UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE
APPLIQUÉES

PAR
FABRICE DÉHOULÉ

ANALYSE DE L'IMPACT DANS LES SYSTÈMES ORIENTÉS ASPECT (SOA) : ÉLABORATION D'UN MODÈLE D'IMPACT

MARS 2014

Université du Québec à Trois-Rivières

Service de la bibliothèque

Avertissement

L'auteur de ce mémoire ou de cette thèse a autorisé l'Université du Québec à Trois-Rivières à diffuser, à des fins non lucratives, une copie de son mémoire ou de sa thèse.

Cette diffusion n'entraîne pas une renonciation de la part de l'auteur à ses droits de propriété intellectuelle, incluant le droit d'auteur, sur ce mémoire ou cette thèse. Notamment, la reproduction ou la publication de la totalité ou d'une partie importante de ce mémoire ou de cette thèse requiert son autorisation.

ANALYSE DE L'IMPACT DANS LES SYSTÈMES ORIENTÉS ASPECT (SOA): ÉLABORATION D'UN MODÈLE D'IMPACT

Fabrice DÉHOULÉ

SOMMAIRE

La programmation orientée aspect (POA) est un paradigme de programmation complémentaire à la programmation orientée objet (POO). Elle permet de réunir toutes les préoccupations transversales du code orienté objet dans des modules appelés aspects. Ces aspects ont la particularité d'être complètement invisibles au code objet et seront introduits à son insu lors d'un processus appelé tissage. La relation entre le code objet et le code aspect introduit donc une nouvelle dimension quant à l'analyse de l'impact des changements. Les techniques utilisées au niveau des systèmes orientés objet ne peuvent pas être directement utilisées pour les systèmes orientés aspect en raison des nouvelles dépendances entre le code objet et le code aspect. Nous nous proposons, dans ce contexte, d'élaborer un modèle d'impact pour les systèmes orientés aspect. Ce modèle aura pour objectif de prédire les impacts d'un changement survenu tant au niveau du code objet qu'au niveau du code aspect. Afin d'appuyer notre approche, nous avons procédé à des tests sur différents programmes orientés aspect. Ces tests ont porté sur les impacts survenus sur le code aspect après des modifications dans le code objet, puis du code aspect vers lui-même et enfin du code aspect vers le code objet. Une comparaison a ensuite été faite entre les impacts réellement observés et ceux prédits par le modèle. Les résultats obtenus montrent que le modèle arrive à indiquer quelles parties du code, objet ou aspect, seront impactées avec une précision plus qu'acceptable. Une implémentation du modèle permettrait de supporter une analyse d'impact dans un environnement réel de développement.

CHANGE IMPACT ANALYSIS IN ASPECT ORIENTED SOFTWARES (AOS): DEVELOPMENT OF AN IMPACT MODEL

Fabrice DÉHOULÉ

ABSTRACT

Aspect oriented programming (AOP) is a programming paradigm that is complementary to object oriented programming (OOP). It allows to put together all the crosscutting concerns of the object oriented code into modules named aspects. Those aspects have the distinction of being completely invisible to the object code and are introduced into it during a process called weaving. The relation between the object code and the aspect code brings a new dimension in change impact analysis. Techniques used in object-oriented systems cannot be directly used for the aspect oriented programs because of new dependencies between the object code and aspect code. We propose, in this context, to develop an impact model for aspect oriented systems. This model aims to predict the impact of any changes brought as well to the object code as in the aspect code. In order to support our approach, we conducted tests on various aspect oriented programs. Those tests focused on the impacts that occurred in the aspect code after a change was made in the object code, then impacts occurring in the aspect code after changes made in aspects modules, and finally impacts occurring within the object code after a change was made in aspect part of system. We then make a comparison between the impacts actually observed and those predicted by the model. The results we obtained show that our model is able to predict which parts of the code, object or aspect, will be impacted with a more than acceptable accuracy. An implementation of the model would support change impact analysis in a real development environment.

REMERCIEMENTS

La réalisation de cette étude n'aurait pu se faire sans l'aide et le support de certaines personnes.

Mes remerciements vont d'abord à Linda Badri et Mourad Badri, tous deux professeurs au Département de Mathématiques et d'Informatique de l'Université du Québec à Trois-Rivières, qui ont été mes guides durant la réalisation de ce mémoire. Un merci particulier à Linda, qui m'a pris sous son aile tout le long de mon parcours académique au Département.

Ce travail n'aurait pas non plus été possible sans mes parents, Omer et Henriette DÉHOULE, qui, par leurs sacrifices, prières et encouragements, m'ont donné la chance de pouvoir avancer dans mes études.

Je tiens enfin à remercier mes frères, ma sœur pour leurs encouragements. Un merci tout particulier à ma femme, qui savait me motiver tout au long de mes études.

Que DIEU vous bénisse tous.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

CHAPITRE 1:
ANALYSE DE L'IMPACT DES PROGRAMMES ORIENTÉS ASPECT: PROBÉLMATIQUE, OBJETCIFS ET DÉMARCHE

I. Introduction

Les applications logicielles sont des collections de modules qui travaillent ensemble afin de procurer des fonctionnalités définies par un ensemble d'exigences [6]. Les techniques de développement de logiciels ont toujours été conduites par le besoin de pouvoir séparer ces fonctionnalités ou préoccupations [14]. C'est ainsi que la programmation orientée objet (POO) fut introduite avec pour objectif d'organiser en différents modules les données d'une application et leurs traitements associés [7], donc de pouvoir séparer le mieux possible les fonctionnalités du logiciel. Cependant, la POO présente une limite. En effet, certaines préoccupations sont difficiles à factoriser; il s'agit de préoccupations transversales [14]. Ces préoccupations transversales correspondent à du code dispersé dans différents modules (code scattering) ou du code enchevêtré dans un même module (code tangling) [14]. Cette dispersion de code va mener à un plus grand risque d'erreur, et le code sera plus difficile à débuguer, à ré-usiner (refactoring), à documenter et à réutiliser [6].

La programmation orientée aspect (POA) fut introduite pour pouvoir palier au problème des préoccupations transversales. La POA est en fait un paradigme de programmation qui vient en complément à la POO afin d'obtenir des programmes mieux structurés et plus clairs [7]. Ce paradigme a été proposé pour permettre de produire des systèmes avec une meilleure modularité [13][15] et qui devraient être plus faciles à comprendre [13]. Ainsi, la POA va regrouper les préoccupations transversales dans des modules appelés « aspects » [11] [13]. Ces aspects seront ensuite injectés dans le code de base lors d'un processus appelé « weaving » (tissage) [10][14][15]. Les programmes développés en programmation aspect vont donc présenter deux parties: une partie regroupant le programme de base développé en objet et une autre partie regroupant les aspects.

Lors de son évolution, un programme est amené à subir de nombreux changements. Les changements sont des opérations essentielles permettant de rajouter de nouvelles fonctionnalités, corriger des bugs ou encore modifier l'implémentation du programme si les spécifications n'ont pas été correctement appliquées [3]. Deux des activités de base de la maintenance des systèmes logiciels sont la compréhension du système et l'évaluation des effets d'un changement [8]. Il va

donc falloir bien comprendre le logiciel et ensuite évaluer de la meilleure façon les effets des différents changements qui seront apportés.

L'analyse d'impact des changements est la partie de la maintenance qui permet aux développeurs d'évaluer les effets possibles d'un changement donné apporté au code source d'un programme [8]. Cette étape de la maintenance est importante car elle permet également d'estimer le coût et les bénéfices possibles qu'une modification aura sur le reste du programme après son implémentation [19].

II. Problématique

À cause des effets hautement intrusifs de la POA sur le code principal, il ne sera pas évident d'estimer l'impact d'un changement dans le système en entier, et il sera dur de comprendre les effets de propagation que ce changement pourrait introduire [13]. En effet, le système obtenu à la suite d'un tissage aura un résultat et une structure pouvant être largement différents du programme de base [11]. Cela pourrait rendre le programme complexe et rendre sa compréhension plus difficile [16], et donc rendre sa maintenance plus ardue [11]. Si la POA est utilisée de façon peu disciplinée, il peut être difficile de pouvoir prédire si un changement effectué dans le code de base et dans le code aspect aura des effets dans le système [11]. De plus, Störzer, dans [18], met en évidence d'autres problèmes. Il mentionne le fait que les aspects et le code de base sont découplés au niveau de la syntaxe, il va falloir donc connaitre les dépendances du système. Il indique aussi que les aspects peuvent interférer entre eux, interférences qui peuvent être difficiles à résoudre. Il ne manque pas non plus d'indiquer que l'évolution du code de base peut modifier ou casser la sémantique des aspects, car ceux-ci reposent sur le code de base.

Ces changements apportés aux programmes orientés aspect peuvent se faire dans le programme de base ou au niveau des aspects. En raison des deux parties d'un programme orienté aspect à savoir la partie orientée objet (programme de base) et la partie des aspects, il y aura quatre possibilités d'impact: (i) impacts de changements introduits dans la partie orientée objet vers elle même (Objet - Objet), (ii) impacts de changements apportés dans la partie objet vers la partie aspect (Objet - Aspect), (iii) impacts de changements apportés dans la partie aspect vers elle même (Aspect - Aspect), (iv) impacts de changements insérés dans la partie aspect vers la partie objet (Aspect - Objet).

Même si la POA est une extension de la programmation orientée objet, nous ne pouvons pas nous limiter aux techniques d'analyse d'impact appliquées aux programmes orientés objet. La mise en lumière de nouvelles dépendances et le fait que les aspects peuvent modifier le flux de contrôle [16] rendent ces techniques peu appropriées. Malgré l'ampleur que prend de plus en plus la POA, de nombreuses recherches sur l'analyse d'impact des changements sont menées, mais la plupart portent sur les programmes procéduraux et orientés objet [3].

III. Démarche

Le travail que nous présentons se donne pour but de pouvoir arriver à une prédiction des impacts possibles suite à des modifications du programme dans un contexte Orienté Aspect. L'analyse d'impact que nous proposons porte à la fois sur les impacts ayant lieu sur le code objet et sur le code aspect. Nous avons établi un modèle d'analyse d'impact des changements pour les programmes orientés aspect. Ce modèle fait suite au MICJ [5] qui s'intéresse aux impacts qu'auront des modifications introduites au niveau du code objet sur lui même (Objet - Objet). Notre modèle mettra l'accent sur les trois autres types d'impacts à savoir Objet-Aspect, Aspect-Aspect et Aspect-Objet.

Dans le chapitre 2 de ce mémoire, nous allons parler de la programmation orientée aspect, présenter ses origines, son lien avec la programmation orientée objet, ainsi que les différents éléments qui la composent. Dans le chapitre 3, il s'agira de présenter les enjeux de la maintenance des systèmes logiciels en général, et la maintenance de programmes orientés aspect en particulier. Nous parlerons, également, de l'importance de l'analyse de l'impact dans le programmes orientés aspect. Au niveau du chapitre 4, nous présenterons notre modèle, avec des détails sur les règles d'impact et comment les utiliser. Le chapitre 5 sera consacré aux expérimentations que nous avons faites, et une interprétation des résultats. Enfin, nous conclurons ce mémoire au niveau du chapitre 6 qui sera suivi des annexes au chapitre 7.

CHAPITRE 2:
LA PROGRAMMATION ORIENTÉE ASPECT

I. Présentation

1. Limites de la programmation orientée objet

La programmation orientée objet (POO) est un paradigme de programmation permettant de séparer les préoccupations d'un programme [6] [7], offrant ainsi une bonne modularité du code [18]. Cependant, il arrive que du code soit répété dans différentes parties du programme. De plus, ce code ne peut pas être généralisé ni par l'héritage, ni par le polymorphisme [22]. Il s'agit de préoccupations transversales.

2. Les préoccupations transversales

Une préoccupation est une exigence ou une fonctionnalité d'un programme [6]. C'est ainsi qu'on distingue deux sortes de préoccupations:

- les préoccupations essentielles (core concerns) qui concernent les fonctionnalités principales d'un module [23]. Chacune de ces préoccupations peut être implémentée dans un module distinct [14][25]
- les préoccupations transversales qui concernent les préoccupations qui se retrouvent dans différents modules du programme et sont impossibles à modulariser [23][25].

Comme indiqué dans [21], ces préoccupations transversales concernent généralement des opérations comme la synchronisation, la sauvegarde de données, la journalisation des états du programme, l'authentification, la persistance des données ou encore la sécurité et divers design patterns [26].

Les préoccupations transversales vont engendrer deux problèmes majeurs:

- La dispersion du code: la préoccupation transversale est implémentée dans différents modules.
- L'enchevêtrement du code: plusieurs préoccupations transversales se retrouvent dans le même module.

Ces deux problèmes vont non seulement altérer la modularité du système, le rendant plus difficile et plus couteux à comprendre, à développer, à maintenir et à faire évoluer [24], mais aussi peu réutilisable par d'autres programmes [21].

Exemple de préoccupation transversale

Soit une classe Compte (figure 1) ayant pour objectif de permettre la gestion des transactions à savoir le dépôt, le retrait et l'obtention du solde.

```
public class Compte
{
        private double solde;

        public Compte(double montant)
        {
                solde = montant;
                System.out.println(("Solde du compte: "+solde);
        }

        public void depot(double montant)
        {
                solde += montant;
                System.out.println("Solde du compte: "+solde);
        }

        public double retrait(double montant)
        {
                if(solde >= montant)
                        solde -=montant;
                else
                        System.out.println("Solde insuffisant");

                System.out.println("Solde du compte: "+solde);
                return solde;
        }

        public double getSolde()
        {
                System.out.println("Solde du compte: "+solde);
                return solde;
        }
```

Figure 1 - Classe Compte

Cette classe a pour but principal la modification du solde du compte lors de la création de celui-ci, ou encore lors d'un retrait ou d'un dépôt. Cependant, il est nécessaire d'afficher le solde à l'utilisateur. Une solution est de mettre le code d'impression dans la classe Compte afin que le solde soit affiché après chaque transaction. Ce code est mis en surbrillance dans l'illustration ci-dessus.

Ce code d'affichage du solde de compte, qui pourrait être aussi du code de journalisation permettant de garder un historique des transactions, ne doit pas vraiment être mis dans cette classe. Il s'agit ici d'une préoccupation transversale.

3. But de la programmation orientée aspect

Développée par la société XEROX au début des années 90 [21], la POA a pour but de rassembler toutes les préoccupations transversales d'un programme dans des modules différents du code source appelés aspects. Le code contenu dans ces aspects sera ensuite injecté dans le code objet grâce à un processus appelé tissage.

II. Mécanisme de la programmation orientée aspect

Avant de commencer à parler d'aspects, il faut d'abord détecter les préoccupations transversales du programme. Ce sont ces préoccupations transversales qui vont être retirées du code orienté objet et mis dans des modules appelés aspect. En ce basant sur le code de la Figure1, le code d'affichage du solde étant une préoccupation transversale, il sera mis dans un aspect.

Rendues dans les aspects, ces préoccupations transversales seront associées à un ou plusieurs éléments du code objet. Cet élément peut être une classe, une méthode ou un attribut.

De plus, le mécanisme de la POA permet de décider à quel moment exécuter le code correspondant à la préoccupation transversale. Ainsi, grâce à la POA, ces préoccupations transversales seront exécutées avant, pendant ou après l'exécution de l'élément du code objet auquel elles ont été associées.

Enfin, lors de la compilation du programme, le code contenu dans les aspects sera introduit dans le code objet grâce à un processus appelé tissage (weaving). La figure 2 suivante illustre le processus du tissage.

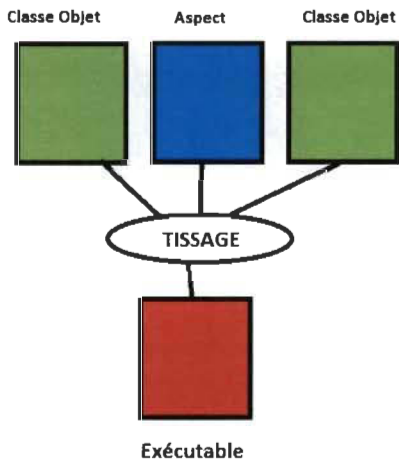

Figure 2 - Tissage

Ainsi, on aura pour l'exemple de la figure 1 le code aspect suivant:

```
public aspect AspectCompte
{
        pointcut pcAffichage():
                execution (double Compte.depot(double )) ||
                execution (double Compte.retrait(double )) ||
                execution (double Compte.getSolde());

        double around():pcAffichage()
        {
                double solde = proceed();
                System.out.println("Solde du compte: "+solde);
                return solde;
        }
}
```

Figure 3 - Aspect AspectCompte

Dans la figure 3, on peut voir que la préoccupation transversale revenant plusieurs fois dans le code objet n'est mise qu'une seule fois dans le code aspect. Cette instruction (en jaune sur la figure 3) sera introduite dans le programme après le tissage, et elle est déclenchée durant l'exécution des méthodes de la classe Compte en bleue sur le figure 3.

