Une aide à la réalisation des évolutions logicielles grâce aux modèles de traçabilité des fonctionnalités

Dolores Diaz, Lionel Seinturier, Laurence Duchien, Pascal Flament

Une aide à la réalisation des évolutions logicielles grâce aux modèles de traçabilité des fonctionnalités

Dolores Diaz[*,**] **— Lionel Seinturier** [*] **— Laurence Duchien** [*] **Pascal Flament** [**]

[*] *Laboratoire d'Informatique Fondamentale de Lille (UMR CNRS 8022)*
INRIA Futurs - Projet Jacquard - USTL Bâtiment M3
F-59655 Villeneuve d'Ascq cedex

{diaz, seinturi, duchien}@lifl.fr

[**] *NORSYS*
1, rue de la Cense des Raines
ZAC du moulin
F-59710 Ennevelin

{ddiaz, pflament}@norsys.fr

RÉSUMÉ. *Les applications développées en entreprise subissent des évolutions logicielles. La réalisation efficace de ces évolutions logicielles est devenue un enjeu crucial. Pour cela, nous exploitons dans cet article la notion de la traçabilité des fonctionnalités qui suit l'élaboration des fonctionnalités d'une application. Cette notion est capturée par un modèle. Ce papier stabilise la définition de la traçabilité des fonctionnalités en proposant une extension UML pour sa modélisation ainsi qu'une méthode de définition. De plus, nous présentons son utilisation sur une application, les apports et les limitations lors de la réalisation d'une évolution logicielle.*

ABSTRACT. *Applications developed in company undergo software evolutions. The realization of these software evolutions is consequently a crucial problem. We use in this article the notion of the functionalities traceability that follows the building of functionalities. This paper stabilizes the definition of functionalities traceability, proposes an extension UML for its modeling and a method for its definition. Moreover, we present his use on an application, its advantages and its limitations during the realization of a software evolution.*

MOTS-CLÉS : *traçabilité des fonctionnalités, évolutions logicielles.*

KEYWORDS: *functionalities traceability, software evolutions.*

1. Introduction

Quel que soit le domaine d'activité considéré (bancaire, santé, secteur public), les applications subissent des évolutions logicielles. Dès lors qu'une application est utilisée dans un environnement réel, elle subit systématiquement des modifications au gré des besoins émergeant ou modifiés des utilisateurs (Lehman, 1996). Selon la nature de ces besoins, l'effort de réalisation des évolutions implique des coûts importants en termes de temps, de ressources matérielles et humaines. C'est pourquoi, maîtriser la réalisation des évolutions logicielles pour les entreprises spécialisées en ingénierie des logiciels est un enjeu primordial.

Notre intérêt porte sur la réalisation des évolutions logicielles sur des applications à grande échelle (J2EE, .NET). Toute évolution suit un processus précis de réalisation (IEEE1219, 1998).

La société d'ingénierie des logiciels, NORSYS, cadre de nos travaux, nous a permis l'observation ainsi que l'identification des difficultés rencontrées lors de la réalisation d'une évolution logicielle. En voici quelques unes :

– la compréhension d'une application. Avant toute modification logicielle, il est fondamental de comprendre ce que fait une application et comment elle le fait. Bien que les modèles et la documentation ont pour fonction de répondre à cette nécessité, il est fréquent de constater que ces artefacts ne sont pas tenus à jour. Au final, la compréhension d'une application basée sur l'exploration de son code est laborieuse et manque de fiabilité ;

– l'identification de l'objet d'une évolution. La localisation des portions de code à modifier est rendue difficile à cause de la taille et de la complexité des applications actuelles. Bien souvent, les nouveaux développeurs perdent un temps significatif dans cette recherche ;

– l'impact des modifications. La réalisation d'une évolution logicielle implique des conséquences importantes sur le reste de l'application. Ces impacts sont fonctionnels et/ou structurels. Or, pour des raisons de temps et de coûts, la modélisation de ces répercussions n'est généralement pas définie. En conséquence, les mainteneurs perdent la vue globale de leurs modifications. Ils se perdent dans les répercussions des modifications à faire et sont par conséquent exposés à multiplier les erreurs.

Ces difficultés sont issues des étapes du processus de réalisation des évolutions logicielles. Ce processus, composé de plusieurs étapes telles que la compréhension du programme, l'application de transformations et d'autres, amène autant d'obstacles qu'il existe d'étapes. Or, pris individuellement, de nombreux travaux sur les évolutions logicielles proposent des solutions. Par exemple, certaines approches se focalisent sur la compréhension d'une application (Soloway *et al.*, 1983) ; (Rugaber, 1995) ; (Lowe *et al.*, 2002) ; (Pacione *et al.*, 2004). D'autres travaux s'intéressent aux transformations d'un programme (Pawlak, 2005) ou à celles de ses modèles (*model refactoring* (Mens *et al.*, 2005a)). Bien que ces approches préparent ou effectuent la réalisation des évolutions logicielles, elles sont malheureusement trop localisées à une étape du

processus des évolutions logicielles pour être réellement utilisables et efficaces. Notre proposition vise le pragmatisme en définissant une aide à la réalisation des évolutions logicielles afin d'améliorer non pas une étape du processus de réalisation, mais tout son ensemble.

Cet article rappelle la notion de traçabilité des fonctionnalités et l'exploite par l'intermédiaire d'un modèle UML (OMG, 2005b) afin de fournir une aide à la réalisation des évolutions logicielles. Chaque modèle de traçabilité des fonctionnalités permet d'identifier et de structurer l'ensemble des artefacts (diagrammes UML, fichiers d'implémentation), produits au cours d'un processus d'ingénierie logiciel. Plus précisément, une instance de ce modèle de traçabilité permet d'organiser et de structurer les artefacts d'une fonctionnalité en particulier. La réalisation d'une évolution technique, fonctionnelle ou corrective est alors facilitée grâce aux instances et au modèle de traçabilité des fonctionnalités.

La section 2 de cet article développe la notion de traçabilité des fonctionnalités. Cette même section introduit le métamodèle de la traçabilité des fonctionnalités dans un objectif de stabiliser cette définition. La section 3 établit un profil UML pour la définition de modèles de traçabilité des fonctionnalités. Deux projections sont fournies pour les processus logiciels RUP et XP. La section 4 présente une méthode de définition des modèles de traçabilité des fonctionnalités. La section 5 explique l'utilisation des instances de tels modèles sur des exemples. Elle met en évidence les avantages et les limites des modèles de traçabilité des fonctionnalités. La section 6 compare notre proposition aux solutions existantes en matière de traçabilité des fonctionnalités. La section 7 conclut cet article.

2. Les modèles de traçabilité des fonctionnalités

2.1. *Définition de la traçabilité des fonctionnalités*

La traçabilité des fonctionnalités a initialement été identifiée en ingénierie des besoins et porte le nom de traçabilité des besoins. De nombreuses définitions sur la traçabilité des besoins sont proposées. La plus connue est celle donnée dans le standard IEEE des spécifications des besoins (IEEE830, 1984) :

Définition 2.1 *Une spécification de besoin est traçable si (i) l'origine de chacun de ses éléments est claire et si (ii) la spécification facilite le référencement du besoin lors des phases de développement et de documentation.*

Les phases d'élaboration (phase de spécification et d'implémentation d'une application) et/ou de maintenance tentent d'exploiter cette définition sur la traçabilité des besoins. Selon la phase, la traçabilité des besoins prend des formes et des appelations différentes. Lors de la phase de l'élaboration d'un logiciel, les méthodes existantes, RUP par exemple, construisent une application sur la base de définitions itératives

et incrémentales d'un besoin exprimé par un utilisateur. La définition de chaque besoin est transformée à travers les différentes phases depuis son expression en langage naturel jusqu'à son implémentation. Cette démarche de transformation de définitions des besoins est identifiée par le nom de traçabilité des fonctionnalités (Jacobson *et al.*, 1994) et ne se limite pas, contrairement à la définition du standard IEEE, à la seule spécification d'un besoin utilisateur. De la même façon, la phase de maintenance, une phase au cours de laquelle l'enjeu de faciliter la compréhension d'une application est essentielle (plus de 50 % du temps de la maintenance est dédié à la compréhension de l'application), se base sur cette notion de traçabilité des fonctionnalités lors de la découverte des fonctionnalités d'une application.

Nous pensons que la plus-value de la notion de traçabilité des besoins est apportée en phase de maintenance et dépasse le cadre de simple spécification des besoins. En effet, dans une optique de faire évoluer les besoins des utilisateurs, la définition de leur traçabilité apporte des informations pertinentes et essentielles lors de la réalisation des évolutions logicielles. Dans la suite de cet article, nous présentons le contenu et la nature de ces informations. De plus, notre intérêt ne se limite pas aux seules évolutions des besoins des utilisateurs mais se veut plus proche de l'architecture logicielle. Pour cette raison, dorénavant, nous utiliserons l'expression de traçabilité des fonctionnalités plutôt que celle de traçabilité des besoins. Passons à la définition de la traçabilité des fonctionnalités.

Nous nous inspirons de la définition proposée par (Gotel *et al.*, 1994), sur la traçabilité des besoins pour l'adapter à celle de la traçabilité des fonctionnalités :

Définition 2.2 *La traçabilité des fonctionnalités est la capacité à décrire et à suivre la vie des fonctionnalités d'une application, selon une double direction : « bottom-up » et « top-down », autrement dit, depuis ses origines jusqu'à son déploiement et ses utilisations en passant par des étapes de spécification et de développement, définies par les phases de raffinement d'un processus logiciel.*

Partant de l'hypothèse que toutes les fonctionnalités d'une application sont élaborées de la même façon, en suivant un unique processus logiciel, nous proposons de décrire et de suivre cette phase d'élaboration des fonctionnalités par l'intermédiaire d'un diagramme de classes stéréotypées. La section suivante introduit les modèles de traçabilité des fonctionnalités.

