UNIVERSITE MONS-HAINAUT

Faculté des Sciences
Département Informatique

Détection des Bads Smells en Delphi

Auteur: *Masse Nicolas*

Promoteur: *Mens Tom*

Co-promoteur: *Real Jean-Christophe*

Année académique 2004-2005

Table des matières

I. Introduction

a) Contexte de travail

Ce mémoire a été effectué dans le service de Génie Logiciel du département informatique de l' Université Mons-Hainaut. Celui-ci a pour objet la création d'un outil capable de mesurer la maintenabilité de programme écrit en Delphi, en utilisant la notion de bad smell (qui sera introduite plus loin).

Ce mémoire a également été réalisé avec l'aide de la société IDLink. Cette dernière y a contribué en diverses façons, notamment en participant à la phase d'évaluation et de validation.

b) Motivations

Motivations dans un contexte de recherche

Introduction à l'évolution des logiciels

L'évolution des logiciels est une discipline qui a pour objet l'étude de la vie des projets informatiques. Cela revient à étudier quels sont les différents cycles dans la vie d'un logiciel, qu'en ce qui fait qu'un logiciel est maintenu plus longtemps, pourquoi certains projets échouent ou finissent par être abandonnés, … Le but de cette discipline est de pouvoir mieux comprendre comment évoluent les logiciels afin de pouvoir mieux les faire grandir et en assurer la maintenabilité à plus long terme. Ce but peut être atteint de diverses manières, par exemple en respectant certaines consignes lors de la programmation, en utilisant certaines procédures lors de l'élaboration du design du programme, … Il peut également être atteint grâce à l'utilisation d'outils spécialisés permettant de plus facilement manipuler le code source ou, comme dans le cadre de ce mémoire, en analysant divers aspects du code.

Analyse du code

Une constatation que l'on peut faire concernant la plupart des environnements de développement est qu'ils n'intègrent généralement pas d'outils permettant d'analyser le code. Généralement ils se contentent d'avoir un éditeur avec la coloration syntaxique et l'achèvement automatique de code (ou auto-completion), ainsi qu'un explorateur de code. Delphi va un peu plus loin puisqu'il intègre également un concepteur d'interface graphique. Mais rares sont ceux qui intègrent des outils permettant de mesurer la qualité du code. Dès lors cette tâche est entièrement à la charge du programmeur. Seulement mesurer la qualité d'un code est quelque chose qui est à la fois assez complexe, subjectif et pouvant demander un certain temps.

C'est pourtant typiquement ce que l'on doit faire dans le cas de certaines sociétés (c'est le cas d' IDlink par exemple) ou certains programmeurs ne sont là que pour une courte période (stagiaires par exemple). Aussi il est difficile de juger s'il peut être intéressant de reprendre leur code dans un nouveau contexte, de le modifier afin de satisfaire de nouvelles exigences, ou s'il est préférable de recommencer depuis le début.

Dans ce contexte, ce mémoire va essayer de démontrer qu'il est tout à fait possible de mettre au point de tels outils (du moins dans le cadre du langage Delphi), capables d'aider le programmeur dans cette tâche.

Motivations personnelles

J'ai personnellement utilisé l'environnement Delphi par le passé. Selon moi il s'agit là d'un environnement de développement tout à fait intéressant. Son intérêt vient surtout du fait qu'il intègre à la fois un concepteur d'interface graphique et un environnement de développement permettant d'éditer le code efficacement. Mais toujours est-il qu'il lui manque quelques outils. En effet, il m'est également arrivé d'utiliser d'autres environnements, et en particulier l'environnement Eclipse (il s'agit d'un IDE pour java). Ce dernier bien que ne disposant pas de concepteur d'interface avait l'avantage de

posséder des fonctions intégrées permettant de renommer des méthodes, générer automatiquement des accesseurs, ... et d'autres choses encore (en fait de telles fonctions font partie de ce que l'on appelle le refactoring). De même il existe un plugin pour cet environnement nommé PMD permettant d'analyser certains aspects de la structure d'un programme. Lorsque je vis cela pour la première fois, je me suis dis « Voilà tout à fait le genre d'utilitaires qui manquent encore à un environnement tel que Delphi ».

A cela s'ajoute le fait que s'intéresser au problème des bads smells permet de se poser et de réfléchir sur les différentes façon de structurer un programme, pourquoi faut-il programmer d'une certaine manière plutôt que d'une autre, à quoi faut-il réfléchir et où porter son attention lorsque l'on programme,... De telles discussions ne sont pas vaines car de cela dépendent des facteurs tels que la réutilisabilité ou la maintenabilité du code que l'on est en train d'écrire.

c) But du mémoire

La première chose que doit réaliser ce mémoire est d'introduire et d'expliciter la notion de bads smells. Cela comprend la définition de ce qu'est un bad smell, une présentation du contexte dans lequel cette notion est utilisée, ansi que l'introduction d'une liste de smell (cette liste est en fait issu du livre de M. Fowler: « Refactoring: Improving the design of existing code[1] »), l'explication et la justification de chacun d'entre eux, ainsi qu'une critique de la liste et des observations que l'on peut en tirer.