III. Éléments de la programmation orientée aspect

1. AspectJ

Afin de mettre en œuvre le mécanisme de la POA, G Kiczales conçut en 1998 AspectJ dans les locaux de la société XEROX [7]. AspectJ joue le rôle d'extension orientée aspect au langage Java [8].

AspectJ sera le langage utilisé dans ce mémoire pour l'implémentation de notre modèle d'impact.

2. Les aspects

Les aspects sont des modules permettant l'implémentation des préoccupations transversales [15]. Les modules aspects seront produits comme des classes objet dans le programme, et c'est là que seront définis tous les éléments relatifs à la programmation orientée aspect, à savoir les points de jointure, les coupes, les advices, etc.

3. Les points de jointure (pointcuts)

Les points de jointure sont des positions précises dans le programme objet. C'est au niveau de ces points de jointure que les aspects vont venir se greffer au code objet. Ces points de jointure vont concerner des classes, des méthodes, des constructeurs, des attributs, des exceptions, des

interfaces ou toute autre instruction du programme de base. Les différents points de jointure possibles sont :

- Les appels ou l'exécution de méthodes
- Les appels de constructeurs
- L'initialisation d'une classe
- La modification de la valeur d'un attribut
- La lecture de la valeur d'un attribut
- La gestion d'une exception

Exemples:

- Pointcut avec exécution de méthode

```
pointcut pcAffichage(): execution (double Compte.depot(double));
```

- Pointcut avec appel de constructeur

```
pointcut pcConstructeur():initialization(Compte.new(double));
```

4. Les coupes (pointcuts)

Une coupe contient un ou plusieurs points de jointure spécifiques. Une coupe permet de spécifier le contexte dans lequel son ou ses points de jointure seront captés. Par exemple, on pourrait atteindre une méthode au moment de son exécution, un attribut au moment de sa modification, une exception au moment de sa gestion.
Il peut arriver qu'un point de jointure appartienne à plusieurs coupes. Dans ce cas, il va falloir définir les priorités afin qu'il n'y ait pas de confusion.

```
pointcut pcAffichage():execution (double Compte.getSolde());
```

5. Les advices

Les advices sont toujours liés à une coupe. Si une coupe permet de définir le contexte dans lequel un point de jointure est atteint, l'advice permettra de spécifier l'ensemble des instructions à exécuter lorsque le contexte de capture d'un point de jointure est atteint. Aussi, un advice permet d'indiquer à quel moment ces instructions seront exécutées, à savoir avant, après, ou pendant que le point de jointure est atteint. Ce sont les lignes de codes se trouvant dans le code aspect qui vont être greffées au code objet lors du tissage. On distingue les trois advices suivants:

- **Before**

  Indique que le code dans l'advice va s'exécuter avant que le point de jointure ne soit atteint. La figure 4 nous présente un exemple d'avice before.

```
pointcut pcGetSolde():
        execution(Compte.getSolde());


before():pcGetSolde()
{
        System.out.print("Obtention du solde");
}
```

Figure 4 - Exemple Advice Before

- **After**

  Indique que le code dans l'advice va s'exécuter après que le point de jointure ne soit atteint. Les advices after permettront également d'utiliser la valeur de retour d'une méthode grâce à l'instruction ***returning.*** Enfin, ces advices after sont les seuls qui sont utilisés quand une gestion d'exception est atteinte. On peut voir dans les figures 5 et 6 deux illustrations d'advices after

```
pointcut pcConstructeur():
            initialization(Compte.new(double));


       after():pcConstructeur()
       {
            System.out.print("Création d'un nouveau compte ");
       }
```

Figure 5 - Exemple Advice After

```
pointcut pcRetrait():
            execution (double Compte.retrait(double ));


      after()returning:pcRetrait()
      {
            System.out.print("Retrait effectué avec succès");

      ]
```

Figure 6 - Exemple Advice After returning

- **Around**

  Indique que le code dans l'advice va s'exécuter durant l'exécution du point de jointure. Il constitue le plus compliqué mais en même temps le plus puissant des advices [6]. Il est en mesure de dire si oui ou non le point de jointure s'exécute et il pourra exécuter un tout autre code. En effet, il contient l'instruction ***Proceed()*** (cf. figure 7). Cette instruction permet de lancer le code du point de jointure. Si certaines conditions ne sont pas réunies, l'instruction ***Proceed()*** n'est pas appelée et un autre code est exécuté éventuellement. Cela peut se produire si par exemple les paramètres passés à une méthode ne sont pas valides. La méthode ne sera donc pas exécutée et un message d'erreur pourrait être affiché.

```
pointcut pcAffichage():
                execution (double Compte.retrait(double;

double around():pcAffichage()
{
        double solde = proceed();
        System.out.println("Solde du compte: "+solde);
        return solde;
}
```

Figure 7 Exemple Advice Around

6. Les déclarations inter-types

Les déclarations inter-types vont permettre de rajouter des méthodes ou des attributs à une classe objet. Ces attributs ou méthodes ne seront visibles par la classe qu'après le tissage.

Les déclarations inter-types permettront également de modifier l'héritage d'une classe. Grâce à ces déclarations, on va pouvoir dire qu'une classe A hérite d'une classe B même si cette classe A héritait déjà d'une autre classe C

Aussi, une déclaration inter-type va pouvoir indiquer qu'une classe implémente une ou plusieurs interfaces.

7. L'héritage dans la programmation orientée aspect

La programmation orientée aspect prend en charge deux types d'héritage :

- Un aspect hérite d'une classe abstraite

  Ce type d'héritage permettra à l'aspect de pouvoir hériter des attributs et des méthodes d'une classe objet et de pouvoir redéfinir des méthodes abstraites
- Un aspect hérite d'un aspect abstrait

  Un aspect pourrait hériter d'un autre aspect à condition que ce dernier soit abstrait. Ainsi l'aspect descendant pourra non seulement redéfinir les méthodes de l'aspect parent, mais

il sera également en mesure de redéfinir les pointcuts abstraits comme illustré dans la figure 8.

```
public aspect Aspect_Carre extends Aspect_Figure
{

   before(): pcPerimetre()
   {
        System.out.println("Périmètre du
                            carré");
        methode();

   }
}
```

```
public abstract aspect Aspect_Figure
{
    protected int att=0;

    protected pointcut pcPerimetre():
        execution(void Carre.calculPerimetre());

    public pointcut pcSurface():
        execution(void Cercle.calculSurface()) ||
        execution(void Carre.calculSurface());

    before(): pcSurface()
    {
        System.out.println("Calcul de la surface");
    }

    public void methode()
        {
                //code
        }
}
```

Figure 8 - Exemple Advice After

8. Les déclarations

La programmation orientée aspect permet de faire les déclarations suivantes:

Declare parents:

Permet d'indiquer à une classe qu'elle hérite d'une autre ou qu'elle doit implémenter une interface.

```
aspect BoundPoint {
        declare parents: Point implements Serializable;
}
```

Figure 9 - Exemple declare parent

Dans la figure 9, il est indiqué à la classe "Point" d'implémenter l'interface "Serializable".

Declare warning et Declare error:

Dans certains cas, on veut indiquer qu'un point de jointure ne doit jamais être atteint. Ces deux déclarations seront utilisées lorsqu'un point de jointure est atteint dans le programme. Il va alors y avoir un message d'avertissement ou d'erreur qui sera indiqué.

```
public aspect FacadePolicyEnforcement {

        pointcut callsToEncapsulatedMethods():
                call(* (Decoration || RegularScreen || StringTransformer).*(..));


        pointcut facade(): within(OutputFacade);

        declare warning: callsToEncapsulatedMethods() && !facade():
                "Calling encapsulated method directly - use Facade methods instead";
}
```

Figure 10 - Exemple declare warning

Dans la figure 10, le point de jointure du pointcut "callsToEncapsulatedMethods" ne devrait pas être atteint si on n'a pas atteint le point de jointure contenu dans le pointcut "facade"

Declare precedence:

Dans un programme orienté aspect, il peut arriver que des points de jointure appartenant à différents aspects pointent sur la même méthode par exemple. Dans ce genre de situations, il nous est impossible de savoir lequel de ces points de jointure sera considéré en premier, ainsi que l'ordre dans lequel les autres suivront. Le mécanisme de déclaration de précédence permet d'indiquer dans quel ordre les aspects seront appelés (cf. figure 11).

```
        declare precedence: Aspect1, Aspect2;
```

Figure 11 - Exemple declare precedence

CHAPITRE 3:
ANALYSE DE L'IMPACT DES PROGRAMMES ORIENTÉS ASPECT

I. Maintenance

La maintenance d'un logiciel est l'ensemble des opérations visant à apporter des modifications à un programme après sa livraison afin d'apporter des corrections, de le faire évoluer, d'ajouter de nouvelles fonctionnalités ou encore le rendre plus performant. Dans le cycle de vie d'un logiciel, la maintenance détient une place assez importante [5]. Son importance est d'autant plus accrue à cause du fait que les systèmes actuellement produits sont de plus en plus complexes et volumineux [5]. De plus, la maintenance est une étape assez couteuse dans la vie d'un logiciel [20].

II. Analyse d'impact

L'analyse d'impact de changement vient soutenir la maintenance afin de la rendre moins ardue. Cette analyse a pour mission de prédire quelles parties du code seront touchées suite à une modification [8]. Bohner and Arnold dans [19] la définissent comme étant l'identification des conséquences d'un changement, ou l'estimation de qu'est ce qui doit être modifié afin que ce changement se fasse. L'analyse de l'impact est d'autant plus importante car elle permet d'estimer le coût de développement d'un projet, d'aider à la gestion des changements, et permet de garder le système stable [19].

L'analyse de l'impact peut se faire après l'introduction du changement; on parle alors de post-analyse. Dans ce cas-ci, le changement a déjà été effectué; il en résultera donc deux versions du même système qu'on pourra comparer afin de voir quelles sont les conséquences des changements apportés. Ce type d'analyse est utile pour les tests de régression [5], mais aussi pour avoir les traces des effets de propagation (ripple effect), et prévoir les cas de test [27].

L'analyse d'impact peut également se faire avant l'introduction du changement. Il s'agit là de l'analyse prédictive [5]. Les impacts de changement seront mis en évidence avant que le changement ne soit effectué. Cela aura pour avantage d'être plus sure, de permettre aux développeurs de pouvoir par exemple choisir une solution parmi d'autres, estimer si oui ou non le changement vaut la peine d'être effectué, ou encore ils auront une meilleure idée des efforts nécessaires à la mise en œuvre du changement ainsi que des coûts qui seront engendrés [8].

Deux techniques principales sont utilisées pour effectuer l'analyse d'impact :

- **Les techniques d'analyse statiques :**

  Ces techniques analysent la syntaxe et les dépendances sémantiques du programme. Elles se basent pour la plupart du temps sur des représentations du système telles que les graphes d'appel, les graphes de contrôle de flot, etc. Le problème de ces techniques statiques est que le volume de données à analyser peut être énorme, rendant ainsi l'analyse d'impact difficile à réaliser [27].

- **Les techniques d'analyse dynamiques :**

  Elles se basent sur les informations recueillies durant l'exécution du programme et présentent l'avantage d'être plus pratiques et plus précises [27].

Comme indiqué dans [28], ces deux techniques présentent malgré tout le même problème, à savoir donner des résultats erronés. En effet, il se peut que des parties non impactées fassent parties de l'ensemble des parties impactées. On parle alors de « faux-positifs » (false positive). Également, l'ensemble obtenu peut en omettre certaines; on parle alors de « faux négatifs » (false negative).

Une analyse d'impact de changement se base sur une bonne compréhension du programme [8]. En effet, il est nécessaire de comprendre les relations entre les différentes entités du système pour être en mesure de prédire les effets.

Au niveau de la POO, les différents modules, à savoir les classes, sont liées entre elles. Grâce à ces dépendances, il est possible de retracer les impacts dus à un changement apporté dans le système. Ce changement peut concerner la suppression d'une classe. La modification d'une méthode, la suppression d'un attribut, etc. De nombreuses techniques [8][29] se basent justement sur les graphes de dépendances pour effectuer leur analyse d'impact.

Avec l'avènement de la POA, une analyse de l'impact des changements dans ces programmes sera nécessaire. La POA introduisant de nouvelles dépendances, la compréhension du système est un peu plus difficile quand on sait que les aspects sont invisibles à la partie orientée objet et que l'exécution du programme peut être complètement modifiée après le tissage. Les approches d'analyse de l'impact présentement utilisées, en particulier les techniques d'analyse d'impact de programmes orientés objet, ne sont pas applicables directement sur des programmes orientés aspect. Il est donc nécessaire de développer de nouvelles techniques tenant compte des spécificités du paradigme aspect.

III. Approches

Différentes approches ont été proposées pour prédire les impacts de changements effectués au niveau d'un programme orienté aspect.

Dans [11], les auteurs ont élaboré un graphe de contrôle aspect inter-procédural (Inter-procedural Aspect Control Flow Graph). Ce graphe a pour but de représenter les relations entre les méthodes des classes et les advices des aspects (cf. Figure 12). Il permet également de montrer l'endroit où le code d'un advice sera inséré dans une méthode lors du tissage et permet également de faire une analyse d'impact de façon statique ou dynamique. Cependant, seule l'analyse statique fut traitée.

Ce graphe de contrôle est composé de sous graphes, chacun représentant le graphe de contrôle d'une méthode ou d'un advice. Les nœuds de ce graphe représentent les éléments suivants:

- Le point d'entrée (Entry node, EN), qui est le premier nœud d'un graphe
- Le nœud final (Final node, FN), qui est le dernier nœud d'un graphe
- Le nœud de sortie (Exit Node, XN) qui est le point de sortie du graphe. Cela peut survenir dans le cas d'un "return" pour les méthodes ou à cause d'un " ***Proceed()*** " qui n'a pas été exécuté dans un advice Around.
- Les nœuds concernant les déclarations (statement node, SN) représentant les déclarations normales
- Les nœuds pour les points de jointure dans le code objet (Joint-point shadow Node, JN) qui sont des nœuds fictifs qui représentent les points de jointure où doit être inséré le code des advices.

L'un des avantages procurés par le graphe selon l'auteur est de permettre de gagner du temps lors des étapes de maintenance, à peu près 20% moins de temps. Cependant, le graphe ne prend en compte ni la gestion des exceptions, ni les déclarations inter-types, ni les initialiseurs statiques; et il peut en résulter un grand nombre de nœuds, rendant l'analyse un peu plus complexe.

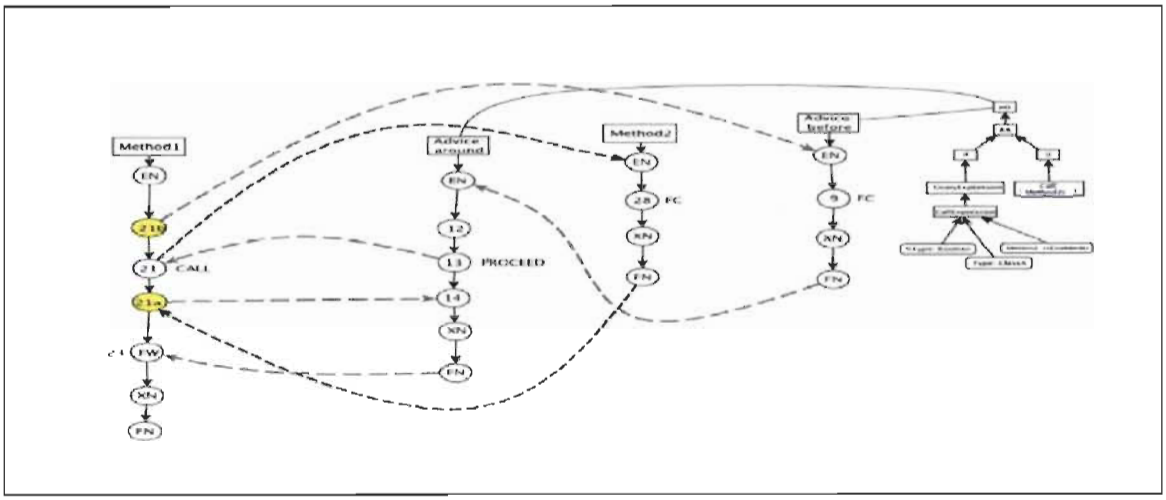

Figure 12 - Graphe de contrôle aspect inter-procédural

Dans [2] Les auteurs ont développé un algorithme (cf. figure 13) ayant pour but de lister les impacts. Cet algorithme commence par générer l'arbre abstrait syntaxique, puis génère à partir de cet arbre le graphe de flux de contrôle. Ensuite, il génère un graphe d'appels des méthodes ou des advices. Un graphe de dépendances est par la suite conçu pour chacune des méthodes en s'appuyant sur le graphe d'appels précédemment généré.