2.2. *Définition d'un modèle de traçabilité des fonctionnalités*

Notre approche préconise la définition d'un modèle de classes UML stéréotypées pour la définition de la traçabilité des fonctionnalités. Sur la base de la définition précédente, ce modèle contient un ensemble structuré des artefacts produits à chaque étape de l'élaboration des fonctionnalités d'une application. Autrement dit, ce modèle permet de réunir et d'organiser les artefacts issus des étapes d'un processus logiciel. Seulement, selon l'étape les artefacts définissent des aspects différents

du logiciel en cours de construction. La définition de ces aspects peut prendre des formes multiples. Par conséquent, nous avons établi une différence entre une unité logicielle et l'ensemble des artefacts qui la définissent. Toute unité logicielle, quelle que soit sa nature, est à considérer sur deux niveaux. Ces deux niveaux à prendre en compte représentent respectivement le quoi et le comment de l'information apportée. Dans (Lehman, 1996), l'utilité de séparer ces deux niveaux est essentielle car l'objet à tracer peut prendre plusieurs formes, manuscrites ou graphiques. Par exemple, un modèle de spécification de besoins, selon la méthode d'élaboration logicielle choisie, peut être exprimé sous la forme d'un simple manuscrit, d'un diagramme de cas d'utilisation ou encore un graphe orienté de nœuds. La figure 1 illustre les deux niveaux que nous associons à chaque unité logicielle.

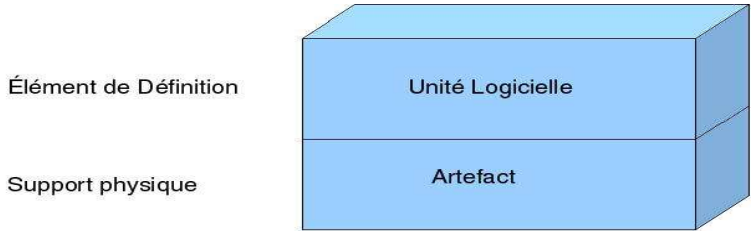

Figure 1. *Unité logicielle sur deux niveaux*

Chaque unité logicielle correspond à un niveau abstrait et les artefacts à un niveau concret. Le niveau abstrait correspond au quoi (l'unité logicielle), et précise ce que l'objet apporte comme type d'information. Le niveau concret correspond au comment (artefact) et précise la forme que prend l'information apportée.

Tout modèle de traçabilité des fonctionnalités prend en compte cette différence de niveau d'abstraction. Celle-ci est explicitement définie par le métamodèle de traçabilité des fonctionnalités, qui permet la définition de modèles de traçabilité des fonctionnalités. Celui-ci est illustré en figure 2.

Le métamodèle de la traçabilité des fonctionnalités (cf. figure 2) met en place la notion d'une unité traçable ainsi que des relations entre ces unités. La première métaclasse à considérer dans ce métamodèle est celle nommée *Object*. Elle désigne la notion d'objet traçable et permet de tracer n'importe quel objet d'une application. Nous précisons la notion d'objet traçable par la métaclasse *SoftwareEntity* qui nous permet de préciser ce qu'il faut tracer (le quoi). La définition d'une unité logicielle sur deux niveaux est réalisée par l'intermédiaire des métaclasses *SoftwareEntity*, *Artefact* et la relation *IsDocumentedBy*. Ainsi, notre métamodèle établit clairement l'association d'une unité logicielle à un ou plusieurs artefacts.

En outre, les unités logicielles peuvent être associées par trois types de relations : *Satisfy*, *DependOn* et *ComposedOf*. L'observation des mécanismes et des liaisons existants entre les artefacts d'un processus logiciel ont permis d'identifier ces trois relations. Chacune de ces relations admet une sémantique précise :

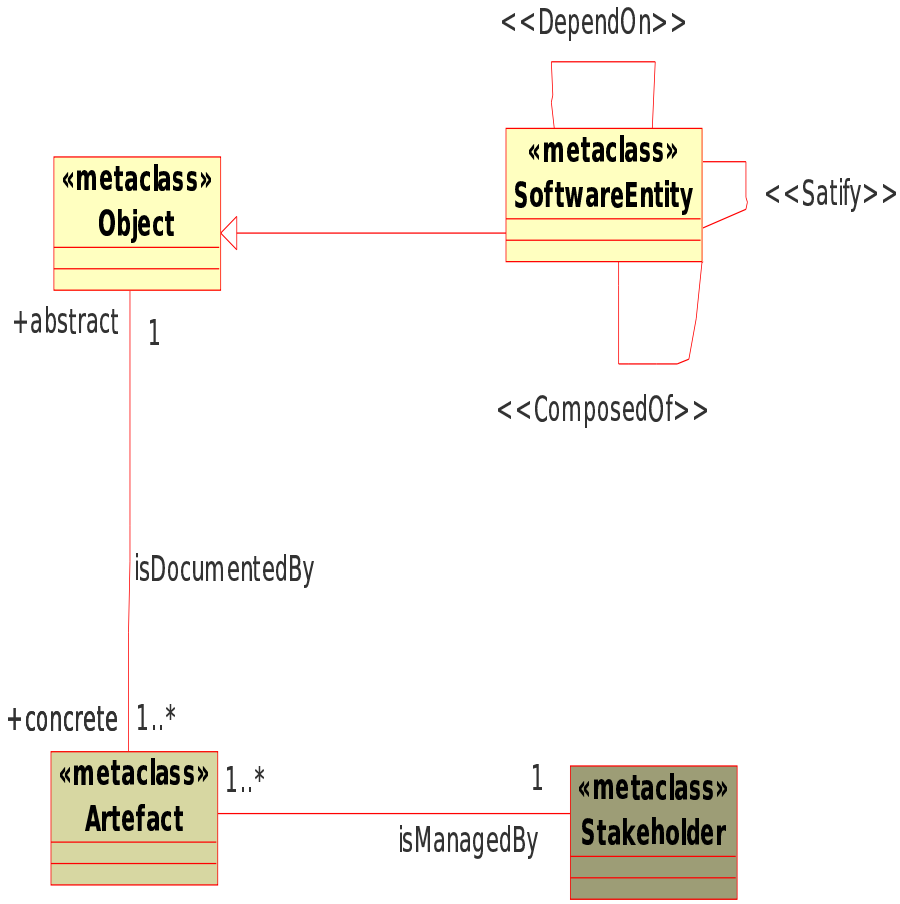

Figure 2. *Métamodèle de la traçabilité des fonctionnalités*

– la relation *Satisfy* met en relation deux ou plusieurs unités logicielles provenant d'étapes logicielles différentes et admettant une relation de « réalisation » (au sens UML). Par exemple, au cours du processus RUP, il est courant d'utiliser un diagramme de collaboration à la phase d'analyse pour réaliser un cas d'utilisation défini lors de la spécification des besoins ;

– la relation *DependOn* met en relation deux ou plusieurs unités logicielles dont les définitions s'influencent. Notre relation ressemble à la relation de dépendance définie dans UML. Ces unités logicielles sont obligatoirement issues de la même phase logicielle. Par exemple, à la phase de conception nous pouvons modéliser par une relation *DependOn*, toute relation de dépendance qui existe entre un diagramme de classes et un diagramme de séquence ;

– la relation *ComposedOf* admet la même sémantique que la relation de composition définie dans la spécification UML 1.5 (OMG, 2005b). Par exemple, nous modélisons par une relation de *ComposedOf*, toute relation existante entre un diagramme de classes et l'une de ses classes.

Enfin, nous complétons la définition d'une unité logicielle par l'ajout du rôle responsable de la définition de ses artefacts, par la métaclasse *Stakeholder* et la relation *IsManagedBy*. Le ou les auteurs des artefacts sont identifiés par ce biais.

Ainsi pour une application, nous offrons la possibilité de stabiliser une méthode d'élaboration des fonctionnalités avec un modèle de traçabilité des fonctionnalités. Ce modèle identifie, énumère et organise l'ensemble des unités logicielles et artefacts à définir lors de l'élaboration des fonctionnalités. La section suivante introduit la définition d'un profil UML permettant la définition de modèles de traçabilité des fonctionnalités.

3. Un profil UML pour la définition de modèles de traçabilité des fonctionnalités

Le profil UML que nous détaillons dans cette section permet la définition de modèles de traçabilité des fonctionnalités. Des profils pour la modélisation de processus logiciels existent déjà et sont parfaitement spécifiés (OMG, 2005a) et outillés (SOFTEAM, 2002). Or, même si ces profils permettent de réunir les notions d'artefact, de rôle et d'activité, ils ne rentrent pas suffisamment dans les détails du processus d'élaboration d'une fonctionnalité. Ils ne permettent pas la modélisation de ce qui nous semble véritablement intéressant dans un processus d'élaboration d'une fonctionnalité, la détermination et l'organisation de ses artefacts.

Nous pensons qu'un processus d'élaboration d'une fonctionnalité est plus efficace s'il définit, en fonction d'une méthode donnée, les artefacts attendus à chacune des étapes de la construction des fonctionnalités. Par là même, il réussit à cadrer l'élaboration et limite les erreurs de spécification.

Ainsi, par l'intermédiaire de notre profil, il est possible de définir un modèle de traçabilité des fonctionnalités. Ce modèle capture un mode d'élaboration des fonctionnalités d'une application, en fournissant l'organisation et la nature des unités logicielles et des artefacts attendus à chaque étape de la phase d'élaboration.

3.1. *Définition du profil*

Nous avons repris le cadre défini dans le projet ACCORD (ACCORD, 2002) pour la définition de notre profil. Celui-ci introduit les éléments suivants pour la définition d'un profil :

– la *description générale* du profil ;

– le *domaine d'utilisation* ;

– la *définition technique* du profil.

Description générale

– *Objectif :* le profil que nous établissons permet la définition de modèles de traçabilité des fonctionnalités. Chaque modèle de traçabilité des fonctionnalités est défini par un diagramme de classes stéréotypées. Ce profil reprend les notions d'unité logicielle, de relation entre les unités logicielles, d'artefact et de rôle afin de permettre la définition détaillée d'une élaboration des fonctionnalités.

