Génération de diagrammes UML à partir d'une analyse dynamique

Travail de Bachelor réalisé en vue de l'obtention du Bachelor HES

par :

Julien Repond

Conseiller au travail de Bachelor :

(Philippe Dugerdil, Professeur HES)

Genève, 5 juin 2009

Haute École de Gestion de Genève (HEG-GE)

Informatique de Gestion

Déclaration

Ce travail de Bachelor est réalisé dans le cadre de l'examen final de la Haute école de gestion de Genève, en vue de l'obtention du titre de Bachelor d'informaticien de gestion HES. L'étudiant accepte, le cas échéant, la clause de confidentialité. L'utilisation des conclusions et recommandations formulées dans le travail de Bachelor, sans préjuger de leur valeur, n'engage ni la responsabilité de l'auteur, ni celle du conseiller au travail de Bachelor, du juré et de la HEG.

« J'atteste avoir réalisé seul le présent travail, sans avoir utilisé des sources autres que celles citées dans la bibliographie. »

Fait à Charmey, le 5 juin 2009

Julien Repond

Remerciements

Je tiens à remercier Messieurs Philippe Dugerdil et Sébastien Jossi qui ont assuré le suivi de mon travail, m'ont apporté leurs conseils et ont toujours fait preuve d'une grande disponibilité à mon égard.

Merci également à Monsieur David Sennhauser qui m'a présenté son travail de Bachelor duquel s'inspire cette réalisation et qui m'a permis de mieux appréhender le sujet.

Je souhaite également remercier mes proches et ma famille qui m'ont soutenu tout au long de la réalisation et m'ont accordé de leur temps pour la relecture et la correction de ce travail.

Finalement, un grand merci à Monsieur Damien Baumgartner, Directeur informatique de la filiale Asset Management chez Pictet & Cie, qui a accepté d'être mon juré.

Sommaire

La réingénierie logicielle est devenue un des principaux outils de compréhension des systèmes existants mis en œuvre lors de la phase de maintenance. Les ingénieurs logiciels requièrent essentiellement à des analyses, dites statiques, qui utilisent le code source du logiciel pour en reconstruire ses modèles et sa documentation. Nous nous focalisons sur l'architecture actuelle, telle qu'elle a été pensée par les concepteurs, sans tenir compte de sa réelle utilisation. Il n'y a par conséquent aucune comparaison avec l'exécution du programme pour évaluer si l'architecture convient au réel emploi. Une autre approche, l'analyse dynamique, tend à s'appuyer sur les informations résultantes de l'exécution des services comme les indicateurs de performances, les interfaces de monitoring fournies par l'environnement d'exécution ou encore les traces d'exécution. A la différence des techniques statiques, nous étudions les processus en cours de réalisation pour reproduire des modèles d'architecture fonctionnels.

Dans mon mandat, j'ai été amené à poursuivre les travaux réalisés sur un analyseur de trace d'exécution (Dugerdil, Jossi, 2009). Cet analyseur apporte une approche originale pour restituer les modèles d'un programme : utiliser une analyse statistique de la trace, basée sur la segmentation de celle-ci, pour produire des regroupements fonctionnels de classes triées grâce à leur facteur de corrélation, leur couplage. Premièrement, l'avantage d'utiliser les traces d'exécution pour produire ce résultat, est de se détacher d'un quelconque environnement d'exécution. La matière première de l'analyse, la trace, est indépendante du langage de développement et de tous les outils s'y référant. De plus, ces ensembles de classes corrélées, nommés clusters, fournissent une architecture fonctionnelle élaborée en observant le comportement réel du programme. Dès lors, une évolution naturelle de l'analyseur pour aboutir l'analyse est de fournir un moyen de comparaison graphique de ses clusters avec l'architecture initiale pour pouvoir évaluer la pertinence de cette dernière et entreprendre des modifications, ainsi que de représenter les flux d'information de la trace. Ce que nous entendons par la représentation des flux d'information, c'est produire un graphique montrant le déroulement d'une fonctionnalité de l'application, l'échange de messages entre les objets durant l'exécution. Mon travail a ainsi porté sur la réalisation de ces deux évolutions.

Concernant le premier point, l'idée initiale était de représenter les composants fonctionnels résultant de l'analyse, les clusters, dans un format d'échange standard, XML Metadata Interchange (XMI). Le modèle pourrait ainsi être inséré dans un logiciel

de modélisation afin de déléguer la représentation graphique à ce dernier. J'ai étudié les langages de modélisation, Unified Modeling Language (UML), et de méta-modélisation, Meta-Object Facility (MOF) et Eclipse Modeling Framework (EMF), ainsi que le format d'échange de modèle. Le manque de ressources pour la manipulation du format XMI et les limites de ce dernier dans le domaine de la représentation graphique m'ont incité à interagir immédiatement avec un environnement de modélisation extensible, Rational Software Architect (RSA). Ce choix m'a tout d'abord contraint à migrer l'analyseur original dans la même technologie que l'outil de modélisation, l'environnement de développement Eclipse, pour ensuite implémenter la fonctionnalité de comparaison de l'architecture fonctionnelle avec l'architecture initiale.

La suite de mon mandat s'est concentrée sur la représentation graphique des flux d'informations d'une trace d'exécution. Au sein de la méthodologie UML, c'est un des rôles du diagramme de séquence de fournir une représentation de ces flux. La principale problématique a été de donner une vue simplifiée mais pertinente de ces flux, malgré les millions d'appels pouvant constitués une trace. J'ai débuté par une analogie entre les éléments présents dans la trace et leur représentation dans un diagramme de séquence. Ensuite, j'ai étudié divers travaux portant sur les procédés de simplification d'une trace et tenter de voir dans quel mesure je pouvais les appliquer à mon cas de figure. J'ai finalement implémenté la fonctionnalité de génération d'un diagramme de séquence représentant les interactions entre les éléments de la trace. Pour ce faire, j'ai retenu certaines techniques de simplification que j'ai adaptées au contexte. J'ai terminé par une étude de cas pour évaluer l'efficacité des fonctionnalités et leur utilisabilité.

Table des matières

Liste des Tableaux

Liste des Figures

Introduction

Dans le cycle de vie d'un projet, « le cabinet de recherche Forrester estime que qu'aujourd'hui les entreprises dépensent 80% de leurs ressources de développement dans la maintenance des systèmes existants » (Murphy, 2008). Le processus de réalisation d'une application passe par l'étape de conception durant laquelle les développeurs, conjointement avec les multiples intervenants techniques et métiers, définissent l'architecture logicielle à l'aide de diagrammes. Malheureusement, lors de corrections ou d'évolutions, les modifications y sont rarement répercutées. La logique est imprimée dans la tête des développeurs et se perd lors de leurs départs, ce qui rend la tâche de maintenance beaucoup plus complexe.

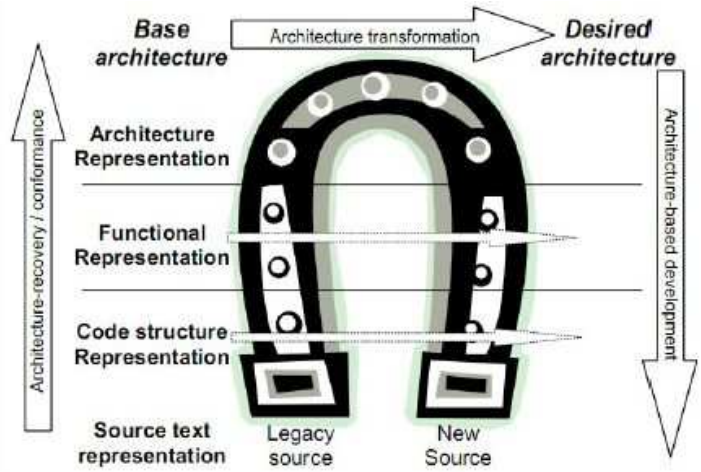
Pourtant, pour intervenir sur un logiciel, il est primordial de comprendre le fonctionnement de ce dernier. Pour éclaircir la notion de « comprendre un logiciel », je vais reprendre cette définition :

> *« Une personne comprend un programme lorsqu'elle est capable d'expliquer ce dernier, sa structure, son comportement, ses effets sur son contexte opérationnel et ses relations au domaine d'application en des termes qui sont qualitativement différents des éléments utilisés pour construire le code source du programme. »*
>
> *(Biggerstaff, Mitbander, Webster, 2004)*

Par conséquent, Il est indispensable de retrouver l'architecture d'un logiciel pour pouvoir ensuite l'expliquer et la comparer avec celle pensée lors de la phase de conception. C'est uniquement une fois la compréhension acquise que nous pouvons entreprendre des évolutions. La rétro-ingénierie est la reconstruction des modèles du système pour apporter cette compréhension. Cette phase « bottom-up » est représentée en figure 1, sur le modèle en fer à cheval du « Software Engineer Institute » (SEI).

Figure 1

Modèle de réingénierie en fer à cheval du SEI



Source : Bergey J., Smith D., Weidermann N., Woods S. *Options Analysis for Reengineering (OAR): Issues and Conceptual Approach*. Software Engineer Institute, Carnegie Mellon University, Technical Report, CMU/SEI-99-TN-014, septembre 1999

Nous constatons sur ce modèle qu'il existe des paliers de transformation lors de la rétro-ingénierie. Il est bien entendu qu'il n'y a pas utilité à générer la documentation de toute l'architecture si nous souhaitons effectuer des modifications ne portant que sur les couches basses des niveaux d'abstraction, comme la correction de l'algorithme d'une fonction précise du système.

Parmi les techniques de rétro-ingénierie, je distinguerais deux catégories : les techniques d'analyses statiques qui s'appuient sur la structure de l'application, comme le code source, et les analyses dynamiques qui consistent à « l'analyse du comportement d'un système logiciel pour en extraire ses propriétés » (Hamou-Lhadj, Lethbridge, 2004), en d'autres termes, l'étude des processus qui s'y déroulent. Bien que de nombreux outils tentent d'implémenter des procédés d'analyse statiques, comme la reconstruction d'un diagramme de classe depuis le code source, il subsiste plusieurs inconvénients à cette approche. Tout d'abord, « le code source ne contient que très peu d'information sur les composants systèmes de hauts niveaux d'abstraction. » (Dugerdil, Jossi, 2009) Nous pouvons très facilement regénérer un diagramme de classe à partir du code source pour se faire une idée de l'implémentation, mais il est beaucoup plus difficile de remonter jusqu'à un diagramme de composant, c'est-à-dire avoir une vue plus générale sur l'architecture. D'autre part, pour autant que nous reproduisions l'architecture actuelle, nous n'avons aucune garantie qu'il y ait eu réel soucis quant à un design spécifique lors de son élaboration.

En d'autres termes, nous ne sommes pas sûrs que l'architecture reproduite soit réellement efficiente pour le cas d'utilisation. Les analyses dynamiques, qui produisent ou vérifient leur résultat en observant l'exécution du programme, utilisant pour cela les indicateurs de performances, les interfaces de monitoring fournies par l'environnement d'exécution ou encore les traces d'exécution, peuvent pallier à ces manques. Attention, il ne faut pas émettre un choix catégorique quant à l'utilisation de l'une ou l'autre des méthodes, mais bien les voir comme des solutions complémentaires à la compréhension d'un programme. Cependant, probablement de part sa jeunesse, sa complexité de mise en œuvre et la masse d'information à manipuler, l'analyse dynamique est nettement moins répandue et exploitée que l'approche statique.

Dans le cadre de ce mandat, je vais poursuivre le travail réalisé sur un analyseur de trace d'exécution (Dugerdil, Jossi, 2009). La fonctionnalité majeure de l'outil est le regroupement des classes présentes dans la trace d'exécution en composants fonctionnels appelés cluster. Le critère de regroupement est le calcul d'un facteur de corrélation des classes. Pour définir ce facteur, la trace est segmentée en bloque de taille égal, puis, nous identifions dans quel segment apparaissent chaque classe, comme démontré au tableau 1.

Tableau 1

Matrice de corrélation

	Segment1	Segment2	Segment3	Segment4	Segment5	Segment6
Classe1	X		X	X		
Classe2	X		X	X		
Classe3		X	X		X	
Classe4	X					X
Classe5		X	X		X	

Lorsque des classes ont un pourcentage de segments communs plus élevés qu'un ratio défini par l'utilisateur, ces classes sont regroupées en cluster. Ainsi, dans le tableau 1, Imaginons que le ratio de corrélation soit configuré à 80%. Les classes 1 et 2 apparaissent chacune dans 3 segments et le nombre de segments communs est également de 3, elles ont donc un facteur de corrélation de 100% et forment un cluster. Idem pour les classes 3 et 5. Cette matrice va ainsi définir des composants fonctionnels, basés sur le fait que les objets travaillent ensemble tout au long de l'exécution du programme. Nous obtenons une architecture fonctionnelle basée sur l'analyse du comportement de l'exécution du logiciel qui est une représentation d'un

niveau d'abstraction plus élevé que le simple diagramme de classe d'implémentation. Ceci peut servir de base à une refonte de l'architecture vers un choix adéquat à l'usage réel en environnement d'exécution.

Conceptuellement, l'objectif de mon mandat a été de :

- Fournir un moyen de comparaison de cette architecture fonctionnelle avec l'architecture statique des packages et des classes. Ainsi, l'utilisateur pourra se représenter, au sein de l'architecture statique, sous forme de diagramme de classe, quelles sont les classes qui collaborent fortement lors de l'exécution d'un cas d'utilisation.

- Donner une représentation aux flux d'information qui se déroulent dans la trace. Ces flux sont les échanges de messages entre les classes. Ainsi, montrer de manière séquentielle à l'utilisateur, le déroulement du programme pour un cas d'utilisation donné.

Technologiquement, la première problématique, abordée au chapitre 1, a été de décider quel est le langage ou le format graphique le plus adapté pour remplir nos objectifs conceptuels et de là, choisir les technologies qui nous permettront de fournir le résultat escompté. Nous verrons ainsi que le format retenu est UML et comme l'approche de la clusterisation est un concept totalement nouveau et absent de ce langage, dans le chapitre 2, nous avons dû réfléchir à son intégration : comment représenté un cluster dans un modèle UML, quel diagramme choisir pour les clusters et pour les flux, et comment comparer l'architecture fonctionnelle avec l'architecture statique. Nous avons également dû faire face à un souci persistent à l'analyse dynamique, la gestion de l'énorme masse d'information. Pour ce faire, au chapitre 3, nous étudierons quelques procédés de simplification de la trace déjà mis en œuvre dans ce contexte. Nous présenterons, au chapitre 4, les techniques retenues et l'implémentation réalisée et nous finirons avec une étude de cas, chapitre 5, pour évaluer l'efficacité des résultats et les éventuelles améliorations qui pourraient être effectuées.

1. Choix technologique

L'analyseur de trace original, dont vous pouvez trouver la documentation utilisateur en annexe 1 et l'architecture en annexe 2, a été développé en Java, avec une interface Swing, j'ai par conséquent limité mes recherches à cette technologie. Partant des deux problèmes à résoudre qui sont la représentation graphique des clusters afin de comparer cette architecture fonctionnelle avec l'implémentation actuelle, ainsi que la représentation graphique des flux d'information de la trace, la première difficulté est de définir les formats des graphes. Pour en faciliter la compréhension aux intervenants, il est important que ces derniers « s'appuient sur des représentations standard des logiciels, semblables à celles qu'on utilise pour le développement » (Dugerdil, 2008). D'une part pour en faciliter l'interprétation aux intervenants grâce à des outils connus et d'autre part, pour permettre une comparaison avec l'architecture de base grâce à des vues de même nature. Ainsi, les diagrammes UML semblent être la meilleure alternative. J'ai annoncé précédemment que les clusters s'apparentaient à des composants fonctionnels. Il est donc naturel d'utiliser un diagramme de composant ou de classe pour représenter cette architecture. Concernant le flux d'information, nous souhaitons représenter le déroulement des appels au sein de la trace, les interactions entre les différentes classes, ou à un niveau d'abstraction plus élevé, les clusters. C'est le rôle du diagramme de séquence que d'apporter cette vision.

Comme nous sommes dans un contexte de diagrammes existants, ceux de classe et de séquence, il s'agit d'identifier les moyens d'interagir avec ces outils. Pour ce faire, il est important de comprendre qu'est-ce que UML et sur quoi se basent les outils de modélisation pour en définir une implémentation.

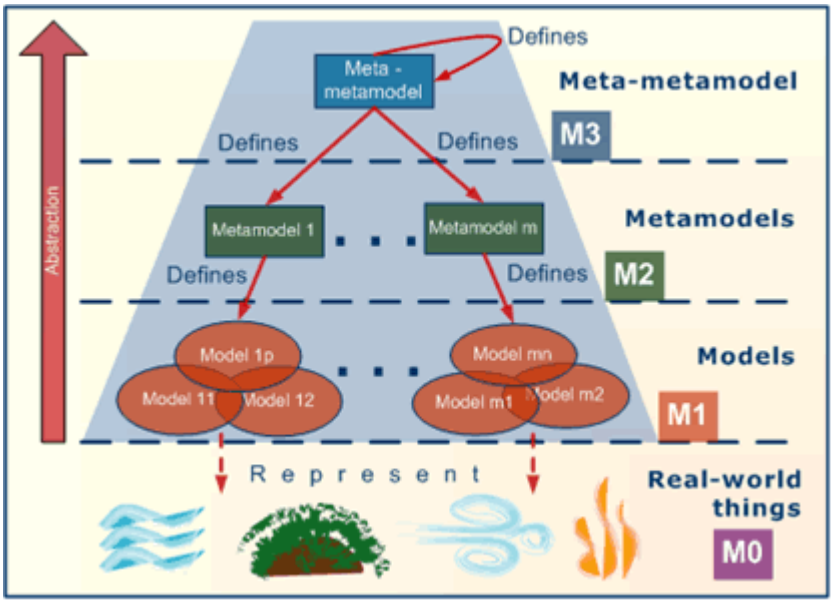
1.1 Meta-Object Facility et XML Metadata Interchange

UML est une spécification de « l'Object Management Group » (OMG). C'est un langage de modélisation graphique devenu un standard industriel. Il en est actuellement à sa version 2.2. Lors de son évolution, à la vue des multiples implémentations qui en ont été faites par les outils de modélisation et de leur incapacité à communiquer entre eux, d'échanger les modèles résultants, l'OMG a entrepris un chantier gigantesque afin d'améliorer l'interopérabilité. Elle a ainsi initié la spécification « Meta-Object Facility » (MOF) qui a pour but de décrire la syntaxe et la structure d'un métamodèle, que ce soit UML ou tout autre métamodèle comme le « Common Warehouse Metamodel » (CWM) chargé de simplifier les échanges de métadonnées d'entrepôts de données ou de « business intelligence », ou encore le

« Software Process Engineering Metamodel » (SPEM) qui s'occupe de décrire le processus de production logiciel. MOF sert également de base pour le « Model Driven Architecture » (MDA). MDA est une approche développement qui passe par l'élaboration des modèles du système pour ensuite générer le code de l'application, d'où le nom d'architecture pilotée par les modèles, car les modifications sont effectuées sur les modèles pour être répercutées par la suite sur le code. Sans entrer en détail dans toutes ces notions, arrêtons-nous sur l'architecture à 4 couches de MDA.