Une fois tous ces graphes conçus, l'algorithme établit une liste de point de départ d'impact pour chacun des aspects. Puis, à partir de chaque point de départ d'impact correspondant à un point de tissage d'aspect (weaving point), on marque les parties impactées par les aspects.

Les inconvénients de cette analyse d'impact sont :

- Elle se fait après tissage des aspects
- Elle ne concerne que les impacts des aspects sur le programme source et ne gère par les impacts du programme source sur les aspects

```
program Impact analysis of weaving
  (1) generate abstract syntax tree and symbol table
  (2) generate control flow graph from the abstract syntax tree
  (3) generate call graph of method/advice relationships
  (4) generate program dependence graph (PDG)
  for all aspects
    (5) list impact starting points for each aspect
    for all weaving points of each aspect
      for all impact starting points at each weaving point
        (6) mark impacted statements by traversing PDG
      endfor
    endfor
  endfor
endprogram
```

Figure 6. Algorithm for the impact analysis

Figure 13 - Algorithme d'analyse d'impact de [2]

La technique d'analyse d'impact proposée dans [3] a pour objectif d'identifier les parties du programme source affectées lorsque celui-ci est modifié.

Cette technique s'appuie sur les changements atomiques qui capturent les différences sémantiques entre deux versions d'un programme. Ces changements atomiques se situent au niveau du programme source, mais également dans le code aspect (cf. figure 14).

On entend par changement atomique des opérations comme ajout/suppression/modification de méthode, d'aspect, de pointcut, de classe ou d'attribut.

Ainsi, pour déterminer quelles parties du programme risquant d'être affectée par une modification (changement atomique), les auteurs dressent une liste des changements apportés. Puis un graphe d'appels du programme est généré. Les fragments de code affectés par une modification seront les nœuds du graphe qui sont reliés au nœud modifié. Il s'agit en fait de la fermeture transitive de chaque nœud modifié.

Encore une fois, cette technique d'analyse d'impact a le désavantage de ne pas être prédictive puisqu'il faut que les modifications soient apportées avant de déterminer quels bouts de code seront affectés.

Abbreviation	Atomic Change Name
AF	Add a field
DF	Delete a Field
AM	Add an Empty Method
DM	Delete an Empty Method
CM	Change Body of Method
AC	Add an Empty Class
DC	Delete an Empty Class
LC	Change Virtual Method Lookup

Abbreviation	Atomic Change Name
AA	Add an Empty Aspect
DA	Delete an Empty Aspect
INF	Introduce a New Field
DIF	Delete an Introduced Field
CIFI	Change an Introduced Field Initializer
INM	Introduce a New Method
DIM	Delete an Introduced Method
CIMB	Change an Introduced Method Body
AEA	Add an Empty Advice
DEA	Delete an Empty Advice
CAB	Change an Advice Body
ANP	Add a New Pointcut
CPB	Change a Pointcut Body
DPC	Delete a Pointcut
AHD	Add a Hierarchy Declaration
DHD	Delete a Hierarchy Declaration
AAP	Add an Aspect Precedence
DAP	Delete an Aspect Precedence
ASED	Add a Soften Exception Declaration
DSED	Delete a Soften Exception Declaration
AIC	Advice Invocation Change

Figure 14 - Listes des changements atomiques au niveau du code source et du code aspect

Les auteurs dans [4] sont les rares à ne pas avoir recours au tissage pour leur analyse d'impact. Leur technique se base sur le slicing d'un programme orienté aspect.

Ils commencent d'abord par établir un graphe de dépendances système orienté aspect (Aspect-Oriented System Dependence Graph) formé de plusieurs graphes de dépendances de modules (Module Dependence Graph) interconnectés. On entend par module une méthode du programme source ou un advice du code aspect. Chacun des graphes de dépendances de module est composé de sommets représentant les déclarations dans le programme et d'arcs représentant soit les dépendances de contrôle (conditions pour l'exécution du code) soit les dépendances de données (le flux de données à l'intérieur d'un module).

Ensuite, pour une modification donnée, ils font le slicing arrière en parcourant chacun des sommets du graphe de dépendances système orienté aspect et répertorient chaque sommet rencontré. Un slicing arrière sera fait également pour chacun des sommets rencontrés.

Une fois le slicing arrière fini, un slicing avant est opéré selon le même principe. Les auteurs n'ont cependant développé aucun outil permettant de visualiser les résultats de leur technique d'analyse.

CHAPITRE 4:
MODÈLE D'IMPACT

I. Objectifs

D'une manière générale, notre modèle d'analyse a pour objectif de faciliter la maintenance de logiciels grâce à une analyse d'impact. Plus précisément, notre modèle permet de prédire les différentes parties d'une application orientée aspect qui seront affectées suite à un changement apporté au programme. Le changement pourra provenir aussi bien de la partie objet que de la partie aspect.

Afin de pouvoir mettre en évidence notre modèle, nous avons utilisé AspectJ pour la réalisation de nos expérimentations.

II. MICJ

Le Modèle d'Impact du Changement pour Java (MICJ) fut développé par N. Joly, L Badri et M. Badri dans [5]. Ce modèle d'impact prend en compte les impacts du code objet vers le code objet. Comme ils l'ont mentionné, le MICJ permet de faire une analyse prédictive d'impact, ainsi qu'une post-analyse. Cependant, l'accent fut mis sur l'analyse prédictive. C'est un modèle d'impact qui permet de faire une analyse d'impact en cascade, répondant ainsi au problème d'effet de propagation des impacts de changement.

1. Modèle de Chammun

Afin de pouvoir concevoir le MICJ, les auteurs se sont inspirés du modèle de Chaummun [30] qui permet de faire une analyse prédictive de l'impact de niveau classe. Ce modèle permet d'indiquer les classes qui seront impactées en fonction de leur relation avec la classe où le changement a eu lieu. Les relations entre classes sont l'association (A), l'agrégation (G), l'héritage (H) et au sein de la classe où il y a le changement, un changement local (L) et l'invocation (I).

Même si le modèle de Chaumun est complet, il présente cependant une lacune, c'est un modèle de niveau classe. Cela signifie que pour un changement à l'intérieur d'une classe, il ne va indiquer que les classes qui seront impactées en indiquant uniquement leurs relations. Il n'indiquera pas quelles parties à l'intérieur des classes impactées seront affectées. Par exemple, lorsque la visibilité d'un attribut de classe va passer de "protégé" à "privé", le modèle de Chammun va indiquer que les classes impactées sont celles qui ont une relation d'association et

d'héritage avec la classe où a eu lieu le changement; on ne saura pas quelles parties au sein de ces classes impactées seront affectées.

Le fait que le modèle de Chammun ne s'arrête qu'au niveau des classes a aussi pour désavantage de ne pas permettre une analyse en cascade (précise). Étant donné qu'il s'arrête au niveau des classes, on ne saura pas quel effet de propagation il aura sur le reste du programme.

2. Relations du MICJ

Le MICJ propose des règles d'impact qui ont pour but d'indiquer quelles parties du système vont être impactées suite à un changement. Ces règles d'impact se basent sur les relations entre les classes. Quatre relations ont donc été distinguées:

- **L'association**
  On parle d'association entre deux classes A et B si la classe A contient un attribut de type B ou encore si une méthode de la classe B a un argument de type A. L'association est également matérialisée par le fait qu'une méthode de la classe A instancie un objet de la classe B ou une méthode de la classe A retourne un objet de type classe B.
- **L'héritage** (hérite de)
  Une classe A hérite d'une classe B
- **L'héritage ascendant** (fait hériter)
  Une classe A est la classe mère d'une classe B
- **La pseudo-relation locale** (la classe elle-même)
  Relations au sein d'une même classe

3. Changements structuraux et non-structuraux

Le MICJ se base sur les changements atomiques pour mener à bien son analyse d'impact. On entend par changement atomique tout changement qui ne peut pas être décrit par plus d'un changement atomique. Comme indiqué par les auteurs, le changement de signature d'une méthode ne serait pas un changement atomique dans le MICJ parce qu'il pourrait être décrit par plusieurs autres changements atomiques du modèle tels que l'ajout d'un paramètre ou le changement de visibilité de la méthode. Les changements atomiques sont divisés en deux groupes

a) Les changements structuraux:

Ce sont les changements qui modifient la structure de la classe et qui sont visibles au niveau du diagramme de classes. Ces changements structuraux se situent à trois niveaux: au niveau des classes, au niveau des méthodes, au niveau des attributs. On peut citer comme exemple le retrait d'un attribut, l'ajout d'une méthode ou encore la suppression d'une classe. Au total, 48 changements structuraux furent alors relevés par les auteurs répartis comme suit: 9 changements au niveau des classes, 24 changements au niveau des méthodes et 15 changements au niveau des attributs.

b) Les changements non structuraux:

Les changements non-structuraux, eux, ne changent pas la structure d'une classe. Ils ne sont donc pas visibles au niveau du diagramme de classes. Ils sont par contre situés dans le corps d'une méthode. On peut prendre comme exemple le retrait d'un appel à une méthode. On dénombre 26 changements non structuraux.

4. Impacts certains ou incertains

Le MICJ introduit la notion de certitude ou d'impact certain. Un impact est dit certain si, suite à un changement, il est inévitable d'apporter des modifications au code afin que celui-ci puisse compiler sans erreur. Cela arrive lorsque par exemple nous supprimons un attribut. Il va falloir alors supprimer toute référence à cet attribut, dans le code, afin que le programme puisse s'exécuter sans problème. De même lorsque nous modifions les paramètres d'une méthode, il va falloir corriger tous les appels à cette méthode à travers le reste du programme.

L'incertitude d'un impact vient du fait qu'un changement apporté au code n'oblige pas à apporter des modifications. Cela arrive très souvent lors de l'ajout d'un attribut ou encore d'une méthode. Lorsque ceux-ci sont ajoutés, le programme peut être exécuté sans erreur, même si on ne leur fait pas appel (l'attribut ou la méthode). Il en est de même au niveau de l'ajout d'une classe.

Ainsi, le MICJ pourra, au travers de ses règles d'impact, indiquer si un impact est certain ou incertain, donnant ainsi plus de précision sur les effets du changement.

5. Les cibles

Les cibles ont la même fonction que les relations entre classes : spécifier les lieux potentiels d'un impact. Par contre, celles-ci donnent une précision supérieure à celle des relations entre classes et peuvent être utilisées en tant que condition sur la certitude de l'impact.

6. Analyse en cascade

Le MICJ est un modèle qui permet de faire une analyse d'impact en cascade. On entend par analyse en cascade, l'analyse d'impacts des impacts [5]. Lorsqu'on effectue un changement, cela aura des impacts sur le reste du programme. À leur tour, ces changements auront eux aussi des impacts sur le programme. L'analyse d'impact effectuée à l'aide du MICJ permettra donc d'avoir une bonne idée de cette propagation d'impacts (ripple effect).

7. Règles d'impact du MICJ

Le MICJ est composé de différents ensembles. C'est grâce à ces ensembles que les règles seront générées. C'est ainsi qu'on distingue tout d'abord l'ensemble des changements atomiques (CA) qui est lui même composé de l'ensemble des changements structuraux (CS) et l'ensemble des changements non structuraux (CNS).
Une règle d'impact est définie par des éléments de changement (EC). Ce sont ces éléments qui indiqueront comment et où est ce que les impacts se produiront. On obtiendra ainsi, pour chaque règle d'impact, un ensemble I d'éléments de changements appartenant à CA. Cet ensemble I représente l'ensemble des changements qui devront ou pourraient être apportés au programme. Chacun des changements structuraux aura une règle d'impact.

Les éléments de changement sont composés des éléments suivants:

- **Un ou plusieurs changements atomiques** qui appartiennent à CA. Ces changements atomiques sont liés par un ET ou un OU. Le ET indique que les changements atomiques doivent être apportés, et le OU indique qu'au moins l'un ou l'autre des changements atomiques doit être apportés.

- **Des indicateurs de relations (IR)** qui représentent une relation qui existe entre classes et qui permet de déterminer dans quelles classes peut se retrouver le EC.

- **Une précision (Pr)** est un élément *e* appartenant à l'ensemble des cibles *Ci*. Une précision permet de préciser où il peut y avoir impact parmi les classes correspondantes aux *IR* du *EC*. Cela a donc pour utilité de savoir où chercher le ou les changements atomiques du *EC* parmi et à l'intérieur des classes.

- **Un élément d'obligation (EO)** qui permet d'indiquer la certitude d'un ou de plusieurs changements atomiques appartenant à l'ensemble des éléments de changement. Ainsi, on pourra dire qu'un ou des changements sont certains de se produire ou pas.

- **Une condition(Co)** est un élément *e* appartenant à l'ensemble des cibles *Ci*. Elle s'applique sur le *EO* du *EC*. Si la *Co* est respectée, alors le *EO* est considéré tel que défini dans le *EC*. Dans le cas contraire, le *EO* est inversé.

III. Extension du MICJ à la programmation orientée aspect

Le modèle d'impact du changement que nous proposons se veut une suite du MICJ adaptée à la programmation orientée aspect et sera utilisé dans un contexte d'analyse prédictive. Compte tenu de l'introduction des aspects, le MICJ, tel que conçu pour la programmation orientée objet, ne sera plus efficace, puisqu'il ne traite que des relations entre classes. De plus, les relations présentées au niveau du MICJ devront être également étendues à la POA. C'est ainsi que nous aurons les relations suivantes:

- **L'association:**

  Dans un programme orienté aspect, un aspect S peut être lié à différentes classes. Cette relation se matérialise par des points de jointure ou par des déclarations inter-type. De plus l'association peut se matérialiser au niveau des advices quand un objet d'une classe A sera instancié au sein de cet advice. Enfin, on aura une association entre un aspect S et une classe A quand une méthode au niveau de S prendra en paramètre un objet de type A. la figure 15 qui suit nous donne un aperçu d'une association.

```
public aspect AspectCompte
{

    pointcut pcConstructeur():
        initialization(Compte.new(double));


    after():pcConstructeur()
    {
        System.out.print("Création d'un nouveau
        compte: ");
    }

    pointcut pcRetrait():
        execution (double Compte.retrait(double ));


    after()returning:pcRetrait()
    {
        System.out.print("Obtention du solde");
    }
```

```
public class Compte
{
    private double solde;

    public Compte(double montant)
    {
        solde = montant;
    }

    public double depot(double montant)
    {
        solde += montant;
        return solde;
    }

    public double retrait(double montant)
    {
        if(solde >= montant)
          solde -=montant;
        else
          System.out.println("Solde insuffisant");

        return solde;
    }
}
```

Figure 15 - Exemple d'association

- **L'héritage :**

  Un aspect S peut hériter d'une classe A, mais également un aspect S peut hériter d'un autre aspect P.

- **Les pseudo-relations locales:**

  Certains impacts peuvent se faire à l'intérieur de l'aspect dans lequel a lieu la modification par exemple. Cela arrive quand par exemple on modifie les paramètres d'une coupe. Il faudra donc modifier également les paramètres des advices liés à cette coupe.

Notre modèle va s'intéresser à trois familles d'impacts:

- Impacts objet vers aspect
- Impacts aspect vers aspect
- Impact aspect vers objet

1. Changements structuraux et non-structuraux

Notre modèle gardera les principes de changements structuraux et non-structuraux présentés par le MICJ. Ainsi, nous aurons de nouveaux changements structuraux représentés par la suppression d'une coupe (pointcut), ou encore la modification des paramètres d'une méthode déclarée dans un aspect. Les changements non-structuraux seront matérialisés par l'ajout d'une coupe, d'un aspect, d'une méthode ou d'un attribut.

2. Notion de certitude et d'incertitude

Dans notre modèle, nous auront également les notions d'impact certain ou impact incertain comme présenté par le MICJ. Dans le contexte de la POA, un impact certain sera matérialisé par exemple lorsqu'un pointcut est supprimé, les advices qui lui sont rattachés seront eux aussi supprimés. Il sera par contre incertain de modifier un advice après l'ajout ou la suppression d'un point de jointure à un pointcut.

3. Analyse en cascade

Tout comme le MICJ, notre modèle sera en mesure d'effectuer une analyse en cascade. Ainsi, quand une modification sera effectuée, nous pourrons voir quelle sera l'effet de propagation de cette modification dans le reste du programme.