L'objectif principal reste néanmoins la réalisation d'un outil de détection de bads smells automatique, conçu pour analyser du code Delphi. Cet outil doit comporter une interface permettant de lancer la détection et d'en afficher les résultats, mais ne doit pas nécessairement s'intégrer à Delphi. Une question intéressante ici qu'il conviendra de se poser sera de savoir si tous les bads smells de la liste de Fowler peuvent être détectés de façon automatique, et si non pourquoi?

La dernière partie concernera l'évaluation de l'outil développé, son utilisation et éventuellement son intégration en vue d'une exploitation dans le milieu professionnel. Egalement ce mémoire aura pour but de montrer par un exemple quel type d'informations il est possible de retirer de par la détection et l'analyse des bads smells. Cela se fera via une analyse statistique des résultats. De même une analyse de la pertinence de chaque bad smell dans le cadre de l'environnement Delphi sera faite avec l'aide de la société IDLink.

II. Notion de bad smell

a) *Littérature*

Tout a commencé dans les années 70, où l'on a commencé à réfléchir sur l'évolution et la maintenance des logiciels. Très vite l'on s'est aperçu que les logiciels avaient continuellement besoin d'être modifiés. De même l'on s'est aperçu que le fait de modifier continuellement un programme avait tendance à en augmenter à la fois la taille et la complexité[2] .

Avec le temps, certains ont commencé à mettre au point une technique appelée refactoring afin de limiter cette augmentation. Le principe est de changer et d'améliorer la structure interne du programme, sans en changer le comportement externe tout au long de son évolution. Bien que tout les bons programmeurs ont toujours fait du refactoring, souvent sans même le savoir, il a fallu attendre pas mal de temps avant que cela ne soit reconnu comme une part importante du développement de logiciel. Cette technique est au départ issue du milieu des programmeurs Smalltalk (un langage de programmation orienté objet) dans le début des années 80, avant d'être repris par l'ensemble des programmeurs, et ce pour l'ensemble des langages de programmation[1, Foreword].

En 1999 est sorti un livre de Martin Fowler, intitulé « Refactoring: Improving the design of existing code ». Ce livre introduit la notion de refactoring et en explique le principe ainsi que son fonctionnement dans les détails. Mais aussi il introduit la notion de bads smells et en donne une liste préliminaire[1]. Il est en effet intéressant de constater que bien que la notion de refactoring soit connue depuis un certain temps, ce n'est que récemment qu'est apparue la notion de bad smell. Depuis, certains travaux ont été effectués et tentent à démontrer l'intérêt de pouvoir disposer d'un outil capable de détecter la présence de bads smells[3][4]. Mais aucun d'entre eux (du moins à ma connaissance) ne s'est intéressé au langage Delphi, ceux-ci généralement se concentrent plutôt sur des langages comme Java, C++ ou encore SmallTalk. De même, bien que les environnements de développement commencent à intégrer des outils permettant d'automatiser le refactoring[1], ce n'est pas le cas des bad smells. Aussi les outils existants permettant d'effectuer cette analyse sont encore peu nombreux et généralement incomplets[5], et (encore une fois à ma connaissance) aucun d'entre eux n'est prévu pour supporter le langage Delphi.

b) *Définitions*

Si l'on traduit littéralement le terme bad smell, cela signifie « mauvaises odeurs ». Ce terme a été introduit pour la première fois par Martin Fowler dans son livre[1]. En effet, les bads smells réfèrent aux parties d'un code source qui peuvent être considérées comme mauvaises. Mais alors la question qui vient à l'esprit est de savoir quelle est la différence entre un bon code source et un mauvais code source?

A ce sujet, il faut avant tout signaler le fait que le code source d'un programme contienne ou non des bads smells ne signifie en rien que ce dernier va ou ne va pas fonctionner correctement.

En fait les bads smells s'attachent plus à l'organisation interne du code plutôt qu'à l'exécution de celui-ci. Dès lors la recherche de bads smells revient à rechercher les endroits où le code devient difficilement compréhensible, et non les endroits ou il est susceptible de générer des erreurs. Aussi la recherche de bads smells n'a rien à voir avec la résolution de bogues dans un programme, bien que la présence du second puisse être une conséquence de la présence du premier (voir chapitre suivant).

En gros, les bads smells peuvent être définis comme un ensemble de structures qui, lorsqu'elles sont présentes dans un code source, peuvent amener ce dernier à être difficile à comprendre, et dès lors difficile à maintenir et à faire évoluer. C'est pourquoi tout (bon) programmeur veillera à faire en sorte que son code source en contienne le moins possible.