– *Public visé :* ce profil est à l'attention des chefs de projets, des concepteurs, des développeurs et des mainteneurs. Son ambition est de réunir ces différents rôles autour d'un même modèle en vue d'améliorer leur communication.

Domaine

Ce profil couvre les activités de modélisation d'une application. Nous préconisons son utilisation le long de la phase d'élaboration ou bien en début de phase de maintenance.

Définition technique

– *Paquetages UML utilisés :* le profil pour la définition de modèles de traçabilité des fonctionnalités étend les paquetages UML 1.5 standard Core et Model Management. Les métaclasses suivantes du paquetage Core ont été étendues :

- artefact
- attribut
- class
- dependency
- package

– *Stéréotypes :* le tableau 1 liste les stéréotypes définis dans notre profil. Ces stéréotypes peuvent s'appliquer aux classes, aux artefacts, aux dépendances ou encore aux attributs.

– *Tagged Values :* le tableau 2 comporte la définition d'un tagged value, inclus dans notre profil.

– *Contraintes structurelles :*

1) seuls les artefacts stéréotypés ModelUML admettent un attribut stéréotypé NatureModel ;

2) s'il existe une dépendance stéréotypée dependOn depuis un élément A vers un élément B alors la dépendance inverse n'existe pas forcément.

Stéréotype	S'applique à	Définition
Software Entity	Class	Toute unité logicielle traçable.
Stakeholder	Class	Tout protagoniste responsable de la définition des artefacts liés à une unité logicielle.
FilePackage	Package	Le regroupement d'un ensemble d'artéfacts de même type.
File	Artefact	Un fichier source qui peut être compilé dans un fichier exécutable.
Executable	Artefact	Un fichier exécutable qui peut être lancé sur un ordinateur.
Library	Artefact	Un fichier librairie statique ou dynamique.
ModelUML	Artefact	Un modèle UML défini au cours de l'élaboration d'une fonctionnalité.
Source	Artefact	Un fichier physique défini au cours de l'élaboration d'une fonctionnalité.
Table	Artefact	Une table de base de données.
isDocumentedby	Dependency	La relation existante entre une unité logicielle et ses artefacts.
isManagedby	Dependency	La relation existante entre des artefacts et des rôles, responsables de leurs définitions.
satisfy	Dependency	La dépendance UML représente un raffinement d'une unité logicielle vers une autre.
dependOn	Dependency	La dépendance UML représente une dépendance entre unités logicielles.
NatureModel	Attribut	La nature d'un modèle UML. Autrement dit, il précise si le modèle effectue une modélisation statique ou dynamique.

Tableau 1. *Tableau des stéréotypes du profil de traçabilité*

– *Métamodèle :* le métamodèle de la figure 3 liste les extensions définies dans ce profil. Ce métamodèle étend celui du langage UML et intègre les notions de notre métamodèle de la traçabilité des fonctionnalités que nous avons présentées en section 2. Certaines métaclasses ne participent pas directement au profil mais aident à sa compréhension. Ainsi, les métaclasses grisées proviennent du métamodèle UML, les autres métaclasses et stéréotypes en clair sont issus de notre profil.

Tagged Value	**S'applique à**	**Définition**
NatureModel	Artefact	Indique la nature du modèle UML : statique ou dynamique.

Tableau 2. *Tableau de tagged values du profil de traçabilité*

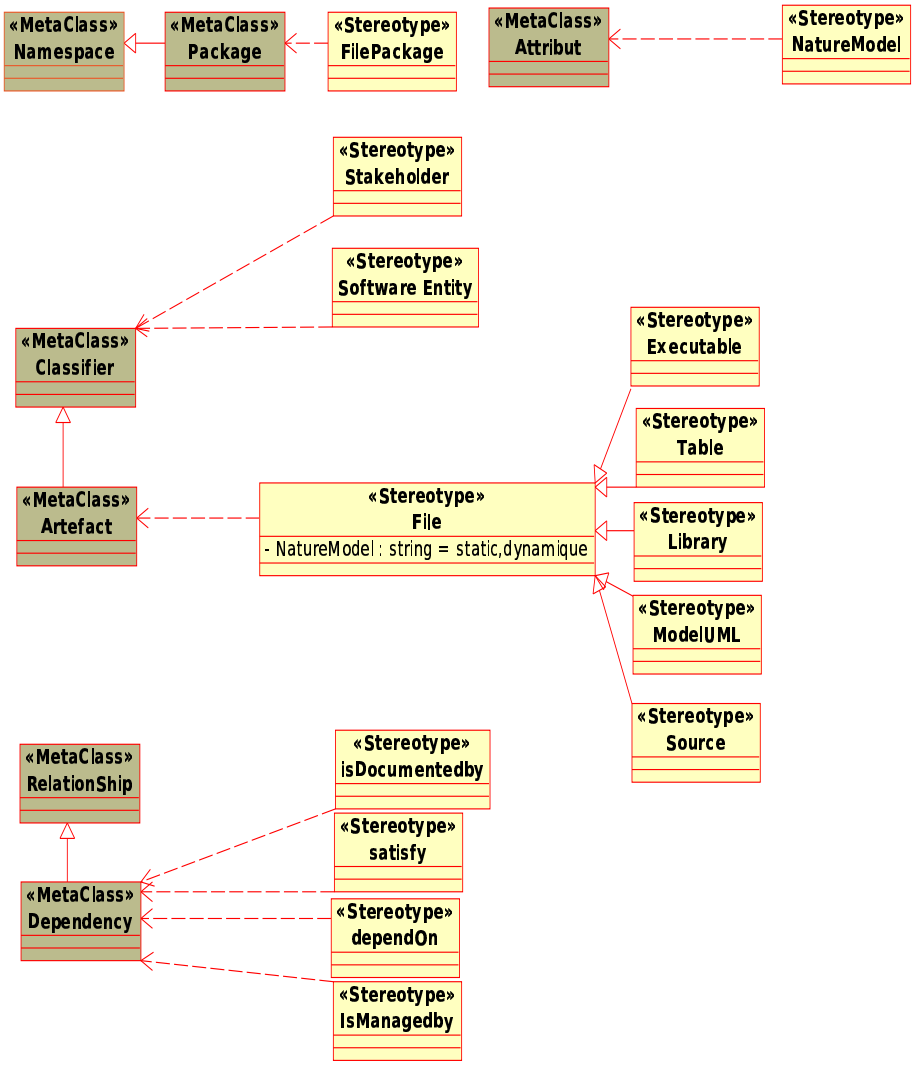

Figure 3. *Métamodèle du profil UML*

3.2. *Utilisation du profil*

Nous illustrons l'usage de notre profil avec deux exemples. Ces exemples définissent deux modèles de traçabilité des fonctionnalités et soulignent l'expressivité du profil. Chaque modèle caractérise une méthode d'élaboration : le *Rational Unified Process* (RUP) (Jacobson *et al.*, 1998) et l'*eXtreme Programming* (XP) (Beck, 1999) à l'aide des unités logicielles et des artefacts. Plus précisément, chaque modèle de traçabilité des fonctionnalités définit un mode d'élaboration des fonctionnalités et précise le type des unités logicielles attendu, ainsi que les artefacts à définir au cours des étapes.

La figure 4 définit un modèle de traçabilité des fonctionnalités selon une méthode RUP. Par manque de place, les protagonistes intervenant dans la définition des artefacts ne sont pas définis. Ce processus de développement logiciel se caractérise par quatre phases qui comprennent la définition d'un ensemble d'artefacts. Ces phases sont représentées à l'aide des entités logicielles « Requirement », « RequirementAnalysis », « RequirementDesign » et « RequirementImplementation ». Quatre groupes d'artefacts sont par conséquent attendus lors de cette phase d'élaboration. La composition des ces groupes d'artefacts peut être soit précisée par la définition d'unités logicielles supplémentaires qui auraient la fonction de venir préciser la nature des unités logicielles ou bien, cette composition peut directement intègrer la définition directe des artefacts à définir à chaque étape du processus. Sur notre exemple (cf. figure 4), la composition de l'unité logicielle nommée « RequirementAnalysis » est précisée par l'intermédiaire de l'unité logicielle « BusinessObject ». Celle-ci désigne la définition des objets métiers de l'application. L'artefact « BusinessClass » indique que cette définition prend la forme de classes UML.

La figure 5 définit un modèle de traçabilité des fonctionnalités selon une méthode XP. Ici, encore les protagonistes, responsables de la définition des artefacts ne sont pas définis. La méthode XP est agile, autrement dit elle admet une grande flexibilité en fonction de la réactivité des concepteurs et des développeurs. Définir un modèle de traçabilité des fonctionnalités peut sembler un peu contradictoire dans la mesure où la flexibité de la méthode incite l'absence de cadre méthodologique. Seulement, nous pensons que ce peut être un atout et un gage de qualité que de stabiliser cette démarche avec un modèle de traçabilité des fonctionnalités. Le cycle de vie de la méthode XP s'appuie sur l'implémentation directe d'un ensemble fini de scénarios (les plus simples possibles). Cette démarche base son atout majeur sur la réalisation rapide d'une solution non générique et se concentre sur l'aspect statique d'une solution (par la définition d'un diagramme de classes). Ajoutons que la définition d'une batterie de tests vient valider ou non l'implémentation proposée pour les fonctionnalités.

De la même façon que pour le modèle de traçabilité des fonctionnalités selon un processus RUP, ce modèle identifie les entités logicielles qui participent à la définition des fonctionnalités et complètent ces dernières par la définition des artefacts adéquats.