Nous allons prendre un exemple afin d'illustrer l'analyse en cascade.

```
aspect BoundPoint {

  private PropertyChangeSupport Point.support = new PropertyChangeSupport(this);


  public void Point.addPropertyChangeListener(PropertyChangeListener listener){
     support.addPropertyChangeListener(listener);
  }

public void Point.addPropertyChangeListener(String propertyName,
                                            PropertyChangeListener listener){

     support.addPropertyChangeListener(propertyName, listener);
  }

pointcut setter(Point p): call(void Point.set*(*)) && target(p);


void around(Point p): setter(p) {
        String propertyName =
      thisJoinPointStaticPart.getSignature().getName().substring("set".length());
        int oldX = p.getX();
        int oldY = p.getY();
        proceed(p);
        if (propertyName.equals("X")){
      firePropertyChange(p, propertyName, oldX, p.getX());
        } else {
      firePropertyChange(p, propertyName, oldY, p.getY());
        }
  }


void firePropertyChange(Point p,String property,double oldval, double newval)
{
        p.support.firePropertyChange(property,new Double(oldval),
                                     new Double(newval));
  }
}
```

Figure 16 - Exemple d'analyse en cascade

Dans la figure 16, l'aspect "BoundPoint" contient une déclaration inter-type qui introduit dans une classe "Point" un attribut nommé "support" et de type "PropertyChangeSupport" (en jaune dans la figure 16).

Si cette déclaration inter-type est supprimée, les deux autres déclarations inter-types contenant l'attribut "support" seront impactées, ainsi que la méthode "firePropertyChange" où apparait cet attribut "support" (les occurrences de l'attribut "support" sont en vert sur la figure). Ces trois impacts constituent des impacts directs.

Cependant, la méthode "firePropertyChange" est utilisée par l'advice around (en bleu dans la figure 16). L'impact au niveau de la méthode "firePropertyChange" va entrainer à son tour un impact sur cet advice.

Notre modèle permettra dans un premier temps de déceler les premiers impacts directs survenus. Puis, pour chacun de ces impacts, on pourra savoir quels impacts ont eu sur le reste du programme.

4. Les règles d'impact

Les impacts traités par le MICJ sont des impacts considérés comme applicables. Certains impacts ne peuvent être détectés qu'en post-analyse. Ces changements ne peuvent pas être calculés. Il s'agit de changements comme l'ajout d'une classe. En effet, on ne peut pas prédire le nombre d'impact qu'aura l'ajout d'une classe puisque qu'on ne sait pas quelles classes seront liées à cette nouvelle classe. On ne pourra le constater qu'en post-analyse.
Notre extension du MICJ n'échappera pas à ce principe. Ainsi, les changements tels que l'ajout d'aspect ou d'un pointcut ne seront pas pris en compte.

4.1. Impacts objet vers aspect

Les aspects sont liés aux classes grâce aux points de jointure.

```
public class Cercle extends Figure
{
        private double rayon;

        public Cercle(String n)
        {
                super(n);
                rayon = 5;
        }

        public void calcul()
        {
                perimetre=2*3.14*rayon;
        }
```

```
public aspect AspectFigure
{

    public pointcut pcCalculCercle():
        execution (void Cercle.calcul());

    before():pcCalculCercle()
    {
        System.out.println("Calcul
CERCLE");
    }
}
```

Figure 17 - Relation classe vers aspect

La figure 17 nous montre comment une méthode est référencée par un point de jointure dans un aspect. Ce point de jointure appartient à une coupe nommée « *pcCalculCercle* ». Et enfin, un advice est lié à cette coupe afin d'exécuter du code.

Ainsi, quand la méthode en question sera supprimée, il y aura impact d'abord sur le point de jointure, ensuite sur la coupe contenant ce point de jointure, et enfin sur l'advice lié au point de jointure. Cet impact sera représenté par notre modèle comme suit : **Mr -> JPr + PCm + ADVm.**

« M » fait référence à une méthode, « JP » pour joinpoint, « PC » pour pointcut et enfin « ADV » pour advice. La lettre « r » veut dire que l'élément qu'elle accompagne est supprimé et la lettre « m » signifie que l'élément est modifié.
Ainsi, la règle se lit comme suit : la suppression d'une méthode (Mr) entraine (->) la suppression du point de jointure lui faisant référence (JPr), suivi de la modification de la coupe contenant ce point de jointure (PCm) et enfin la modification de l'advice lié à la coupe (ADVm).

4.1.1. Exemples de règles :

a) Modification du type d'un attribut :

Modifier le type d'un attribut aura pour effet d'entrainer une modification directement sur les points de jointure qui y font référence. Nous aurons donc la règle suivante :

Atm ->JPm + PCm
Cette règle se lit come suit : *Modifier le type d'un attribut (Atm) va entrainer la modification des points de jointure pointant sur lui (JPm). Les pointcuts contenant ces points de jointure sont eux aussi modifiés (PCm).*

Dans la figure 18 ci-dessous, un point de jointure est lié à l'attribut "cote" situé au niveau de la classe "Carre". La modification de cet attribut entrainera donc la modification du point de jointure qui lui est lié, ainsi que le pointcut auquel appartient le point de jointure en question.

Avant modification du type de l'attribut

```
public class Carre extends Figure
{
        private double cote;

        public Carre(String n, double c)
        {
                super(n);
                cote = c;
        }
        public double getCote()
        {
                return cote;
        }
}
```

```
public aspect AspectFigure
{

   pointcut pcCoteGet():
        get(double Carre.cote);

   after(): pcCoteGet()
   {
        System.out.println("Accès à la
        valeur de l'attribut cote");
   }
}
```

Après modification du type de l'attribut

```
public class Carre extends Figure
{
        private int c;

        public Carre(String n, double c)
        {
                super(n);
                this.c = c;
        }

        public int getCote()
        {
                return c;
        }
}
```

```
public aspect AspectFigure
{

   pointcut pcCoteGet():
        get(int Carre.c);
   after(): pcCoteGet()
   {
        System.out.println("Accès à la
        valeur de l'attribut cote");
   }

}
```

Figure 18 - Exemple de règle: Modification du type d'un attribut

b) Retrait d'une interface :

Lorsqu'on retire une interface à une classe, on n'est pas obligé de supprimer les méthodes implémentées. Si tel est le cas, on pourrait avoir des impacts sur les points de jointure correspondant à ces méthodes. Un point de jointure impacté entraine la modification ou la suppression du pointcut auquel il appartient. Et enfin, l'advice lié au pointcut en question sera soit modifié soit supprimé.

Nous avons donc la règle suivante : **Cir -> [Mr + JPr + PCr||PCm + ADVr||ADVm]**

Après retrait d'une interface à une classe (Cir), il peut y avoir éventuellement ([]) le retrait de méthodes (Mr) ainsi que les points de jointure liées à ces méthodes (JPr). Il s'en suivra la suppression des pointcuts (PCr) ou (||) leur modification (PCm), puis la suppression ou la modification des advices qui pointent sur les pointcuts impactés (ADVr || ADVm).

Dans cet exemple, on peut voir l'utilisation des crochets « [] » qui indiquent l'incertitude de l'impact, ainsi que l'utilisation des deux barres verticales « || » qui indiquent qu'il y aura soit une modification du pointcut (PCm) soit un retrait d'un pointcut (PCr). Aussi, soit un advice sera modifié (ADVm), soit un advice sera supprimé (ADVr).

4.2. Impact aspect vers aspect :

Des liens existent entre les différents membres d'un aspect. En effet, un point de jointure est utilisé le plus souvent dans un pointcut. Ainsi, quand ce point de jointure est modifié, le pointcut le contenant est automatiquement modifié. De plus, une fois qu'un point de jointure est atteint, un advice doit exécuter le code voulu. Cet advice est lié au pointcut contenant le point de jointure atteint. Dans certaines situations, un pointcut peut contenir plusieurs autres pointcuts. Toute modification au niveau de ces derniers entraine la modification du pointcut les englobant.
On parle également d'impact d'aspect vers aspect quand une modification dans un aspect S touchera les aspects qui héritent de cet aspect S.

```
public aspect Intro
{
    public void Point.faireSomme()
    {
        System.out.println("Somme: " + (getY()+getX()));
    }

    pointcut xSet():
        set(int Point.x);
    after(): xSet()
    {
        System.out.println("Modification de L'attribut X");
    }


    pointcut ySet():
        set(int Point.y);

    pointcut Set():
        xSet()||
        ySet();

    after(): Set()
    {
        System.out.println("Modification des attibuts");
    }
}
```

Figure 19 - Exemple impact aspect vers aspect

La figure 19 ci-dessus montre que la suppression ou la modification du nom de la coupe "xSet" entrainera des impacts sur l'advice after lié à cette coupe, ainsi que sur la coupe "Set" contenant la coupe "xSet".

4.2.1. Exemples de règles :

a) Suppression d'un pointcut :

Règle d'impact :

PCr -> PJr + ADVr + PCm(L) + PCr(H) + PJr(H) + ADVr(H)

La suppression d'un pointcut (PCr) entraine automatiquement la suppression des points de jointure le composant (PJr), ainsi que la suppression des advices qui lui sont rattachés (ADVr). Aussi, les pointcuts qui font appels au pointcut supprimé seront modifiés (PCm(L)). Tout pointcut redéfinissant le pointcut supprimé dans un aspect descendant sera aussi supprimé (PCr(H)) ainsi que les points de jointure et advices de cet aspect descendant (PJr(H) + ADVr(H)).

Avant retrait du pointcut

```
public aspect Intro
{
    public void Point.faireSomme()
    {
        System.out.println("Somme: " +
(getY()+getX()));
    }

    pointcut xSet():
        set(int Point.x);
    after(): xSet()
    {
        System.out.println("Modification de
        L'attribut X");
    }


    pointcut ySet():
        set(int Point.y);

    pointcut Set():
        xSet()||
        ySet();

    after(): Set()
    {
        System.out.println("Modification des
attibuts");
    }
}
```

Après retrait du pointcut

```
public aspect Intro
{
    public void Point.faireSomme()
    {
        System.out.println("Somme: " +
(getY()+getX()));
    }

    pointcut xSet():
        set(int Point.x);
    after(): xSet()
    {
        System.out.println("Modification de
        L'attribut X");
    }


    pointcut ySet():
        set(int Point.y);

    pointcut Set():
        xSet()||
        ySet();

    after(): Set()
    {
        System.out.println("Modification des
attibuts");
    }
}
```

Figure 20 - Exemple règle d'impact: suppression d'un pointcut

Dans cette règle d'impact, le « L » indique que la modification se fait au niveau du même aspect, c'est à dire au niveau local. Le « H » indique quant à lui qu'il s'agit d'impacts situés dans un aspect héritant de celui où a eu lieu la modification.

Dans la figure 20, on peut voir un pointcut nommé "xSet" (en jaune dans la figure 20). Lorsque ce pointcut est supprimé, le point de jointure lui appartenant est automatiquement supprimé, ainsi que l'advice rattaché au pointcut "xSet". Aussi, le pointcut "Set" sera également impacté par cette modification puisqu'il fait appel au pointcut "xSet".

b) Modification de la signature de la méthode (nom, paramètres, type de retour)

Règles d'impact :

Mnm | Mpa | Mpm | Mpr -> ADVm(L) + Mm(L) + Mm(H) + ADVm(H)

Lecture de la règle: Toute modification de la signature d'une méthode entrainera des impacts sur les advices (ADVm) et les méthodes Mm(L) du code aspect où la méthode est invoquée, ainsi que les méthodes Mm(H) et advices ADVm(H) se situant dans un aspect qui hérite de l'aspect dans lequel il y a eu la modification.

Exemple : Modification de la méthode « message »

Avant modification

```
public aspect Aspect_Carre extends Aspect_Figure
{

    public pointcut pcPerimetre():
        execution(void Carre.calculPerimetre());

    before(): pcPerimetre()
    {
        System.out.println("Valeur du périmètre du
                            carré");
        message();
    }
}
```

```
public abstract aspect Aspect_Figure extends Figure
{
    public abstract pointcut pcPerimetre();

    public pointcut pcSurface():
        execution(void Cercle.calculSurface()) ||
        execution(void Carre.calculSurface());

    before(): pcSurface()
    {
        System.out.println("Calcul de la
surface");
        message();
    }

    public void message()
    {
        //code
    }
```

Après modification

```
public aspect Aspect_Carre extends Aspect_Figure
{

    public pointcut pcPerimetre():
        execution(void Carre.calculPerimetre());

    before(): pcPerimetre()
    {
        System.out.println("Valeur du périmètre
du                      carré");
        message();
    }
}
```

```
public abstract aspect Aspect_Figure
{
    public abstract pointcut pcPerimetre();

    public pointcut pcSurface():
        execution(void Cercle.calculSurface()) ||
        execution(void Carre.calculSurface());

    before(): pcSurface()
    {
        System.out.println("Calcul de la
                            surface");
        message();
    }

    public void message(double i)
    {
        //code
    }
```

Figure 21 - Exemple règle d'impact: Modification de la signature de la méthode

Au niveau de la figure 21, la modification des paramètres de la méthode "message" située dans l'aspect "Aspect_Figure" aura un impact au niveau local étant donné que la méthode en question est appelée dans un advice du même aspect. De plus, cette modification aura un impact dans l'aspect "Aspect_Carre" qui hérite de l'aspect "Aspect_Figure" étant donné que l'advice contenu dans l'aspect "Aspect_Carre" fait appel à cette même méthode "message".

4.3. Impact aspect vers objet:

Dans les relations entre les classes objet et les aspects, les aspects dépendent des classes objet. De plus, la création d'aspects n'a aucune incidence sur le code objet d'un point de vu statique (avant le tissage). Ainsi, toute modification, ajout ou suppression de point de jointure, de pointcut ou d'advice dans un aspect n'aura pas d'impact sur le code objet auquel est rattaché l'aspect en question.

Toutefois, ces modifications seront observées durant la post-analyse puisque les effets des éléments modifiés dans la partie aspect ne seront visibles dans la partie objet seulement qu'après le tissage. De plus, toute modification dans le code aspect n'empêchera pas le programme d'être exécuté.

Cependant, il y aura un impact du code aspect vers le code objet au niveau des déclarations intertype. En effet, toute déclaration intertype insérée dans le code aspect peut être utilisée dans le code objet. Ainsi, toute modification ou suppression de ces déclarations intertype va se répercuter sur le code objet.

```
public aspect Aspect_Limousine
{
	int Chauffeurs.iNbMaxTrajet = 100;

	public pointcut pcNewTrajet():
		initialization(Trajet.new(String, int, Limousine));

	after(): pcNewTrajet()
	{
		System.out.println("Création d'un nouveau trajet");
	}

	public pointcut pcAddTrajet(Trajet t):
		execution(void Chauffeurs.addTrajet(Trajet)) && args(t);

	before(Trajet t): pcAddTrajet(t)
	{
		System.out.println("\n\nAjout d'un trajet");
		t.toString();
		System.out.println("\n\n");
	}
```

```
public class Chauffeurs {

	private Trajet aTrajet[];
	private int iNbTrajet;
	private int iAnneeEmbauche;
	private String sNom;
	private String sPrenom;
	private String sAdresse;
	private boolean bLibre;

	// Contructeur
	public Chauffeurs(int anneeEmbauche, String nom,
					String prenom, String adresse)
	{
		iNbTrajet = -1;
		iAnneeEmbauche = anneeEmbauche;
		sNom = nom;
		sPrenom = prenom;
		sAdresse = adresse;
		bLibre = true;
		aTrajet = new Trajet[iNbMaxTrajet];
	}

	public void addTrajet(Trajet objTrajet)
	{
		if((bLibre == true) && (iNbTrajet < iNbMaxTrajet - 1))
		{
			bLibre = false;
			iNbTrajet ++;
			aTrajet[iNbTrajet] = objTrajet;
			System.out.println("Le trajet du chauffeur " +
					this.getNoIdentification() + " a été créé.");
		}
		else
		{
			System.out.println("Impossible d'ajouter le trajet de " +
					this.getNoIdentification());
		}
	}
```

Figure 22 - Exemple impact aspect vers objet

Dans l'aspect "Aspect_Limousine" (cf. figure 22), un attribut est introduit dans la classe "Chauffeur" grâce à une déclaration intertype. Toute modification ou suppression de cette déclaration intertype aura des répercutions dans la classe "Chauffeur".

Règles d'impact :

I-TYr | I-TYm -> Mm

Lecture de la règle: Toute suppression ou modification d'une déclaration intertype entrainera des impacts sur les méthodes (Mm) du code objet où la déclaration intertype est invoquée.

CHAPITRE 5:
ÉVALUATION EMPIRIQUE

I. Introduction

Après l'élaboration des règles du modèle, nous allons faire des expérimentations visant chacune des règles. Ces expérimentations vont d'abord porter sur des impacts objet vers aspect, suivies des impacts aspect vers aspect et enfin des impacts aspect vers objet.