1 JBuilder ainsi que la prochaine version de Delphi devrait intégrer des outils de refactoring, tout comme certains autres éditeurs.

c) Utilité des bads smells

Quantification de la qualité du code

Les bads smells peuvent avoir différentes utilités. L'une d'entre elles est de pouvoir quantifier, ou plus raisonnablement donner une idée de la qualité d'un code source. C'est cette approche qui a motivé certains membres de la société IDlink à collaborer pour ce projet. En effet il arrive dans cette entreprise que de jeunes programmeurs (stagiaires ou autres) soient amenés à développer certains projets pour eux. Aussi le fait de savoir qu'un code source contient un certain nombre de bads smells peut les aider à déterminer ce qu'ils doivent faire avec le code qui a été développé. Aussi peuvent-ils dès lors choisir de l'intégrer tel quel dans leur programmes, de l'intégrer via quelques modifications ou bien tout simplement de le mettre à la poubelle si celui-ci est trop mal structuré et dès lors est trop difficile à maintenir et à réutiliser aux yeux des autres programmeurs de la société.

Bads smells et refactoring

Le refactoring est quand à lui un ensemble de méthodes permettant de changer la structure interne du code source d'un programme sans en changer le fonctionnement lors de son exécution. Généralement cela revient à subdiviser le code différemment de ce qu'il est au départ en vue d'en augmenter l'expressivité.

Une autre utilisation des bads smells est quand à elle issue de la notion de refactoring. C'est d'ailleurs généralement de là que l'on part pour définir un bad smell, en disant que celui-ci est un endroit dans le code source ou il y a lieu de faire du refactoring.

« Structures in the code that suggest (sometimes scream for) the possibility of refactoring [Beck1999] »

Ici les bads smells ont un intérêt indirect lié à celui du refactoring. Parmi les diverses utilités du refactoring, on peut citer l'amélioration continue du design d'un programme qu'il apporte, le fait que cela amène généralement à un code plus facilement compréhensible, ce qui permet à la fois de programmer plus vite et de trouver plus facilement les bugs.

Le problème c'est que pour pouvoir « refactorer » correctement un code, le programmeur a besoin de connaître les endroits où il y a lieu de faire du refactoring, mais aussi la manière de le faire. C'est là qu'interviennent les bads smells, en aidant le programmeur à mettre le doigt sur les parties du programme ou le code est mal structuré.

D'une certaine manière, le refactoring et les bads smells sont issus d'une même idée qui consiste à améliorer le design du programme au cours de sa vie. Les bads smells permettent dès lors de mettre en évidence ce qu'il ne faut pas faire, quelles sont les structures de programmation à éviter, tandis que le refactoring s'intéresse plus à ce qu'il faut faire et par quoi faut-il remplacer les structures mises en évidence lors de l'étape précédente.

d) Approfondissement de la notion de bad smell

Liste de Fowler

Avant d'aller plus loin, il peut être utile d'approfondir cette notion de bad smell. On sait qu'il s'agit de structures pouvant amener un code source à être difficilement compréhensible. Mais quelles sont ces structures?

Dans son livre[1], Martin Fowler (aidé de Kent Beck) introduit une liste de différentes structures que lui et Beck considèrent comme étant des bads smells. Certes cette liste peut être discutée et approfondie (elle le sera plus loin), néanmoins pour le moment elle servira de base.

Code dupliqué

Il s'agit ici du bad smell considéré comme étant le pire par Fowler et Beck. Le code dupliqué concerne toutes les parties du code ayant une même structure. Par exemple, si 2 méthodes utilisent une même structure, alors le code peut être simplifié en unifiant ces 2 méthodes ou du moins, la partie de code qu'elles ont en commun.

A noter que ce bad smell a déjà à lui seul fait l'objet d'un travail réalisé par un autre étudiant de l'université, c'est pourquoi je n'en parlerai que très peu.[6]

Méthode longue

En effet, plus une méthode est longue, plus il est difficile de savoir ce qu'elle fait et ce à quoi elle sert. Idéalement, toute fonction devrait se contenter de réaliser 1 et 1 seule action.

A cela peut s'ajouter une autre considération d'ordre pratique : la taille de l'écran. Il est en effet plus difficile de pouvoir comprendre une méthode lorsqu'il faut sans cesse scroller pour aller voir diverses parties de son code que lorsque l'entièreté du code de la fonction peut être affiché sur 1 seul écran.

Large classe

Le principe est ici le même que pour les méthodes. Le fait de créer des classes trop importantes et réalisant trop de tâches différentes ou tenant compte de trop de choses peut rapidement rendre cette classe difficile à comprendre dans son ensemble, et amener également à réaliser des méthodes trop complexes.

Longue liste de paramètres

Une longue liste de paramètres rend souvent une fonction plus difficile à comprendre que si elle n'en comporte que 2 ou 3. Qui plus est, cela se fait ressentir à la fois dans la fonction en elle-même mais aussi à chacun des endroits ou celle-ci est utilisée.

Changements divergents

On parle de changements divergeants quand une classe est amenée à être modifiée à différents niveaux, de différentes manières et pour des raisons différentes. Il devient dès lors difficile de maintenir la classe en question et d'assurer les différents changements.