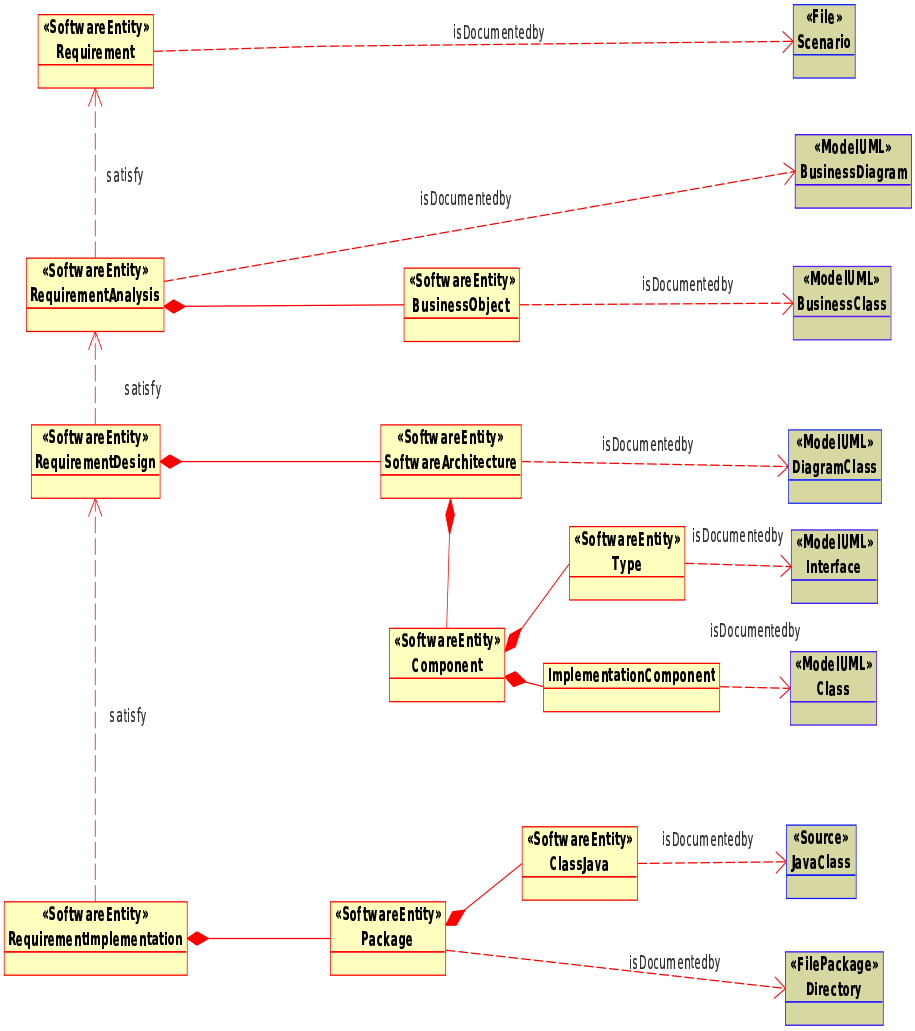

Figure 4. *Modèle RUP de traçabilité des fonctionnalités*

3.3. *Conclusion sur le profil*

Le profil que nous venons d'introduire permet la définition de modèles de traçabilité des fonctionnalités. Chaque modèle énumère les unités logicielles et les artefacts attendus pour la définition des fonctionnalités, selon une méthode d'élaboration donnée. En plus d'apporter une vue détaillée sur la modélisation générale d'un processus logiciel, la définition d'un modèle de traçabilité des fonctionnalités nous sert de base pour la compréhension de l'élaboration des fonctionnalités.

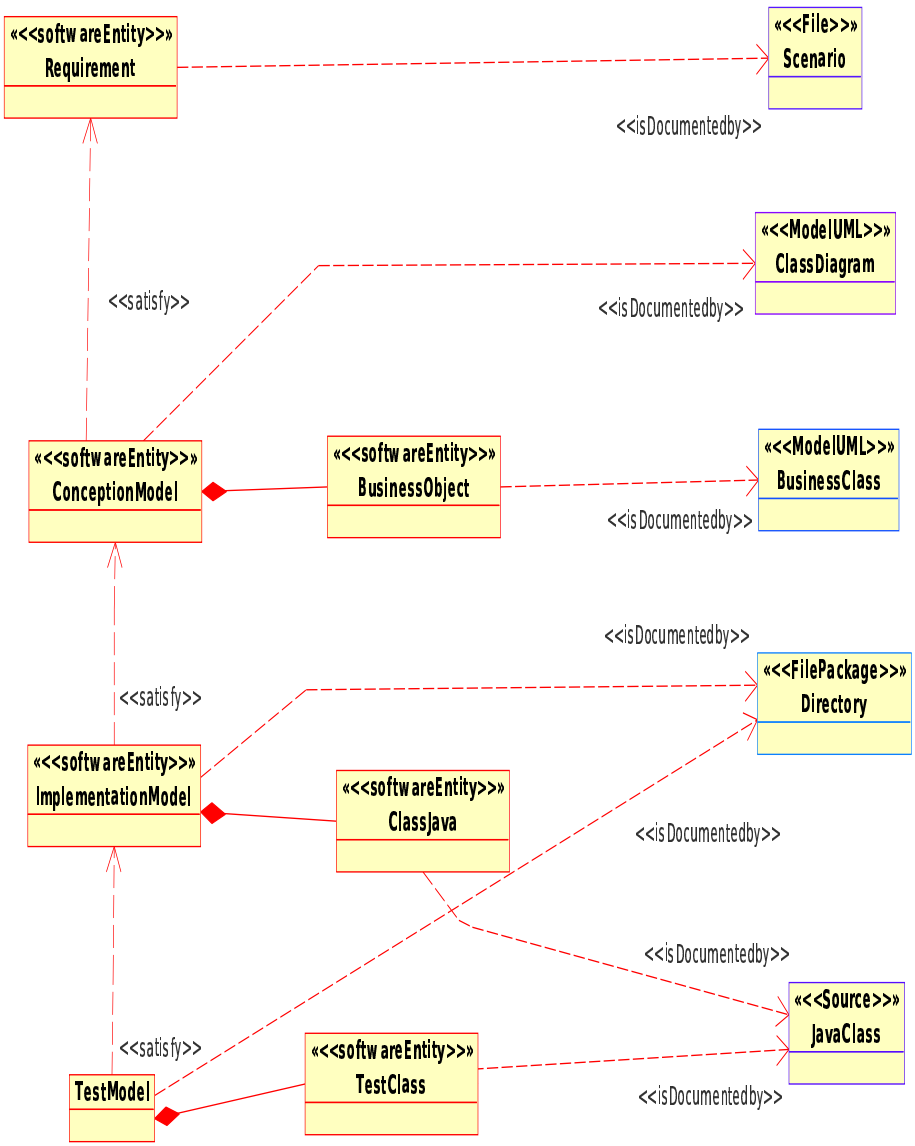

Figure 5. *Modèle XP de traçabilité des fonctionnalités*

La section suivante présente un cadre méthodologique pour la définition des modèles de traçabilité des fonctionnalités. Elle explique également quand et qui sont responsables de la définition de ces modèles de traçabilité des fonctionnalités.

4. Méthode pour la définition de modèles de traçabilité des fonctionnalités

A plusieurs reprises, nous avons émis l'idée que la définition d'un modèle de traçabilité des fonctionnalités était dépendante d'un processus logiciel. Cette section vise à préciser cette dépendance et introduit une méthode de définition des modèles de traçabilité des fonctionnalités. Pour ce faire, nous répondons à trois questions de base : quand, comment définit-on un modèle de traçabilité et qui définit un modèle de traçabilité ?

4.1. *Quand définit-on un modèle de traçabilité des fonctionnalités ?*

Notre objectif est l'utilisation de modèles de traçabilité des fonctionnalités en phase de maintenance. Deux cas se présentent pour la définition de ces modèles : ils peuvent soit être définis le long de la phase d'élaboration de l'application, soit en début de phase de maintenance. Le premier cas, le plus simple et le plus intuitif, implique une définition itérative et incrémentale. La définition du modèle de traçabilité des fonctionnalités s'établit au rythme de cette phase d'élaboration de façon à ce que chaque étape de cette phase d'élaboration indique les artefacts définis. Le deuxième cas implique une définition un peu plus longue car elle se base sur une application stable et finie. Toute la difficulté réside dans la recherche des informations, à savoir comment les unités logicielles et artefacts ont été définies et quel a été le processus logiciel employé.

Que ce soit le long du processus d'élaboration ou en début de phase de maintenance, la méthode de définition reste la même. La section suivante présente cette méthode de définition.

4.2. *Comment définit-on les modèles de traçabilité des fonctionnalités ?*

La définition d'un modèle de traçabilité des fonctionnalités se base sur deux éléments essentiels :

– *description des fonctionnalités.* Cette description liste clairement les unités logicielles et les artefacts attendus pour la définition des fonctionnalités. C'est ici qu'analystes et concepteurs se mettent d'accord sur la composition de la définition des fonctionnalités. Les modèles, ainsi que les supports utilisés pour leur définition sont à cette étape établis. Par exemple, pour une petite application, très souvent, une seule modélisation statique suffit pour la définition des fonctionnalités. Pour ce faire un diagramme de classes est le support approprié. Voici ci-après, la définition d'une unité logicielle et de son artefact associés à notre petite application ;

```
<application name="petite application">
  <functionality>
   <software unit name="modelisation statique">
     <artefact type="ClassDiagramm"/>
```

```
   </software unit>
  </functionality>
</application>
```

– *définition d'un processus d'élaboration.* Ce processus définit la méthode employée pour l'élaboration des fonctionnalités. Pour une application, elle est commune à l'ensemble des fonctionnalités et garantit la construction identique de l'ensemble des fonctionnalités.

Ainsi que ce soit le long d'un processus d'élaboration ou en début de maintenance, pour les fonctionnalités d'une application, la définition d'un modèle de traçabilité des fonctionnalités se base sur :

1) l'identification de ses unités logicielles, de ses artefacts utilisés et des protagonistes responsables de leurs définitions ;

2) l'organisation de ces éléments précédemment identifiés sur la base d'une méthode d'élaboration logicielle.