Le but de ces tests est de pouvoir vérifier la capacité des règles à pouvoir prédire de façon fiable les impacts suite à une modification. De plus, ces expérimentations permettront de pouvoir apporter des corrections aux éventuelles règles ayant besoin d'ajustement. Enfin, nous seront en mesure de déceler les limites du modèle.

Ces expérimentations porteront sur chacune des règles. Ainsi, pour une règle donnée, nous allons créer via un petit programme le contexte de celle-ci. Suite à une modification, nous allons donc pouvoir vérifier si les éléments énoncés dans la règle répondent correctement aux résultats que nous allons constater effectivement.

Certaines règles vont nécessiter plus d'un exemple. En effet, pour une modification donnée, il n'est pas toujours évident que les situations d'impact prévues par la règle soient toutes rassemblées en même temps. Nous allons donc présenter différents exemples pour une même règle d'impact afin de passer en revue toutes les situations possibles de cette règle.

Par exemple, pour la suppression d'une classe, la règle d'impact indique que les impacts dans le code aspect vont concerner les points de jointure, les pointcuts, les advices, les déclarations inter-types et enfin des classes aspects héritant de la classe supprimée (*Cr -> JPr + [PCr] + [ADVr|| ADVm] + I-TYr + ASm(H)*). Cependant, un aspect peut avoir des pointcuts, des advices, sans avoir de déclarations inter-types, ou encore les aspects n'hériteront pas forcement de la classe objet en question.

II. Protocole d'évaluation

Afin de mener nos expérimentations, nous avons élaboré de petits programmes à l'aide d'AspectJ. Pour certains programmes, nous avons rajouté un second aspect qui hérite d'un autre. De plus nous avons utilisé les programmes contenus dans le dossier des exemples fournis avec le compilateur aspectJ afin de réaliser nos tests. Nous avons également utilisé le programme "DocFetcher" disponible sur www.sourceforge.net. Nous avons introduit des modifications dans chacun de ces programmes et avons observé les conséquences. Le tableau 1 ci-dessous nous donne une description des programmes utilisés.

Nom du programme	#Classes	#Aspect
Bean	3	1
Introduction	1	3
Observer	6	2
Spacewar	17	10
Telecom	10	3
TJP	1	1
Tracing (version 1)	5	3
Figure	3	2
Limousines	5	1
Quicksort	2	1
DocFetcher	79	7
Stack	2	1
Êtres vivants	6	2
Bank	2	1
Gestion matrice	2	1

Tableau 1 - description des programmes de test

Les expérimentations se sont déroulées comme suit :

- **Phase 1** :

  Nous avons apporté des modifications aux programmes en suivant la règle testée, et ce, pour chacune des règles de notre modèle. Il s'agit ici de faire ressortir les impacts prévus par le modèle.

- **Phase 2** :

  Nous avons apporté les mêmes changements dans ces mêmes programmes et avons constaté les modifications qui en découlent. Ces changements sont effectués sans tenir compte des règles d'impact du modèle. Cette étape nous permet de déterminer le nombre d'impacts réels sur le programme.

- **Phase 3** :

  Une comparaison des résultats obtenus a été effectuée après les phases précédentes. Nous pourrons donc déterminer le ratio d'impacts prévus par rapport aux impacts réels. Nous saurons également quel est le pourcentage d'impacts correctement prévus par notre modèle après une modification par rapport aux impacts prévus par le modèle..

III. Mesures d'évaluation

Afin de pouvoir évaluer notre modèle, nous avons calculé la **précision et le rappel** (precision and recall) [5] pour chacune de nos expérimentations.

a. La précision (precision)

La précision est le pourcentage d'éléments bien identifiés d'une approche par rapport au nombre total d'éléments identifiés [8].

La précision permettra donc de connaitre le pourcentage d'impacts observés et prévus par le modèle par rapport au total des impacts constatés réellement suite aux modifications apportées au programme. Cette mesure permet de voir si le modèle arrive à prévoir tous les impacts. La précision idéale, qui est de 100%, indique que tous les impacts survenus après un changement ont été correctement identifiés par le modèle. Une précision de 80% indique que sur 10 impacts constatés, par exemple, le modèle en a prévu 8. D'un autre côté, une précision de 20% indique que le modèle n'a prédit que 2 impacts sur 10, ce qui est peu efficace. Notre modèle sera donc précis s'il arrive à prédire le plus d'impacts possible parmi tous les impacts survenus.

b. Le rappel (recall)

Le rappel est la mesure permettant d'évaluer la capacité du modèle à prédire tous les impacts réels [5].

Rapporté à notre modèle, le rappel permettra la mise en évidence du pourcentage d'impacts effectifs prévus par le modèle par rapport aux impacts prévus par la modèle. Obtenir un rappel de 100% signifierait que notre modèle est suffisamment efficace pour déceler tous les impacts effectifs.

Notre modèle serait assez efficace, si la précision et le rappel que nous aurons pour chacune des expérimentations se rapprochent le plus possible de 100%.

IV. Exemples d'expérimentations

Nous allons, dans cette partie du mémoire, présenter quelques expérimentations.

1. Impacts objet vers aspect

1.1. Suppression d'une classe

Règle d'impact : Cr -> JPr + [PCm || PCr] + [ADVr|| ADVm] + I-TYr + DCLm

Suppression de classe (Cr) entraine la suppression des points de jointures (JPr) suivi éventuellement (représenté par les crochets) de la suppression de pointcut ou de sa modification ([PCm || PCr]). On aura ensuite éventuellement la suppression ou la modification d'advice ([ADVr|| ADVm]), le retrait des déclarations inter-type (I-TYr), et enfin la modification des déclarations (DCLm)

Le programme que nous allons tester (programme EtresVivants) est composé de 6 classes et de deux aspects.

La classe "Vivant" (cf. figure 23) est une classe abstraite dont héritent les classes "Humain", "Animal", et "Plante". La classe "TestVivant" est la classe qui permet de démarrer le programme, et enfin la classe "ComplexiteIncorrecte" qui permet de gérer des exceptions.

Du coté des aspects, nous avons l'aspect "Aspect_EtresVivants" (cf. figure 24) qui contient des points de jointure liés aux classes "Vivant" , "Humain", "Plante" et "ComplexiteIncorrecte". L'autre aspect, "Aspect_Animal", contient des points de jointure liés à la classe "Animal" et hérite de l'aspect "Aspect_EtresVivants".

```
public abstract class Vivant implements Comparable
{
	public enum enumMilieudeVie { eau, terre, airTerre };
	protected String nom;
	protected enumMilieudeVie milieuDeVie;
	protected int complexite;
	//protected final short DEFAULTCOMPLEX = 0; /* Complexite par defautt */

	public abstract void deplacement();

	public Vivant()
	{
		nom = "Undefined";
		milieuDeVie = enumMilieudeVie.terre;
		complexite = DEFAULTCOMPLEX;
	}

	public Vivant(String name, enumMilieudeVie milDeVie, short complexity) throws
		ComplexiteIncorrecte
	{
		ComplexiteIncorrecte problemeComplexite;
		this.nom = name;
		this.milieuDeVie = milDeVie;
		this.complexite = DEFAULTCOMPLEX;
		problemeComplexite = new ComplexiteIncorrecte
			("La complexité est hors limites (limites: 0 à 10) pour " +
				this.nom);


		if ((complexity < 1) || (complexity > 10))
			throw problemeComplexite;
		else
			this.complexite = complexity;
	}

	public int compareTo(Object autreObjet)
	{
		Vivant secondObjet;

		secondObjet = (Vivant) autreObjet;
		return (this.complexite - secondObjet.complexite);
	}

	public String toString()
	{
		String result;

		result =   "Nom : " + nom + "\n"
			 + "Milieu de vie : " + milieuDeVie + "\n"
			 + "Complexité : " + complexite + "\n";
		return result; /* Retourne la chaine de caractères */
	}
```

Figure 23 - Classe Vivant

Ainsi, en se basant sur la règle, nous devons avoir les impacts suivants:

- Suppression des points de jointure liés à la classe "Vivants". Il y en a 3,
- Modification ou suppression des pointcuts contenant les points de jointure supprimés. Il y en a 3,
- Modification ou suppression des advices liés aux pointcuts modifiés ou supprimés. Il y en a 3,
- Suppression des déclarations intertype liées à la classe "Vivants". Il y en a 1.

En nous basant sur la règle appliquée à cet exemple, nous devrions avoir 10 impacts. La figure 24 ci-dessous montre en jaune les impacts réellement observés.

Suppression de la classe Vivant :

```
public abstract aspect Aspect_EtresVivants
{

	final short Vivant.DEFAULTCOMPLEX = 0;

	public pointcut pcCheckComplexcite(int c, Animal a):
		set(int Vivant.complexite) && args(c) && this(a);

	after(int c, Animal a): pcCheckComplexcite(c, a)
	{
		System.out.println("Complexité: "+c);
		a.toString();
	}

	public pointcut pcCheckComplexcite2():
		set(int Vivant.complexite);

	after(): pcCheckComplexcite2()
	{
		System.out.println(" Attribut modifié ");
	}


	public pointcut pcNewHumain(int i, String s):
		initialization(Humain.new(int, String)) && args(i, s);

	after(int i, String s): pcNewHumain(i, s)
	{
		System.out.println("Création d'un humain");
	}

	public pointcut pcDeplacement():
		execution(void Vivant.deplacement());

	after(): pcDeplacement()
	{
		System.out.println("DEPLACEMENT DE L'ETRE VIVANT");
	}

}
```

Figure 24 - Aspect Aspect_EtresVivants

Au niveau du programme, nous avons supprimé de façon concrète la classe "Vivant". Nous avons constaté les éléments suivants dans l'aspect "Aspect_EtresVivants".

- Les points de jointure qui indexaient des parties de code liées à la classe "Vivant" ne peuvent plus s'appliquer, donc ils sont à supprimer.
- Les pointcut contenant les points de jointure à supprimer sont à supprimer, puisque n'ayant plus de point de jointure.
- Les advices liés aux pointcuts supprimés sont aussi à supprimer.
- La déclaration intertype contenue dans l'aspect "Aspect_EtresVivants" est à supprimer.

Nous avons aussi constaté des impacts sur le reste des classes qui héritent de la classe Vivant, soit les classes "Humain", "Animal" et "Plante", ainsi que sur la classe associée à la classe Vivant, soit la classe " TestVivant". Le tableau 2 résume nos observations.

Nombre d'impacts prévus par le modèle	Nombre d'impacts effectifs	Nombre d'impacts effectifs prévus par le modèle
10	10	10

Tableau 2 - Résultat expérimenttion: suppression d'une classe

Au tableau 2, nous avons les mesures suivantes:

- Précision: (10 / 10) * 100 = 100%
- Rappel: (10 / 10) * 100 = 100%

Dans cet exemple, le modèle a réussi à prédire parfaitement les 10 impacts survenus du coté de l'aspect "Aspect_EtresVivants".

1.2. Modification des paramètres d'une méthode

Nous avons déterminé différentes situations pour cette modification. Pour chacune des situations, une règle d'impact a été établie.

L'exemple que nous allons prendre pour les différentes situations est constitué d'un aspect et d'une classe "Cercle" comprenant deux méthodes.

a- Le pointcut ne prend aucun paramètre

Dans cette situation, le pointcut ne prend aucun paramètre, on ne fait que capter la méthode par le biais d'un point de jointure, comme illustré dans la figure 25.

Règles d'impact : Mpm -> JPpm + PCm

La modification de paramètre d'une méthode (Mpm) entraine la modification du type du paramètre du point de jointure faisant référence à la méthode (JPpm). Il s'en suit donc une modification du pointcut contenant ce point de jointure (PCm).

Avant modification des paramètres du constructeur de la classe Cercle

```
public class Cercle extends Figure
{
        private double rayon;

        public Cercle(String s)
        {
                super(s);
                rayon = 5;
        }

        public double getRayon()
        {
                return rayon;
        }
}
```

```
pointcut pcConstructeurCercle():
   initialization(Cercle.new(String);

   after():pcConstructeurCercle()
   {
       System.out.println("Création d'un
       cercle ");
   }
```

Après modification des paramètres du constructeur de la classe Cercle

```
public class Cercle extends Figure
{
        private double rayon;

        public Cercle(String s, double r)
        {
                super(s);
                rayon = r;
        }

        public double getRayon()
        {
                return rayon;
        }
}
```

```
pointcut pcConstructeurCercle():
   initialization(Cercle.new(String,
   double));

   after():pcConstructeurCercle()
   {
       System.out.println("Création d'un
       cercle ");
   }
```

Figure 25 - Exemple de règle: Modification des paramètres d'une méthode - cas pointcut sans paramètre

Lorsque les paramètres du constructeur sont modifiés, on aura les impacts suivants:

- Modification du point de jointure lié à la méthode. Il y en a 1,
- Modification du pointcut contenant le point de jointure modifié. Il y en a 1.

En nous basant sur la règle d'impact, nous avons deux impacts prévus par le modèle (cf. tableau 3).

Nombre d'impacts prévus par le modèle	Nombre d'impacts effectifs	Nombre d'impacts effectifs prévus par le modèle
2	2	2

Tableau 3 - Résultat expérimentation: Modification des paramètres d'une méthode - cas pointcut sans paramètre

Au vu de ce tableau, nous avons les mesures suivantes:

- Précision: (2 / 2) * 100 = 100%
- Rappel: (2 / 2) * 100 = 100%

Nous obtenons pour cet exemple une précision et un rappel tous deux de 100%. Tous les impacts ont été prédits correctement par notre modèle.

b- Le point de jointure, le pointcut et les advices sont affectés par la modification

Dans cette situation, des paramètres sont passés au pointcut. Bien entendu, ces paramètres sont les mêmes que ceux de la méthode capturée par le point de jointure. De plus, ces paramètres sont également passés à l'advice relié au pointcut. Cela a pour but de pouvoir les utiliser dans l'advice lié au pointcut. On pourrait alors les vérifier, les utiliser pour un traitement, ou juste les afficher.

> **Règle d'impact : Mpm-> JPpm +PCpm + PCargm + ADVpm**
> *La modification d'un paramètre dans une méthode entrainera la modification des paramètres du point de jointure captant cette méthode (JPpm). On aura ensuite la modification des paramètres du pointcut (PCpm), ainsi que la modification d'arguments de ce pointcut (PCargm). Enfin une modification des paramètres des advices rattachés au pointcut (ADVpm).*

La règle indique les impacts suivants:

- Modification du point de jointure
- Modification du pointcut
- Modification des arguments du pointcut
- Modification de ou des advices liés au pointcut modifié

Avant modification des paramètres du constructeur de la classe Cercle

```
public class Cercle extends Figure
{
        private double rayon;

        public Cercle(String s)
        {
                super(s);
                rayon = 5;
        }

        public double getRayon()
        {
                return rayon;
        }
}
```

```
pointcut pcConstructeurCercle(String s):
   initialization(Cercle.new(String))&&
   args(s);

after(String s):pcConstructeurCercle(s)
{
   System.out.println("Création d'un cercle
   nommé: "+ s);
}
```

Après modification des paramètres du constructeur de la classe Cercle

```
public class Cercle extends Figure
{
        private double rayon;

        public Cercle(double r)
        {
                super("nom");
                rayon = r;
        }

        public double getRayon()
        {
                return rayon;
        }
}
```

```
pointcut pcConstructeurCercle(double r):
   initialization(Cercle.new(double))&&
   args(r);

after(double r):pcConstructeurCercle(r)
{
   System.out.println("cercle de rayon "+r
   );
}
```

Figure 26 - - Exemple de règle: Modification des paramètres d'une méthode - cas point de jointure, pointcut et advice modifiés

Dans la figure 26, nous avons en jaune du côté Java la modification du paramètre du constructeur et en vert du côté aspect les impacts observés après la modification. On dénombre donc les impacts suivants:

- Les paramètres du point de jointure sont à modifier (1 impact)
- Les paramètres du pointcut doivent être changés (1 impact)
- Les arguments du pointcut doivent également être modifiés (1 impact)
- Les paramètres de l'advice lié au pointcut sont à modifier (1 impact)

Nombre d'impacts prévus par le modèle	Nombre d'impacts effectifs	Nombre d'impacts effectifs prévus par le modèle
4	4	4

Tableau 4 - Résultat expérimentation: Modification des paramètres d'une méthode - cas point de jointure, pointcut et advice modifiés

Au vu du tableau 4, nous avons les mesures suivantes:

- Précision: (4/4) * 100 = 100%
- Rappel: (4 / 4) * 100 = 100%

Nous précisons que nous considérons comme un seul impact, tous les impacts qu'il y a eu dans un même advice, que ce soit au niveau des paramètres ou au dans le corps de cet advice.