« Shotgun surgery »

On a un cas de shotgun surgery lorsqu'un changement dans une classe/méthode est susceptible d'entraîner un grand nombre de changements dans l'ensemble du code d'un programme. Aussi lorsque cela arrive, il faut s'armer de courage et de patience afin de retrouver tous les endroits ou la classe/méthode en question est utilisée et adapter le code.

Envie de caractéristique

En anglais « feature envy ». C'est le nom que Fowler et Beck utilisent pour caractériser le fait que certaines méthodes vont parfois chercher la majorité des variables dont elles ont besoin dans une autre classe que celle à laquelle elles appartiennent. Cela est souvent le signe que ces méthodes ne se trouvent pas là où elle devraient se trouver.

« Data clumps »

Les data clumps sont en fait des variables que l'on retrouve fréquemment ensemble, que ce soit à l'intérieur d'une classe ou lors du passage de paramètres. Le fait de regrouper de telles données en 1 seule classe permet souvent de simplifier l'appel de certaines fonctions. De même le fait d'ajouter des membres à cette nouvelle classe permet de simplifier le travail fait par les fonctions utilisant ces variables mais aussi de limiter la redondance du code.

Obsession de primitives

Les primitives sont les types de variables définis comme étant les types de base d'un langage de programmation. Généralement, dans les langages de programmation orienté objet on redéfini une classe pour chaque type de donnée que l'on peut rencontrer. On parle d'obsession de primitive lorsqu'un programmeur utilise de façon abusive les types inclus dans le langage de programmation plutôt que de définir un nouveau type de donnée. (Ex: utiliser des entiers pour représenter un code postal, ou un numéro de téléphone)

Utilisation des switchs

Ce qu' ont voulu mettre en évidence Fowler et Beck, c'est le fait que si une classe possède plusieurs états qui sont gérés de façon différentes par certaines méthodes de la classe, alors il vaudrait mieux créer une sous-classe pour chacun des états de cette classe. Ils parlent de la clause switch car dans un langage tel que java (qui est celui sur lequel s'appuie Fowler dans son livre), une telle chose se caractérise par la présence de clause switch dans le code de diverses fonctions; chacun de ces switchs testant un même champ de la classe.Cela ne remet pas pour autant en cause l'utilisation du *« switch »* en java (*« case of »* en Delphi).

Hiérarchies d'héritage parallèles

Une telle structure peut se reconnaître de par le fait qu'a chaque fois que l'on crée ou que l'on modifie une sous-classe d'une classe, on se retrouve obligé de créer ou de modifier une sous-classe d'une autre classe. Cela complique en effet la tâche du programmeur qui doit créer une nouvelle classe, ainsi que la taille de l'arbre d'héritage des classes qui grandit plus qu'il ne le devrait.

Classes fainéantes

Ce sont des classes qui pour une raison ou pour une autre se retrouvent à n'avoir quasi plus aucun rôle, et n'assurent plus aucun travail. Cependant, leur présence continue d'alourdir la hiérarchie des classes et dès lors complique la compréhension du fonctionnement global du programme.

Généralité spéculative

La généralité spéculative se produit lorsque l'on crée une nouvelle classe en spéculant sur les besoins et les différents cas qu'elle sera amenée à traiter. Dès lors on surcharge inutilement cette classe en y ajoutant du code qui ne sera peut-être jamais utilisé.

Champ temporaire

Fowler et Beck ici parlent de champs d'une classe qui ne sont utilisé que par certaines fonctions de celle-ci, et qui ne sont pas initialisées le reste du temps. De tels champs ne reflètent dès lors l'état dans lequel se trouve l'objet en question comme ils devraient le faire. Aussi comprendre quel peut être le rôle d'une variable dans une classe alors qu'elle n'est pas utilisée peut s'avérer périlleux...

Chaîne de messages

Typiquement, une chaîne de message est constitué d'une classe qui accède à une seconde en demandant une troisième, puis qui à partir de cette troisième en demande une quatrième, ... et ainsi de suite. Au niveau du code, cela peut se voir comme état une liste de get du style A.getB.getC.getD...

Le problème d'une telle chaîne est que cela crée une dépendance entre la classe A et les classes B, C,D,... Aussi chaque changement dans une des classes B, C, D,... est dès lors susceptible d'affecter la classe A, ce qui rend le code plus difficile à maintenir et à comprendre.

L'homme du milieu

Cela se dit d'une classe qui joue plus le rôle de ciment entre les différentes briques du programme qu'autre chose. Une telle classe se caractérise par le fait qu'elle délègue la majorité du travail qu'elle devrait assurer à d'autres classes. Dès lors, on peut à partir d'un certain point considérer que cette dernière surcharge inutilement le diagramme des classes et que le programme serait plus simple à comprendre si elle n'existait pas.

Intimité inappropriée

Fowler et Beck considèrent que chaque classe ne doit pas connaître trop de chose concernant les autres classes. Aussi ils considèrent que 2 classes ne devraient pas passer leurs temps à utiliser les champs l'une de l'autre.

