logicielle, les travaux existants proposent la définition d'opérations de refactoring consistant à faire des transformations de base (Sunyé *et al.,* 2001). De telles transformations réduisent les possibilités d'erreurs de restructuration. Pour prouver la préservation du comportement de notre schéma de refactoring, nous proposons de l'exprimer par un enchaînement de ces opérations de refactoring de base :

– *Add Class* : Elle permet d'ajouter une classe composante, appelée Compte, de la classe Banque. A ce stade, la classe Compte est une classe vide, sans attribut ni méthode. Son adjonction ne modifie pas le comportement du modèle.

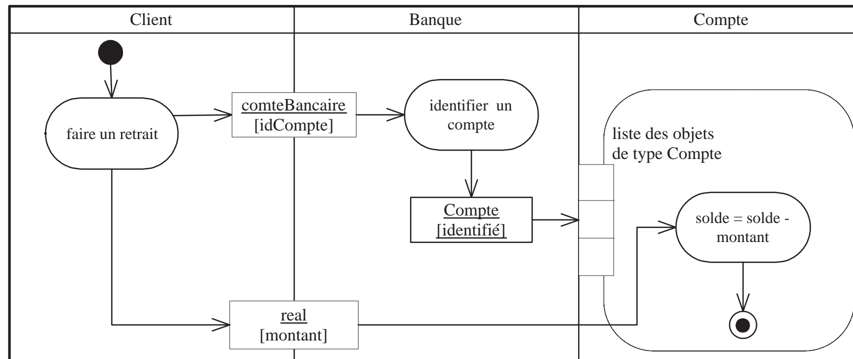

Figure 6. *Diagramme d'activités de la méthode* retirer *après refactoring*

– *Move Operation* : Nous appliquons cette opération sur chaque méthode identifiée reliée au concept à extraire. Ces méthodes sont déplacées vers la nouvelle classe Compte. L'opération *Move Operation* exige de conserver un lien dans la classe Banque, pour chaque méthode déplacée. Pour cela, nous gardons les signatures des opérations déplacées dans la classe Banque.

– *Add Attribute* : Nous appliquons cette opération sur chaque attribut identifié relié au concept à extraire. Ces attributs sont ajoutés à la classe Compte, avec une cardinalité égale à 1. Ceci est permis puisque la classe Compte a été initialement créée sans attribut.

– *Remove attribute* : Nous appliquons cette opération sur chaque attribut identifié relié au concept à extraire. Les attributs sont supprimés de la classe Banque. Ceci est permis puisque ces éléments ne sont référencés dans l'ensemble du modèle que par les méthodes déplacées vers la classe Compte.

6. Conclusion

Cet article propose un schéma de refactoring de diagrammes de classes basé sur la notion de délégation, permettant son évolution tout en préservant sa cohérence et son comportement, afin d'améliorer les critères de qualité logicielle tels que la complexité et la réutilisabilité des classes. L'idée consiste à redistribuer le contenu d'une classe d'un diagramme de classes par déplacement dans une nouvelle classe d'un ensemble d'attributs et de méthodes associées à un concept, au sens type abstrait de données. Il prend en compte les contraintes OCL attachées au diagramme de classes. Pour cela, des règles de réécriture et de transfert d'expressions OCL ont été définies. Nous proposons une préservation de la cohérence intra-modèles qui consiste à revoir tous les diagrammes associés au diagramme de classes refactorisé. Cette étape n'est pas prise en compte par le schéma de refactoring, mais elle doit être prise en compte par l'utilisateur.

L'activité de refactoring ne sera réellement efficace que si son application est supportée par des outils. Certaines étapes du processus sont automatisables. Concernant la vérification de la cohérence interne au diagramme de classes et intra-modèles, nous proposons de définir une relation entre les schémas de refactoring des modèles et la notion de raffinement dans les méthodes formelles et en particulier event-B, en vue d'utiliser les outils de preuve associés.

L'identification du concept que l'on cherche à extraire est un problème ouvert. Une piste consiste à utiliser les résultats obtenus dans le cadre du refactoring de code, avec la détection des défauts structurels, *"bad smells"*. (Astels, 2002) utilise les diagrammes UML pour les détecter.

Pour enrichir les schémas de refactoring disponibles, nous travaillons sur le problème de la fusion et la généralisation de classes.

7. Bibliographie

Allem K., Mens T., « Refactoring des modèles : concepts et défis », *Proc. IDM 2007*, Hermes Science Publications, Lavoisier, 2007.

Astels D., « Refactoring with UML », *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering (XP)*, p. 67-70, 2002. Alghero, Sardinia, Italy.

Fowler M., Beck K., Brant J., Opdyke W., Roberts D., *Refactoring : Improving the Design of Existing Code*, Addison-Wesley Professional, June, 1999.

Hacene M. R., Dao M., Huchard M., Valtchev P., « Analyse formelle de données relationnelles pour la réingénierie des modèles UML », *in* , I. Borne, , X. Crégut, , S. Ebersold, , F. Migeon (eds), *LMO*, Hermès Lavoisier, p. 151-166, 2007.

Kerievsky J., *Refactoring to Patterns (Addison-Wesley Signature Series)*, Addison-Wesley Professional, August, 2004.

Marković S., Baar T., « Synchronizing Refactored UML Class Diagrams and OCL Constraints », *1st Workshop on Refactoring Tools , ECOOP07 Conference Workshop, July 31, 2007, Berlin, Germany*, TU Berlin Technical Report, ISSN 1436-9915, p. 15-17, 2007.

Marković S., Baar T., « Refactoring OCL annotated UML class diagrams », *Software and Systems Modeling*, vol. 7, n˚ 1, p. 25-47, February, 2008.

Mens T., Tourwe T., « A Survey of Software Refactoring », *IEEE Trans. Softw. Eng.*, vol. 30, n˚ 2, p. 126-139, February, 2004.

Meyer B., *Object-Oriented Software Construction*, 2nd edn, Prentice Hall PTR, March, 2000.

OMG, *UML 2.0 OCL Specification*. Juin, 2005a.

OMG, *UML 2.0 Superstructure Specification*. August, 2005b.

Opdyke W. F., Refactoring : A Program Restructuring Aid in Designing Object-Oriented Application Frameworks, PhD thesis, University of Illinois at Urbana-Champaign, 1992.

Refactoring Community, *Refactoring home page. http ://www.refactoring.com*. 2005.

Simmonds J., Ragnhild, Jonckers V., Mens T., « Maintaining Consistency between UML Models Using Description Logic », *Série L'objet - logiciel, base de données, réseaux*, vol. 10, n˚ 2-3, p. 231-244, 2004.

Sunyé G., Pollet D., Le Traon Y., Jézéquel J.-M., « Refactoring UML Models », *Proceedings of UML 2001*, vol. 2185 of *LNCS*, Springer Verlag, p. 134-148, 2001.