1.3. Changement du type de retour d'une méthode de « void » à type (primitif ou objet)

Règle d'impact : Mtr_{vo} -> JPm + PCm + [ADVm]

Changement de type de retour d'une méthode de void à objet (Mtr_{vo}) entraine la modification des points de jointure pointant sur cette méthode (JPm). Le pointcut contenant le point de jointure en question est lui aussi modifié (PCm). L'advice rattaché au pointcut sera éventuellement modifié ([ADVm]).

Le programme utilisé pour l'expérimentation est composé de trois classes: la classe "LinkList" (cf. figure 27), la classe "Link" et la clase "LinskListApp". Ce programme contient également un aspect, "Aspect_Link".

Dans cet aspect, un point de jointure est lié à la méthode "insertFirst" qui se trouve dans la classe "LinkList" et qui renvoie un résultat de type "void".

```
public class LinkList
{
        public Link first;

        public LinkList()  {
           first = null;
        }

        public void insertFirst(int id, double dd)  {
           Link newLink = new Link(id, dd);
           newLink.next = first;
           first = newLink;
        }

        public Link deleteFirst()   {
              Link temp = first;
              first = first.next;
              return temp;
        }
...
```

Figure 27 - Classe LinkList

Modifier le type de retour de la méthode "insertFirst" (en jaune dans la figure 27) de « void » à « double » va donner les impacts suivants selon le modèle:

- Modification du point de jointure lié à la méthode. Il y en a 1,
- Modification du pointcut contenant le point de jointure modifié. Il y en a 1,
- Modification de l'advice lié au pointcut modifié. Il y en a 1.

Après la modification du type de retour, il y a eu les impacts suivants (en vert dans la figure 28):

- La méthode "insertFirst" a été impactée. Il faut lui ajouter une valeur de retour. Cet impact se situe au niveau du code java (1 impact).
- Le point de jointure lié à la méthode doit être modifié pour prendre en compte le nouveau type de retour de la méthode (1 impact)
- Le pointcut contenant le point de jointure en question sera impacté (1 impact).
- L'advice lié au pointcut ci-dessus sera lui aussi impacté (1 impact).

```
public aspect Aspect_Link
{
        public pointcut pcInsertItem(LinkList l):
                execution(void LinkList.insertFirst(int, double))&& target(l) ;

         after(LinkList l): pcInsertItem(l)
         {
                System.out.print("Insertione de: ");
                l.displayList();
         }
```

Figure 28 - Exemple règle d'impact: Changement du type de retour d'une méthode de void à type (primitif ou objet)

Nombre d'impacts prévus par le modèle	Nombre d'impacts effectifs	Nombre d'impacts effectifs prévus par le modèle
3	3	3

Tableau 5 - Résultat expérimentation: Changement du type de retour d'une méthode de void à type (primitif ou objet)

Au vu du tableau 5, nous avons les mesures suivantes:

- Précision: (3 /3) * 100 = 100%
- Rappel: (3 / 3) * 100 = 100%

La précision dans cet exemple est de 100% et le rappel de 100%. Étant donné que notre modèle ne prend pas en compte les impacts au niveau objet, nous nous retrouvons donc avec 3 impacts observés comme indiqué dans le tableau 5.

1.4. Suppression d'un lancement d'exception

Règle d'impact : Nrtc -> JPr + PCr || PCm + ADVr || ADVm

Enlever le lancement d'une exception, ou retirer un try-catch (Nrtc) aura pour conséquence le retrait du point de jointure (JPr) faisant référence à cette exception. Le pointcut contenant ce point de jointure est lui aussi supprimé (PCr) ou modifié (PCm). Les advices liés au pointcut sont eux aussi supprimés (ADVr) ou modifiés (ADVm).

Dans l'exemple qui suit (cf. figure 29), le lancement d'une exception dans le code java (en vert dans la figure 29) est capté par un point de jointure (en jaune dans la figure 29). Ce point de jointure est contenu dans un pointcut qui lui même comprend un autre point de jointure captant une autre exception.

```
private static void valeurs()
{
   cont = 0;
   try
   {
        System.out.println("Entrez X ");
        x = new Scanner(System.in).nextInt();
        cont = 1;

        System.out.println("Entrez Y ");
        y = new Scanner(System.in).nextInt();
        cont = 2;

        System.out.println("Entrez  Z ");
        z = new Scanner(System.in).nextInt();
        cont = 3;
   }
   catch(java.util.InputMismatchException e)
   {
        System.out.println("\nEntrez les
                        valeurs) ");
        valeurs();
   }
}
```

```
pointcut pcException():
   handler(java.util.InputMismatchException
   )|| handler(IOException);

before(): pcException()
{
   System.out.println("Interception d'une
                exception");
}
```

Figure 29 - Exemple règle d'impact: Suppression d'un lancement d'exception

En s'appuyant sur la règle d'impact, on devrait avoir les impacts suivants:

- Suppression du point de jointure captant l'exception supprimée
- Modification du pointcut où se trouve le point de jointure
- Modification de l'advice lié au pointcut modifié

La suppression effective de l'exception entraine les impacts suivants:

- Le point de jointure captant l'exception est impacté (1 impact)
- La pointcut contenant le point de jointure est impacté (1 impact)
- L'advice lié au pointcut est automatiquement affecté (1 impact)

Nombre d'impacts prévus par le modèle	Nombre d'impacts effectifs	Nombre d'impacts effectifs prévus par le modèle
3	3	3

Tableau 6 - Résultat expérimentation: Suppression d'un lancement d'exception

Au vu de ce tableau, nous avons les mesures suivantes:

- Précision: (3 /3) * 100 = 100%
- Rappel: (3 / 3) * 100 = 100%

Dans ce cas, la règle d'impact a pu prédire tous les impacts dus à la suppression d'une exception.

2. Impacts aspect vers aspect

2.1. Modification du nom d'un aspect

Règle d'impact : ASnm -> AShm(H) + PCm(A) + ADVm(A) + DCLm(A) + DCLm(L)
Modifier le nom d'un aspect provoque une modification du lien d'héritage des aspects qui en héritent (AShm(H)). Il y aura également la modification des pointcuts des aspects associés (PCm(A)), la modification des advices dans les aspects associés (ADVm(A)) et enfin la modification des déclarations contenues dans les aspects associés (DCLm(A)) ainsi qu'au niveau local (DCLm(L)).

Le programme présenté dans la figure 30 ci-dessous est composé des aspects "Billing" et "Timing". Au niveau de l'aspect "Billing", on déclare l'ordre dans lequel les aspects seront exécutés si jamais ils pointent sur le même point de jointure (en jaune). Aussi, dans ce même aspect "Billing", on définit un advice lié au pointcut "endTiming" (en vert) qui se trouve dans l'aspect "Timing".

```
public aspect Billing {
    // precedence required to get advice on endtiming in the right order
    declare precedence: Billing, Timing;

    public static final long LOCAL_RATE = 3;
    public static final long LONG_DISTANCE_RATE = 10;

    public Customer Connection.payer;
    public Customer getPayer(Connection conn) { return conn.payer; }

    after(Customer cust) returning (Connection conn):
        args(cust, ..) && call(Connection+.new(..)) {
        conn.payer = cust;
    }

    after(Connection conn): Timing.endTiming(conn) {
        long time = Timing.aspectOf().getTimer(conn).getTime();
        long rate = conn.callRate();
        long cost = rate * time;
        getPayer(conn).addCharge(cost);
    }
...
```

```
public aspect Timing {

    public long Customer.totalConnectTime = 0;

    public long getTotalConnectTime(Customer cust) {
        return cust.totalConnectTime;
    }

    private Timer Connection.timer = new Timer();
    public Timer getTimer(Connection conn) { return conn.timer; }

    pointcut endTiming(Connection c): target(c) &&
        call(void Connection.drop());

    after(Connection c): endTiming(c) {
       getTimer(c).stop();
       c.getCaller().totalConnectTime += getTimer(c).getTime();
       c.getReceiver().totalConnectTime += getTimer(c).getTime();
    }
```

Figure 30 - Exemple règle d'impact: Modification du nom d'un aspect

Quand le nom de l'aspect "Timing" sera modifié on aura les impacts suivants (illustrés en bleu) selon la règle:

- Modification de la déclaration de précédence
- Modification de l'advice lié au pointcut se trouvant dans l'aspect "Timing".

On constate les impacts suivants après la modification effective du nom de l'aspect "Timing":

- Impact au niveau de la déclaration de précédence, il y en a 1
- Impact au niveau de l'advice lié au pointcut se trouvant dans l'aspect "Timing", il y en a 1

Nombre d'impacts prévus par le modèle	Nombre d'impacts effectifs	Nombre d'impacts effectifs prévus par le modèle
2	2	2

Tableau 7 - Résultat expérimentation: Modification du nom d'un aspect

On aura les mesures suivantes au vu du tableau 7:

- Précision: (2 /2) * 100 = 100%
- Rappel: (2 / 2) * 100 = 100%

Nous avons ici une précision et un rappel de 100% chacun, ce qui montre que notre modèle a pu déceler tous les impacts correctement.

2.2. Passage d'aspect abstrait à non abstrait

Dans un programme orienté aspect, un aspect B peut hériter d'un aspect A. Cela se fera uniquement si l'aspect A est déclaré comme abstrait. Dans ce cas de figure, on pourra déclarer des pointcut abstraits et des méthodes abstraites dans l'aspect A. Ces méthodes et pointcuts abstraits seront redéfinis dans l'aspect B. De plus, on pourra définir un pointcut dans l'aspect A et lui rattacher un advice dans l'aspect B. Enfin, tout attribut défini dans l'aspect A pourra être utilisé dans l'aspect B.

Règle d'impact : ASt_{an} -> Mt_{an}(L) + PCt_{an}(L) + ADVa(L) + AShr(H) + [Mr(H) | Mm(H)] + [PCr(H) | PCm(H)] + [ADVr(H) | ADVm(H)]

Faire passer un aspect d'abstrait à non abstrait (AStan) entraine le passage de ses méthodes (Mtan) et pointcuts (PCtan) abstraits à non abstraits. Un ou des advices sont rajoutés (ADVa(L)) si on ne supprime pas les pointcuts qui ont subit le changement. Puis, il y aura le retrait du lien d'héritage au niveau des aspects héritant de l'aspect modifié (AShr(H)), et éventuellement le retrait ou la modification des méthodes redéfinies dans les aspects héritant (Mr(H) | Mm(H)), et des pointcuts dans les aspects héritiers (PCr(H) | PCm(H)). Ensuite, les advices liés au différents pointcuts modifiés ou supprimés pourraient être modifiés ou supprimés au niveau des aspects héritiers ([ADVr(H) | ADVm(H)]).

Le programme utilisé pour l'expérimentation comprend quatre classes, soit les classes "FiguereApp", "Figure", "Cercle" et "Carre". Les classes "Cercle" et "Carre" héritent de la classe "Figure". Nous avons aussi dans le programme l'aspect "Aspect_Figure" qui est un aspect abstrait, et un aspect "Aspect_Carre" qui hérite de l'aspect "Aspect_Figure".

L'aspect "Aspect_Figure", illustré au niveau de la figure 31, comprend un attribut, un pointcut défini avec un point de jointure, un autre pointcut abstrait et deux méthodes abstraites.

Avant passage d'aspect abstrait à non abstrait

```
public abstract aspect Aspect_Figure
{
	pointcut pcPerimetre():
		execution(void Carre.calculPerimetre());

	abstract void calculPerimetre();

	abstract pointcut pcSurface();

	abstract void calculSurface();
}
```

```
public aspect Aspect_Carre extends Aspect_Figure
{
	pointcut pcSurface():
		execution(void Carre.calculSurface());

	before(): pcSurface()
	{
		System.out.println("Calcul de la surface");
		calculSurface();
	}

	before(): pcPerimetre()
	{
		System.out.println("Valeur du périmètre du carré");
		calculPerimetre();
	}

	void calculSurface()
	{
	}

	void calculPerimetre()
	{
	}
}
```

Figure 31 - Exemple règle d'impact: passage d'aspect abstrait à non abstrait (avant modification)

En se basant sur la règle d'impact. Nous aurons les impacts suivants:

- Les méthodes abstraites vont devenir non abstraites, il y en a 2.
- Le pointcut abstrait va devenir non abstrait.
- Un ou des advices devront être rajoutés et rattachés aux pointcuts devenus non abstraits.
- L'aspect "Aspect_Carre" va perdre son lien d'héritage avec l'aspect "Aspect_Figure".
- Les méthodes redéfinies dans l'aspect "Aspect_Carre" seront éventuellement modifiées ou supprimées. Il y en a 2.
- Le pointcut défini dans l'aspect "Aspect_Carre" sera supprimé ou modifié éventuellement.
- L'advice contenu dans l'aspect "Aspect_Carre" et qui est rattaché au pointcut défini dans l'aspect "Aspect_Figure" sera supprimé.

Après passage d'aspect abstrait à non abstrait

```
public aspect Aspect_Figure
{
	pointcut pcPerimetre():
		execution(void Carre.calculPerimetre());

	abstract void calculPerimetre();

	abstract pointcut pcSurface();

	abstract void calculSurface();
}
```

```
public aspect Aspect_Carre extends Aspect_Figure
{
	pointcut pcSurface():
		execution(void Carre.calculSurface());

	before(): pcSurface() {
		System.out.println("Calcul de la surface");
		calculSurface();
	}

	before(): pcPerimetre(){
		System.out.println("Calcul du périmètre");
		calculPerimetre();
	}

	void calculSurface()
	{ }

	void calculPerimetre()
	{}

}
```

Figure 32 - Exemple règle d'impact: passage d'aspect abstrait à non abstrait (après modification)

Une fois la modification effectuée, les résultats obtenus, marqués en jaune dans la figure 32, sont les suivants:

- Au niveau de l'aspect "Aspect_Figure" (local)
  - Les méthodes abstraites sont impactées au niveau de l'aspect "Aspect_Figure". Il y en a 2,
  - Le pointcut abstrait est impacté au niveau de l'aspect "Aspect_Figure" (1 impact).
- Au niveau de l'aspect "Aspect_Carre" (héritage)
  - Le lien d'héritage doit être enlevé (1 impact),
  - L'advice relié au pointcut "pcPerimetre" est impacté, puisque le lien d'héritage entre les deux aspects n'existe plus (1 impact).

Nombre d'impacts prévus par le modèle	Nombre d'impacts effectifs	Nombre d'impacts effectifs prévus par le modèle
8	5	5

Tableau 8 - Résultat expérimentation: passage d'aspect abstrait à non abstrait

Le tableau 8 nous permet d'avoir les mesures suivantes:

- Précision: (5 /5) * 100 = 100%
- Rappel: (5 /8) * 100 = 62.5%

Au vu de ces résultats, nous tombons sur une précision 100% et un rappel de 62.5. Les 5 impacts effectifs ont tous été prévus par la règle. Cependant, la règle en avait prédit trois de plus qui ne sont pas survenus. Cela s'explique par le fait que ces impacts sont incertains. En effet, la redéfinition des deux méthodes dans l'aspect "Aspect_Carre" ne dépendra que du programmeur. Il en est de même pour l'ajout d'advices à rattacher aux pointcuts devenus non abstraits.

2.3. Suppression d'un pointcut :

Règle d'impact : PCr -> ADVr(L) + PCm(L) + PCm(A) + ADVr(A) + PCr(H) + PCm(H)

Retrait d'un pointcut(PCr) entraine le retrait d'advices (ADVr) et la modification de pointcuts (PCm(L)) qui contenaient le pointcut supprimé au niveau local.)). Il y aura également la modification des pointcuts situés dans les aspects associés (PCm(A)), la suppression des advices dans les aspects associés (ADVr(A)). Enfin, il y aura la suppression des pointcuts redéfinissant le pointcut supprimé au niveau des aspects héritiers (PCr(H)), et la modification des pointcuts contenant le pointcut supprimé au niveau des aspects héritiers (PCm(H))

L'aspect de la figure 33 contient trois pointcuts. Un de ces pointcuts (en vert dans la figure 33) contient les deux autres.

```
public aspect Intro {
    public void Point.faireSomme(){
        System.out.println("Somme: " + (getY()+getX()));
    }
    pointcut xSet():set(int Point.x);
    after(): xSet(){
        System.out.println("Modification de L'attribut X");
    }

    pointcut ySet():set(int Point.y);

    pointcut Set():xSet()||ySet();

    after(): Set(){
        System.out.println("Modification des attributs");
    }
}
```

Figure 33 - Exemple règle d'impact: suppression d'un pointcut

En se basant sur la règle d'impact, on devrait avoir les impacts suivants après la suppression du pointcut "xSet" en jaune dans la figure 33:

- Suppression d'advice lié au pointcut supprimé,
- Modification du pointcut faisant référence au pointcut supprimé.