Ce genre de problème se rencontre surtout avec les sous-classes, qui connaissent en général trop de chose de leur classe parente.

Classes alternatives avec interfaces différentes

Cela se dit de 2 classes capables d'assurer le même rôle au sein d'un programme, mais qui pour se faire utilisent des interfaces différentes. Le problème est que si l'on désire remplacer une classe par une autre équivalente mais dont l'interface est différente, alors il faut également modifier tous les endroits où était utilisée la classe à remplacer, ce qui peut être long et fastidieux.

Librairie incomplète

Le problème qui peut se poser ici vient de l'utilisation de classes développées par quelqu'un d'autre. En effet, lorsque quelqu'un développe une librairie, il peut arriver que celui-ci tourne cela de telle manière que la librairie en question ne soit pas ou peu adaptée au travail que l'on désirerait lui faire faire.

Classe de données

Il s'agit de l'existence de classes qui contiennent des champs, des méthodes d'accès à ces champs et rien d'autre. Cela signifie que ces classes se contentent de stocker des données, lesquelles sont probablement manipulées par d'autres classes.

« Refused bequest »

Cela concerne les classes enfants qui ne réutilisent pas toutes les fonctions héritées de leurs parents. Le fait de ne pas réutiliser systématiquement les méthodes héritées dans une classe, bien que contraire à la philosophie des langages de programmation orienté objet n'est pas considéré comme étant un bad smell en soi. Par contre, une classe doit toujours respecter et rester conforme à l'interface dont elle hérite, et ne doit pas redéfinir la portée des fonctions.

Commentaires

En fait la présence de commentaires dans le code n'est pas un bad smell en soi. Néanmoins Fowler et Beck considèrent que si un code nécessite à un endroit d'être commenté, cela est un signe selon lequel le code à cet endroit n'est pas bien structuré. Aussi préconisent-ils la façon de procéder suivante:

à chaque fois que l'on a envie d'ajouter un commentaire quelque part, alors il faut d'abord regarder sa fonction, voir s'il n'y a pas moyen de la décomposer ou de la renommer, et bien souvent une fois que cela est fait le code devient plus clair et le commentaire devient dès lors superflu.

(« ..comments are often used as a deodorant. » [M. Fowler])

Observations

L'un des premiers intérêts de cette liste est que celle-ci a le mérite d'exister et de nous permettre d'avoir un point de départ. Néanmoins, le fait qu'elle constitue un point de départ ne signifie pas qu'elle est exempte de défaut.

Tout d'abord il faut tenir compte que cette liste a été introduite par M. Fowler. Cela signifie qu'elle ne représente en rien une liste ultime et exhaustive de tous les bads smells que l'on peut rencontrer. En effet, quand Fowler soutient dans son livre une liste de 22 bads smells, il se peut que d'autres personnes en rajoutent d'autres, de même il peut arriver que certains bads smells soient ou ne soient pas applicable (entendre par là utile) en fonction du projet particulier auquel on s'intéresse.

Qui plus est, cette liste a été formulée dans le cadre du langage de programmation java. Le présent travail portant sur le langage de programmation Delphi, il se peut que certains bads smells aient besoin d'une adaptation ou soient tout simplement inapplicable dans le cadre de Delphi. De même, il se peut que l'on trouve certains bads smells typiques à Delphi. (Cet aspect sera vu plus en détail plus loin). Cela restera néanmoins ici dans des proportions limitées dans la mesure ou Java et Delphi sont 2 langages de programmation orientés objets et ont de ce fait certains principes de fonctionnement en commun. Il va de soi que cela aurait des proportions plus importantes si l'on voulait se préoccuper de langages de programmation logique ou fonctionnelle.

Un autre point qui peut être discuté est la pertinence de chacun de ces bads smells. Même si la logique peut nous faire penser que les bads smells dont parlent Fowler et Beck peuvent bien amener

à des erreurs de programmation, rien ne permet cependant d'en être complètement sur. Malheureusement, on manque encore d'études permettant de mettre en évidence la pertinence de ces critères. Aussi certaines structures du langage comme le casting sont considérées par certains comme étant des bads smells tandis que d'autres ne voient aucun inconvénient quant à leur utilisation. Un autre problème vient du manque de précision de certains bads smells. Quand faut-il considérer qu'une classe est trop importante ou qu'une méthode a trop de paramètres?

Enfin, je formulerai une autre critique concernant non pas la liste en elle même mais plutôt sa formulation générale. Les bads smells ont étés présentés comme étant une liste plate de différentes structures n'ayant aucun lien entre elle. Mais à bien y regarder, on peut constater que certaines ont des points en commun, parce qu'elles sont issues d'une même idée de la programmation ou parce qu'elles s'appliquent au même type de donnée. C'est pourquoi je vais dans la section suivante essayer de classifier ses différents bads smells en fonction du type d'erreur (de mauvaise structure) auxquels ils font références. Cela devrait permettre d'avoir une meilleur compréhension de chacun des bads smells et de leur raison d'être, mais aussi en plus du fait de savoir qu'un programme contient un nombre important de bads smells, d'avoir une idée de quel type d'erreur de structuration est généralement commise.