Gardons à l'esprit qu'un seul modèle de traçabilité des fonctionnalités suffit par application. La section 5 détaille l'utilisation de ce modèle de traçabilité des fonctionnalités lors d'une évolution logicielle.

4.3. *Qui définit les modèles de traçabilité des fonctionnalités ?*

Le long de la phase d'élaboration d'une application, tout protagoniste intervenant au cours de cette phase peut participer à la définition du modèle de traçabilité des fonctionnalités. Autrement dit, nous retrouvons des analystes, des concepteurs et des développeurs. En début de maintenance, les développeurs sont les seuls responsables de cette tâche avec l'aide des concepteurs. Ces mêmes protagonistes, responsables de la définition de ces modèles de traçabilité des fonctionnalités sont également les premiers utilisateurs. La définition d'un modèle de traçabilité des fonctionnalités favorise la communication entre tous ces protagonistes dans la mesure où elle met en commun différentes unités logicielles définies par plusieurs rôles.

Une fois définis, ces modèles sont censés nous fournir une aide à la réalisation des évolutions logicielles. Deux questions se posent alors auxquelles nous répondons dans la section suivante : comment un seul modèle de traçabilité des fonctionnalités par application peut-il remplir cette tâche, quels sont les apports et les limitations liés à ces modèles de traçabilité des fonctionnalités lors de la réalisation d'une évolution logicielle.

5. Utilisation, apports et limitations des modèles de traçabilité lors de la réalisation d'une évolution logicielle

Cette section se compose de deux parties. Une première partie développe l'utilisation des modèles de traçabilité des fonctionnalités et l'illustre sur un exemple d'appli-

cation de guichet automatique. Une seconde partie recense les apports et les limitations des instances d'un modèle de traçabilité des fonctionnalités lors de la réalisation d'une évolution logicielle.

5.1. *Utilisation d'un modèle de traçabilité des fonctionnalités*

Un modèle de traçabilité des fonctionnalités définit une méthode d'élaboration en termes d'unités logicielles et d'artefacts. Selon un processus d'élaboration, ce modèle liste les unités logicielles et les artefacts attendus pour chaque fonctionnalité de l'application. Or, en l'état, ce modèle est inutilisable puisqu'il est commun à l'ensemble des fonctionnalités : il reflète un modèle d'élaboration des fonctionnalités et non pas l'élaboration d'une fonctionnalité en particulier. Par contre, une instanciation de ce modèle, en rapport avec les unités logicielles d'une fonctionnalité offre l'état d'élaboration de cette fonctionnalité à un instant t.

5.1.1. *La traçabilité d'une fonctionnalité*

Pour ce faire, nous avons clarifié l'expression de traçabilité *d'une* fonctionnalité :

Définition 5.1 *La traçabilité d'une fonctionnalité f représente la vie d'une fonctionnalité donnée. Celle-ci prend la forme d'un graphe d'objets, que nous nommons graphe de traces de f, et fournit une organisation précise de ses unités logicielles et de ses artefacts. Ce graphe de traces est conforme à un modèle de traçabilité des fonctionnalités par un mécanisme d'instanciation. Ainsi, à chaque date t, chaque graphe de traces admet un état global de la fonctionnalité t.*

Précisons ce que nous voulons dire par état global d'une fonctionnalité à la date t. En phase de maintenance, les fonctionnalités d'une application sont déjà spécifiées et construites. Nous le savons au cours de cette phase, les artefacts des fonctionnalités changent, ils évoluent. Chaque artefact d'une fonctionnalité traverse ainsi avec le temps une séquence de différents états de définition. Si une séquence d'états peut être attribuée à un artefact, nous pouvons alors définir l'état global d'une fonctionnalité par la réunion des états de ses artefacts à une date t. Ainsi, dès lors qu'un graphe de traces structure les unités logicielles et les artefacts d'une fonctionnalité, nous considérons que l'état global de la fonctionnalité à une date t est fourni par l'ensemble des états de ses artefacts à la même date t.

La construction du graphe de traces pour chaque fonctionnalité f d'une application est donc réalisée à partir d'un modèle de traçabilité des fonctionnalités et à partir de l'ensemble des artefacts définis à l'instant t. D'une part, le modèle de traçabilité des fonctionnalités est construit selon la méthode introduite en section 4, à l'aide du profil UML défini en section 3. D'autre part, l'ensemble des artefacts associés à la fonctionnalité f traduit l'état global de f, à la date t. Puis, automatiquement, et de manière incrémentale, le graphe de traces de f, est partiellement instancié à partir du modèle de traçabilité des fonctionnalités. L'instanciation est partielle car elle est soumise à

une vérification de la présence des artefacts attendus. En effet, cette vérification permet d'instancier une unité logicielle si et seulement si cette dernière admet au moins un artefact lui correspondant et existant à la date t. Voici ci-dessous l'algorithme de construction d'un graphe de traces pour une fonctionnalité f à la date t.

Soient,
- f, le nom de la fonctionnalité pour laquelle on construit le graphe de traces ;
- $T(S, A)$, le graphe de traces de la fonctionnalité f ;
- $P(N, E)$, un modèle de traçabilité des fonctionnalités, transformé en graphe. Cette transformation est formellement définie dans les travaux de (Mens *et al.*, 2005a) ;
- $U_{f,t}$, l'ensemble des artefacts définis pour la fonctionnalité f à la date t.

construireGrapheTrace ($U_{f,t}$,P,f) : Graphe{
1- initialisation de la traçabilité de f
le graphe est initialement vide :
T := graphe vide ;

2- identification et instanciation des unités logicielles définies dans
le modèle de traçabilité des fonctionnalités :
ajouter dans S les éléments stéréotypés « Software Unit » ;

3- identification et instanciation des relations entre unités logicielles :
ajouter dans A des instances des relations stéréotypées
« Satisfy »,« DependOn », « ComposedBy » ;
si et ssi $\exists$ **une relation** r **et un couple** (a, b) **de** S **tels que** $r(a, b)$
existe dans E

4- pour chaque unité logicielle présente,
vérification et instanciation des artefacts présents dans
U_f dans le graphe T
pour toute instance d'un élément stéréotypé « Software Unit »
du graphe T,
a- identifier le ou les artefact(s)
attendu(s) (depuis le graphe P) ;
b- vérifier s'il ou s'ils existent dans U_f,
des représentants de ces d'artefacts ;
c- si c'est le cas,
les ajouter dans S ;
et ajouter dans A, les liens correspondants
et stéréotypés « isDocumentedBy »
sinon supprimer
l'élément stéréotypé « Software unit » ;

5- pour chaque artefact défini dans T
définition et organisation des rôles, responsables de leur définition
pour toute instance d'artefact de graphe T,
a- Identifier le rôle, responsable de sa définition ;
b- Instancier un élément stéréotypé « Stakeholder »
L'ajouter dans S ;
et ajouter dans A les liens correspondants
et stéréotypés « isManagedBy » ;

retourner T ; }

Voici un exemple de construction d'un graphe de traces d'une fonctionnalité.

5.1.2. *Exemple du guichet automatique*

Nous considérons l'exemple simple du distributeur automatique d'argent, aussi connu sous le nom de guichet automatique. Ses fonctionnalités sont le retrait, le dépôt et le versement d'argent.

L'objectif de cet exemple est la définition des tracabilités de chaque fonctionnalité f du guichet automatique. Chacune d'elles prend la forme d'un graphe d'objets, nommé graphe de traces de f. Ainsi, pour la fonctionnalité du retrait d'argent, son graphe de traces est basé sur :

– le modèle de traçabilité des fonctionnalités, illustré en figure 4. Ce modèle définit le processus RUP simplifié en termes des unités logicielles et des artefacts. Il établit la liste des artefacts attendus par unité logicielle ;

– l'ensemble des artefacts $U_{retrait}$ définis au moment de l'exemple (après le développement complet de l'application). L'ensemble ne regroupe que les artefacts liés au retrait d'argent.

$U_{retrait} = \{$
$Besoins.UseCase.Retrait$,
$Analyse.ClassDiagram.Bank$,
$Analyse.Class.Retrait$,
$Analyse.Class.Client$,
$Analyse.Class.Transaction$,
$Analyse.Class.Compte$,
$Conception.ClassDiagram.Bank$,
$Implementation.Package.bancaire$,
$Implementation.Class.Retrait.java$,
$Implementation.Class.Transaction.java$,
$Implementation.Class.Compte.java$,
$Implementation.Class.Client.java$,
$Implementation.Class.GuichetAutomatique.java$
$\}$

Puis, de manière automatique, incrémentale et itérative, l'instanciation partielle du modèle de traçabilité des fonctionnalités de la figure 4 fournit le graphe de traces de la figure 6. Ce graphe de traces illustre la traçabilité du retrait d'argent. Il présente :

– une organisation claire des artefacts du retrait d'argent. Cette organisation est conforme au modèle de traçabilité des fonctionnalités défini en figure 4 et vient par là même valider l'utilisation de la méthode RUP ;

– l'état global du retrait d'argent après son développement. Ce graphe liste les artefacts réellement définis pour le retrait d'argent. Nous remarquons, par exemple que l'élaboration de la fonctionnalité n'a pas effectué de spécification de composants logiciels, comme cela était attendu lors de la phase de conception (cf. modèle RUP de traçabilité des fonctionnalités). Ce graphe de traces évalue qualitativement l'utilisation réelle de la méthode RUP.

Ainsi, l'application admet un modèle de traçabilité des fonctionnalités. Chaque fonctionnalité de l'application est décrite par son graphe de traces. Celui-ci reflète l'état global de la fonctionnalité par l'intermédiaire de ses artefacts. Lorsqu'une fonctionnalité subit une évolution, un autre graphe de trace est dérivé à partir du précédent afin de prendre en compte le nouvel état de la fonctionnalité. Au résultat, nous avons un graphe de traces à chaque évolution de la fonctionnalité qui prend en compte le ou les transformations réalisées. La réunion de ces graphes de traces permet de modéliser l'évolution temporelle de chaque fonctionnalité d'une application. La section suivante présente leurs apports lors de la réalisation d'une évolution logicielle.