Une fois la suppression du pointcut effectuée, on constate les impacts qui suivent:

- L'advice lié au pointcut supprimé est impacté,
- Le pointcut qui fait appel au pointcut supprimé est impacté.

Nombre d'impacts prévus par le modèle	Nombre d'impacts effectifs	Nombre d'impacts effectifs prévus par le modèle
2	2	2

Tableau 9 - Résultat expérimentation: suppression d'un pointcut

Le tableau 9 nous permet d'avoir les résultats qui suivent:

- Précision: (2 /2) * 100 = 100%
- Rappel: (2 / 2) * 100 = 100%

Nous avons ici une précision et un rappel de 100% chacun. Ici aussi, notre modèle a pu identifier tous les impacts.

2.4. Suppression d'un attribut:

Règles d'impact : Ar -> ADVm(L) + ADVm(H) + Mm(L) + Mm(H)
La suppression ou modification d'un attribut entrainera des impacts sur les advices qui utilisent l'attribut aussi bien dans l'aspect lui même que dans les aspects héritiers (ADVm(L), (ADV(H)), sur les méthodes faisant appel à cet attribut au niveau de l'Aspect (Mm(L)) ainsi que sur les méthodes dans les aspects héritiers

Dans l'aspect "DiskWritingSuppressor", au niveau du programme DocFetcher, une variable de type booléen est déclarée (en jaune dans la figure 34). Celle-ci est utilisée dans 5 pointcuts anonymes contenu dans cet aspect "DiskWritingSuppressor".
D'après la règle d'impact de notre modèle, la suppression de cet attribut va entrainer des impacts sur chacun de ces pointcuts anonymes.
La partie de la règle d'impact indiquant les méthodes ne sera pas prise en compte, car l'aspect "DiskWritingSuppressor" n'en contient pas.

```
public privileged aspect DiskWritingSuppressor {

	private boolean writable = true;

	after(): set(Boolean Const.IS_PORTABLE) {
		if (Const.IS_PORTABLE && ! Const.USER_DIR_FILE.canWrite())
			writable = false;
	}

	after(): execution(* DocFetcherm.createTemporaryIndexes(..)) {
		if (! writable)
			DocFetcher.getInstance().setStatus(Msg.write_warning.value());
	}

	void around(): execution(* Pref.save()) {
		if (writable) proceed();
	}

	void around(): execution(* Serializer.save(..)) {
		if (writable) proceed();
	}

	void around(): execution(* ScopeRegistry.save(..)) {
		if (writable) proceed();
	}

	declare warning: call(FileWriter+.new(..))&& !withincode(* Pref.save())
	&& !withincode(* ScopeRegistry.save(..))&& !withincode(*ExceptionHandler.appendError(..))
	&& !within(CommandLineHandler):"Don't write to disk without updating
					net.sourceforge.docfetcher.aspect.DiskWritingSuppressor.";
}
```

Figure 34 - Exemple règle d'impact: suppression d'un attribut

Après la suppression de la variable "writable", on a constaté que tous les pointcuts anonymes faisant appel à cet attribut ont été impactés. Il y en a cinq.

Nombre d'impacts prévus par le modèle	Nombre d'impacts effectifs	Nombre d'impacts effectifs prévus par le modèle
5	5	5

Tableau 10 - Résultat expérimentation: suppression d'un attribut

On aura les mesures suivantes:

- Précision: (5 /5) * 100 = 100%
- Rappel: (5 / 5) * 100 = 100%

2.5. Ajout, modification ou suppression des paramètres d'une méthode:

Règles d'impact : Mpa | Mpm | Mpr -> ADVm + Mm(L) + Mm(H)

L'ajout (Mpa), la modification(Mpm) ou le retrait(Mpr) de paramètre d'une méthode aura des impactes sur les advices (ADVm) et les méthodes au niveau local (Mm(L)) ou de l'aspect héritier (Mm(H)) qui font appel à cette méthode.

Afin d'illustrer cette règle, nous allons prendre le programme "TJP Example" qui comprend une classe "Demo" et un aspect "GetInfo" présenté dans la figure 35. Au niveau de l'aspect "GetInfo", nous avons une déclaration intertype, deux pointcut, un advice et une méthode "printParameters". Cette méthode est appelée dans l'advice. Selon la règle d'impact, si on ajoute,

modifie ou supprime des paramètres de la méthode "printParameters", l'advice qui appelle cette méthode est automatiquement impacté.

```
aspect GetInfo {

   static final void println(String s){ System.out.println(s); }
   pointcut goCut(): cflow(this(Demo) && execution(void go()));
   pointcut demoExecs(): within(Demo) && execution(* *(..));

   Object around(): demoExecs() && !execution(* go()) && goCut() {
      println("Intercepted message: " + thisJoinPointStaticPart.getSignature().getName());
      println("in class: " + thisJoinPointStaticPart.getSignature().getDeclaringType().getName());
      printParameters(thisJoinPoint);
      println("Running original method: \n" );
      Object result = proceed();
      println("  result: " + result );
      return result;
   }

   static private void printParameters(JoinPoint jp) {
      println("Arguments: " );
      Object[] args = jp.getArgs();
      String[] names = ((CodeSignature)jp.getSignature()).getParameterNames();
      Class[] types = ((CodeSignature)jp.getSignature()).getParameterTypes();
      for (int i = 0; i < args.length; i++) {
         println("  " + i + ". " + names[i] +" : " +types[i].getName() + " = " + args[i]);
      }
   }
}
```

Figure 35 - Exemple règle d'impact: Ajout, modification ou suppression des paramètres d'une méthode

Après la modification des paramètres de la méthode "printParameters" (en jaune dans la figure 35), nous avons recensé un impact: l'advice qui fait appel à la méthode "printParameters" est impacté (en vert dans la figure 35).

Nombre d'impacts prévus par le modèle	Nombre d'impacts effectifs	Nombre d'impacts effectifs prévus par le modèle
1	1	1

Tableau 11 - Résultat expérimentation: Ajout, modification ou suppression des paramètres d'une méthode

On aura les mesures suivantes:

- Précision: (1 /1) * 100 = 100%
- Rappel: (1 / 1) * 100 = 100%

3. Impacts aspect vers objet

Dans un programme, il peut y avoir la situation où la partie objet fait appel à des éléments qui ont été déclarés dans la partie aspect. Il s'agit de déclarations intertype qui seront appelées dans une classe. Les seules modifications qui peuvent se faire à ce niveau sont la modification ou la suppression de la déclaration intertype.

Règles d'impact : I-TYr | I-TYm -> Mm
La suppression ou modification d'une déclaration intertype entrainera des impacts sur les méthodes (Mm) du code objet où la déclaration intertype est invoquée

3.1. Exemple 1

Nous allons prendre un premier exemple présenté dans la figure 36, dans lequel une méthode déclarée dans un aspect est utilisée dans le code objet.

Dans l'aspect "Aspect_Bank", il y a une déclaration intertype qui définit une méthode appartenant à la classe " BankAccount". Cette méthode est utilisée dans des méthodes de ladite classe.

Ainsi, selon notre règle d'impact, on aura les impacts suivants si on supprime la méthode définie par la déclaration intertype dans l'aspect "Aspect_Bank" (en jaune dans la figure 36):

- Toutes les méthodes faisant appel à la méthode seront impactées. Il y en a deux. Nous rappelons que les constructeurs sont considérés comme des méthodes normales.

```
public aspect Aspect_Bank {
        public long nbComptes = 0;

        public pointcut pcCheckBalance(double montant, BankAccount account):
                execution(void BankAccount.withdraw(double)) && args(montant) && target(account);

        void around(double montant, BankAccount account): pcCheckBalance(montant, account)      {
                if((montant - account.balance) > 0)
                        proceed(montant, account);
                else
                        System.out.println("Fonds insuffisants");
        }
        public pointcut pcBalancee():
                set(double BankAccount.balance) ;

        after(): pcBalancee(){
                System.out.println("Modification de la balance du compte ");
        }

        public void BankAccount.display(){
                System.out.println("balance=" + balance);
        }
}
```

```
public class BankAccount
{
        public double balance;

           public BankAccount(double openingBalance)
           {
              balance = openingBalance;
           }

           public void deposit(double amount)
           {
              balance = balance + amount;
              display();
           }

           public void withdraw(double amount)
           {
              balance = balance - amount;
              display();
           }

}
```

Figure 36 - Exemple règle d'impact: Impact aspect vers objet (Exemple 1)

Après la suppression de la déclaration intertype, on constate les impacts au niveau des deux méthodes qui appellent la méthode déclarée dans l'aspect. Ceux-ci sont indiqués en vert dans la figure 36.

Nombre d'impacts prévus par le modèle	Nombre d'impacts effectifs	Nombre d'impacts effectifs prévus par le modèle
2	2	2

Tableau 12 - Résultat expérimentation: Impact aspect vers objet (Exemple 1)

On aura les mesures suivantes:

- Précision: (2 /2) * 100 = 100%
- Rappel: (2 / 2) * 100 = 100%

3.2. Exemple 2

Dans ce second exemple, nous avons pris le programme "Bean Example". Il est composé de deux classes "Demo" (cf. figure 37) et "Point", et d'un aspect nommé "BoundPoint".
Cet aspect, illustré à la figure 37, est composé d'un pointcut, d'une méthode, d'un advice around et de plusieurs déclarations intertype.
Parmi ces déclarations intertype, une d'entre elles implémente l'interface "Serializable" par le biais de la classe "Point" (en jaune dans la figure 37). La méthode "save" (en vert dans la figure 37) située au niveau de la classe " Demo" se charge ensuite d'effectuer la sérialisation. Cette méthode prend en paramètre un objet de type Serializable et un autre de type String. La méthode "save" est enfin appelée dans la méthode "main" de classe démo, où on lui passe comme

argument un objet de la classe "Point" et une valeur de type String. Cela est possible car la classe "Point" implémente "Serializable".

Supprimer cette déclaration intertype entrainera, selon notre règle d'impact, des conséquences au niveau des méthodes se servant de la déclaration intertype. Ainsi, toujours en nous basant sur la règle d'impact, la méthode "main" devrait être impactée parce qu'on passe un objet de type "Point" à la méthode "save" qui prend en paramètre un objet de type Serializable; ce qui n'est plus possible car la classe "Point" n'implémente plus "Serializable".

```
aspect BoundPoint {

  private PropertyChangeSupport Point.support = new PropertyChangeSupport(this);

  public void Point.addPropertyChangeListener(PropertyChangeListener listener){
    support.addPropertyChangeListener(listener);
  }
  ...

  declare parents: Point implements Serializable;

  pointcut setter(Point p): call(void Point.set*(*)) && target(p);

  void around(Point p): setter(p) {
        String propertyName =
      thisJoinPointStaticPart.getSignature().getName().substring("set".length());
        int oldX = p.getX();
        int oldY = p.getY();
        proceed(p);
        if (propertyName.equals("X")){
      firePropertyChange(p, propertyName, oldX, p.getX());
        } else {
      firePropertyChange(p, propertyName, oldY, p.getY());
        }
  }

  void firePropertyChange(Point p, String property,double oldval,double newval) {
        p.support.firePropertyChange(property,new Double(oldval), new Double(newval));
  }
}
```

```
public class Demo implements PropertyChangeListener {

    static final String fileName = "test.tmp";

    public void propertyChange(PropertyChangeEvent e){
        System.out.println("Property " + e.getPropertyName() + " changed from " +
                           e.getOldValue() + " to " + e.getNewValue() );
    }

    public static void main(String[] args){
        Point p1 = new Point();
        p1.addPropertyChangeListener(new Demo());
        System.out.println("p1 =" + p1);
        p1.setRectangular(5,2);
        System.out.println("p1 =" + p1);
        p1.setX( 6 );
        p1.setY( 3 );
        System.out.println("p1 =" + p1);
        p1.offset(6,4);
        System.out.println("p1 =" + p1);
        save(p1, fileName);
        Point p2 = (Point) restore(fileName);
        System.out.println("Had: " + p1);
        System.out.println("Got: " + p2);
    }

    static void save(Serializable p, String fn){
        try {
            System.out.println("Writing to file: " + p);
            FileOutputStream fo = new FileOutputStream(fn);
            ObjectOutputStream so = new ObjectOutputStream(fo);
            so.writeObject(p);
            so.flush();
        } catch (Exception e) {
            System.out.println(e);
            System.exit(1);
        }
    }
...
```

Figure 37 - Exemple règle d'impact: Impact aspect vers objet (Exemple 2)

Après la suppression de la déclaration intertype, nous avons constaté un impact au niveau de la classe "Demo", là où la méthode "save" est appelée avec comme paramètre un objet de type "Point" et un objet de type String.

Nombre d'impacts prévus par le modèle	Nombre d'impacts effectifs	Nombre d'impacts effectifs prévus par le modèle
1	1	1

Tableau 13 - Résultat expérimentation: Impact aspect vers objet (Exemple 1)

On aura les mesures suivantes:

- Précision: (1 /1) * 100 = 100%
- Rappel: (1 / 1) * 100 = 100%

V. Résultats et discussion

1. Impacts objet – aspect

Pour chacun des programmes testés au niveau de la section des exemples d'expérimentions, nous avons effectué une modification dans le code objet en nous basant sur les différentes règles d'impact présentées comme exemple. Le tableau 14 ci-dessous fait un résumé des impacts prévus, observés et pertinents recueillis, ainsi que la précision et le rappel obtenus:

Règles d'impact		Nombre d'impacts prévus par le modèle	Nombre d'impacts effectifs	Nombre d'impacts effectifs prévus par le modèle	Précision	Rappel
Suppression d'une classe		10	10	10	100%	100%
Modification des paramètres d'une méthode	Exemple a	2	2	2	100%	100%
	Exemple b	4	4	4	100%	100%
Changement du type de retour d'une méthode de « void » à type (primitif ou objet)		3	3	3	100%	100%
Suppression d'un lancement d'exception		3	3	3	100%	100%
Total		22	22	22	100%	100%

Tableau 14- Synthèse résultats: impacts Objet vers Aspect

Au total, 4 règles ont été testées, dont une avec deux situations différentes. Avec les exemples utilisés, nous arrivons à une précision totale de 100% et un rappel total de 100% (cf. tableau 14).

2. Impacts aspect – aspect

Dans cette seconde phase de test, nous avons effectué des modifications dans le code aspect et nous avons recensé les impacts sur cette même partie aspect du programme utilisé. Les impacts se sont produits soit dans l'aspect dans lequel le changement a été effectué, soit dans les aspects héritant de l'aspect comprenant le changement, soit dans des aspects associés.

Le tableau 15 ci-dessous fait un récapitulatif des impacts prévus, effectifs, effectifs prévus par le modèle, ainsi que les pourcentages obtenus pour la précision et le rappel.

Règles d'impact	Nombre d'impacts prévus par le modèle	Nombre d'impacts effectifs	Nombre d'impacts effectifs prévus par le modèle	Précision	Rappel
Modification du nom d'un aspect	2	2	2	100%	100%
Passage d'aspect abstrait à non abstrait	8	5	5	100%	62.5%
Suppression d'un pointcut	2	2	2	100%	100%
Suppression d'un attribut	5	5	5	100%	100%
Ajout, modification, suppression de paramètre d'une méthode	1	1	1	100%	100%
Total	**18**	**15**	**15**	**100%**	**83.3%**

Tableau 15 - Synthèse résultats: impacts Aspect vers Aspect

Les programmes testés nous donnent une précision de 100% et un rappel de 83%, montrant ainsi que dans le cas d'impacts aspect vers aspect, notre modèle peut être très efficace.

Cependant, le rappel peut être affecté dans le cas de règles présentant des impacts éventuels. Ces règles vont prédire des impacts qui ne se réaliseront que par la volonté du programmeur.

3. Impacts aspect – objet

Les impacts aspect vers objet n'ont qu'une seule règle. En effet, cette situation ne se produit que lorsqu'une déclaration intertype est utilisée au niveau du code objet. Cette déclaration intertype pourra concerner un attribut ou une méthode, ou modifier l'héritage d'une classe, ou encore faire implémenter une interface par une classe.

Règles d'impact		Nombre d'impacts prévus par le modèle	Nombre d'impacts effectifs	Nombre d'impacts effectifs prévus par le modèle	Précision	Rappel
Suppression d'une déclaration intertype	Exemple 1	2	2	2	100%	100%
	Exemple 2	1	1	1	100%	100%
Total		3	3	3	100%	100%

Tableau 16 - Synthèse résultats: impacts Aspect vers Objet

Suite à nos expérimentations, une précision et un rappel de 100% chacun ont été obtenus. Chacune des méthodes impactées par la suppression de la déclaration intertype au niveau des classes objet a été détectée.