Figure 2

Architecture MDA à quatre couches



Source : Djuric, D., Gaševic, D., Devedžic, V. The Tao of Modeling Spaces. *Journal of Object Technology*, 2006, volume 5, numéro 8, pp 125-147.

Sur le schéma de la figure 2, MOF se situe au niveau de la couche M3 des niveaux d'abstraction, il est le méta-métamodèle. Comme on peut le constater, le méta-métamodèle « s'auto définit », c'est-à-dire qu'il utilise sa propre terminologie, composée de méta-classe représentant les concepts et de méta-association représentant les liens entre ces concepts, pour se définir et qu'il la respecte. Il est ensuite utilisé pour définir le métamodèle UML, se situant à la couche M2. UML va finalement nous permettre de définir les modèles spécifiques à notre domaine d'application et qui eux, représenteront la réalité de la couche M0. De par cette architecture, il a été possible de définir un protocole commun pour l'échange d'information entre les modèles de la couche M1, le « XML Metadata Interchange »

(XMI). C'est grâce à cette spécification de l'OMG que les outils de modélisation sont capables d'échanger les modèles qu'ils génèrent.

Ainsi, pour interagir avec les environnements de modélisation, il est nécessaire d'utiliser l'interface d'échange XMI. Il y a donc lieu de produire du XMI représentant nos modèles, nos diagrammes, et qui pourra ensuite être transmis à l'outil pour affichage. Malheureusement, après avoir mené une étude sur les API Java, « Application Programming Interface », permettant de générer du XMI, il s'avère qu'il n'en existe que très peu. Plusieurs outils ont leur propre implémentation de la spécification JSR-000040 « Java Metadata Interface » (JMI) qui définit les interfaces standards Java pour la manipulation des composants MOF, y compris XMI, mais ce sont des implémentations propriétaires qui ne sont pas accessibles aux développeurs. A ma connaissance, je n'ai trouvé que le projet « MetaData Repository » (MDR) de Netbeans qui offre la possibilité d'interaction.

Néanmoins, après une étude plus approfondie de XMI, il s'avère qu'elle se distingue en deux spécifications : XMI qui sert de format d'échange des modèles uniquement, et « XMI-Diagram Interchange » (XMI-DI) ou « UML Diagram Interchange », plus récente et en cours d'élaboration au prêt de l'OMG, qui elle, sert à l'échange des diagrammes. Je précise « en cours d'élaboration », car d'après les informations sur le site de l'OMG, la version actuelle, 1.0, correspond à la version de travail (OMG, 2009). Ainsi, la fonctionnalité d'échange de diagramme n'a pas pu être implémentée dans le projet MDR et l'unique usage de XMI ne nous permet pas la représentation des diagrammes.

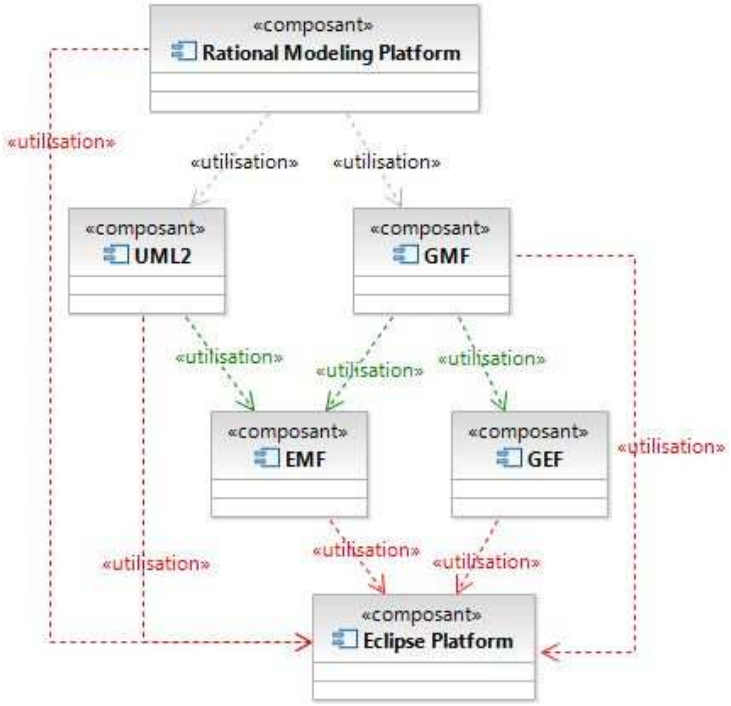
Suite à ce constat, il est nécessaire de s'orienter vers une autre alternative : l'interaction immédiate avec un environnement de modélisation extensible. Au lieu de me focaliser sur la génération du XMI, je vais communiquer directement avec l'outil afin de créer mes diagrammes. Je lui déléguerai la tâche de sauvegarde des diagrammes et de génération du XMI en vue d'un échange de modèle avec une autre solution. En analysant les outils du marché et en réalisant quelques prototypes, je me suis focalisé sur « Rational Software Architect » (RSA) qui est livré avec son API « Rational Modeling Platform » (RMP). D'une part, j'ai pu réaliser les diagrammes qui nous intéressent, d'autre part, il utilise un format proche du XMI pour la sauvegarde des fichiers et intègre la sauvegarde des diagrammes dans ce format, et finalement, permet l'import et l'export au format XMI (sans les diagrammes pour ce cas, étant donné que la spécification n'est pas aboutie). Intéressons-nous de plus prêt à l'architecture de la « Rational Modeling Platform ».

1.2 Rational Modeling Platform

RSA utilise la « Rational Modeling Platform » (RMP), pour la création et la manipulation des modèles et diagrammes UML. Cette plateforme a été construite sur les technologies Eclipse et fournit des API permettant l'interaction avec l'environnement de modélisation.

Figure 3

Architecture de la « Rational Modeling Platform »



Comme on peut le constater sur la figure 3, RMP et les composants Eclipse utilisés reposent avant tout sur la plateforme Eclipse. Ainsi, l'application est exécutée dans un plan de travail Eclipse, son « workbench », et vient s'intégrer parfaitement à l'interface utilisateur standard de l'outil. Cela signifie également que lors de notre implémentation, nous avons à disposition toutes les API de développement Eclipse, principalement par l'intermédiaire du « Plug-in Development Environment » (PDE), mais également par le biais de tous les projets maintenus par la fondation dont UML2, le « Graphical Modeling Framework » (GMF), le « Eclipse Modeling Framework » (EMF) et le « Graphical Editing Framework » (GEF) font partis.

Alors que MOF est une spécification, EMF en est une implémentation réalisée par IBM et la fondation Eclipse. C'est un framework de modélisation dédié à la construction d'outils basés sur une structure de modèle de données. Les éditeurs, en s'appuyant

sur la terminologie MOF, ont redéfini leur propre méta-métamodèle nommé « Ecore ». Une des principales fonctionnalités d'EMF est de générer une API, en se basant sur un métamodèle défini à l'aide d'un fichier Ecore (très proches du XMI), d'interfaces Java annotées ou encore d'un schéma XML, et permettant de créer et manipuler les modèles qui respecteront le métamodèle défini initialement. De plus, il génère un éditeur basique en arborescence qui permet d'interagir avec l'API pour la création et la modification de ces entités. En supplément d'Ecore, un méta-métamodèle « Essential MOF » (EMOF) ainsi que les outils de manipulation inhérents ont été implémentés pour faciliter les échanges entre le monde EMF et MOF. UML2 est simplement une de ces APIs générées représentant le métamodèle UML en version 2.0. Ce composant contient ainsi la structure nécessaire pour la création et la modification de modèles UML et les méthodes permettant de les manipuler.

Concernant la partie visuelle, GEF est un composant destinée à la création d'éditeur graphique riche depuis un modèle, qu'il soit ou non réalisé à l'aide d'EMF. GMF est la dernière brique graphique venant coupler GEF et EMF, afin de faciliter la génération d'éditeur basé sur un modèle EMF.

En plus « d'augmenter significativement l'utilisabilité de l'outil grâce à son intégration dans un environnement de développement intégré » (Hamou-Lhadj, Lethbridge, 2004), j'ai choisi cet outil en raison de :

- Son API accessible pour la manipulation des modèles.
- Le format de sauvegarde Ecore qui se trouve être très proche du XMI et qui autorise la persistance des diagrammes.
- Les possibilités d'échange entre le monde MOF et EMF, ainsi que les évolutions que nous pouvons attendre de la part de l'éditeur pour se maintenir à jour avec les évolutions de XMI et la prise en charge du format d'échange des diagrammes si la spécification « Diagram Interchange » aboutie.

Néanmoins, un point important à souligner, son utilisation me contraint à migrer l'analyseur de trace dans cette technologie.

2. Intégration au métamodèle UML

Dans l'état actuel, l'analyseur de trace fournit déjà une fonctionnalité de représentation graphique des clusters. Mais celle-ci n'est pas élaborée à l'aide de « représentations standards des logiciels, semblables à celles qu'on utilise pour le développement » (Dugerdil, 2008), et ne permet pas la comparaison avec l'architecture initiale. Il nous faut donc trouver une alternative au sein des diagrammes UML. Comme le concept de cluster est nouveau et étranger à UML, il n'est pas possible de le représenter avec la notation de base. Je vais tout d'abord définir la terminologie liée à ce concept en étendant la notation UML. Nous pourrons ainsi intégrer les nouvelles entités au sein d'un modèle UML. Je décrirai ensuite comment j'ai entrepris la représentation des clusters et comment il est possible de les comparer avec l'implémentation de base. Je terminerai en expliquant quelle est la représentation la plus adaptée pour décrire les flux d'informations au sein de la trace.

2.1 Définition d'une terminologie

Pour intégrer le concept de cluster dans un modèle UML, nous avons besoin de définir sa terminologie. Pour ce faire, nous avons recours à un profil qui étendra la notation UML. Un profil est un ensemble de stéréotypes appliqués à des métaclasses ou métaassociations pour représenter un domaine d'application précis. Dans le langage UML, il existe des profils prédéfinis, comme celui d'analyse, qui permet de réaliser un diagramme de robustesse et d'ajouter un stéréotype à une classe pour définir si elle représente une entité, une limite ou un contrôleur. Dans notre cas, nous allons créer notre propre profil, définir les stéréotypes qui nous seront utiles, et quels types de métaclasses ou métassociations du modèle UML ils étendront.

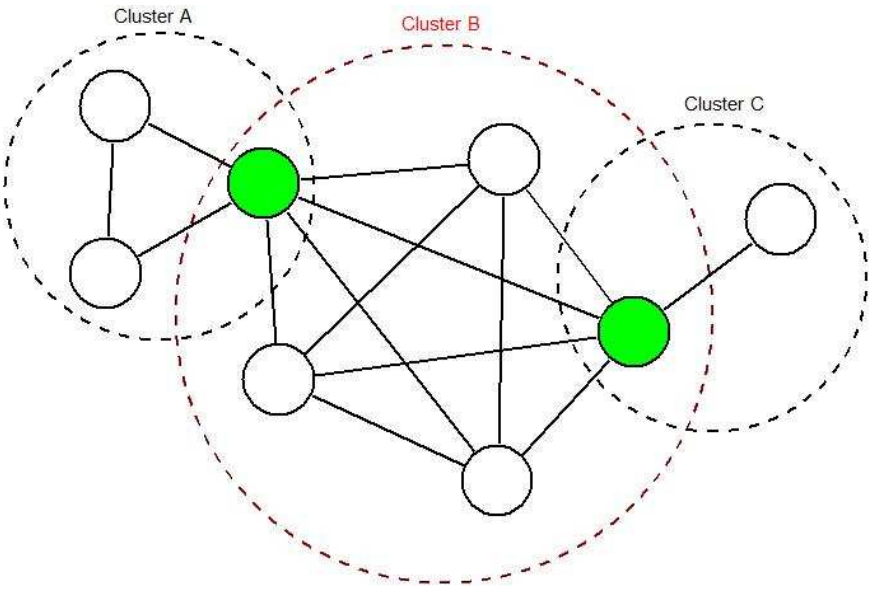
En me basant sur le travail lié à la clusterisation (Dugerdil, Jossi, 2009), je débuterai en donnant une définition d'un cluster relatif à notre concept : « Un cluster est un composant fonctionnel, constitué de classes ayant un facteur de corrélation minimum avec l'ensemble des classes présentes dans ce dernier et qui est calculé grâce à la technique de segmentation d'une trace d'exécution ». Par conséquent, un cluster est avant tout un composant et il nous faut créer un stéréotype « cluster » qui étendra la métaclasse UML « composant ».

Détaillons maintenant la suite de la définition portant sur « la sélection des classes en fonction de leur facteur de corrélation minimum ». Nous avons vu précédemment que le facteur de corrélation de deux classes correspond à leur fréquence de présence

commune dans les segments d'une trace. Le résultat du calcul est un ratio qui est comparé à un pourcentage prédéfini par l'utilisateur et qui doit être supérieur à celui-ci pour autoriser le regroupement en cluster. C'est-à-dire que les classes qui constituent un cluster doivent avoir un facteur de corrélation supérieur à la valeur configurée par l'utilisateur, et ce, avec l'ensemble des classes présentes au sein du cluster.

Figure 4

Graphe G de corrélation des classes



La figure 4 représente cette situation à l'aide d'un graphe G. Les nœuds sont les classes présentes dans l'analyse. Les arêtes représentent les corrélations supérieures au pourcentage défini et les cercles pointillées les clusters. Ainsi, un cluster est possible lorsqu'on peut réaliser un sous-graphe maximal complet du graphe G (Dugerdil, Jossi, 2009). Dès lors, on constate le problème qui se pose : à la différence du composant UML, une classe peut appartenir à plusieurs clusters, comme démontré sur le graphe à l'aide des nœuds coloriés en vert.

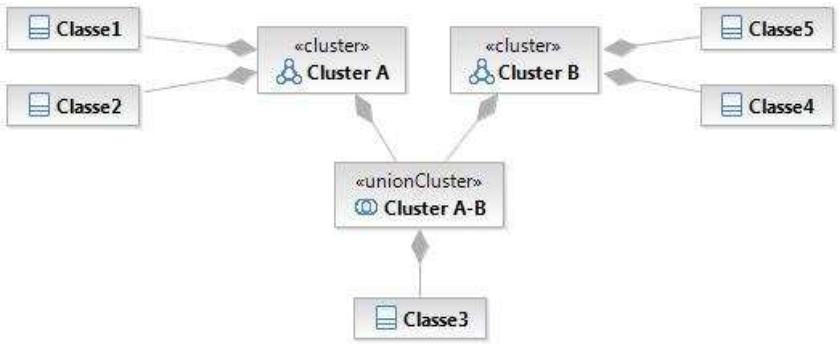
Il faut donc définir comment seront représentées les relations d'appartenance d'une classe à un cluster. Dans le cas d'UML, l'appartenance d'une classe à un composant est représentée par un cercle du côté du containeur avec un symbole « + » en son centre. Néanmoins, la classe doit avoir été créée à l'intérieur du composant pour pouvoir représentée cette association sur un diagramme. Cela soulève un second problème : nous verrons plus tard que la méthodologie d'analyse débute par une transformation du code source Java vers un modèle UML pour posséder l'architecture de base afin de la comparer avec nos clusters. Nous souhaitons donc conserver cette

architecture intacte et nous ne pouvons pas modifier l'emplacement des classes pour les intégrer à un quelconque cluster. J'ai donc opté pour l'association de composition, car un composant cluster est composé de classes.

Nous devons également faire un choix de présentation pour les jonctions des ensembles : est-ce qu'au sein d'un modèle UML, une classe peut appartenir à plusieurs clusters, auquel cas elle possèdera une association pour chaque appartenance, ou doit-on distinguer ce cas particulier ?

Figure 5

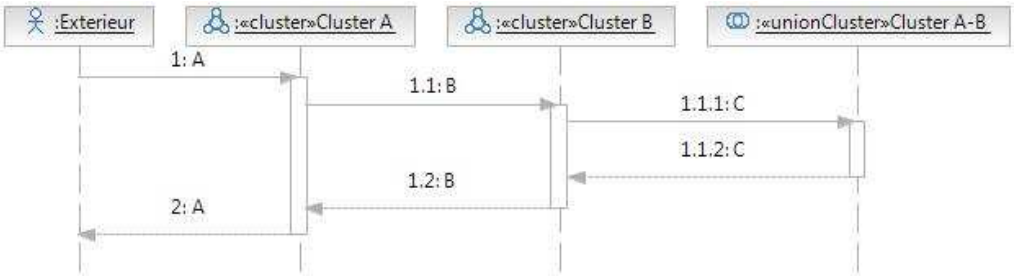
Représentation des sous-clusters



La figure 5 montre comment nous avons choisi de représenter ce cas de figure. Lorsqu'il y a regroupement, un sous-cluster est créé, stéréotype « unionCluster » qui étendra « composant », et il aura une association « composé de » sur les classes faisant parties de l'ensemble. Comme nous le verrons dans l'implémentation, les clusters seront utilisés pour être représentés dans un diagramme de séquence, afin de montrer les flux d'information. Cependant, lorsqu'un message est envoyé à une classe appartenant à deux clusters, sur quelle ligne de vie, représentant les clusters, envoie-t-on le message ?

Figure 6

Envoie de message à un sous-cluster



La figure 6 démontre que l'utilisation d'un sous-cluster résous ce problème. Il suffit d'envoyer les messages destinés à des classes appartenant à deux clusters sur la ligne de vie du sous-cluster.

Finalement, pour organiser l'ensemble des éléments créés par notre analyse, j'ai conçu un stéréotype « clusteringPackage » qui étend « package » et qui servira de containeur à l'ensemble des clusters, sous-clusters, association et différents diagrammes afin de ne pas éparpiller tous ces objets à travers le modèle.

2.2 Choix des représentations

2.2.1 Représentation des clusters

Gardons à l'esprit notre objectif qui est de donner la possibilité à l'utilisateur de comparer les classes constituant un cluster avec une représentation de l'implémentation initiale. Précisons que dans ce cas, l'implémentation initiale se présente sous la forme d'un diagramme de classe. Grâce à l'utilisation de la plateforme Rational, il est possible de récupérer cette structure par le biais d'une transformation de Java vers UML. Ainsi, l'utilisateur aura à disposition, au sein d'un modèle UML, les classes telles qu'elles ont été réparties par package lors de la phase de développement. Il est ensuite libre de créer un diagramme de classe en n'en sélectionnant qu'un sous-ensemble ou la totalité. Pour éviter à l'utilisateur de devoir comparer lui-même ce résultat avec un nouveau diagramme de classe contenant les mêmes objets mais regroupés différemment, il est plus judicieux de mettre en évidence l'appartenance de classes à un cluster immédiatement au sein du même schéma. Cela n'empêche pas l'utilisateur de créer un nouveau diagramme uniquement avec les clusters, et leur contenu, pour n'avoir qu'une vue des classes impliquées dans la clusterisation. Ainsi, l'idée est de lister les clusters et leur contenu dans une nouvelle vue Eclipse, en parallèle du diagramme de classe représentant l'implémentation initiale, et de laisser l'utilisateur mettre en évidence, à l'aide de cette vue, les clusters désirés immédiatement dans le diagramme de l'implémentation initiale. Le choix de l'implémentation finale sera présenté dans le chapitre « Implémentation ».