Classification des différents smells

Les métriques du logiciel

La notion de métrique du logiciel est quelque chose de connu et considéré depuis longtemps comme étant un bon prédicateur de la maintenabilité d'un logiciel. Il existe d'ailleurs certaines études montrant la corrélation entre les deux[4].

Les métriques du logiciel sont un ensemble de paramètres tout à fait mesurable dans le code source d'un logiciel. J'entends par là qu'il s'agit de choses telles que le nombre de lignes de code, le nombre de variables,... ou des choses plus complexes comme la complexité cyclomatique ou encore les calculs de couplage et de cohésion.

Historiquement parlant, les métriques du logiciel ont existé avant que ne soit introduite la notion de bad smell. Cependant, il convient aujourd'hui de considérer les métriques du logiciel comme une catégorie de bads smells dans la mesure où nombre de ces derniers peuvent être exprimés sous forme d'une ou plusieurs métriques. Il suffit pour cela de déterminer que l'on est en présence d'un bad smell à partir du moment où une métrique atteint une certaine valeur considérée comme étant une borne (généralement inférieure). Par exemple on peut considérer qu'un bad smell serait le fait d'avoir une méthode comptant plus de 200 lignes de code, ou une classe possédant plus de 10 variables d'instances.

Il convient dès lors de placer dans cette catégorie tous les smells étant issus ou pouvant se ramener à une ou plusieurs métriques du logiciel. Par exemple les bads smells méthode longue et large classe peuvent tous deux être mesurés (en comptant par exemple le nombre de lignes d'une méthode ou de variables d'instances d'une classe). Il en va de même pour le fait qu'une fonction peut avoir un trop grand nombre de paramètres. A noter également les cas des bads smells Shotgun surgery et changement divergent. En fait, si l'on se fie uniquement à la définition qu'en donne Fowler, ces bads smells ne sont à ranger dans cette catégorie. Cependant, la définiton de M.Fowler ne me permettant pas de détecter ces derniers (v. plus loin), j'ai pris la liberté de les redéfinir de la façon suivante : On se trouve dans un cas de shotgun surgery à partir du moment où une classe est utilisée dans beacoup d'autre, car chaque changement dans cette classe est susceptible de provoquer un changement dans une des classes qui utilisent la classe considérée. De même le changement divergent est le même que shotgun surgery, mais juste l'inverse à savoir que l'on se retrouve dans un cas de changement divergent quand une classe est liée à trop d'autres classes, chaque changement dans chacune de ces classes pouvant affecter la classe courante.

En se basant sur ces présentes définitions, ces deux bads smells peuvent être désormais repéré en utilisant des métriques du logiciel, telle que le couplage par exemple. Ils sont donc bien à leur place.

Non respect des « règles » de programmation

Lorsque l'on utilise un langage de programmation, il est bon de savoir que généralement celui-ci a été conçu afin d'être utilisé d'une certaine manière et en respectant certaines directives. Par exemple les langages de programmation orientés objet ont comme philosophie de regrouper ensemble les

différents types de données que l'on peut rencontrer ainsi que les différentes manipulations que l'on peut leur appliquer. Les bads smells que je vais lister ici sont simplement le reflet du non-respect de cette philosophie de programmation. Parmi eux, on peut citer les bads smells tels que chaîne de message, data clumps ou encore Intimité inappropriée. En effet une des idées sous-jacente de la programmation orientée objet est le fait que les objets doivent avoir des dépendances entre eux limitées. La présence de l'un de ces bads smells est dès lors symptomatique d'un problème à ce niveau.

Classons aussi ici des bads smells comme champ temporaire, utilisation de switchs, ou obsession de primitives qui témoigne tous d'une mauvaise utilisation de la programmation orientée objet.

Enfin les bads smells tels que hiérarchies d'héritage parallèles et refused bequest témoignent de problèmes au niveau de la hiérarchie des classes du projet. Pour le cas des hiérarchies d'héritage parallèles, cela est évident (c'est la hiérarchie même des classes qui est en cause). Pour le cas du refused bequest, cela est dû au fait qu'une classe est censée accepter les méthodes dont elle hérite et éventuellement les surcharger, mais pas les redéfinir totalement.

Autres

Sont regroupés ici des smells que je n'ai pu classer dans les catégories juste au dessus.

Librairie incomplète: Personnellement, je ne considère pas cela comme un bad smell. Si l'on se retrouve à un moment donné coincé car les librairies que l'on utilise sont incomplètes, cela signifie selon moi qu'il y a eu une erreur dans le choix de cette librairie, et non pas un problème au niveau du code.

Généralité spéculative : Le problème ici est que l'on veut faire faire à son code plus de chose que nécessaire. Bien que le problème ne vienne pas du code lui-même, le fait d'y ajouter la gestion de cas qui ne sont pas utilisés y entraîne une plus grande complexité alors que cela est inutile.