CHAPITRE 6:
CONCLUSION

La programmation orientée aspect prend de plus en plus en plus d'ampleur. Il est donc nécessaire de se pencher sur la maintenance des logiciels élaborés sous ce paradigme de programmation, plus particulièrement sur l'analyse d'impacts de changement. En effet, les nouvelles relations entre classes objet et aspects de ce type de programmation font ressortir un autre niveau de complexité de cette analyse d'impact.

Quatre sortes de relations d'impact émanent de ce type de programme : les impacts objet - objet, les impacts objet – aspect, les impacts aspect – aspect et les impacts aspect – objet.

Un premier modèle, le modèle d'impact du changement pour Java (MICJ), avait déjà été développé dans [5] afin de pouvoir effectuer une analyse d'impact de changements pour ce qui concerne les impacts objet - objet.

Le modèle que nous proposons se veut une suite complémentaire du MICJ, mais adaptée à la programmation orientée aspect. Notre modèle permet de dire, pour chacune des relations d'impact objet – aspect, aspect – aspect et aspect – objet, quelles parties du code seront affectées suite à un changement.

Notre modèle d'analyse d'impact a pour objectif de prendre en compte l'analyse d'impact en cascade. Ainsi, les programmeurs seront en mesure d'avoir une meilleure idée de l'effet de propagation qu'auront tous les changements qu'ils apporteront à un programme. Cela leur permettra aussi de prendre une décision quant à l'application ou non de cette modification.

Ces règles d'impact concernent les changements comme la suppression d'une classe, la modification des paramètres d'une méthode, la suppression ou la modification d'un pointcut, ou encore la suppression de déclarations intertypes.

Afin de montrer l'efficacité de notre modèle, des expérimentations ont été faites pour chacune de nos règles d'impact. Les résultats obtenus montrent que notre modèle est assez efficace quant à la prédiction d'impacts suite à toute modification apportée au code, que ce soit dans la partie objet ou dans la partie aspect. Même si certaines règles indiquent des impacts incertains, elles permettront tout de même au programmeur d'avoir une bonne idée des modifications à faire pour que le système demeure cohérent.

Notre modèle permettra d'améliorer l'analyse de l'impact des changements opérés dans un programme orienté aspect, et ce, avant que le tissage ne soit effectué (analyse statique); ce qui représente un avantage par rapport à la plupart des approches effectuées qui elles, font une analyse après le tissage du programme (analyse dynamique). Combiné au MICJ, notre modèle d'impact pourrait s'avérer très utile aux programmeurs.

Afin de rendre notre modèle encore plus précis, un programme implémentant celui-ci devra être conçu. Ce programme aura pour but de faire ressortir de façon plus explicite, les éléments impactés dans le code aspect après une modification apportée dans le code objet ou le code aspect lui-même.

ANNEXE 1
LISTE DES CHANGEMENTS

Au niveau du code Java

Abréviations	Changement
Au niveau des classes	
Cr	Retrait d'une classe
Cnm	Changement du nom d'une classe
Chr	Retrait d'un héritage
Cir	Retrait d'une interface
Mr	Retrait d'une méthode
Au niveau des méthodes	
Mnm	Modification du nom d'une méthode
Mpa	Ajout de paramètre à une méthode
Mpm	Modification de paramètre d'une méthode
Mpr	Retrait de paramètre d'une méthode
Mtr_m	Changement du type de retour d'une méthode de void à objet
Mv_m	Changement de la visibilité d'une méthode de privé à public
Mt_{sn}	Changement type d'une méthode de statique à non statique
Mt_{ns}	Changement type d'une méthode de non statique à statique
Au niveau des attributs	
Ar	Retrait d'un attribut
Anm	Modification du nom d'un attribut
Atm	Modification du type d'un attribut
Av_m	Modification de la visibilité d'un attribut de privé à public
At_{sn}	Modification d'un attribut de statique à non statique
At_{ns}	Modification du type d'un attribut de non statique à statique
At_{fn}	Modification d'un attribut de final à non final
At_{nf}	
Au niveau des exceptions	
Nrtc	Retrait d'un try/catch

Au niveau du code Aspect

Abréviations	Changement
Au niveau des aspects	
ASr	Retrait d'un aspect
ASnm	Changement du nom d'un aspect
ASt_{an}	Changement d'un aspect d'abstrait à non abstrait
ASha	Ajout d'un lien d'héritage d'un aspect
AShr	Retrait d'un lien d'héritage d'un aspect
ASia	Ajout d'une interface à un aspect
ASir	Retrait d'une interface à un aspect
Au niveau des pointcuts	
PCa	Ajout d'un pointcut
PCr	Retrait d'un pointcut
PCnm	Modification du nom d'un pointcut
PCpm	Modification des paramètres d'un pointcut
PCargm	Modification des arguments d'un pointcut
At_{sn}	Modification d'un attribut de statique à non statique
At_{ns}	Modification du type d'un attribut de non statique à statique
At_{fn}	Modification d'un attribut de final à non final
At_{nf}	Modification du type d'un attribut de non final à final
Au niveau des points de jointure	
JPr	Retrait d'un point de jointure
JPm	Modification d'un pointcut
JPm{dg}	Modification d'un pointcut lié à un assesseur d'un attribut
JPm{mu}	Modification d'un pointcut lié à un mutateur d'un attribut
Au niveau des advices	
$ADVt_{ra}$	Modification d'un advice de after à around
$ADVt_{ba}$	Modification d'un advice de before à around
$ADVt_{ab}$	Modification d'un advice de around à before
$ADVt_{ar}$	Modification d'un advice de around à after
Au niveau des déclarations	
I-TYm	Modification d'une déclaration intertype
I-TYr	Retrait d'une déclaration intertype
DCLm	Modification d'une déclaration (warning ou error)
DCLr	Retrait d'une déclaration (warning ou error)
Au niveau des méthodes	
Mr	Retrait d'une méthode
Mnm	Modification du nom d'une méthode
Mpa	Ajout de paramètre à une méthode
Mpm	Modification de paramètre d'une méthode
Mpr	Retrait de paramètre d'une méthode
Mtr_m	Modification du type de retour d'une méthode
Au niveau des attributs	
Ar	Retrait d'un attribut
Anm	Modification du nom d'un attribut
Atm	Modification du type d'un attribut
Avm	Modification de la visibilité d'un attribut de privé à public

ANNEXE 2
LISTE DES RÈGLES D'IMPACT

Abréviations	Règles d'impact
Cr	Cr -> JPr + [PCr] + [ADVr| | ADVm] + I-TYr + DCL
Cnm	Cr -> JPr + [PCr] + [ADVr| | ADVm] + I-TYr + DCL
Chr	Chr -> Mpm(L) + Mm(L) + JPm + PCm + ADVm + Mm(AS)
Cir	Cir -> [Mr + JPr + PCr| |PCm + ADVr| |ADVm + I-TYr]
Mr	Mr -> JPr + PCm + ADVm
Mnm	Mnm -> JPm + PCm + ADVm
Mpa	Mpa | Mpm | Mpr -> JPpm + PCm +ADVm
Mpm	
Mpr	
Mtr_m	Mtrm -> JPm + PCm + ADVm
Mv_m	Mv_m-> [JPm + PCm + ADVm]
Mt_{sn}	Mt_{sn} | Mt_{ns} -> [JPm + PCm + ADVm]
Mt_{ns}	
Ar	Ar -> JPr{dg} | | JPr{mu} +PCm + ADVm
Anm	Anm -> JPm{dg} | | JPm{mu} +PCm + ADVm
Atm	Atm ->Mpm{mu} + Mtrm{a c} + JPm + PCm + ADVm
Av_m	Av_m -> JPm + PCm + ADVm + Mm{AS}
At_{sn}	At_{sn} | At_{ns} -> JPm{dg} | | JPm{mu} +PCm + ADVm
At_{ns}	
At_{fn}	At_{fn} | At_{nf} -> JPm{dg} | | JPm{mu} +PCm + ADVm
At_{nf}	
Nrtc	Nrtc -> JPr + PCr + ADVr

Abréviations	Règle d'impact
ASr	ASr -> AShr(H) || AShM(H) + PCm(A) + ADVm(A) + Cm
ASnm	ASnm -> AShr(H)
ASt_{an}	ASt_{an} -> Mt_{an}(L) + PCt_{an}(L) + AShr(H) + [Mr(H) || Mm(H) + PCr(H) ||PCm(H)+ADVr(H) || ADVm(H)] + [ADVa(L)]
ASha	AShca -> ASka(L)
AShr	AShcr -> ASm + ASkr + [Mr] + [Ar]
ASia	ASia -> Ma
ASir	ASir -> Mr + ADVm + PCm + Mm
PCa	PCa -> PJa + ADVa
PCr	PCr -> PJr + ADVr + PCm(L) + PCr(H) + PJr(H) + ADVr(H)
PCnm	PCnm -> ADVm + PCm(L) + PCm(H) + ADVm(H)
PCpm	PCpm -> PCm(L) + ADVm + PCm(H) + ADVm(H)
At_{sn}	At_{sn} | At_{ns} -> JPm{dg} || JPm{mu} +PCm + ADVm
At_{ns}	
At_{fn}	At_{fn} | At_{nf} -> JPm{dg} | JPm{mu} +PCm + ADVm
At_{nf}	
$ADVt_{ab}$	$ADVt_{ab}$ | $ADVt_{ar}$ -> [ADVm]
$ADVt_{ar}$	
I-TYm	I-TYr | I-TYm -> Mm
I-TYr	
Mr	Mr -> ADVm + Mm(L) + Mm(H)
Mnm	Mnm -> ADVm + Mm(L) + Mm(H)
Mpa	Mpa | Mpm | Mpr -> ADVm + Mm(L) + Mm(H)
Mpm	
Mpr	
Mtr_m	Mtr_m -> ADVm + Mm(L) + Mm(H)
Ar	Ar -> ADVm + Mm(L) + Mm(H)
Anm	Anm -> ADVm + Mm(L) + Mm(H)
Atm	Atm -> ADVm + Mm(L) + Mm(H)
Avm	Avm -> ADVm(H) + Mm(H)

BIBLIOGRPHIE

[1] Zhang D, Duala-Ekoko E, Hendren L., *Impact Analysis and Visualization Toolkit for Static Crosscutting in AspectJ*, IEEE 17th International Conference on Program Comprehension, pages 30-69, Vancouver, BC, Canada, 17-19 Mai 2009

[2] Shinomi I., Tamai T., *Impact Analysis of Weaving in Aspect-Oriented Programming*, Proceedings of the 21st IEEE International Conference on Software Maintenance,pages 357-660, 26-29 Septembre 2005

[3] Zhang S, Gu Z, Lin Y, Zhao J., *Change Impact Analysis for AspectJ Programs*, IEEE International Conference on Software Maintenance,pages 87-96, Beijing, Sept. 28 2008-Oct. 4 2008

[4] Zhao J., *Change Impact Analysis for Aspect-Oriented Software Evolution*, Proceedings of the International Workshop on Principles of Software Evolution, New York, NY, USA, 2002

[5] M Badri, L. Badri, N Joly, *Un modèle d'analyse d'impact du changement pour java,* thèse de maîtrise, Université du Québec à Trois-Rivières, Canada, 2010

[6] J D Gradecki, N. Lesiecki, *Mastering AspectJ, Aspect Oriented Programming in Java.* Indianapolis, Indiana, USA, 2003

[7] Pawlak, R., Retaillé, J-Ph., Seinturier, L. *Programmation orientée aspect pour Java J2EE.* Paris, France 2004

[8] Badri M., Badri L., St-Yves D., *Dépendances et gestion des modifications dans les systèmes orientés objet : utilisation des graphes de contrôle,* thèse de maîtrise, Université du Québec à Trois-Rivières, Canada, Décembre 2007.

[9] Ackermann C., Lindvall M., *Understanding Change Requests to Predict Software Impact*, 30th Annual IEEE/NASA Software Engineering Workshop, pages 66 - 75, Columbia, MD, USA, Avril 2006.

[10] Zhao J., *Measuring Coupling in Aspect-Oriented Systems*, Information Precessing Society of Japan (IPSJ), pages 14-15, Japon, 2004.

[11] Bernardi M.L., Di Lucca G.A, *An Interprocedural Aspect Control Flow Graph to Support the Maintenance of Aspect Oriented Systems*. IEEE International Conference on Software Maintenance, pages 435-444, Paris, France, 2-5 Octobre 2007.

[12] Wang Y, He X., Wang Q., *Lifecycle based Study Framework of Software Evolution*, International Conference on Computer Application and System Modeling, V3-262 - V3-266, 22-24, Taiyuan, Chine, Octobre 2010.

[13] Bernardi M. L., Di Lucca G. A., Ceccato M., *Workshop on Maintenance of Aspect Oriented Systems*, 13th European Conference on Software Maintenance Reengineering, pages 273-274, Kaiserslautern, Germany, 24-27 Mars 2009.

[14] Przybylek A., *Impact of aspect-oriented programming on software modularity*, 15th European Conference on Software Maintenance and Reengineering, pages 369-372, Oldenbourg, Allemagne, 1-4 Mars 2011.

[15] Dong Z., *Aspect Oriented Programming Technology And The Strategy Of Its Implementation*, International Conference on Intelligence Science and Information Engineering, pages 457-460, Wuhan, Chine, 20-21 Août 2011.

[16] Liu C-H., Chen S-L., Jhu W-L., *Change Impact Analysis for Object-Oriented Programs Evolved to Aspect-Oriented Programs*, Proceedings of the 2011 ACM Symposium on Applied Computing, pages 59-68, New York, NY, USA, 2011.

[17] Burrows R., Ferrari F. C., Lemos O. A.L., Garcia A., Taïani F., *The Impact of Coupling on the Fault-Proneness of Aspect-Oriented Programs: An Empirical Study*, IEEE 21st International Symposium on Software Reliability Engineering, pages 329-338, San Jose, CA, USA, 1-4 Novembre 2010

[18] Störzer M., *Impact Analysis for AspectJ A Critical Analysis and Tool-Based Approach to AOP*, Dissertation, Passau, 2007

[19] Ali H. O., Abd Rozan M. Z. A., Sharif A. M., *Identifying Challenges of Change Impact analysis for software projects*, International Conference on Innovation Management and Technology Research, pages 407-411, Malacca, Malaisie, 21-22 Mai 2012

[20] M Badri, L. Badri, N Fréchette, *Tests de régression dans les systèmes orientés objet : Une approche st a tique basée sur le code*, thèse de maîtrise, Université du Québec à Trois-Rivières, Canada, Avril 2010

[21] J-Y Guyomarc'h, *Une architecture pour l'évaluation qualitative de l'impact de la programmation orientée aspect*, thèse de maîtrise, Université de Montréal, Canada, Mai 2006

[22] Tuong Vinh HO, Manh Tien NGUYEN, *Programmation Orientée Aspect*, Institut de la Francophonie pour l'Informatique, Hanoï, Vietnam, Juillet 2005

[23] http://bambangpdp.wordpress.com/2009/10/25/modularizing-crosscutting-concerns-using-aspect-oriented-programming/ [En ligne]

[24] M. Eaddy, *An Empirical Assessment of the Crosscutting Concern Problem*, Submitted in partial fulfillment of the Requirements for the degree of Doctor of Philosophy in the Graduate School of Arts and Sciences, University of Columbia, 2008

[25] L. Rosenhainer, *Identifying Crosscutting Concerns in Requirements Specifications*, Department of Computer Science, Chemnitz University of Technology, Germany, 2004

[26] R. J. Walker, S. Rawal, J. Sillito, *Do Crosscutting Concerns Cause Modularity Problems*?, Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, Article No. 49, New York, NY, USA 2012

[27] Xiaobing Sun, Bixin Li, Chuanqi Tao, Wanzhi Wen, Sai Zhang, *Change Impact Analysis Based on a Taxonomy of Change Types*, 2010 IEEE 34th Annual Computer Software and Applications Conference, pages 373-382, Seoul, 19-23 Juillet 2010

[28] M. C. O. Maia, R. A. Bittencourt, J. C. Abrantes de Figueiredo, D. D. S. Guerrero, *The Hybrid Technique for Object-Oriented Software Change Impact Analysis*, 2010 14th European Conference on Software Maintenance and Reengineering, pages 252-255, Madrid, 15-18 March 2010.

[29] ZHOU Xiao-bo, JIANG Ying, WANG Hai-tao, *Method on Change Impact Analysis for Object-Oriented Program*, 2011 Fourth International Conference on Intelligent Networks and Intelligent Systems, pages 161-164, Kunming, 1-3 Nov 2011.

[30] M. Ajmal Chaumun, H. Kabaili, Rudolf K. Keller, F. Lustman, *A change impact model for changeability assessment in object-oriented software systems*, Département d'informatique et de recherche opérationnelle, Université de Montréal, Canada, 1998