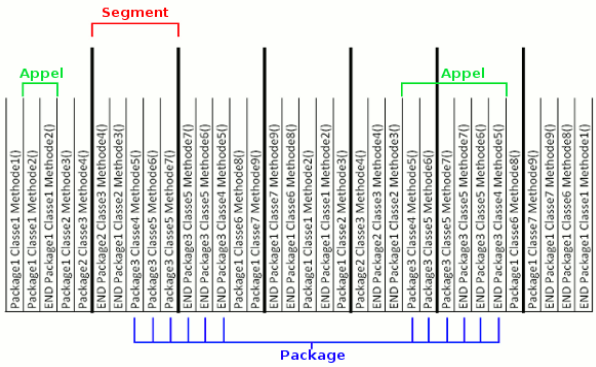
2.2.2 Représentation des flux d'information

Analysons tout d'abord le procédé de collecte d'information de la trace. La trace d'exécution est obtenue grâce à une instrumentation du code source. La technique est similaire à celle de la journalisation d'événements ou d'erreurs d'une application. L'instrumenteur est un logiciel ajoutant des instructions au début et à la fin de chaque

méthode du code source analysé. Ces instructions seront exécutées lors du lancement du code instrumenté et inscriront dans un journal les événements de début et de fin d'exécution d'une méthode avec les informations désirées comme le nom de la classe contenant la méthode, le nom du package ou encore les paramètres d'entrée. La définition d'une syntaxe identique à chaque entrée rend la trace manipulable par un logiciel tiers.

Figure 7

Représentation d'une trace d'exécution



Dans la figure 7, nous avons la représentation d'une trace d'exécution. La ligne, ou entrée, est la plus petite unité temporelle de la trace. Elle correspond au début ou à la fin d'un appel, différenciés grâce au mot clé END en début de ligne pour déterminer le second cas. Pour chaque entrée, on retrouve :

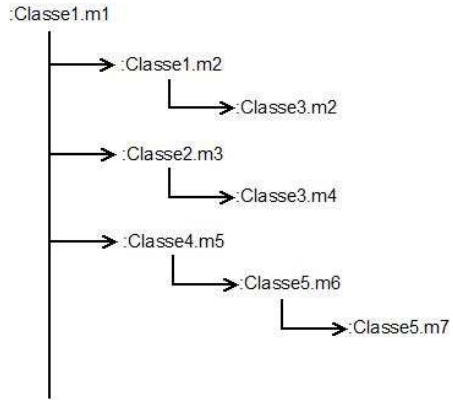
- Le nom du containeur ou package.
- Le nom de la classe.
- La signature de la méthode.

Les entrées sont regroupées pour former une unité temporelle d'un niveau supérieur, le segment. Chaque segment est composé d'un nombre identique d'entrées défini par l'utilisateur lors de la clusterisation, si ce n'est le dernier, dans la mesure où la trace n'est pas un multiple du nombre d'entrées défini pour un segment.

Cette trace correspond donc à la suite des appels aux procédures de l'application. « Un chemin plus pratique pour la représentation de traces d'appels à des méthodes est l'utilisation d'une structure en arbre » (De Pauw, Lorenz, Vlissides, Wegman, 1998). A la figure 8, nous avons une illustration de la trace dans une structure arborescente.

Figure 8

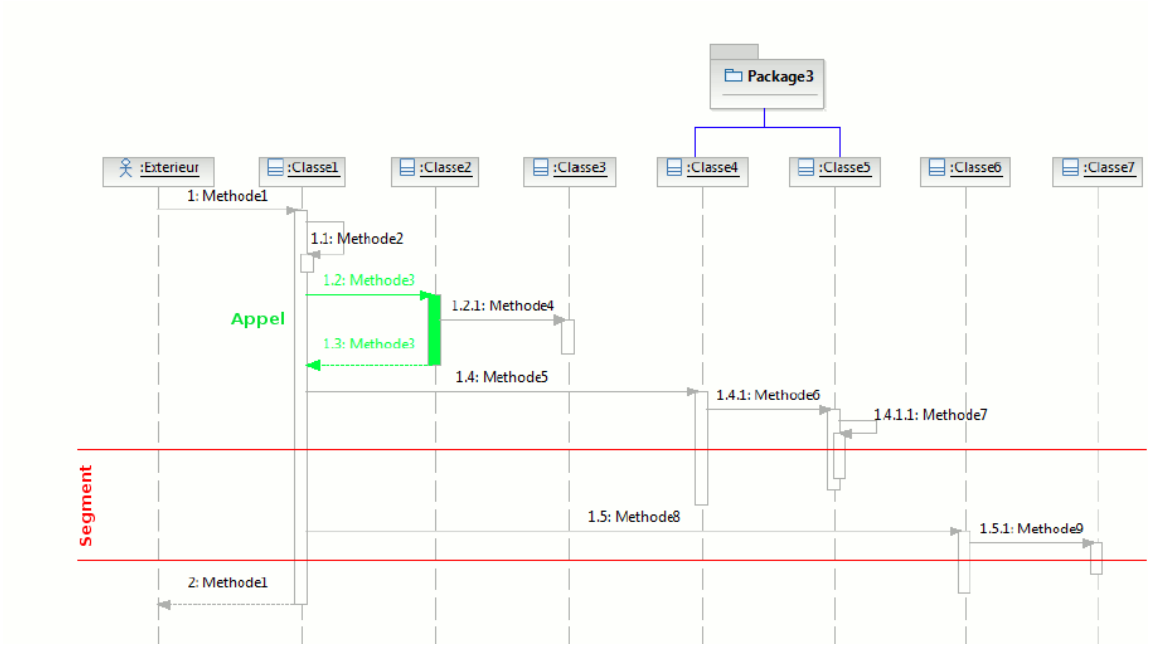
Structure arborescente d'une trace



Cette structure arborescente représente tout simplement l'échange des messages entre les objets. C'est précisément cet échange qui est synonyme de flux d'information et que nous souhaitons représenté dans un graphique. Le graphique UML en charge de montrer les interactions entre les objets est le diagramme de séquence. Dès lors, il est très facile d'effectuer la transition de la représentation arborescente de la trace en un diagramme de séquence, comme le montre la figure 9.

Figure 9

Diagramme de séquence d'une trace d'exécution



3. Procédés de simplification de la trace

La principale problématique lors de la manipulation d'une trace d'exécution, et de manière plus générale, dans le contexte d'une analyse dynamique, c'est l'énorme masse d'information à manipuler et à afficher. Une trace peut se composer de plusieurs dizaines voir de plusieurs centaines de milliers d'événements. Il est donc nécessaire d'avoir recours à des procédés de simplification. Ces techniques peuvent être classées en deux catégories (Hamou-Ladhj, 2004), celles « d'Exploration de la Trace », qui, fortement couplées avec les outils de représentations graphiques, permettent de naviguer au travers de la trace, et celles de « Compression de la Trace », qui tentent d'identifier quelles sont les informations inutiles à la compréhension du programme et susceptibles d'être retirer sans impacter la pertinence du résultat.

Malheureusement, concernant la deuxième catégorie, l'analyse statistique de clusterisation requiert de conserver la totalité des interactions entre les objets pour produire des regroupements fonctionnels. Si nous retirions des événements de la trace, le calcul des corrélations en serait faussé. Cependant, plutôt que d'appliquer ces procédés en phase de « preprocessing », nous pouvons très bien les mettre en œuvre en « postprocessing », et réutiliser les clusters en tant que support à la simplification de la trace.

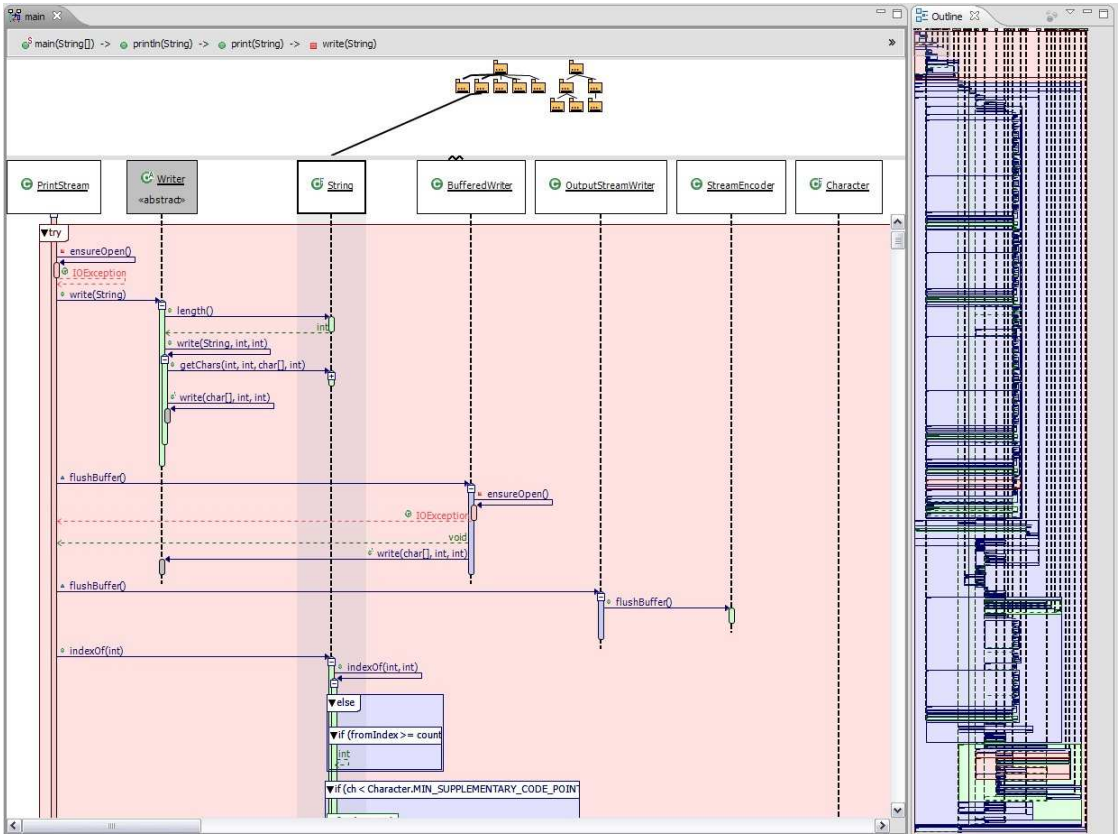
3.1 Sampling

Il s'agit de n'afficher à l'utilisateur qu'une fenêtre sur la trace et d'offrir les outils de navigation nécessaires. Il y a différentes approches comme l'idée d'utiliser une vue synthétisée de l'ensemble de la trace avec une loupe qui permet de parcourir cette dernière. A l'aide d'une seconde vue, nous représentons les informations détaillées de la zone mise en évidence par la loupe. (Benett, 2007)

Sur la figure 10, la vue synthétisée est affichée sur la droite de l'écran. Dans la zone principale, nous retrouvons la vue détaillée de la zone en évidence mise en forme à l'aide d'un diagramme de séquence. Dans ce cas-là, le problème de la taille persiste car il y a lieu de représenter l'ensemble de la trace dans la première vue.

Figure 10

Zest Sequence Viewer



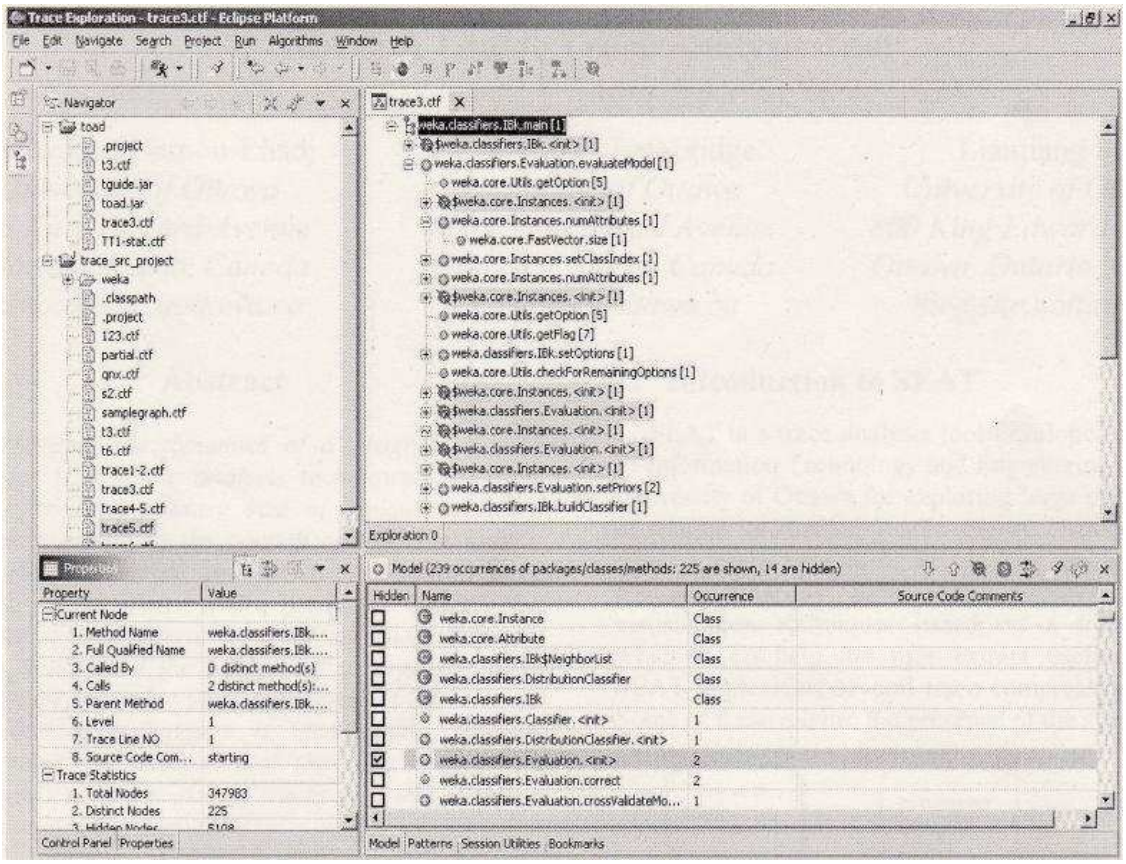
Source : Benett C., Myers D., Storey M.-A., German D. Working with “Monster” Traces: Building a Scalable, Usable Sequence Viewer. *Proc. of the 3rd International Workshop on Program Comprehension through Dynamic Analysis*. 2007.

Une alternative est d’abandonner la vue synthétisée et développer un composant représentant la trace dans une structure arborescente avec un objet capable d’effectuer un chargement tardif. A la différence d’un outil traditionnel qui charge l’ensemble des informations, ce composant est capable de ne charger que les informations représentées dans la zone en cours et de ne procéder au chargement de celles requises qu’au moment du déplacement de la zone active à l’aide des ascenseurs. (Hamou-Ladhj, 2004)

Sur la figure 11, nous pouvons voir l’implémentation de cette stratégie avec le composant présent dans le coin supérieur droit de l’écran. Nous avons dans le cadre inférieur gauche une fenêtre de propriétés du nœud sélectionné dans le composant d’exploration de la trace et dans la zone de droite, une fenêtre de filtrage des éléments à afficher et masquer dans la trace.

Figure 11

Software Exploration and Analysis Tool



Source: Hamou-Lhadj A., Lethbridge T. C., Fu L. SEAT: A Usable Trace Analysis Tool. *Proc. of the 13th International Workshop on Program Comprehension (IWPC'05)*. 2005.

Nous pouvons tout de même nous poser la question de la réelle utilisabilité de techniques d'exploration de la trace dans un contexte où il s'agit à l'utilisateur de se situer au sein d'énormes quantités d'événements, ce qui demande non seulement d'excellentes facultés d'abstraction, mais également une connaissance de l'architecture de base pour savoir sur quelle zone focaliser son attention. Comment savoir quelles interactions méritent notre attention et à quel moment elles surviennent si nous n'avons aucune connaissance du système existant. Cependant, dans la deuxième technique, un élément intéressant à relever est la représentation en arbre qui se fait à l'aide d'un « Compact Trace Format » (CTF), un format de la trace en arborescence que ses concepteurs tentent de normaliser. Nous reviendrons plus tard sur CTF.

3.2 Collection des données

La collection des données pour construire une trace est la phase de l'instrumentation du code source et de la construction de la trace. Pour diminuer la quantité de données qui sera extraites du code, il est possible d'influencer l'instrumenteur pour ne sélectionner que certaines données. Nous pourrions effectuer une sélection en fonction des éléments présents dans la signature d'une méthode comme ses paramètres ou son type de retour, ou alors, élever le niveau d'abstraction de la vue résultante en changeant les informations extraites comme remplacer le nom de la classe par le nom du package. Seulement, en plus d'être synonyme de perte d'information car nous ne récoltons plus la totalité des méthodes exécutées, cela nécessite d'avoir une connaissance de l'architecture initiale pour savoir quelles sont les méthodes pertinentes et que nous souhaitons voir apparaître dans la trace. La phase de collection des données est une étape qui se déroule obligatoirement avant l'analyse statistique par segmentation de la trace. Nous ne pouvons donc pas la mettre en œuvre dans notre situation auquel cas notre clusterisation serait erronée.

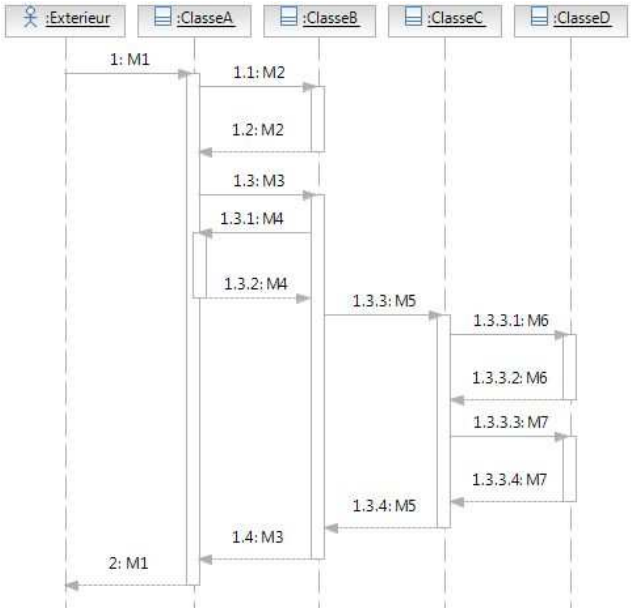
3.3 Filtrage au niveau architectural

Une autre approche est de montrer l'interaction entre les composants architecturels plutôt qu'entre chaque objet. Certains outils proposent la possibilité à l'utilisateur de regrouper les objets de la trace en composant pour effectuer l'analyse. Le problème est le même que pour la collection de données : cela demande à l'utilisateur d'avoir des connaissances suffisantes de l'architecture initiale pour effectuer ces regroupements. C'est précisément dans cette situation que la clusterisation s'avère être un réel atout. La clusterisation nous apporte un regroupement fonctionnel des classes qui s'appuie sur l'exécution réelle du programme. Nous possédons ainsi un facteur de tri pertinent pour procéder à une analyse de la trace. Si nous transposons cela sur notre diagramme de séquence, au lieu d'utiliser les classes comme lignes de vie, nous pouvons les remplacer par nos clusters et ainsi, ne représenter que la communication entre ces composants.

La figure 12 représente le diagramme de séquence d'une trace au niveau le plus détaillé, celui des classes.