Classes alternatives avec interfaces différentes: Ce bad smell intervient si l'on désire remplacer une classe existante par une autre. Cela peut aussi arriver si l'on crée une classe alors qu'une autre existait déjà. Cela peut mener alors à un code désordonné ou dans certains cas l'on utilise une classe et dans d'autres on utilise l'autre classe. Le problème est que comme ces 2 classes ont des interfaces différentes, le fait de remplacer l'une par l'autre peut être compliqué et demande un certain temps.

En ce qui concerne les commentaires, j'ai placé ce bad smell ici dans la mesure où les commentaires ne font pas réellement partie du code et où il était dès lors difficile de le placer ailleurs. Cependant il faut faire attention, bien que les commentaires ne fassent pas partie du code et n'influence pas le comportement du programme, ceux-ci doivent être maintenus afin de refléter l'état actuel du code. Aussi un changement dans ce dernier peut parfois impliquer de devoir modifier le commentaire en conséqence.

Tableau récapitulatif

	Liste des Bads Smells	Commentaires
Métriques du logiciel	Méthode Longue	Se ramène à un calcul du nombre de variables d'instances et du nombre de lignes de code de chaque méthode.
	Large classe	Se ramène à un calcul du nombre de méthodes et de champs appartenant à la classe.
	Longue liste de paramètres	Simple comptage du nombre de paramètres.
	Changements divergents	Peut se ramener à compter le nombre de classes utilisées par une classe donnée.
		Peut se ramener à compter le nombre de classes utilisant une classe donnée. (métrique

	Shotgun Surgery	de couplage)
	Classes fainéantes	Se ramène à calculer le nombre de champs et la complexité moyenne de s méthodes de la classe.
	Code dupliqué	Les différentes techniques de programmation et en particulier la programmation orientée objet prônent la réutilisation du code existant plutôt que le copier/coller.
	Envie de caractéristique	Ce bad smell est caractéristique d'une classe qui accède à une donnée à laquelle elle ne devrait pas avoir accès.
	Data clumps	Dans la mesure du possible, il est toujours préférable de regrouper des données qui se retrouvent souvent ensemble dans une même classe.
	Obsession de primitives	Utiliser des objets pour encapsuler les actions à faire sur des données simples plutôt que de recommencer ce travail à chaque fois.
	Utilisation de switch	Une classe se doit d'avoir, autant que possible, toujours le même comportement. Ce dernier ne doit pas dépendre uniquement de la valeur d'un champ particulier de la classe.
	Hiérarchie d'héritage parallèle	Symptomatique d'un problème au niveau de la façon dont les classes sont organisées.
Non respect des « règles » de programmation	Champ temporaire	Ce type de champs est à éviter car ils ne reflètent pas l'état d'un objet.
	Chaîne de message	Ce type de structure rend trop dépendant la méthode où elle se trouve vis-à-vis de l'organisation des classes.
	Homme du milieu	Ce type de classe surcharge inutilement les diagrammes et la hiérarchie des classes.
	Intimité inappropriée	La justification est la même que dan le cas de l'envie de caractéristique.
	Classe de donnée	Une classe est faite pour encapsuler à la fois les données et les méthodes à appliquer sur ces données, pas seulement les données.
	Refused Bequest	Si une classe n'hérite pas proprement des méthodes de sa classe parente, cela risque d'entraîner des incohérences dans son fonctionnement.
	Généralité spéculative	Cela est lié au fait que l'on veut prendre en compte trop de cas particulier, complexifiant inutilement le code.
Autres	Classe alternative avec interface différente	A ne considérer que dans le cas où 2 classes sont censées être interchangeables.
	Librairie incomplète	Erreur dans le choix de la librairie, pas dans la façon dont le code est organisé.

Commentaires	Les commentaires ne font pas réellement partie du code, ou du moins n'influence pas l'exécution de celui-ci. Néanmoins, ils doivent être maintenus.

Interprétation des bads smells

En fait, à l'heure actuelle il est difficile de savoir dans quelle mesure la présence d'un ou de plusieurs smells combinés peuvent amener tôt ou tard à des problèmes de stabilité(bugs) ou de maintenabilité d'un programme. La raison en est simple: très peu d'études ont été faites sur le sujet. Il existe bien quelques études portant sur les métriques du logiciel à ce sujet[4], mais à ma connaissance aucune étude prenant en compte un ensemble de bads smells (plus général que les métriques) et tentant de faire le lien entre eux et la maintenabilité de logiciels n'a été faite. Qui plus est, les études portant sur les métriques du logiciel le font généralement sur un ensemble de métriques restreint, et sur un nombre d'applications restreintes.