5.2. *Modèle de traçabilité des fonctionnalités et instances au service de la réalisation des évolutions logicielles*

Les activités de maintenance sont essentielles pour la réalisation des évolutions logicielles. Selon leur nature, ces évolutions sont plus ou moins difficiles à réaliser. Le temps et l'investissement des mainteneurs varient significativement selon qu'ils ont à effectuer une simple réécriture d'une opération d'une classe ou qu'ils ont à changer la mise en œuvre d'une fonctionnalité dans son intégralité. Or, que ce soit dans un cas ou dans un autre, les outils pour assister ces démarches de maintenance manquent et la qualité des évolutions en est souvent affectée. Grâce à nos graphes de traces, représentant la tracabilité de chaque fonctionnalité d'une application, la réalisation des évolutions logicielles est significativement simplifiée voire même améliorée. Afin d'illustrer ces apports, nous déroulons la réalisation d'une petite évolution logicielle sur la fonctionnalité du retrait d'argent de notre application précédente. Les apports ainsi que les limitations fournis par le graphe de traces du retrait d'argent sont ensuite listés et commentés dans une seconde partie.

5.2.1. *Présentation d'une évolution logicielle*

Nous reprenons l'exemple de l'application bancaire de la section 5.1.2 afin d'y effectuer une évolution de la fonctionnalité du retrait d'argent. Plus précisément, nous

Figure 6. *Graphe de traces du retrait d'argent*

désirons faire évoluer le retrait d'argent par l'ajout d'une vérification supplémentaire du solde de compte avant toute transaction bancaire. Reprenons le graphe de traces du retrait d'argent de la figure 6 et voyons comment ce graphe peut nous rendre service dans les trois étapes d'un processus simplifié de réalisation d'une évolution comme celui du standard (IEEE1219, 1998) :

1) la compréhension d'une fonctionnalité ;

2) l'identification d'une évolution sur la fonctionnalité ;

3) l'analyse d'impact de l'évolution.

Ces apports sont confrontés aux pratiques utilisées en SSII.

5.2.2. *Les apports*

1) La compréhension de la fonctionnalité

Cette étape est loin d'être triviale pour les applications d'entreprise. Elle occupe significativement 50 à 60 % du temps de maintenance. En effet, deux facteurs sont importants à considérer : la taille et l'architecture logicielle d'une application. Concernant la taille de l'application, les SSII développent des applications à grande échelle. Ces applications admettent plus de 10 000 classes et compliquent la compréhension de la fonctionnalité. L'architecture logicielle peut être dans ce cas une aide à la compréhension mais bien que chaque composant logiciel permette l'implémentation d'une ou plusieurs opérations, elle ne permet pas la description explicite d'une fonctionnalité dans sa globalité. Notons de plus, qu'en pratique, les outils existants, fournissant une aide à la compréhension, ne sont pas utilisés à cause de leur prise en main difficile. La compréhension est basée dans le meilleur des cas sur les modèles définis en conception, et dans le pire des cas, par l'exploration du code. Grace à notre graphe de traces, la compréhension d'une fonctionnalité est immédiate puisque notre approche préconise un graphe par fonctionnalité. Très rapidement, tout mainteneur peut accéder aux modèles métier mais aussi à l'implémentation d'une fonctionnalité. De plus, ce graphe apporte de la précision puisqu'il permet d'énumérer l'ensemble des artefacts qui participent à la définition de la fonctionnalité. En l'occurrence pour notre exemple du retrait d'argent bancaire, le graphe de la figure 6 donne un détail complet de tous les artefacts définis à chaque étape de son élaboration ainsi que les éléments d'implémentation.

2) Identification de l'évolution sur la fonctionnalité

La fonctionnalité une fois comprise, l'évolution à réaliser est formulée en termes de transformations. L'ensemble de ces transformations est théoriquement projeté sur la fonctionnalité. L'ensemble des unités logicielles et les artefacts directement touchés par les transformations doivent être rapidement identifiés. Notre graphe de traces lié au retrait d'argent permet de faire directement cette identification. En l'occurrence pour notre exemple, l'évolution concerne l'ajout d'une vérification métier sur le compte de tout client avant toute transaction de retrait. En conséquence, les transformations sont des ajouts de régles de gestion et touchent les artefacts suivants du retrait d'argent :

$\{Besoins.UseCase.Retrait$,
$Analyse.ClassDiagram.Bank$,
$Analyse.Class.Retrait$,
$Conception.ClassDiagram.Bank$,
$Implementation.Class.Retrait$
$\}$

Depuis le graphe de traces du retrait d'argent, ces artefacts sont directement identifiables et accessibles puisque chaque unité logicielle est liée à son ou ses artefacts qui la définissent. Cette étape importante d'identification de l'évolution sur la fonctionnalité cible est allégée par nos graphes de traces.

3) Analyse d'impact d'une évolution

L'analyse d'impact d'une évolution est une étape délicate dans un processus d'évolution logicielle car elle consiste à quantifier les conséquences structurelles et fonctionnelles provoquées à la suite d'une ou plusieurs transformations logicielles. Dans la littérature, les méthodes d'analyse d'impacts sont nombreuses (Doyle, 1987) ; (Yau *et al.*, 1987) ; (Collofello *et al.*, 1988) ; (Knethen, 2001). En général, elles se basent sur un graphe de dépendances entre les objets d'un programme et se différencient les unes des autres par les critères et les algorithmes utilisés pour la détermination des impacts. Dans le cadre de cet article, nous nous intéressons plus particulièrement aux impacts provoqués par une évolution et à la manière de les déterminer avec un graphe de traces. Plus précisément, nous cherchons à déterminer l'ensemble des artefacts à mettre à jour à la suite d'une évolution logicielle.

Dans cette optique, nous réduisons la notion de degré d'impact par la détermination du nombre des modifications logicielles à faire. Plus précisément un degré d'impact est le nombre d'artefacts à modifier à la suite d'une évolution logicielle. Ainsi, à partir d'un graphe de traces, nous déterminons ce degré d'impact à partir des relations (*« satisfy »*) et de composition définies entre les unités logicielles d'une fonctionnalité. Ainsi, pour chaque unité logicielle $unitA$ modifiée, un algorithme de parcours fournit l'ensemble des artefacts associés à $unitA$. Puis il recherche l'ensemble des unités logicielles liées à cette unité logicielle $unitA$ par les relations « satisfy » et/ou de composition. Cet algorithme réitère cette recherche sur chaque unité logicielle rencontrée. Sur notre exemple d'application bancaire, en dehors des unités logicielles directement concernées par la modification du retrait d'argent, son évolution touche l'évolution des artefacts suivants :
$\{$ $Analyse.Class.Client$,
$Analyse.Class.Transaction$,
$Analyse.Class.Compte$,
$Implementation.Package.bancaire$,
$Implementation.Class.Transaction.java$,
$Implementation.Class.Compte.java$,
$Implementation.Class.Client.java$,

$Implementation.Class.GuichetAutomatique.java$
$\}$

La figure 7 illustre les unités logicielles et artefacts impactés (en gras) par l'évolution du retrait d'argent. L'algorithme débute avec la classe du retrait d'argent et identifie toutes les unités logicielles et tous les artefacts impactés. Remarquons toutefois que cette analyse est à la fois exhaustive et grossière puisqu'elle surestime l'ensemble des unités logicielles et artefacts à modifier en déterminant un ensemble plus grand que l'ensemble des unités logicielles et artefacts réellement impactés par l'évolution.

Le degré d'impact lié à cet ajout de vérification de solde de compte est de treize puisque treize artefacts sont concernés par l'impact de l'évolution.

Pour une fonctionnalité donnée, notre graphe de traces permet d'identifier les artefacts à mettre à jour à la suite d'une évolution logicielle. Il facilite la définition du degré d'impact associé à une évolution logicielle. Ceci étant dit, notre graphe ne permet pas de répercuter le contenu d'une modification logicielle sur le reste de l'application mais uniquement la présence d'une modification sur le reste des unités logicielles et artefacts d'une fonctionnalité. Dans un processus de réalisation des évolutions logicielles cette analyse d'impact permet de cibler les transformations à réaliser au niveau des spécifications et des fichiers d'implémentation d'une fonctionnalité.

5.2.3. *Les limitations*

Les apports des graphes de traces sont certains. Ils fournissent une aide précieuse à la réalisation des évolutions logicielles. Cependant, nous sommes également conscients que ces graphes sont limités sur plusieurs points. Nous les détaillons ci-après :

1) ces graphes ne réalisent pas automatiquement les transformations liées à une évolution logicielle. La raison principale en est que la granularité de ces graphes est trop grosse. Sa plus petite granularité est la classe. Nous avons intentionnellement fait ce compromis afin d'avoir une vue globale sur l'élaboration d'une fonctionnalité et sur ses unités logicielles ;

2) l'analyse des impacts ne s'effectue qu'au sein d'une fonctionnalité. Nous sommes bien conscients que la réalisation d'une évolution logicielle partant d'une fonctionnalité n'impacte pas uniquement cette dernière. Les répercussions peuvent s'étendre sur le reste de l'application. Ces répercussions sont à la fois fonctionnelles et structurelles. Or, notre volonté est de focaliser l'attention des mainteneurs sur une fonctionnalité à la fois afin de faciliter leur tâche. Prendre en compte, plusieurs fonctionnalités expose le mainteneur aux erreurs. De plus, la traçabilité d'une fonctionnalité ne doit concerner que son élaboration. Modéliser ses dépendances avec les autres fonctionnalités de l'application devient hors de notre propos et peut amener de la confusion ;

3) les graphes de traces n'effectuent pas de validation des évolutions logicielles réalisées. Nous pensons que seules des méthodes et des spécifications formelles autres que UML peuvent correctement répondre à ce besoin. Partant de ce fait, il est difficile