Figure 12

Diagramme de séquence niveau classe

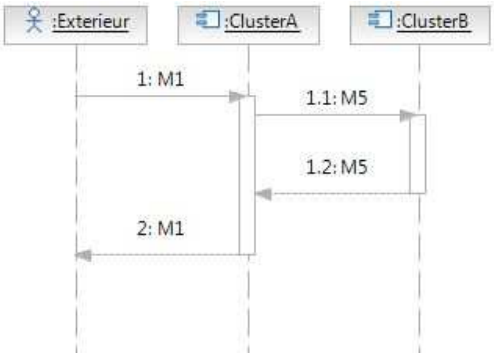


Lors de la phase de clusterisation, les classes A et B apparaitront dans les mêmes segments car elles collaborent fortement ensemble et idem pour les classes C et D. Nous aurons ainsi un cluster A, composé des classes A et B, ainsi qu'un cluster B composé lui des classes C et D. Comme nous sommes à un niveau d'abstraction plus élevé, ce n'est plus la communication entre les classes qui nous intéresse, mais celle entre les composants détectés.

Ainsi, comme le démontre la figure 13, nous pouvons effacer de notre représentation tous les événements internes à un composant et n'afficher que le fonctionnement des clusters, les uns avec les autres.

Figure 13

Diagramme de séquence niveau cluster



3.4 Masquage de composants

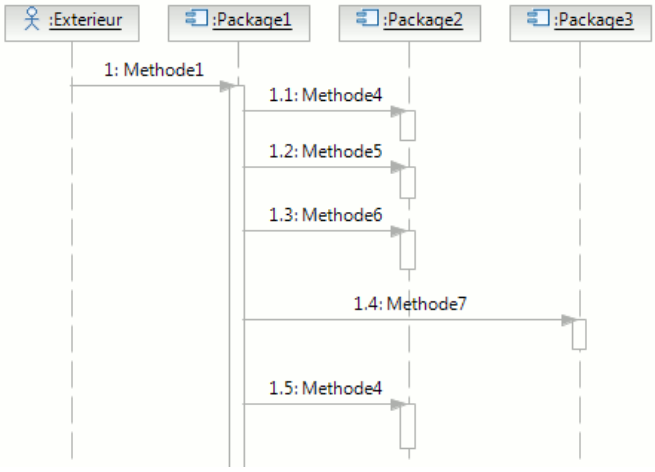
Le masquage des composants peut s'effectuer à deux niveaux : nous pouvons décider de ne pas tenir compte de certains éléments de la trace lors de l'analyse, comme des appels à une méthode spécifique par exemple, ou alors, par le biais de l'interface graphique, nous pouvons masquer certaines parties du résultat, du diagramme, et attendre une action de l'utilisateur pour les afficher. La première fonctionnalité est déjà implémentée par notre analyseur de classe qui permet de sélectionner les classes à tester lors de la clusterisation. Dans la mesure où nous ne sélectionnons pas certaines classes, elles sont masquées, et donc, n'apparaitront pas dans les clusters.

Pour ce qui est de l'interface graphique, j'ai étudié les cas dans lesquels une optimisation du diagramme de séquence pourrait être apportée sans modifier la pertinence de ce dernier. Sachant que l'on souhaite mettre en évidence les flux d'informations, il arrive qu'il y ait une multitude d'échanges de messages successifs entre deux composants.

Sur la figure 14, on constate que durant une certaine période, les composants A et B collaborent uniquement ensemble. Pour simplifier la représentation et souligner qu'un flux d'information important transite entre ces deux éléments, il pourrait être envisagé de regrouper ces messages sous une seule entité.

Figure 14

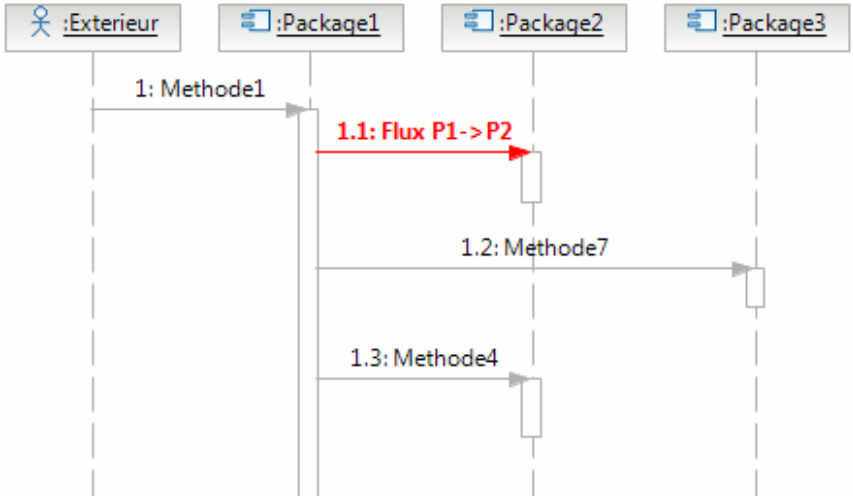
Représentation des appels successifs



Nous aurions ainsi, comme démontré à la figure 15, un message représentant ce flux qui pourrait être différencié par des codes couleurs ou une épaisseur de flèche différente selon le volume d'information échangé et pourrait être détaillé à la suite d'une action utilisateur.

Figure 15

Regroupement des appels successifs



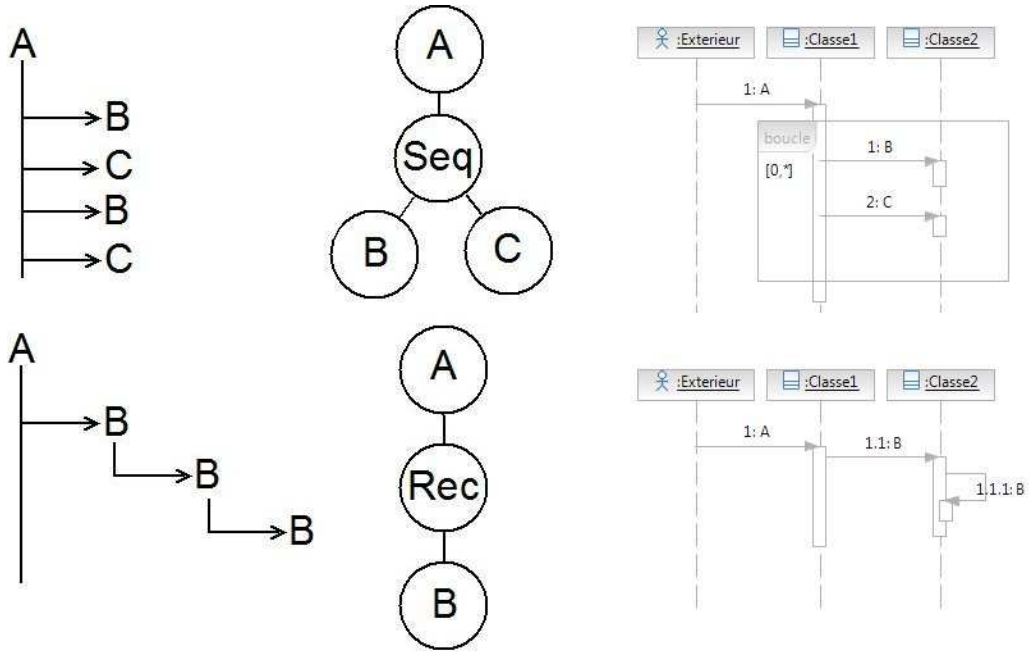
3.5 La détection de patterns

La détection de pattern est probablement une des techniques les plus mises en œuvre et une des plus efficientes pour faire face au problème de la taille de la trace. Le principe est de détecter les séquences d'événements similaires au sein de la trace et de ne les représenter qu'une seule fois. Il existe différents critères et algorithmes pour détecter ces patterns. Certains s'appliquent au format séquentiel de la trace avec, par exemple des algorithmes de recherche de sous-chaîne dans une chaîne mais une des approches les plus prometteuses est amenée par le format de trace compacte CTF cité précédemment. (Hamou-Ladhj, 2004)

L'idée d'élaboration de ce format part d'un constat : il persiste un gros manque de cadre de travail commun pour la manipulation de trace. Il faut donc débuter en normalisant la représentation d'une trace. Le format CTF tente d'apporter cette standardisation à l'aide d'une structure en arborescence. Afin de diminuer la taille, en plus des nœuds représentant des appels, le format comporte des nœuds dits de « contrôle » pour représenter la récursivité et la répétition.

Figure 16

Transition CTF-UML2



Logiquement, comme c'est un format de stockage de la trace, les méthodes de compression sont mises en œuvre immédiatement lors de la récolte d'information. Ceci dit, nous pouvons appliquer un algorithme de détection des répétitions et de la récursivité qui s'applique à cette structure arborescente. Une fois la compression effectuée, comme nous pouvons le voir sur la figure 16, UML 2 nous apporte la notation nécessaire pour représenter ces deux cas de figure : le message récursif à une ligne de vie pour la récursivité et la boucle de fragment regroupé pour les répétitions. Passons maintenant à l'implémentation, chapitre dans lequel je présenterai l'algorithme de compression élaboré.

4. Implémentation

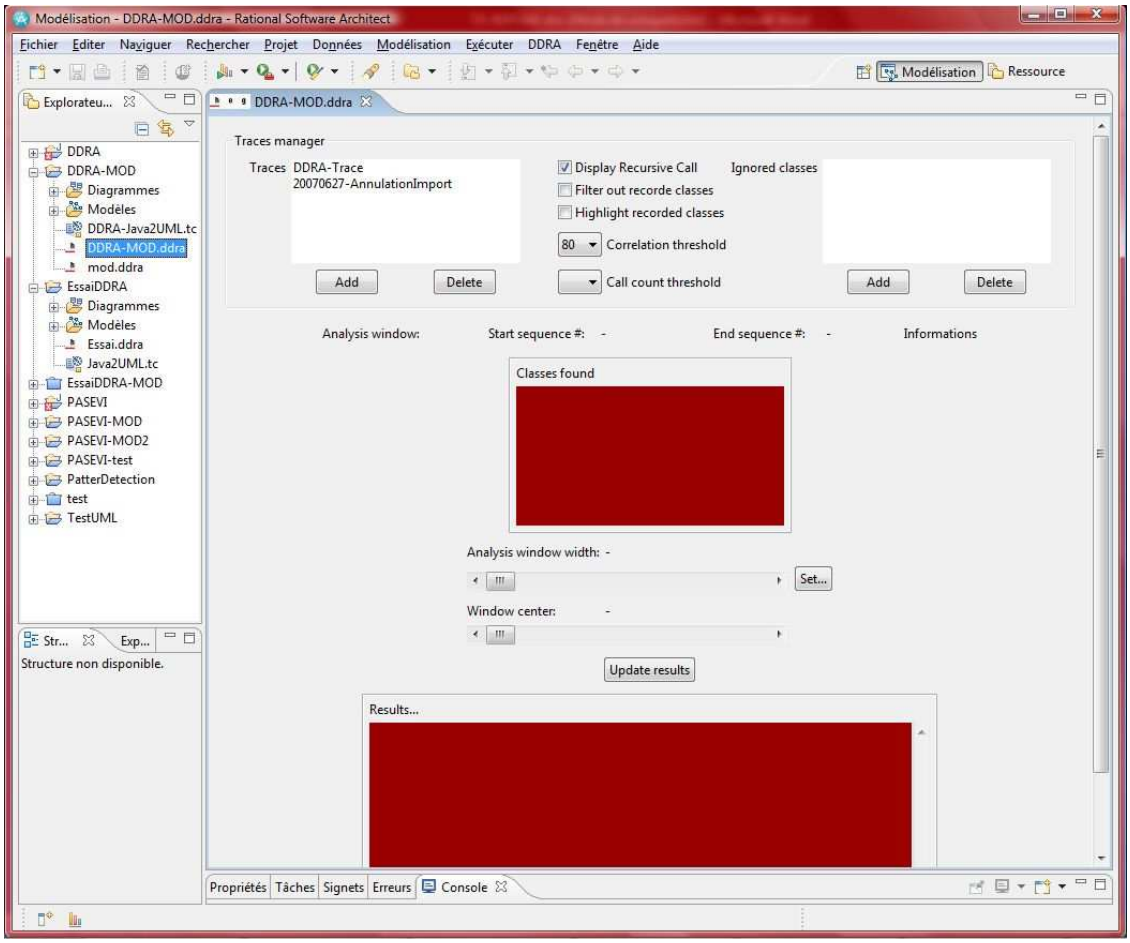
Le choix technologique porté sur « Rational Modeling Platform » nous a contraint à migrer l'ancien analyseur dans l'environnement de développement intégré Eclipse, sur lequel est basé la plateforme Rational. Nous verrons donc la transition des différents composants dans ce nouvel environnement. Ensuite, je présenterai les choix d'implémentation pour la comparaison des clusters et la représentation des flux d'information. Nous finirons par présenter les procédés de simplification qui ont été retenus pour manipuler la masse d'information et comment ils ont été mis en œuvre.

4.1 Migration de l'analyseur

L'analyseur a été converti en un plugin Eclipse. C'est donc un jar qui peut-être déployé dans un environnement Eclipse pour en étendre les fonctionnalités. Les couches métiers n'ont quasiment pas été modifiées, c'est uniquement la couche graphique qui a été remplacé pour être convertie en SWT. Le plugin est, au sens Eclipse, un éditeur de fichier. Cela signifie que l'écran principal de l'analyseur est une interface graphique associée à son propre type de fichier, dont l'extension a été définie à « *.ddra », et qui est automatiquement ouverte lorsque l'utilisateur tente de manipuler ce type. Le fichier est un document XML dans lequel est stocké le paramétrage des filtres comme le ratio de corrélation ou l'affichage de la récursivité, qui se faisait depuis le menu « Filter ». C'est un plus, car dans l'ancienne version, ces paramètres n'étaient pas sauvegardés et il fallait les reconfigurer à chaque lancement de l'application. Pour initialiser le fichier, une analyse débute à l'aide d'un assistant qui permet de définir le premier paramétrage, procède à la création du fichier et ouvre ce dernier avec l'analyseur.

Figure 17

Fenêtre éditeur de l'analyseur

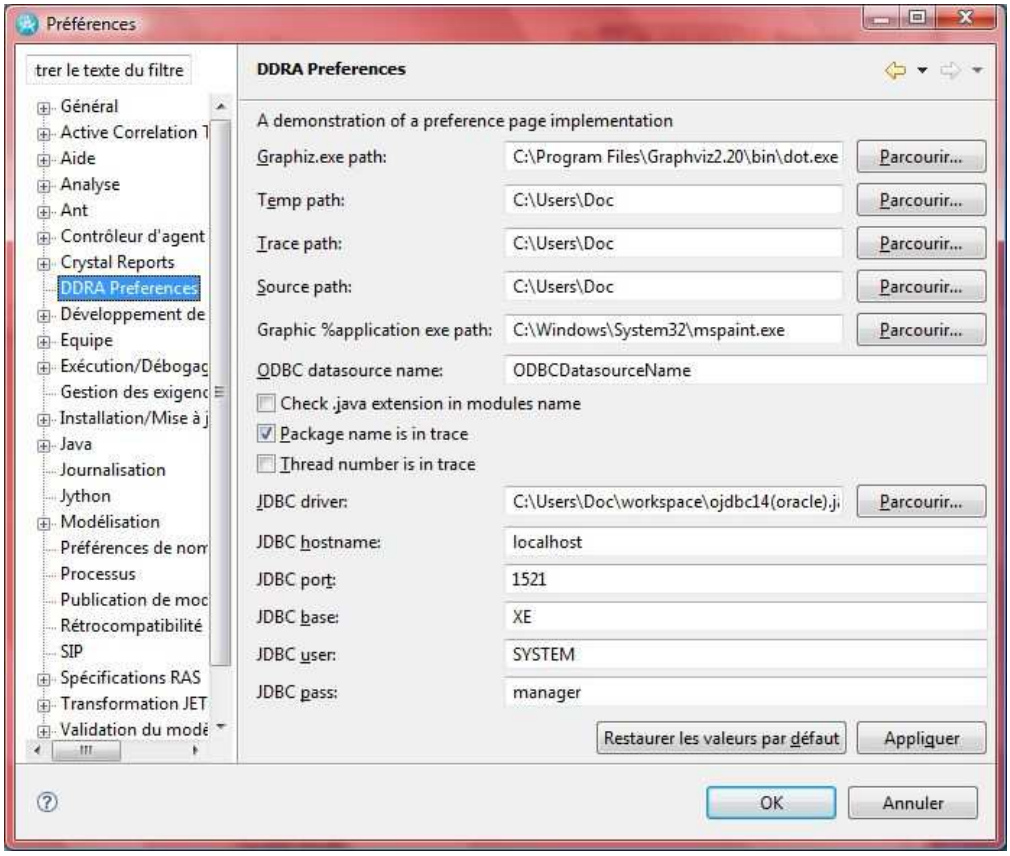


A la figure 17, nous pouvons voir que l'écran de gestion de traces ainsi que l'écran des classes à ignorer lors d'une analyse ont tous les deux été directement intégrés dans l'éditeur. Pour le reste, l'ergonomie a été conservée. Afin de respecter les standards Eclipse, le menu de l'éditeur ne fait plus partie intégrante de la fenêtre d'analyse mais vient s'ajouter au menu Eclipse, au haut de la fenêtre.

Finalement, le paramétrage de l'outil, devenu le paramétrage du plugin, vient s'intégrer en tant qu'onglet dans les préférences d'Eclipse, comme on peut le voir à la figure 18.

Figure 18

Onglet de paramétrage du plugin



En plus du paramétrage initial présent dans l'ancien analyseur comme le chemin de l'exécutable « Graphiz », utilisé pour générer le graphe des clusters, les différents répertoires de travail ou encore le paramétrage du format de la trace, une rubrique est venue s'ajouter qui permet de définir la configuration de la base de données. Dans l'ancien analyseur, ces éléments étaient codés en dur dans la source, alors que dans la nouvelle version, l'instanciation du driver se fait de manière dynamique, cela permet ainsi de facilement changer la source données, immédiatement depuis cet écran, sans modifier l'exécutable.

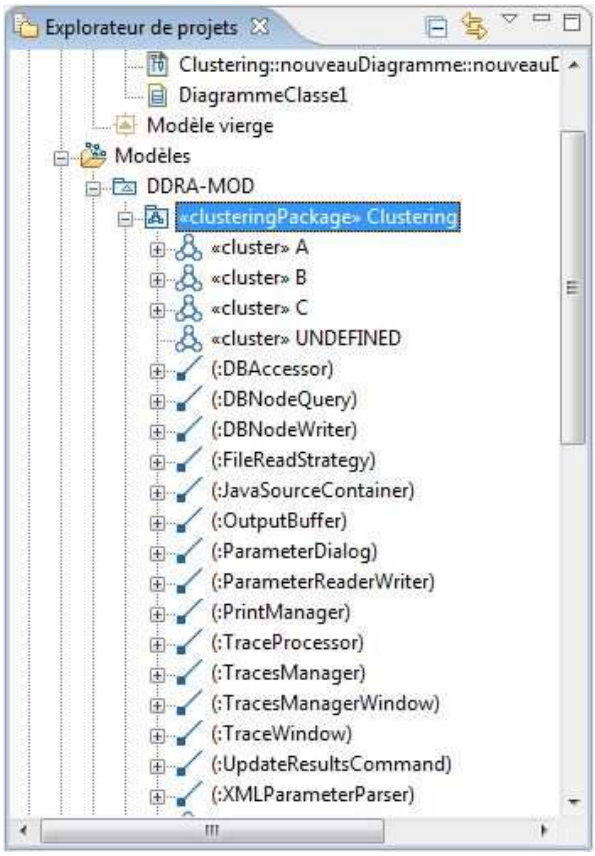
Si l'on isole cette partie du plugin, elle est totalement indépendante de la surcouche Rational est pourrait faire l'objet d'un projet indépendant en vue d'un déploiement sur une plateforme Eclipse dépourvue de RSA.

4.2 Comparaison des clusters

Pour procéder à une comparaison des clusters, il faut tout d'abord changer le format du résultat de la clusterisation, actuellement c'est une simple matrice, pour pouvoir l'intégrer au métamodèle. Pour procéder à cette transformation, cela requiert que le profil créé pour étendre la notation UML avec notre terminologie soit appliqué au modèle. Ensuite, afin de respecter les normes de l'analyseur, j'ai créé une nouvelle stratégie d'analyse qui étend celle de la clusterisation mais qui me permet de conserver le résultat de cette dernière. Je peux ensuite, en utilisant ce résultat comme source, exécuter une commande Rational de génération des clusters au sein de notre modèle UML.

Figure 19

Résultat de la clusterisation

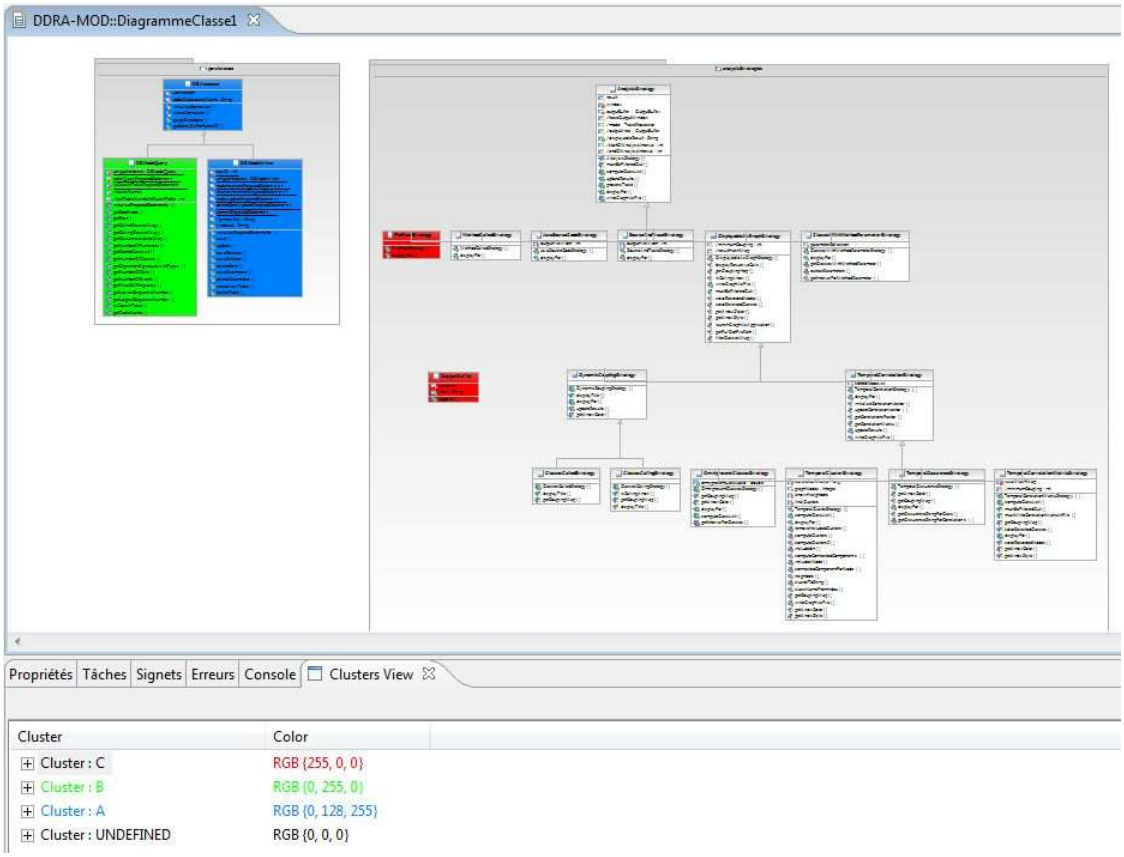


La figure 19 illustre le résultat de notre clusterisation dans la fenêtre d'explorateur de projets. Nous avons notre package « Clustering » servant de containeur à notre analyse et identifié à l'aide du stéréotype « clusteringPackage », qui contient l'ensemble des clusters créés avec leurs associations sur les classes Java, précédemment générées à l'aide d'une transformation Java vers UML.

Il est maintenant libre à l'utilisateur de créer un diagramme de classe et de sélectionner les classes qu'il souhaite y voir apparaître.

Figure 20

Représentation des clusters



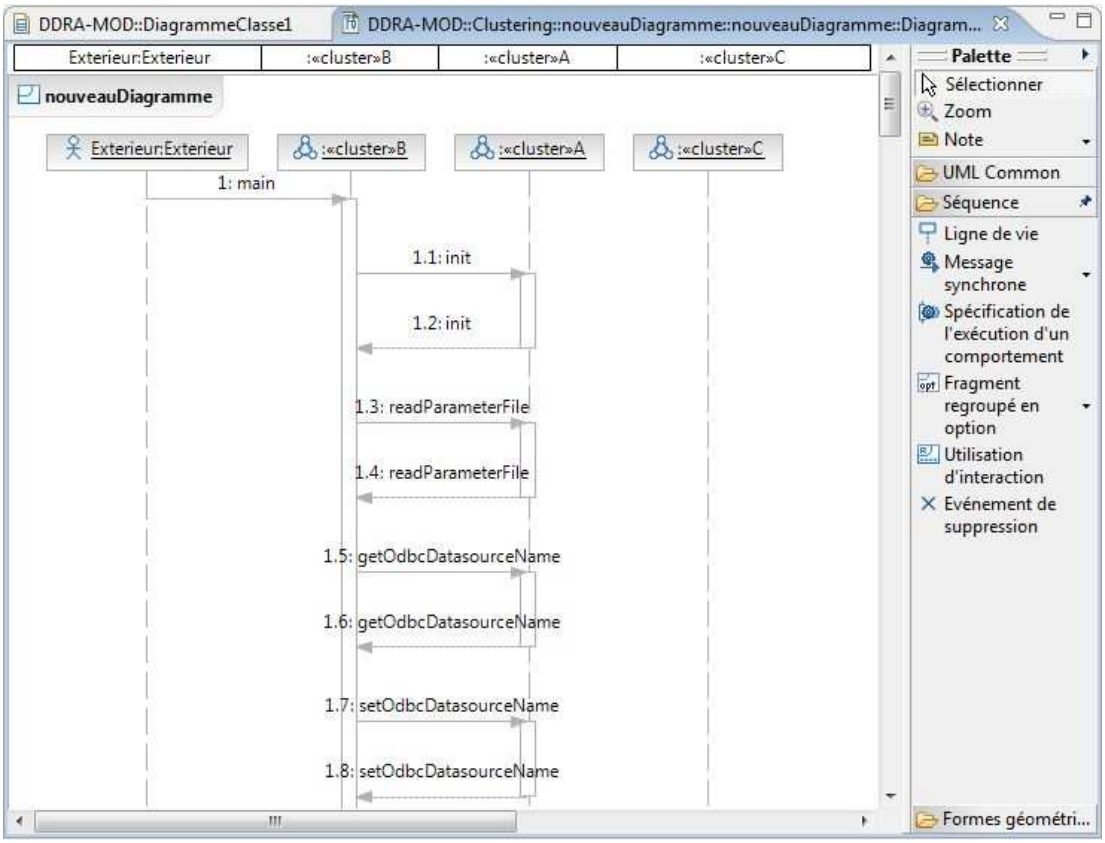
Lorsque l'utilisateur a son diagramme de classe ouvert, il peut sélectionner le package de clusterisation, stéréotypé par « clusteringPackage », pour l'ouvrir dans la vue des clusters, représentée au bas de la figure 20. Cette vue lui présente les clusters dans un format arborescent avec une colonne qui lui permet de changer les couleurs des clusters, ainsi, ils seront mis en évidence au sein du diagramme. Il peut également créer un diagramme à l'aide des clusters et n'afficher que les éléments liés à ces derniers.

4.3 Représentation des flux d'information

La figure 21 nous montre la représentation des flux d'information dans un diagramme de séquence UML2. Ce type est déjà implémenté dans la plateforme Rational et il suffit de piloter les API RMP pour le générer.

Figure 21

Représentation des flux d'information



L'utilisateur, lorsqu'il génère les clusters peut directement générer le diagramme de séquence correspondant à ces derniers. Dans la boîte de dialogue, il lui sera demandé quelles techniques de simplification il souhaite mettre en œuvre. C'est sur ce point qu'un gros effort d'implémentation a été investi.

4.4 Techniques de compression de la trace

Le diagramme de séquence est généré au niveau des clusters. Malgré qu'il y ait une diminution de la quantité d'information à afficher, il reste tout de même énormément de message à modéliser. J'ai donc dû mettre en œuvre certains des procédés de simplification, en plus du filtrage architectural, pour parvenir à un résultat efficient.

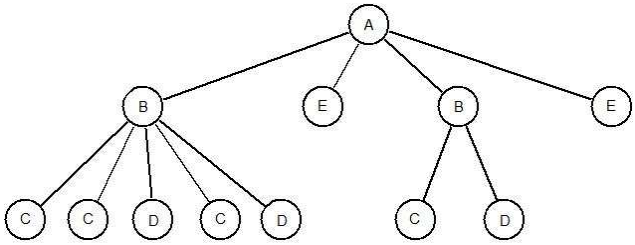
En générant les premiers diagrammes de séquence, j'ai constaté qu'il y avait énormément d'accesseurs qui étaient présents. Ces opérations n'apportent rien à la compréhension du comportement du programme, la première technique consiste à ôter les événements liés à ces derniers. Les facteurs sur lesquels nous nous appuyons pour identifier les accesseurs sont :

1. Le nom de la méthode doit débuter par les termes « get » ou « set ».
2. La méthode doit être une feuille de l'arbre d'appel. En effet, une méthode peut très bien avoir un nom qui débute par les mêmes termes sans pour autant être un accesseur. Pour distinguer les deux cas de figure, nous définissons que l'accesseur est une méthode qui n'effectue aucun appel dans son corps.

La deuxième technique implémentée est une technique de détection de patterns, et plus précisément, les patterns de boucle. Le but est de détecter au sein de la trace les boucles dans lesquelles est passée l'exécution du programme et d'en donner une représentation simplifiée. La manipulation de la trace se fait à l'aide d'une structure de données en arbre qui s'inspire du format de trace compact.

Figure 22

Arbre d'appel



L'algorithme analyse l'arbre en débutant par les feuilles, pour détecter des répétitions au plus bas niveau, jusqu'à atteindre le nœud racine. Il calcule tout d'abord la profondeur de l'arbre pour savoir à quel niveau doit débuter l'analyse. A la figure 22, la profondeur est de 3. Ainsi, nous allons nous focaliser sur les parents des feuilles de profondeur 3, pour détecter les répétitions de leurs enfants par fenêtrage, ou slicing.

Tableau 2

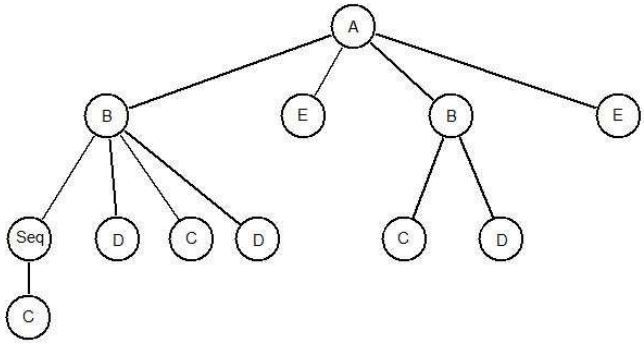
Fenêtrage de taille 1

C	C	D	C	D	
	C	C	D	C	D

Dans le tableau 2, nous comparons chaque sous-arbre avec son successeur, et s'ils sont identiques, nous supprimons la redondance et insérons une séquence.

Figure 23

Arbre d'appel avec un nœud séquence

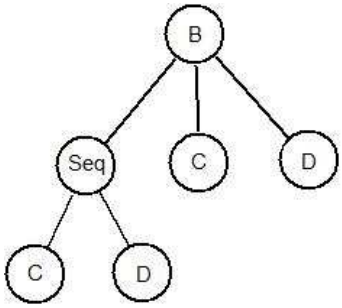


Nous pouvons voir à la figure 23 la représentation de la répétition à l'appel C. L'analyse se poursuit en incrémentant la taille de la fenêtre pour effectuer une comparaison avec un décalage de deux, en d'autres termes, trouver des répétitions de deux appels comme par exemple C-D, et ce, tant que la taille de la fenêtre d'analyse est inférieure ou égale à la moitié du nombre d'enfants d'un parent testé. Cette sentinelle est élaborée sur le fait que nous ne pouvons pas trouver de répétition dans un ensemble d'enfant si nous ne pouvons comparer que deux segments et qu'ils sont de taille différente. Imaginons que nous ayons une liste d'enfant tel que « A ;B ;C ;A ;B » et que nous cherchions avec une taille de fenêtre de 3, auquel cas nous devrions comparer un segment « A ;B ;C » avec un segment « A ;B » et comme la taille des deux segments est différentes, il est impossible qu'il y ait répétition.

Il y a d'autres heuristiques à souligner. Un nœud séquence est un nœud de contrôle. Par conséquent, lorsque nous comparons un sous-arbre contenant une séquence, ce sont les fils de cette dernière qui font l'objet de la comparaison. Lorsque nous recherchons des répétitions de taille 2 dans les enfants du parent B de la figure 24, nous allons comparer les deux enfants de la séquence, donc équivalent à une taille 2, avec les deux enfants suivants du parent B, appel C et appel D. Dans ce cas-là, il y a donc répétition et nous pouvons supprimer les appels C et D déjà représentés dans la séquence.

Figure 24

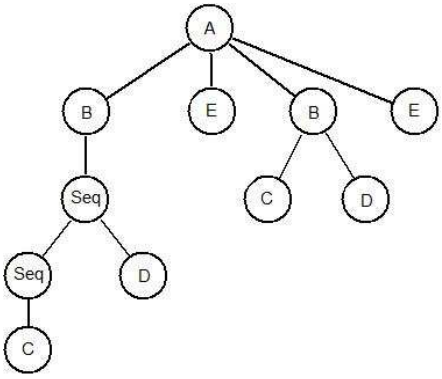
Comparaison de séquence



De plus, lors de la comparaison d'une séquence avec un appel simple, ce dernier doit être pris en compte comme une éventuelle séquence dans laquelle il n'y aurait qu'un seul appel. Pour illustrer ce cas, reprenons le résultat de la figure 23. Lorsque nous comparons les nœuds fils du premier appel B avec une fenêtre de taille 2, nous allons comparer un segment composé de la séquence de C et un appel D avec un segment composé d'un appel C et un appel D. Il faut alors interpréter que dans le deuxième segment, l'appel C est éventuellement une répétition mais dans laquelle l'exécution du programme n'est passée qu'une fois. Cela nous permet ainsi de détecter répétition entre ces deux segments.

Figure 25

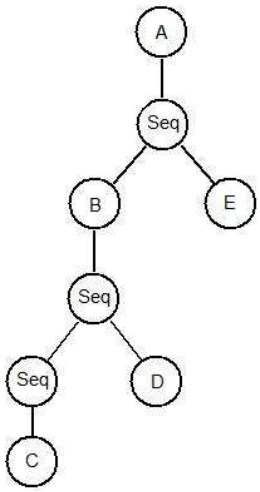
Arbre d'appel après analyse du premier niveau



Après une analyse du niveau le plus bas, dont le résultat est illustré à la figure 25, l'algorithme poursuit en changeant de pallier, ainsi de suite, jusqu'à atteindre le nœud racine.

Figure 26

Arbre d'appel après simplification des répétitions



Lors de la transposition de cet arbre en diagramme de séquence, nous allons utiliser la terminologie UML 2 et représenter les séquences par des « boucles de fragment regroupé » et les appels par des messages d'interaction. Ainsi, si l'on compare la représentation de l'arbre initial et celui simplifié de la figure 26, la représentation des échanges de message est passé de 12 nœuds d'appel à 5 nœuds d'appel à représenter au sein du diagramme.

Il est difficile d'établir un ratio moyen de compression car ce dernier est fortement dépendant de l'implémentation qui a produit la trace. Cependant, nous pouvons distinguer que le meilleur cas est N, où N est le nombre de nœud contenu dans l'arbre, sachant qu'une trace ne représentant qu'une répétition d'appel pourra être simplifié par un fragment de boucle constitué d'un seul appel, et que le pire cas est 1, dans le cas où il n'existe aucune répétition dans la trace.

5. Etude de cas

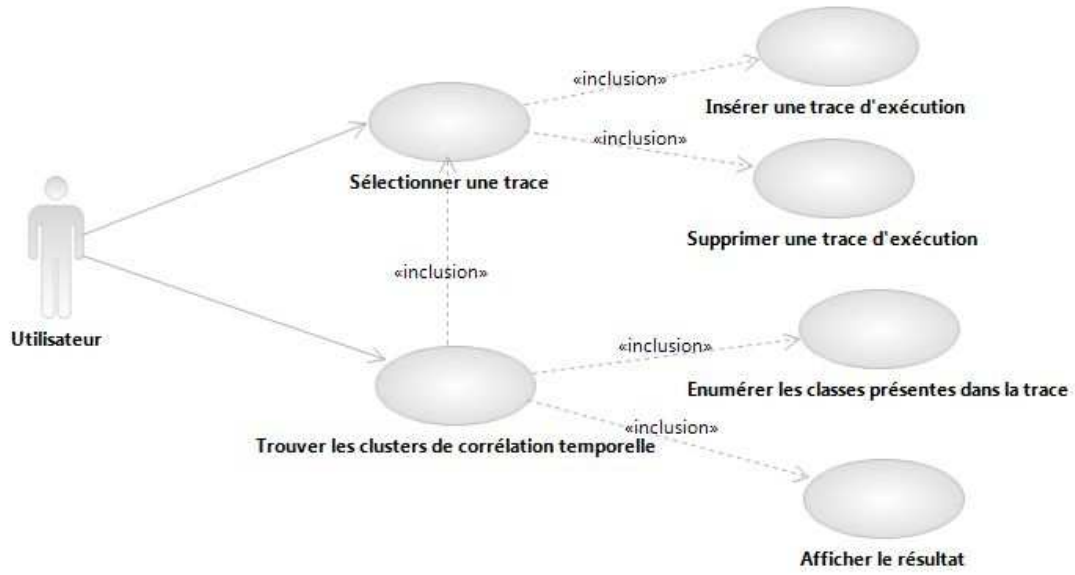
Pour cette étude de cas, nous allons observer l'implantation originale de l'analyseur de trace développé dans le labo de Philippe Dugerdil. Nous nous focaliserons sur une fonctionnalité, celle des « clusters de corrélation temporelle », c'est-à-dire, l'analyse statistique par segmentation de la trace.