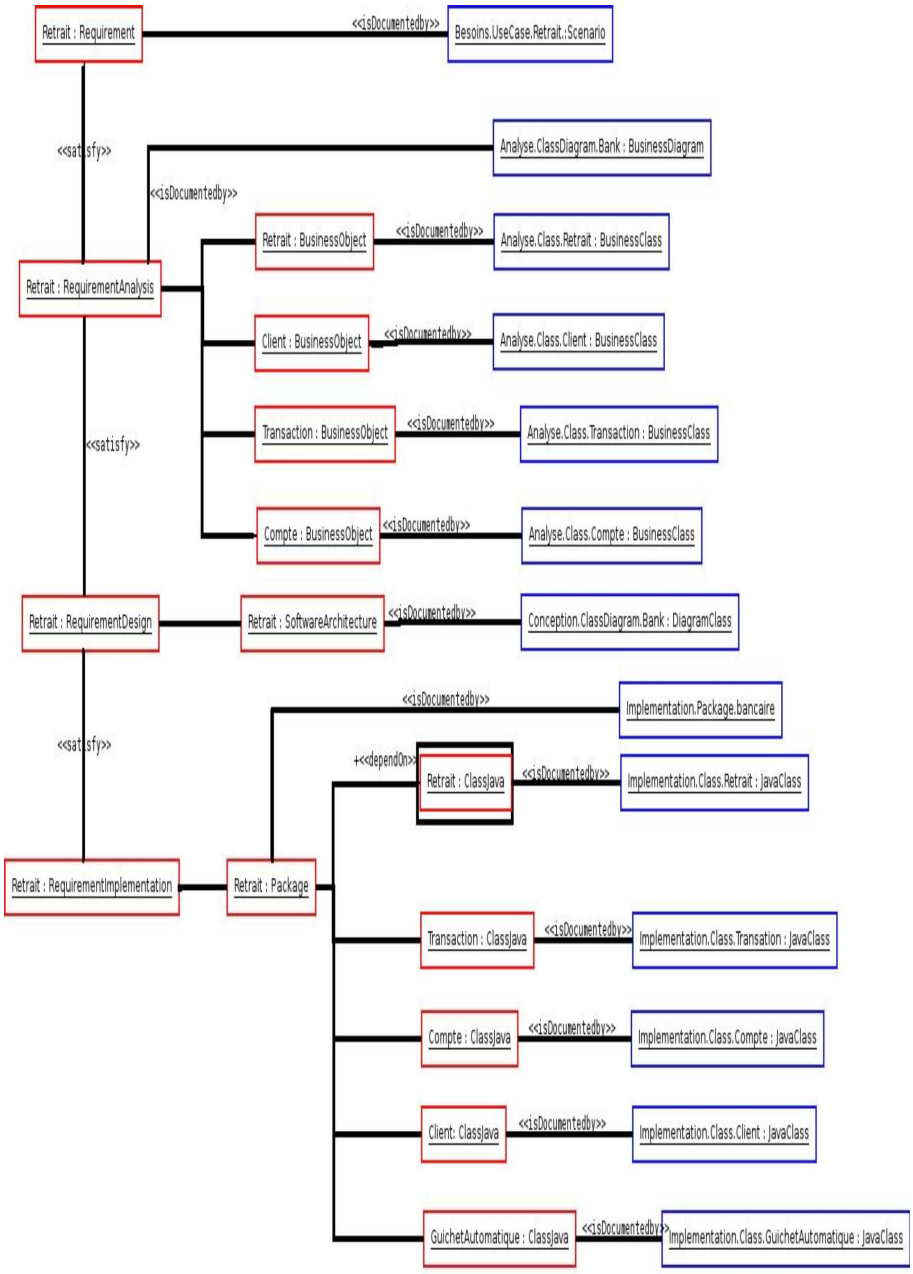

Figure 7. *Répercussions provoquées par l'évolution du retrait d'argent*

d'évaluer quantitativement la qualité des évolutions logicielles. Ceci étant dit, la propriété de conformité des graphes de traces par rapport à un modèle de traçabilité des fonctionnalités permet de vérifier que toute évolution logicielle ne vienne pas altérer la méthode d'élaboration initialement appliquée à une fonctionnalité. Autrement dit, nous avons la possibilité de vérifier qu'aucun artefact lié à une unité logicielle d'une fonctionnalité n'a été malencontreusement supprimé.

6. Discussion

Cette section discute de travaux autour de la traçabilité des besoins. Par conséquent, elle n'utilise pas exactement l'expression de la traçabilité des fonctionnalités. Dans le cadre de cette discussion et pour éviter toute confusion, nous effectuons un abus de langage en assimilant un besoin à une fonctionnalité. Ceci étant dit, cette littérature sur la traçabilité des besoins a servi de base pour nos travaux et conforte à la fois notre changement de vocabulaire, pour une utilisation de la traçabilité dédiée à la réalisation des évolutions logicielles.

Les problématiques et les enjeux associés à la notion de traçabilité des besoins sont étudiés depuis plusieurs années. De nombreux travaux (Johnson *et al.*, 1991) ; (Hamilton *et al.*, 1991) ; (Ramesh *et al.*, 2001) ; (Knethen, 2001) s'accordent pour dire qu'ils portent à la fois sur la phase d'élaboration d'un besoin ainsi que sur la phase de maintenance. Nous l'avons dit, nos travaux s'intéressent à l'utilisation de la notion de la traçabilité des fonctionnalités lors de la réalisation des évolutions logicielles et se concentrent principalement sur son exploitation en phase de maintenance.

Selon (Gotel *et al.*, 1994), l'essentiel des difficultés rencontrées lors de l'exploitation de la traçabilité des besoins s'explique par une mauvaise définition de la *traçabilité des préspécifications des besoins*. L'évolution d'un besoin est d'autant mieux réussie que si cette traçabilité est bien définie et bien maintenue. Bien que nos travaux s'inspirent de leur définition sur la traçabilité des besoins, nous ne partageons pas tout à fait ce point de vue, car la réalisation des évolutions logicielles actuelles se focalise plus sur des modifications de l'implémentation des besoins plutôt que sur leurs identifications et leurs formulations. Ainsi l'évolution d'un besoin revient aujourd'hui à modifier les modélisations UML et l'implémentation de ce besoin plutôt que sa capture effectuée en amont de la phase d'élaboration. Notre démarche s'attache à la *traçabilité postspécifications des besoins* pour reprendre le vocabulaire introduit dans (Gotel *et al.*, 1994) afin de l'exploiter dans un contexte d'évolution logicielle. Ceci explique pourquoi, nous préférons utiliser le vocabulaire de « traçabilité des fonctionnalités » au lieu de « traçabilité des besoins ».

D'autres travaux exploitent la notion de traçabilité afin de suivre non plus seulement l'élaboration d'un besoin mais de tout objet. Celui-ci peut être de granularité quelconque, du moment que son suivi apporte une plus-value. (Lindvall *et al.*, 1996) présentent quatre natures de traçabilités relevées au cours de l'élaboration d'une application orientée objet, qui permettent de tracer un objet, une association, un cas d'uti-

lisation et la dépendance entre plusieurs cas d'utilisation. Ce travail met en évidence la diversité des traçabilités définissables au cours de l'élaboration d'une application. Or, bien que ces identifications soient basées sur une expérimentation de l'élaboration d'une application à grande échelle, nous doutons que ces traçabilités puissent être véritablement définies. Pour des raisons de temps, de disponibilité de ressources, la plupart de ces traçabilités ne peuvent être définies et encore moins être maintenues. Ceci relève le constat que tout n'est pas traçable dans une application.

Enfin, nous attirons l'attention sur la notion de *type de traçabilité* définie dans de nombreux travaux (Lindvall *et al.*, 1996) ; (Ramesh *et al.*, 2001). Ces travaux identifient deux types de traçabilité des besoins : *traçabilité verticale* et *traçabilité horizontale*. La première caractérise le suivi d'un objet au sein d'un même niveau d'abstraction. Par exemple, toutes les dépendances d'un objet, à la phase de conception avec le reste des objets de l'application. La seconde caractérise le suivi des objets de niveau d'abstraction différent tel que la réalisation d'un cas d'utilisation par un diagramme de collaboration. A l'inverse, nous qualifions la traçabilité des fonctionnalités de traçabilité verticale et non d'horizontale car contrairement aux travaux précédents, nous avons fortement associé la définition d'un modèle de traçabilité des fonctionnalités à un processus de développement logiciel. Or comme ceux-ci se veulent soient *top-down* ou *bottom-up*, la traçabilité des fonctionnalités définie à l'aide de ces processus ne peut être que verticale. Nous identifions également *la traçabilité temporelle* qui correspond à la séquence de versions des spécifications et des implémentations d'un besoin définis à travers le temps. (Antoniol *et al.*, 1999) exploitent cette traçabilité temporelle pour la réalisation des évolutions logicielles. Dans leur méthode, deux ou plusieurs modèles de conception de haut niveau correspondant chacun à une version de l'élaboration d'un besoin sont comparés pour constituer cette traçabilité temporelle. Notons que le support, le plus naturel pour ce type de traçabilité est un système de versionning tel que CVS (Fogel, 1999) ou plus récemment SVN (Collins-Sussman *et al.*, 2006).