Pour ce faire, nous établirons la modélisation métier de ce cas, sans représenter l'ensemble du logiciel, mais uniquement les diagrammes liés à cette action. Ensuite, nous procéderons à la clusterisation et nous utiliserons les représentations RSA pour afficher les clusters dans un diagramme de classe et pour les comparer avec l'implémentation initiale. Finalement, nous générerons le diagramme de séquence de la trace avec et sans les simplifications pour voir le ratio de compression dans cette situation.

5.1 Modélisation métier

Figure 27

Diagramme de cas d'utilisation système



La figure 27 nous donne le diagramme de cas d'utilisation système. Nous avons représenté le use case principal qui nous intéresse, « Trouver les clusters de corrélation temporelle » ainsi que tous les use cases de sous-fonction liés. Décrivons maintenant les flots intéressants.

5.1.1 UC : Trouver les clusters de corrélation temporelle

Acteur principal : Utilisateur

Déclencheur : l'utilisateur sélectionne la fonctionnalité "clusters de corrélation temporelle".

Précondition : l'utilisateur a sélectionné une trace.

Flot principal

1. L'utilisateur clique sur "clusters de corrélation temporelle".
2. Le système énumère les classes présentes dans la trace.
3. L'utilisateur configure le nombre de segments souhaité.
4. L'utilisateur sélectionne les classes qui feront l'objet de l'analyse.
5. Le système parcourt les lignes de la trace pour construire la matrice de corrélation.
6. Le système construit les clusters.
7. Le système affiche le résultat sous forme de clusters constitués de classes.
8. Le flot se termine.

Flots alternatifs

8.a. L'utilisateur sélectionne d'autres classes.

8.a1. Le flot reprend en point 5.

8.b. L'utilisateur modifie la configuration du nombre de segment souhaité.

8.b1. Le système demande à l'utilisateur de cliquer sur "mise à jour".

8.b2. L'utilisateur clique sur "mise à jour".

8.b3. Le flot reprend en point 5.

5.1.2 UC : Sélectionner une trace

Niveau : sous-fonction

Acteur principal : Utilisateur

Déclencheur : L'utilisateur clique sur "sélection d'une trace"

Flot principal

1. L'utilisateur clique sur "sélection d'une trace" sur l'écran principal des traces.
2. Le système se connecte à la base de données.
3. Le système récupère la liste des traces présentes dans la base.
4. Le système ouvre l'écran de gestion des traces.
5. Le système affiche la liste des traces.
6. L'utilisateur sélectionne une trace.

7. L'utilisateur valide sa sélection.

8. L'écran de gestion des traces retourne la sélection.

9. Le système initialise la configuration pour cette trace.

10. Le système ferme l'écran de gestion des traces.

11. Le flot se termine.

Flots alternatif

6.a. L'utilisateur clique sur "ajouter une trace".

6.a1. Le système insère une trace d'exécution.

6.a2. Le flot reprend au point 5.

7.a. L'utilisateur clique sur "supprimer cette trace".

7.a1. Le système supprime la trace d'exécution.

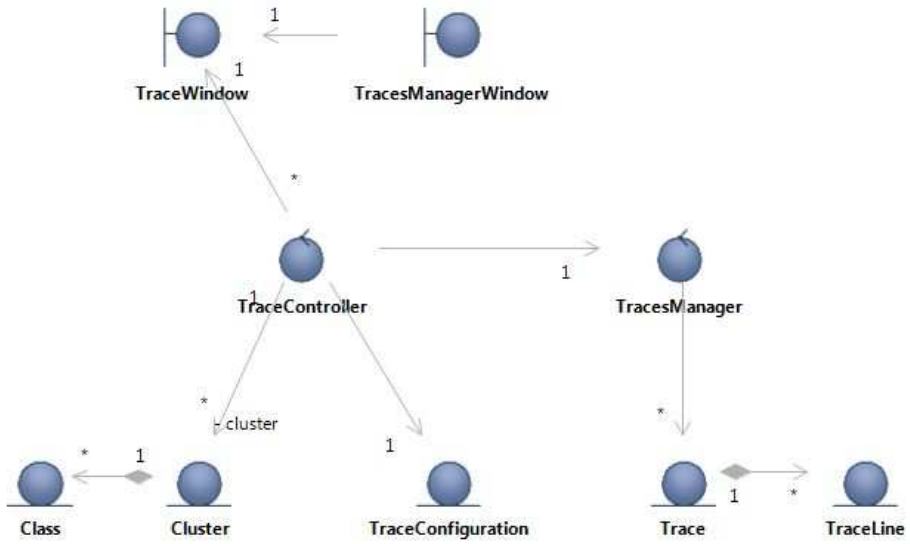
7.a2. Le flot reprend au point 5.

5.1.3 Diagramme de robustesse

Après avoir analysé les flots des deux cas d'utilisation précédents, nous avons pu identifier certains éléments manipulés qui sont représentés par le diagramme de robustesse en figure 28.

Figure 28

Diagramme de robustesse



Sur le haut de l'image, nous retrouvons nos deux interfaces utilisateurs (limites), l'écran principal et celui de gestion des traces. Sur le bas du diagramme, ce sont

toutes les entités manipulées dans les flots : les clusters composés de classes, la configuration d'une trace ainsi que la trace, elle-même constituée de lignes. Finalement, au centre, nous avons les deux contrôleurs : celui pour les opérations d'analyse sur la trace ainsi que celui pour la gestion de la trace.

5.2 Clusterisation

Nous avons instrumenté le code source de cette application et générer une trace en réalisant le cas d'utilisation cité précédemment. Cette trace se compose d'environ 730'000 lignes, et comme pour chaque opération, nous avons une ligne pour le début de l'appel et une ligne pour la fin de l'appel, cette trace représente donc 365'000 appels.

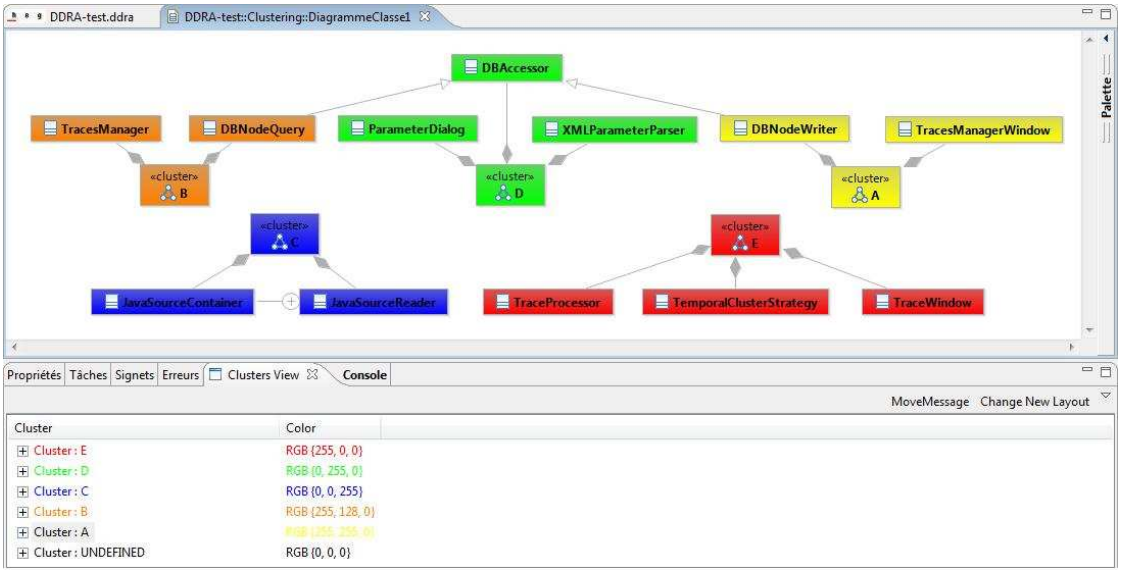
Pour débuter l'utilisation de notre analyseur, nous créons un projet UML dans la plateforme Rational et nous procédons à une transformation Java vers UML du code source. Nous avons ainsi une représentation des packages et des classes au format UML. Nous pouvons ensuite créer notre fichier d'analyse à l'aide de l'assistant et procéder à une clusterisation de la trace d'exécution. Comme paramétrage, nous découpons notre trace en 5'132 segments et chaque segment sera composé d'une septantaine d'appels. Nous configurons le ratio de corrélation à 80%, ce qui signifie que deux classes doivent avoir 80% des segments où elles apparaissent communément.

L'opération de clusterisation se déroule de la même manière que dans l'ancien analyseur en affichant le résultat dans la zone d'affichage. Mais dès lors, nous avons maintenant la possibilité d'intégrer ce résultat dans le modèle UML. Une fois l'opération exécutée, nous avons nos clusters disponibles et nous pouvons les représenter à l'aide d'un diagramme de classe, comme démontré à la figure 29.

Dans le diagramme, nous avons inclus uniquement les clusters et les classes qui leurs étaient liées. Nous avons pu les colorer grâce à la vue des clusters qui se situe au bas de l'écran. Bien que ce diagramme représente toutes les classes fortement impliquées dans l'exécution du cas d'utilisation, il est beaucoup plus intéressant de les mettre en évidence au sein de l'architecture initiale.

Figure 29

Diagramme de classe représentant les clusters

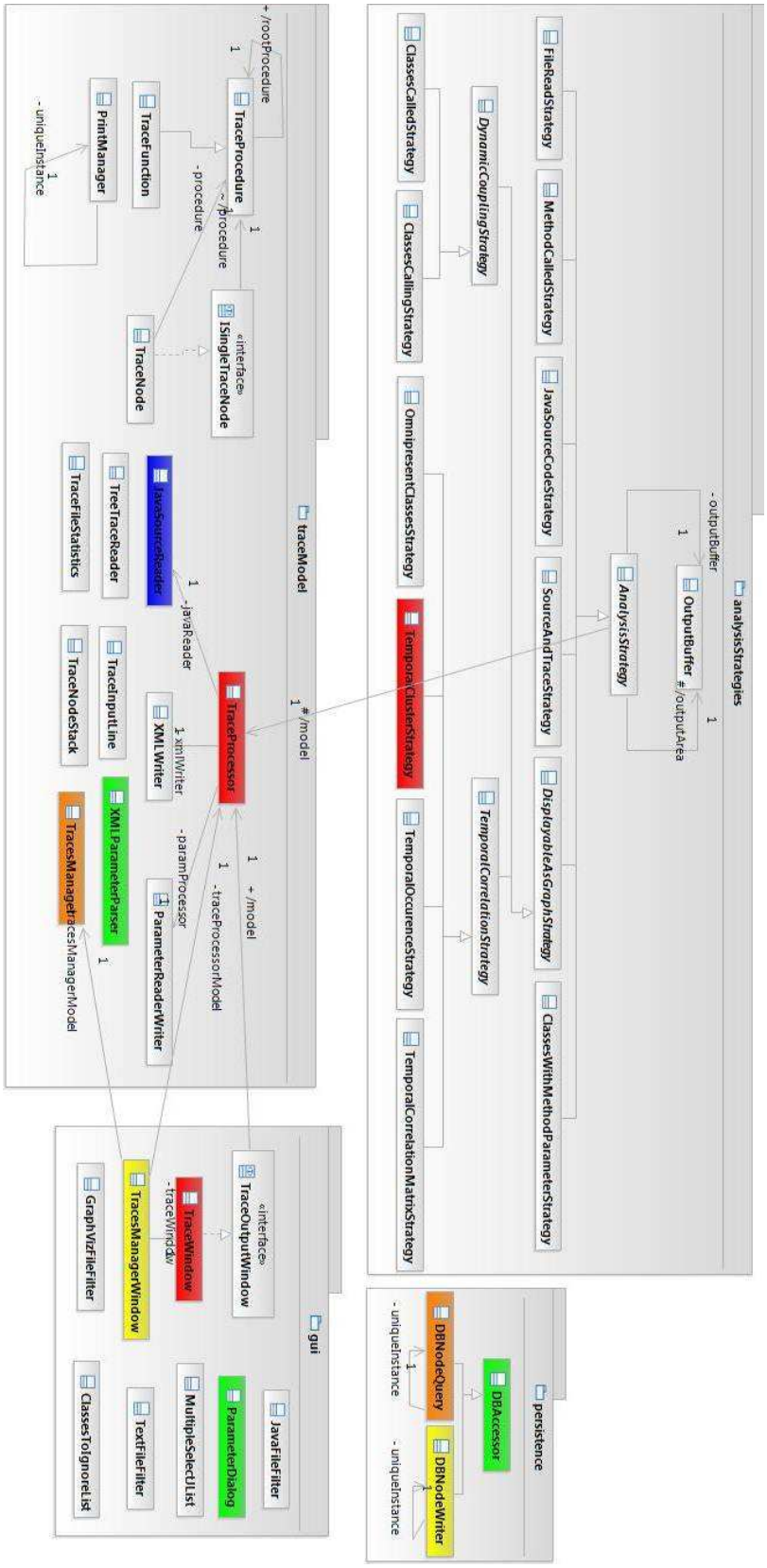


La figure 30 est le diagramme de classe représentant cette implémentation initiale sur lequel nous avons appliqué la coloration de nos clusters. Grâce au nommage des classes et à ma connaissance de l'architecture, je vais distinguer la fonction de chaque cluster et évaluer la pertinence du résultat. Le cluster vert implique les classes liées au paramétrage. Nous pouvons voir la classe « ParameterDialog » qui est une boîte de dialogue permettant de setter les valeurs, et cet écran travaille conjointement avec la classe « XMLParameterParser » qui se charge de lire et d'écrire le fichier de paramètre. Le cluster orange est le composant chargé de la gestion des traces. Nous retrouvons le contrôleur identifié dans le diagramme de robustesse, « TracesManager », qui utilise la classe de persistance « DBNodeQuery » pour effectuer les requêtes SQL sur la trace stockée en base. En jaune, la classe « TracesManagerWindow » est utilisée uniquement au lancement de l'application pour gérer les traces, et dans notre cas d'utilisation, pour sélectionner une trace, et elle collabore avec la classe « DBNodeWriter » uniquement lors de l'initialisation des classes SQL de « PreparedStatement ».

Finalement, en rouge, nous retrouvons les classes les plus utilisées lors de l'exécution. L'écran principal « TraceWindow », qui capture l'action utilisateur pour définir quelle analyse doit être effectuée et se charge d'afficher le résultat. La classe « TraceProcessor » est en faite le contrôleur « TraceControler » identifié dans le diagramme de robustesse, et il travaille conjointement avec la stratégie d'analyse sélectionnée, « TemporalClusterStrategy » pour produire le résultat.

Figure 30

Comparaison des clusters



Premier constat, le découpage en cluster semble pertinent, si ce n'est la classe « DBAccessor » qui se retrouve clusterisée avec les classes de paramétrage sans avoir une quelconque relation avec ces éléments. Pour le reste des composants, il y a un réel regroupement fonctionnel qui démontre les principales collaborations entre les objets.

5.3 Représentation des flux d'information

Maintenant que nous avons nos clusters, nous allons analyser leurs interactions en produisant le diagramme de séquence. Pour pouvoir évaluer le taux de compression des techniques de simplification, nous générons un diagramme de séquence avec et sans ces procédés.

Tableau 3

Ratios de compression des flux d'information

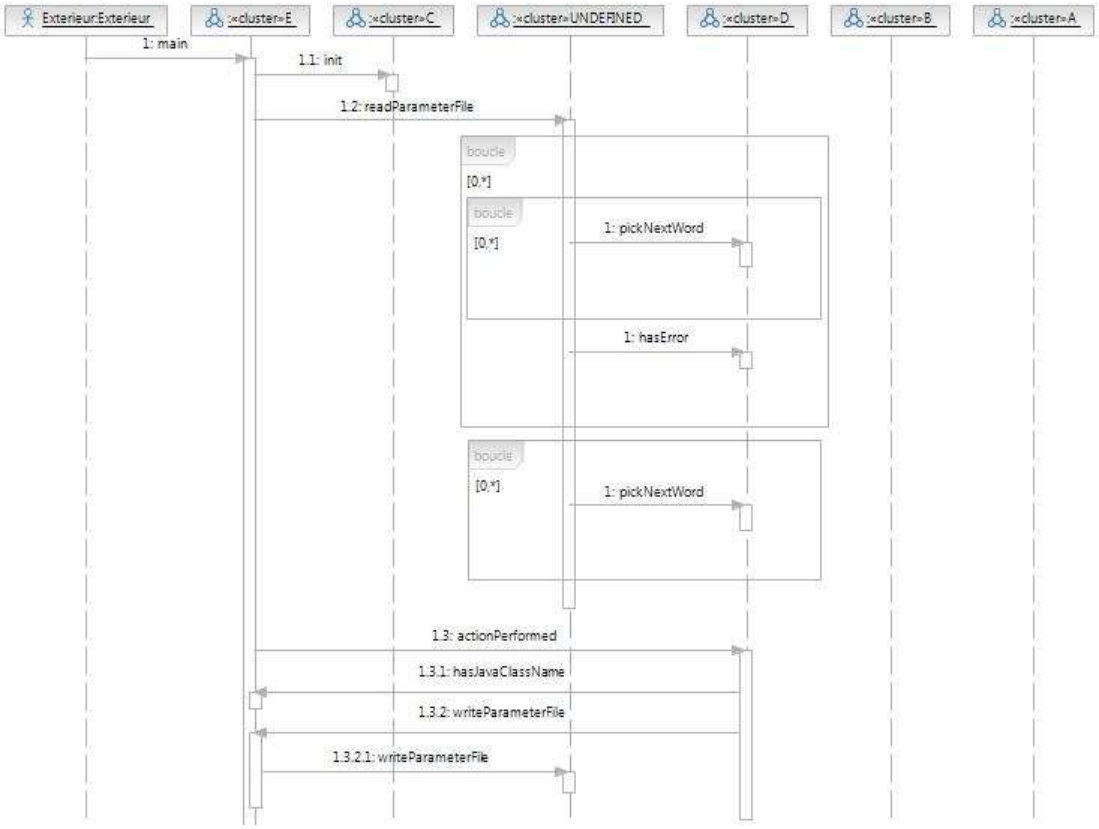
Simplification	Temps de lecture	Temps de simplification	Temps écriture	Nb. Messages
Normal	~ 3 min.	-	13.5 sec.	324
Accesseur	~ 3 min.	~ millisecondes	5 sec.	166
Répétition	~ 3 min.	~ millisecondes	2 sec.	103
Access. + Répét.	~ 3 min.	~ millisecondes	1 sec.	45

Premièrement, nous constatons que les temps d'exécution des algorithmes de simplification sont significativement bas pour être considérés comme nuls. Deuxièmement, la représentation, sans procédés de simplification, mais uniquement avec le regroupement par cluster nous permet de passer de 365'000 appels à 324 appels. Et finalement, la compression à l'aide des procédés de simplification aboutit à une représentation ne comportant que 45 messages, ce qui est un résultat tout à fait utilisable.

Malgré cela, j'ai décelé, lors de la tentative d'ouverture de diagrammes avec RSA, que des graphiques de trop grandes tailles, notamment plus de 100 appels, peuvent prendre énormément de temps à s'afficher.

Figure 31

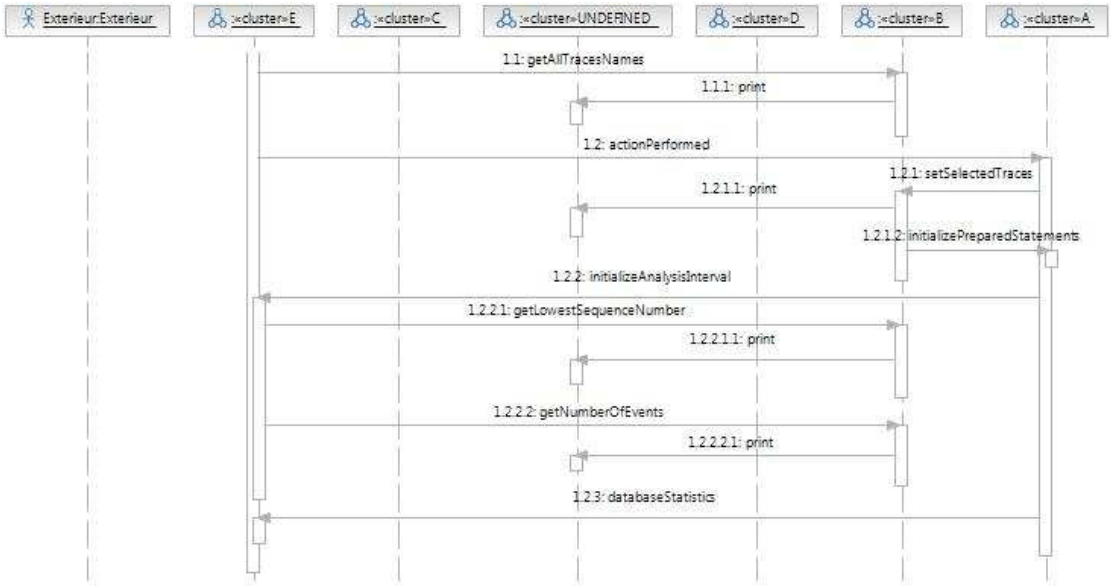
Diagramme de séquence de l'étude de cas – 1ère partie



Pour présenter dans ce document le diagramme de séquence généré, je l'ai découpé en 4 phases importantes. Un cluster « UNDEFINED » est utilisé pour représenter toutes les classes qui ne font partie d'aucun cluster. A la figure 31, nous voyons la première phase, avec l'initialisation de l'application, la méthode « main ». Lors de l'initialisation, l'écran des paramètres est automatiquement affiché à l'utilisateur. Nous avons donc la lecture séquentielle du fichier de paramétrage, « readParameterFile ». Lorsque l'utilisateur valide le paramétrage, « actionPerformed », le paramétrage est sauvegardé dans un fichier, « writeParameterFile ».

Figure 32

Diagramme de séquence de l'étude de cas – 2ème partie

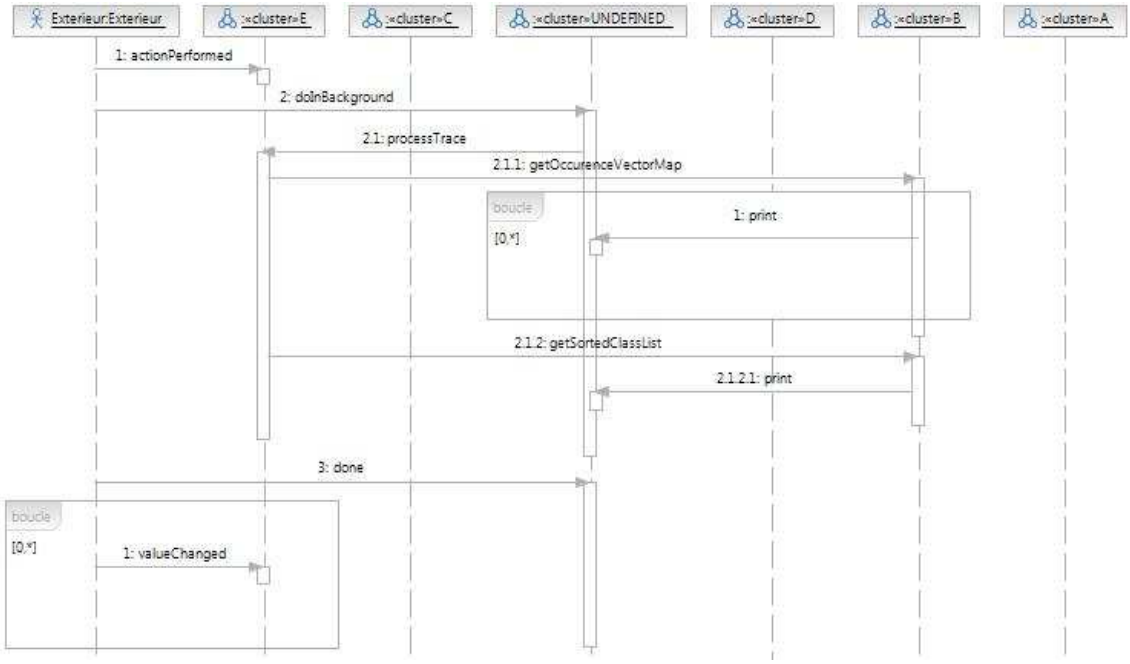


Lorsque la phase de paramétrage est terminée, l'écran de gestion des traces vient également s'afficher à l'utilisateur pour sélection de la trace. Cela débute par la récupération des traces présentes dans la base de données, « getAllTracesnames ». Lorsque l'utilisateur a validé la sélection de sa trace, « actionPerformed », on sette la trace sélectionnée, « setSelectedTraces », on initialise les « PreparedStatements » avec cette trace, « initializePreparedStatements », et on initialise les éléments de l'interface, « initializeAnalysisInterval », « getLowestSequenceNumber », « getNumberOfEvents ». Une dernière action s'effectue lors du démarrage de l'application, c'est la récupération des statistiques de la base qui s'effectue à chaque sélection de trace pour fournir ses informations à l'utilisateur s'il le souhaite.

Nous voyons également apparaître l'événement « print » dans le diagramme. Cette méthode est uniquement une méthode de gestion des logs propre à l'analyseur. C'est donc juste l'impression des logs dans la console.

Figure 33

Diagramme de séquence de l'étude de cas – 3ème partie

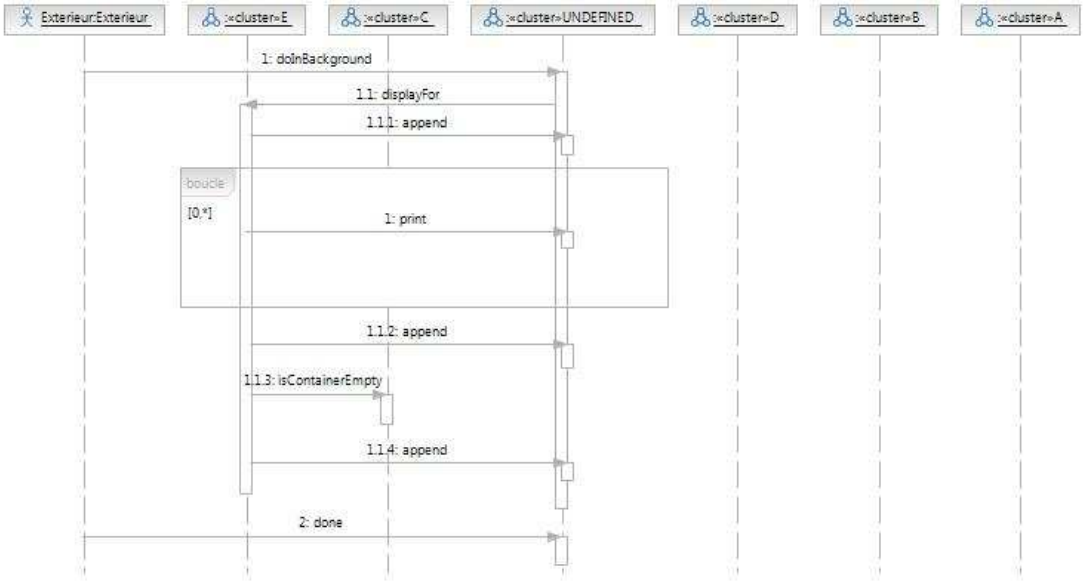


Cette partie est la réalisation de l'analyse. L'utilisateur a lancé l'exécution de l'analyse, « actionPerformed ». Elle parcourt la trace, « processTrace », pour récupérer la matrice d'occurrences, « getOccurenceVectorMap ». Une fois la matrice construite, on récupère les classes présentes dans la matrice pour les afficher à l'utilisateur dans un composant liste, « getSortedClassList ».

Le composant liste, lorsque l'utilisateur sélectionne des classes, parcours la matrice d'occurrence et construit les clusters. La méthode « valueChanged » montre la sélection effectuée par l'utilisateur dans la liste des classes, mais les événements liés à la récupération des clusters sont masqués car toutes les classes intervenant dans ce processus appartiennent au cluster E. C'est un problème, car l'algorithme qui s'exécute dans cette zone est une partie importante de l'exécution. Il serait donc nécessaire de mettre en évidence le message « valueChanged » afin de spécifier à l'utilisateur qu'un important nombre d'échange s'effectue dans cette partie du diagramme. Si l'utilisateur le souhaite, il pourrait alors détailler, avec un nouveau diagramme de séquence, les événements qui se déroulent entre les classes du cluster E en aval du message « valueChanged ».

Figure 34

Diagramme de séquence de l'étude de cas – 4ème partie



Finalement, dans cette dernière phase, le programme procède à l'affichage des résultats, « displayFor ». Il ajoute chaque cluster, les classes contenues, dans la zone d'affichage, « append ».

Nous pouvons retirer de ce cas pratique que le diagramme de séquence comporte un nombre tout à fait tolérable d'élément sans impacter l'efficience du résultat. L'utilisateur retrouve le fonctionnement général de l'application au traves de la collaboration des clusters identifiés. Le seul bémol à souligner, et le masquage de toutes les interactions internes au composant chargé de récupérer la liste des clusters. Pour résoudre ce cas de figure, il faut envisager une évolution qui permet de mettre en évidence les endroits où d'importantes masses d'information sont échangées.

6. Conclusion

Les fonctionnalités graphiques apportées à l'analyseur renforcent mon sentiment que l'analyse statistique par segmentation de la trace est une approche très prometteuse dans le domaine de la réingénierie logicielle. Si l'outil original possédait déjà cette fonctionnalité majeure, une utilisation dans un contexte réelle de maintenance d'application n'était probablement pas envisageable sans les fonctionnalités de représentation graphique et de comparaison avec l'architecture initiale. La migration de l'outil sur des logiciels, tels qu'Eclipse ou RSA, largement répandus dans le monde industriel, permet également de fournir au développeur un outil d'analyse de trace ayant une réelle utilité et cela, directement dans l'environnement de développement et de modélisation qu'ils utilisent tous les jours.

Néanmoins, même si le diagramme de séquence généré se révèle efficace, principalement grâce à ces regroupements fonctionnels que sont les clusters et aux techniques de simplification mises en œuvre, je pense que l'état actuel du projet doit être perçue comme une étape intermédiaire. D'une part, la représentation des flux d'information souffre de quelques lacunes décelées dans l'étude de cas, notamment lorsque certains gros flux sont masqués par la clusterisation, et parce que sa taille peut encore être diminuée en y appliquant d'autres procédés de simplification. Mais surtout, maintenant que les bases d'une intégration dans UML sont posées en s'ouvrant la voie d'un puissant environnement de développement, nous pouvons envisager de nombreuses autres améliorations. Tout particulièrement dans les possibilités de passer à des vues plus détaillées depuis le diagramme de séquence, comme par exemple, se focaliser sur un cluster et ne représenter que l'échange de message interne à ce composant, ou par une autre approche, isoler un message du diagramme de séquence et détailler les appels qu'il effectue dans un nouveau graphe. Le mode de calcul des clusters pourrait également intégrer les techniques d'analyses financières, pour avoir des clusters plus précis (Dugerdil, Sennhauser, 2009).

Je ne souhaite pas énumérer tous les efforts qui peuvent encore être menés, mais tout simplement montrer le potentiel qui se cache derrière un tel outil, surtout dans le contexte actuel, où la pression pesant sur les développeurs les force à négliger de plus en plus la documentation des systèmes et impacte la phase de maintenance.

Bibliographie

Benett C., Myers D., Storey M.-A., German D. Working with “Monster” Traces: Building a Scalable, Usable Sequence Viewer. *Proc. of the 3rd International Workshop on Program Comprehension through Dynamic Analysis*. 2007.

Bifferstaff T. J., Mitbander B. G., Webster D. E. Program Understanding and the Concept Assignment Problem. *Communications of the ACM*. CACM 37(5), 1994.

De Pauw W., Lorenz D., Vlissides J., Wegman M. Execution Patterns in Object-Oriented Visualization. *Proc. of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*. 1998.

Djuric D. Gaševic, D., Devedžic, V. The Tao of Modeling Spaces. *Journal of Object Technology*, 2006, vol. 5, num. 8, pp 125-147.

Dugerdil P., Jossi S. Computing Dynamic Clusters. *Proc. of the 2nd India Software Engineering Conference (ISEC’09)*. 2009.

Dugerdil P. *Domain-Driven Reverse Architecting (DDRA)*. Genève : Haute école de gestion de Genève, 2008. 32 p.

Dugerdil P., Sennhauser D. *Applying Financial Time Series Analysis to the Dynamic Analysis of Software*. Genève : Haute école de gestion de Genève, 2009. 8 p.

Hamou-Lhadj A., Lethbridge T. C. A Survey of Trace Exploration Tools and Techniques. *Proc. of the Conference of the Centre for Advanced Studies on Collaborative Research*. 2004.

Hamou-Lhadj A., Lethbridge T. C. Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System. *Proc. of the 14th IEEE International Conference on Program Comprehension (ICPC’06)*. 2006.

Hamou-Lhadj A., Lethbridge T. C., Fu L. *Challenges and Requirements for an Effective Trace Exploration Tool*. Ottawa : University of Ottawa. 2004. 9 p.

Murphy, P. *Got Legay? Migration Options For Applications*. Forrester Research Inc. 12 septembre 2006.

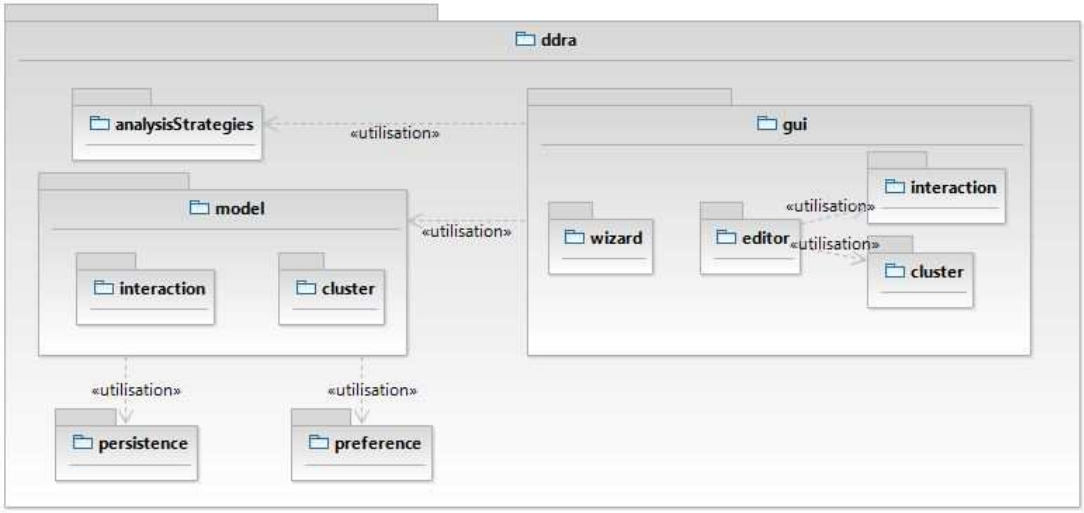
Object Management Group. Catalog of OMG Modeling and Metadata Specifications [en ligne]. 19 mai 2009. http://www.omg.org/technology/documents/modeling_spec_catalog.htm (consulté le 23 mai 2009)

Annexe 1
Documentation utilisateur de l’ancien analyseur

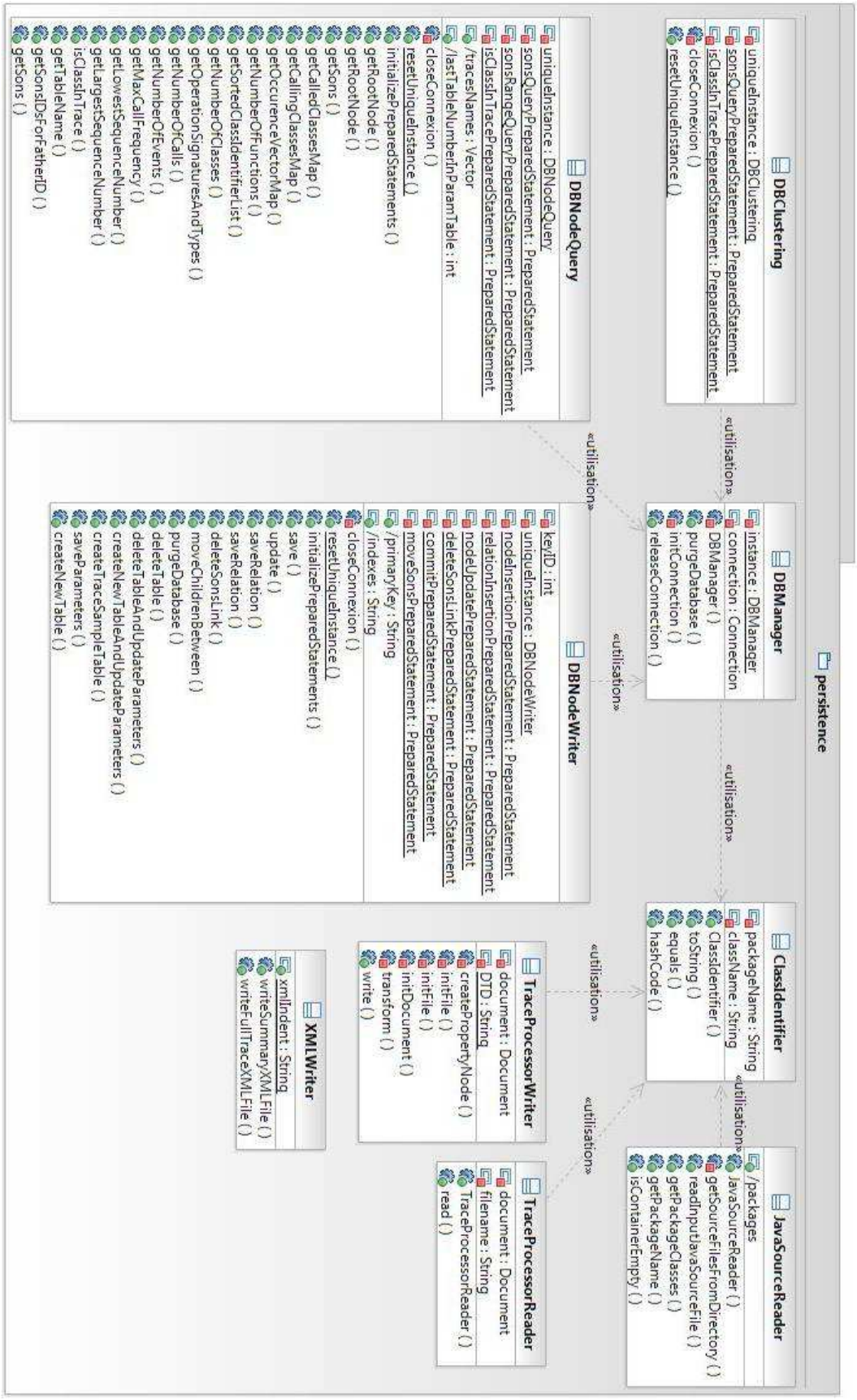
Annexe 2
Architecture de l’ancien analyseur