7. Conclusion

Dans cet article, nous avons présenté et établi le concept de traçabilité des fonctionnalités pour fournir une aide à la réalisation des évolutions logicielles. Sa définition s'inspire des secteurs industriels qui suivent le cycle de vie d'un produit depuis son expression jusqu'à sa destruction. Nous nous sommes également inspirés de travaux en ingénierie des besoins sur la notion de la traçabilité des besoins. La traçabilité des fonctionnalités correspond à la capacité de décrire et de suivre l'élaboration des fonctionnalités d'une application. Cette notion peut être capturée par l'intermédiaire d'un modèle, défini grâce à un profil UML. Ce modèle permet de décrire une méthode d'élaboration de l'ensemble des fonctionnalités d'une application. Une instanciation de ce modèle permet de décrire l'élaboration d'une fonctionnalité en particulier. Chaque fonctionnalité est par conséquent caractérisée par un graphe de traces qui structure l'ensemble de ses unités logicielles et ses artefacts (modèles UML et fi-

chiers d'implémentation). Enfin, l'utilisation, les apports et limitations de ces graphes de traces ont été mis en évidence par le déroulement d'une évolution logicielle sur un exemple classique. En facilitant la compréhension, l'identification et une relative analyse d'impact de toute évolution logicielle, nous offrons, à chaque mainteneur un cadre et une méthode de réalisation des évolutions. La prochaine étape de notre travail consiste justement à s'intéresser au contenu de ces artefacts afin de garantir un degré de qualité des évolutions logicielles. Nous rentrons dans le domaine de la transformation des applications avec les travaux sur la transformation de modèles (OMG, 2003) ; (Blanc *et al.*, 2004) ou ceux sur la transformation de programme (Pawlak, 2005).

8. Bibliographie

ACCORD P., « Conventions communes aux profils UML », 2002, http ://www.infres.enst.fr/projets/accord/lot2/index.html.

Ajila S., « Software Maintenance : An Approach to Impact Analysis of Objects Change », *Software - Practice and Experience*, vol. 25, n° 10, p. 1155-1181, 1995.

Antoniol G., Canfora G., Lucia A. D., « Maintaining Traceability During Object-Oriented Software Evolution : A Case Study », *ICSM*, p. 211-219, 1999.

Beck K., *Extreme Programming explained*, Addison-Wesley Professional ; 1st edition, 1999.

Blanc X., Caron O., Georgin A., Muller A., « Transformation de modèles : d'un modèle abstrait aux modèles CCM et EJB », *Langages, Modèles et Objets, LMO 2004*, L'objet, Hermès Sciences, p. 161-174, March, 2004.

Bohner S. A., « Software Change Impacts - An Evolving Perspective. », *ICSM*, p. 263-272, 2002.

Brooks R., « Towards a Theory of the Comprehension of Computer Programs », *International Journal of Man-Machine Studies*, vol. 18, p. 543-554, 1983.

Clarke P., Malloy B., Gibson P., « Using a Taxonomy Tool to Identify Changes in OO Software », , citeseer.ist.psu.edu/clarke03using.html, n.d.

Collins-Sussman B., Fitzpatrick B. W., Pilato C. M., *Version Control with Subversion for Subversion 1.3*, O'Reilly Edition, 2006.

Collofello J., Orn M., « A Pratical Software Maintenance Environment », *In Proceedings of the Conference on Software Maintenance*, p. 45-51, 1988.

Consortium W. W. W., « Resource Description Framework Specification », , http ://www.w3.org/RDF/, 10 February 2004.

Doyle J., « A truth maintenance system », , vol. , p. 259-279, 1987.

Fogel K. F., *The CVS Book : Open Source Development with CVS*, Coriolis Inc., 1999.

Gotel O. C., Finkelstein A. C., « An analysis of the requirements traceability problem. », *IEEE Conputer Society Press.*, vol. , p. 94-101, 1994.

Hamilton V., Beeby M. L., « Issues of traceability in integrating tools », *Tools and Techniques for Maintaining Traceability During Design, IEEE Colloquium*, vol. , p. 4/1-4/3, 1991.

IEEE1219, *Standard for Software Maintenance-IEEE 1219-1998*, IEEE Society Press, 1998.

IEEE830, *Standard for Software Requirements Specifications 830-1984*, IEEE Society Press, 1984.

Jacobson I., Booch G., Rumbaugh J., *The Unified Software Development Process*, Addison Wesley Longman, 1998.

Jacobson I., Christerson M., Jonsson P., Overgaard G., *Object-Oriented Software Engineering. A use case driven approach*, Addison - Wesley, 1994.

Johnson W. L., Feather M. S., Harris D. R., « Integrating Domain Knowledge, Requirements and Specifications », *Journal of Systems Integration*, vol. 1, p. 283-320, 1991.

Knethen A. V., « A Trace Model for System Requirements Changes on Embedded Systems », *IWPSE '01 : Proceedings of the 4th International Workshop on Principles of Software Evolution*, ACM Press, New York, NY, USA, p. 17-26, 2001.

Kung D., Gao J., Hsia P., Wen F., Toyoshima Y., Chen C., « Change Impact Identification in Object Oriented Software Maintenance », *Proceedings of the International Conference on Software Maintenance 1994*, IEEE Computer Society Press, p. 202-211, 1994.

Lehman M., « Laws of Software Evolution Revisited », , citeseer.ist.psu.edu/255362.html, 1996.

Lindvall M., Sandahl K., « Practical Implications of Traceability », *Software Practice and Experience*, vol. 26, n° 10, p. 1161-1180, 1996.

Lowe W., Ericsson M., Lundberg J., Panas T., « Software Comprehension - Integrating Program Analysis and Software Visualization », *In Software Engineering Research and Practice (SERPS)*, 2002.

Mens T., Eetvelde N. V., Demeter S., Janssens D., « Formalizing refactorings with graph transformations : Research Articles », *J. Softw. Maint. Evol.*, vol. 17, n° 4, p. 247-276, 2005a.

Mens T., Wermelinger M., Ducasse S., Demeyer S., Hirshfeld R., « Challenges in software evolution », , ChaSE Workshop 2005 report, 2005b.

OMG, « Model Driven Architecture (M.D.A) », 2003, http ://www.omg.org/mda/.

OMG, « SPEM Specification 1.1 », 2005a, http ://www.omg.org/technology/documents /formal/spem.htm.

OMG, « UML Specification 1.5 », 2005b, http ://www.omg.org/technology/documents.

Omote H., Sasaki K., Kaiya H., Kaijiri K., « Software Evolution Support Using Traceability Link between UML diagrams », *in* V. Stefanuk, K. Kaijiri (eds), *Knowledge-Based Software Engineering*, IOS Press, p. 15-23, Aug., 2004. Proc. of the 6th JCKBSE.

Pacione M. J., Roper M., Wood M., « A novel software visualisation model to support software comprehension », *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE)*, IEEE Computer Society Press, p. 70-79, 2004.

Parnas D. L., « Software aging », *ICSE '94 : Proceedings of the 16th international conference on Software engineering*, IEEE Computer Society Press, Los Alamitos, CA, USA, p. 279-287, 1994.

Pawlak R., « Spoon : Annotation Driven Program Transformation - The AOP Case », *In Proceedings of the first First Workshop on Aspect-Orientation for Middleware Development. Middleware*, 2005.

Ramesh B., Jarke M., « Toward Reference Models of Requirements Traceability », *Software Engineering*, vol. 27, n° 1, p. 58-93, 2001.

Rugaber S., « Program Comprehension », *Encyclopedia of Computer Science and Technology, Marcel Dekker*, vol. 35, n° 20, p. 341-368, 1995.

SOFTEAM, « Objecteering », 2002, http ://www.objecteering.com/.

Soloway E., Bonar J., Ehrlich K., « Cognitive Strategies and Looping Constructs : An Empirical Study », *Communications of the ACM*,, IEEE Computer Society Press, p. 853-860, 1983.

Yau S., Collofello J. S., MacGregor T., « Ripple effect analysis of software maintenance », *Compsac, IEEE Computer Society Press, Los Alamitos*, p. 60-65, 1978.

Yau S. S., Kishimoto Z., « A method for revalidating modified programs in the maintenance phase », *In Proceedings of the 11th International Computer Software and Applications Conference (COMP-SAC)*, IEEE Computer Society Press, p. 272-277, 1987.

ANNEXE POUR LE SERVICE FABRICATION
A FOURNIR PAR LES AUTEURS AVEC UN EXEMPLAIRE PAPIER
DE LEUR ARTICLE ET LE COPYRIGHT SIGNE PAR COURRIER
LE FICHIER PDF CORRESPONDANT SERA ENVOYE PAR E-MAIL

1. ARTICLE POUR LA REVUE :
RSTI - L'objet. Volume 13 – n°1/2007. Evolution du logiciel

2. AUTEURS :
Dolores Diaz[*,**] *— Lionel Seinturier* [*] *— Laurence Duchien* [*]
Pascal Flament [**]

3. TITRE DE L'ARTICLE :
Une aide à la réalisation des évolutions logicielles grâce aux modèles de traçabilité des fonctionnalités

4. TITRE ABRÉGÉ POUR LE HAUT DE PAGE MOINS DE 40 SIGNES :
Traçabilité et évolutions logicielles

5. DATE DE CETTE VERSION :
15 janvier 2007

6. COORDONNÉES DES AUTEURS :

– adresse postale :

[*] Laboratoire d'Informatique Fondamentale de Lille (UMR CNRS 8022)
INRIA Futurs - Projet Jacquard - USTL Bâtiment M3
F-59655 Villeneuve d'Ascq cedex

{diaz, seinturi, duchien}@lifl.fr

[**] NORSYS
1, rue de la Cense des Raines
ZAC du moulin
F-59710 Ennevelin

{ddiaz, pflament}@norsys.fr

– téléphone : 03.28.76.56.00

– télécopie : 03.28.76.57.00

– e-mail : {ddiaz,pflament}@norsys.fr, {diaz,duchien,seinturi}@lifl.fr

7. LOGICIEL UTILISÉ POUR LA PRÉPARATION DE CET ARTICLE :
LaTeX, avec le fichier de style `article-hermes.cls`,
version 1.23 du 17/11/2005.

8. FORMULAIRE DE COPYRIGHT :
Retourner le formulaire de copyright signé par les auteurs, téléchargé sur :
`http://www.revuesonline.com`

SERVICE ÉDITORIAL – HERMES-LAVOISIER
14 rue de Provigny, F-94236 Cachan cedex
Tél. : 01-47-40-67-67
E-mail : revues@lavoisier.fr
Serveur web : http://www.revuesonline.com