Annexe 3
Architecture finale du plugin

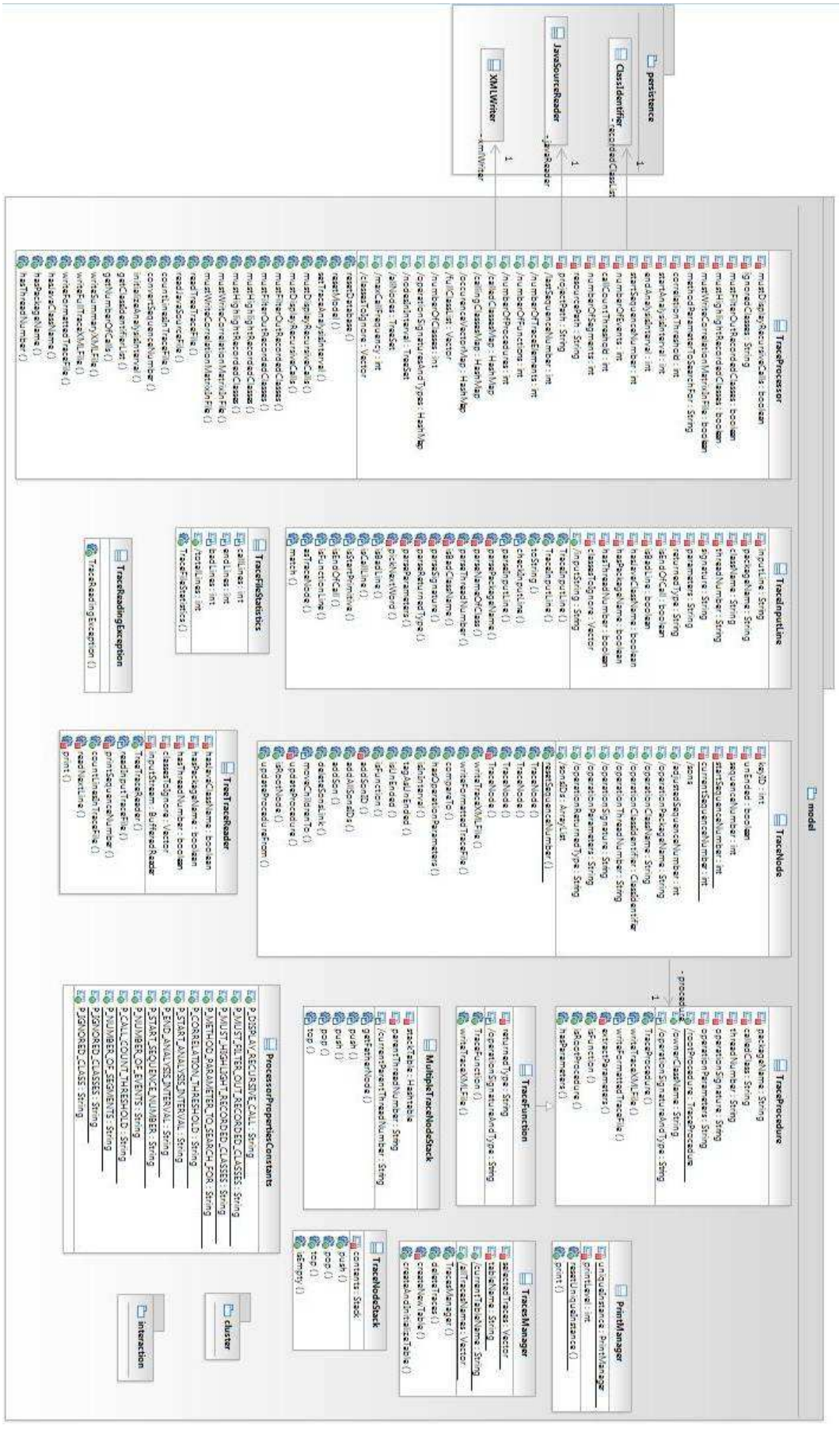
Architecture des packages



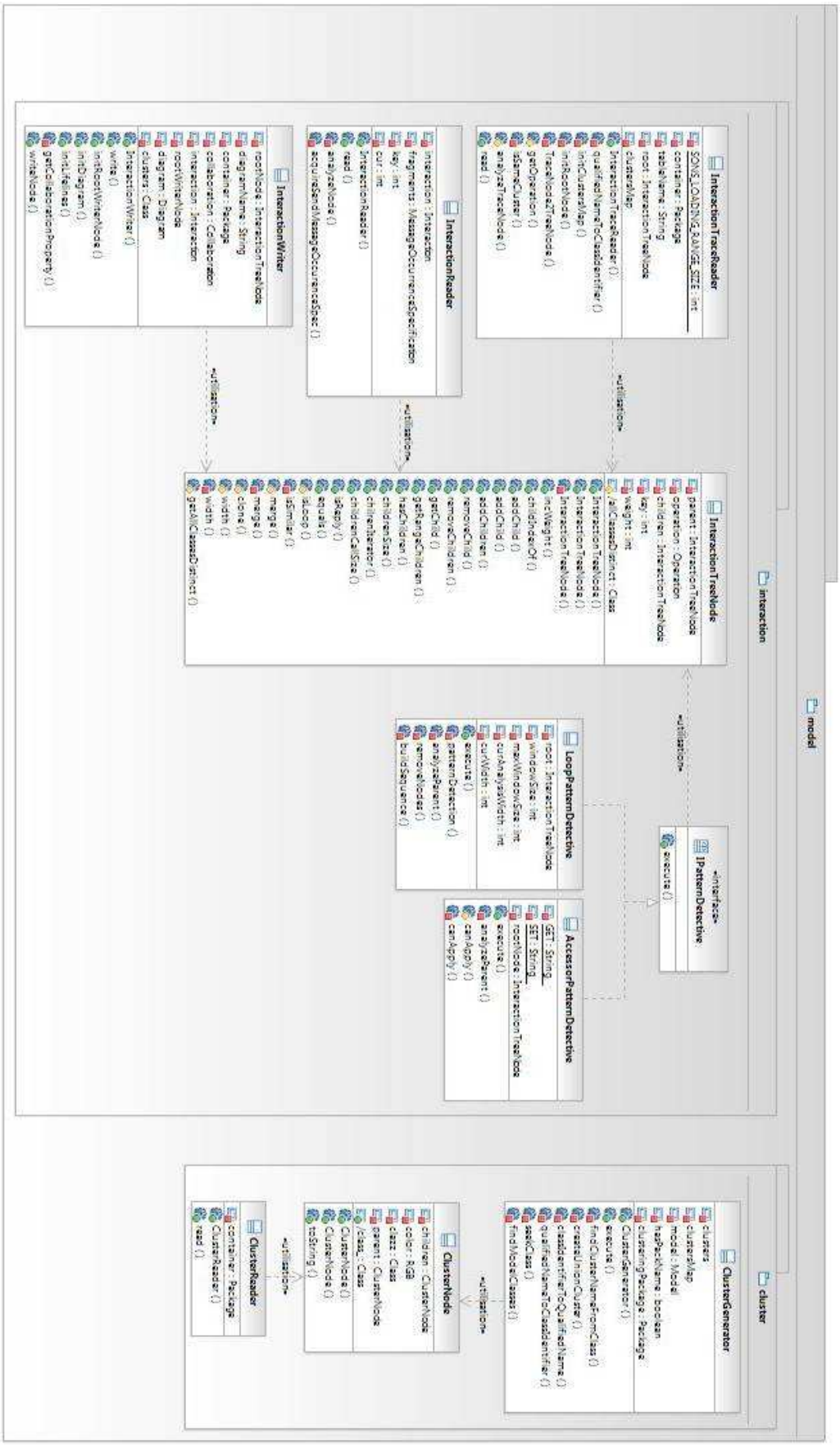
Architecture du package « persistence »



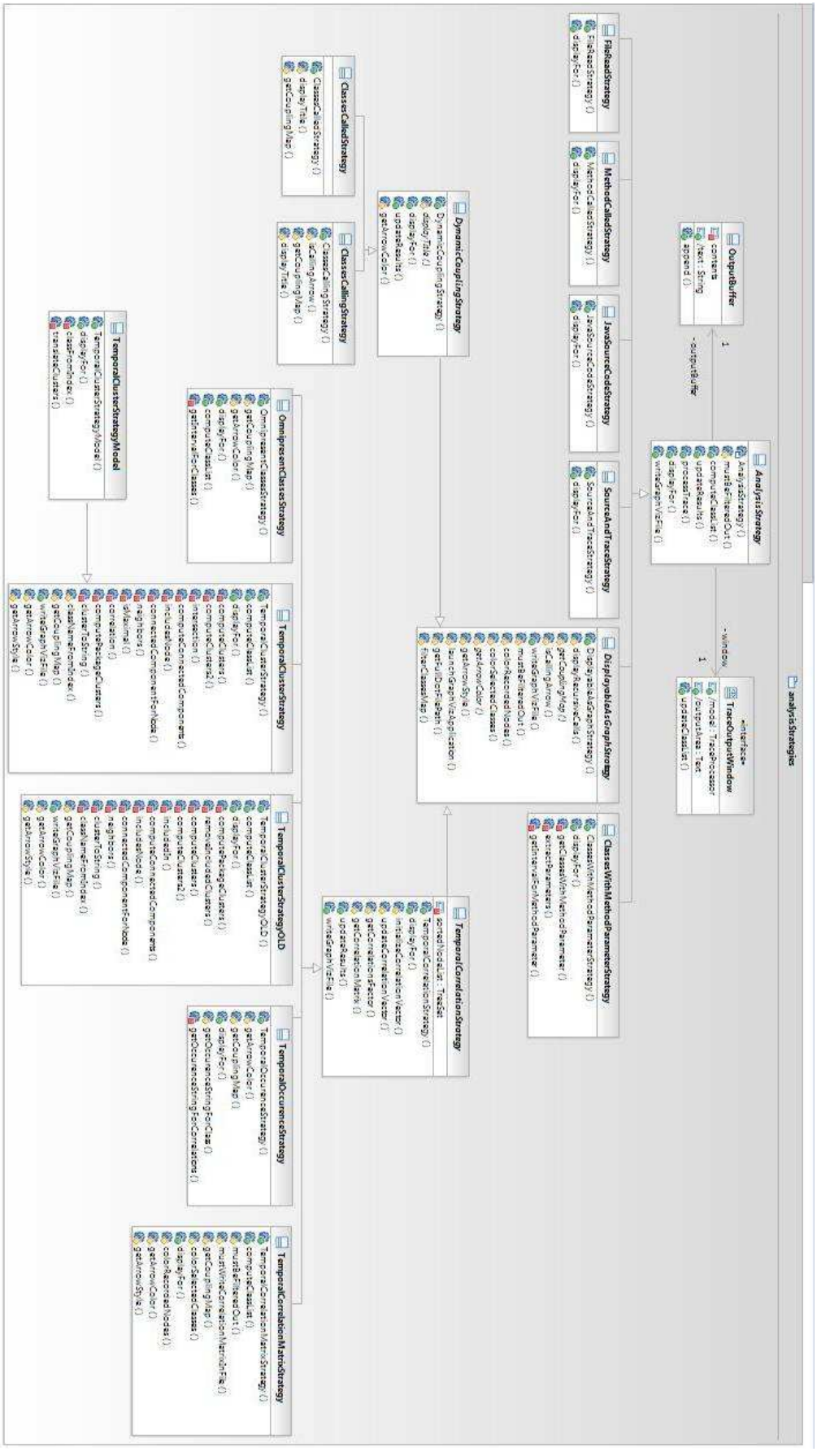
Architecture du package « model »



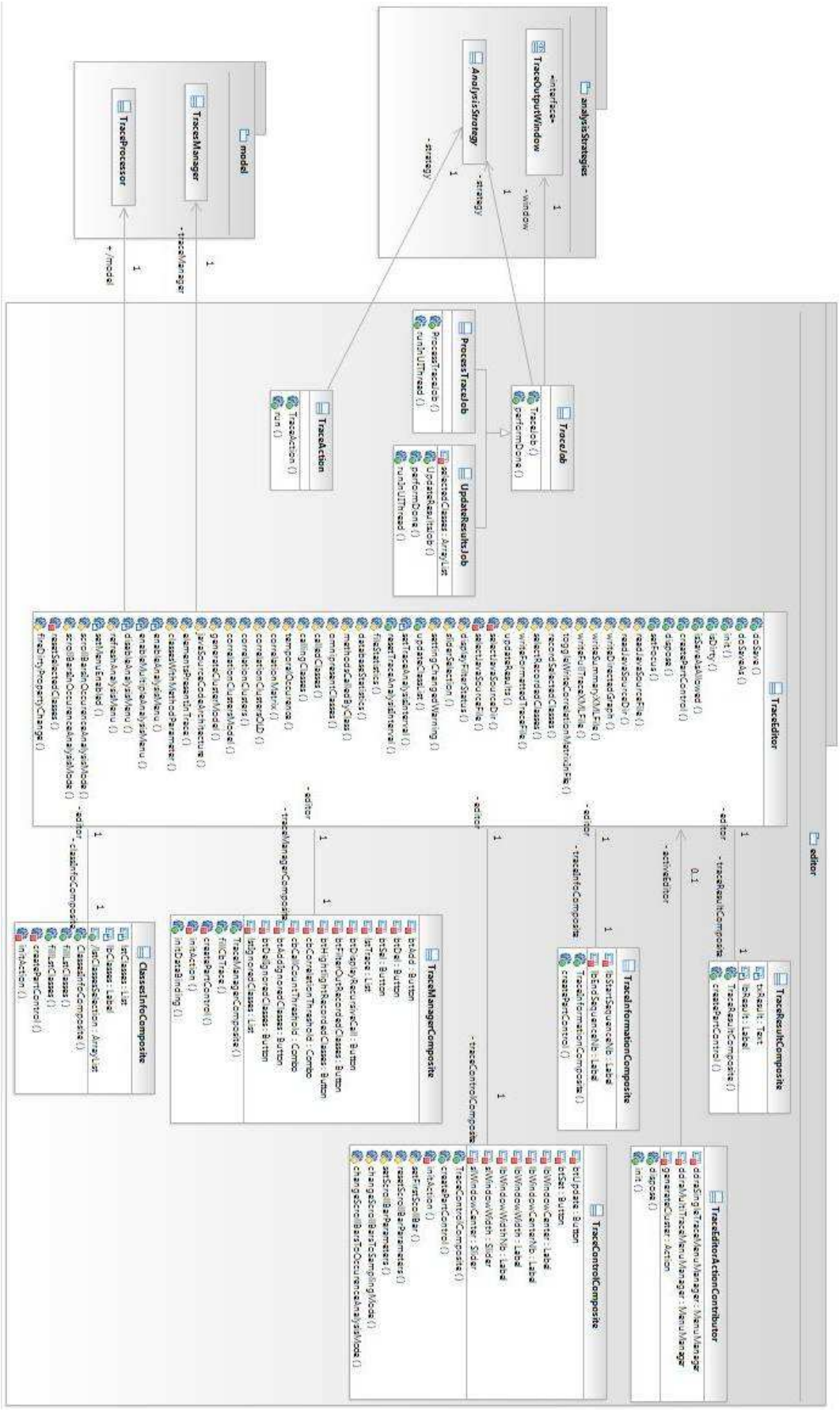
Architecture des packages « model.interaction » et « model.cluster »



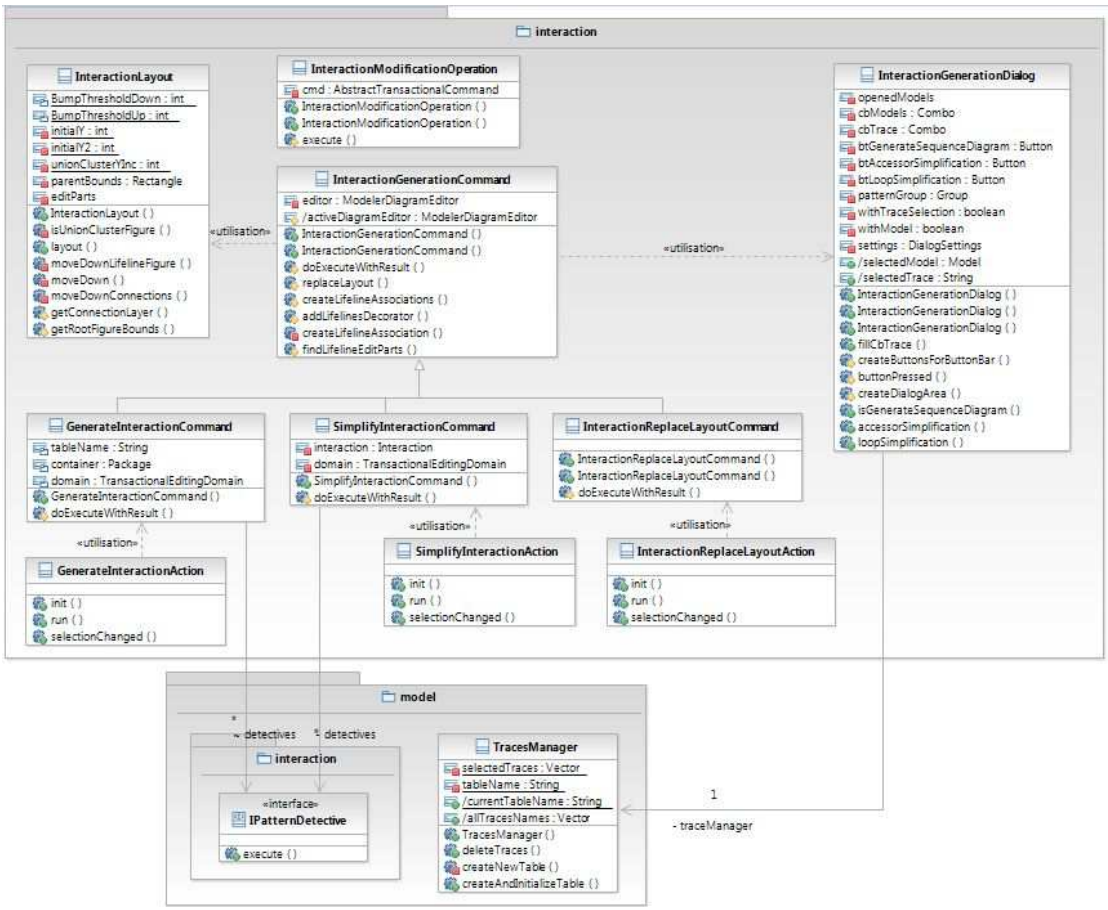
Architecture du package « analysisStrategies »



Architecture du package « gui.editor »



Architecture du package « gui.interaction »



Architecture du package « gui.cluster »