De tout cela, on est forcé de retirer le fait qu'il n'existe pas de règles empiriques en la matière et qu'il est dès lors du ressort du ou des programmeurs d'interpréter les résultats et de pondérer l'importance des bads smells détéctés en fonction de leurs expériences. Cela a comme conséquence qu'il sera nécessaire (et il s'agit là de l'un des objectifs du présent travail) de mettre au point une interface permettant au programmeur d' avoir connaissance des résultats et de pouvoir les analyser facilement. Notons que pour se faire il existe essentiellement deux types d'interfaces[1]:

- La première consiste en une sorte de navigateur qui permet de voyager au travers des bads smells et de les classer/regrouper selon différents critères (classe/fichier auquel ils appartiennent, type de bad smell, ...). Ce type d'interface est plutôt destiné au programmeur même car il s'intègre facilement dans son environnement de travail.

- La seconde consiste en un outil de visualisation capable de représenter graphiquement l'ensemble des bads smells d'un projet. L'avantage de ce type de représentation est qu'elle permet de plus facilement mettre en évidence l'omniprésence d'un bad smell dans un projet ou de voir par exemple quel bad smell est typique de l'une ou l'autre partie du programme. Ce type d'outil est plutôt orientée pour le gestionnaire d'un projet informatique si ce dernier veut par exemple en controller l'évolution.

Pour le présent travail, l'interface mise au point sera du premier type pour diverses raisons. Tout d'abord une telle interface est plus facile à concevoir et la création d'une interface de visualisation aurait débordé du cadre de ce mémoire. Ensuite le personnel de la société Idlink était plutôt orienté vers ce type de solution. En ce qui concerne l'analyse des résultats, une analyse statistique sera néamoins réalisée afin de montrer que ceux-ci peuvent être extrait sans pour autant nécessiter la création d'une interface dédiée.

[1] Dans leur document, Eva Van Emden et Leon Moonen [5] parlent également de ce type d'interface. Ils mentionnent aussi le fait que l'utilisateur peut ne pas être une personne mais un programme tiers pour lequel seul compte l'accès à l'information.

III. Recherche de smells dans du code Delphi

a) *Présentation de Delphi*

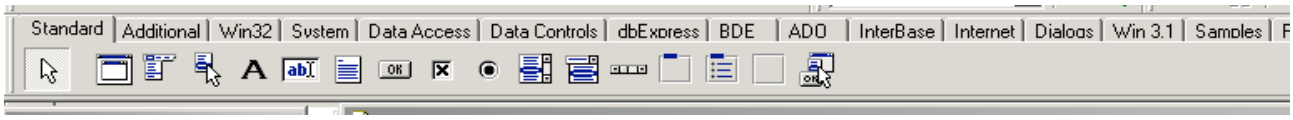
Présentation de l'IDE

Delphi est en fait le nom donné à un environnement de développement appartenant à la société Borland. Il comporte entre autre un concepteur d'interface graphique, un éditeur de code intégré, un compilateur, un debuggueur, … En voici un aperçu:

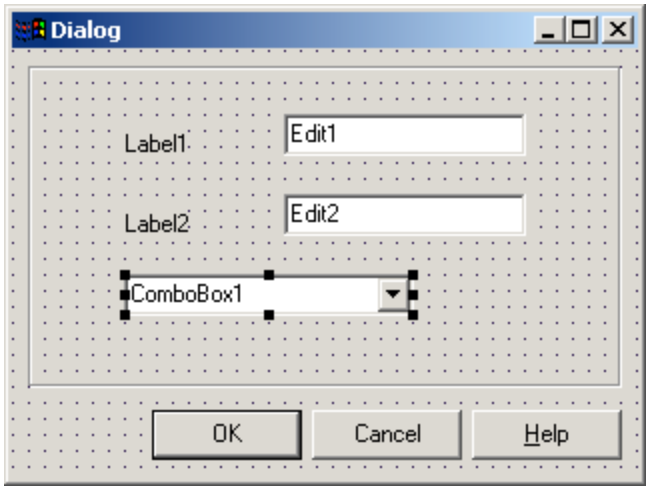
Concepteur d'interface

Celui-ci se compose essentiellement de 3 outils:

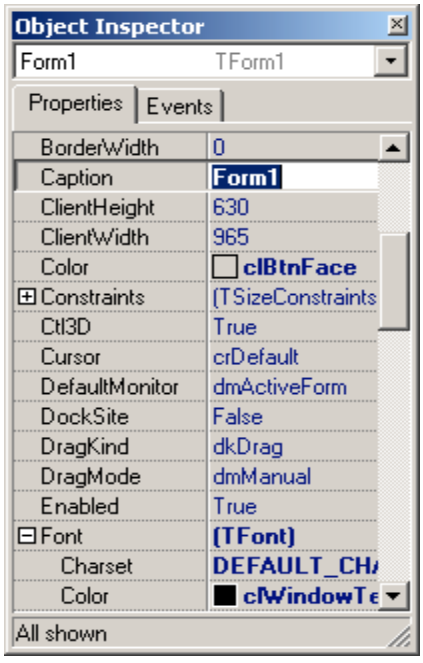
- La palette de composants



- Le concepteur de fiche



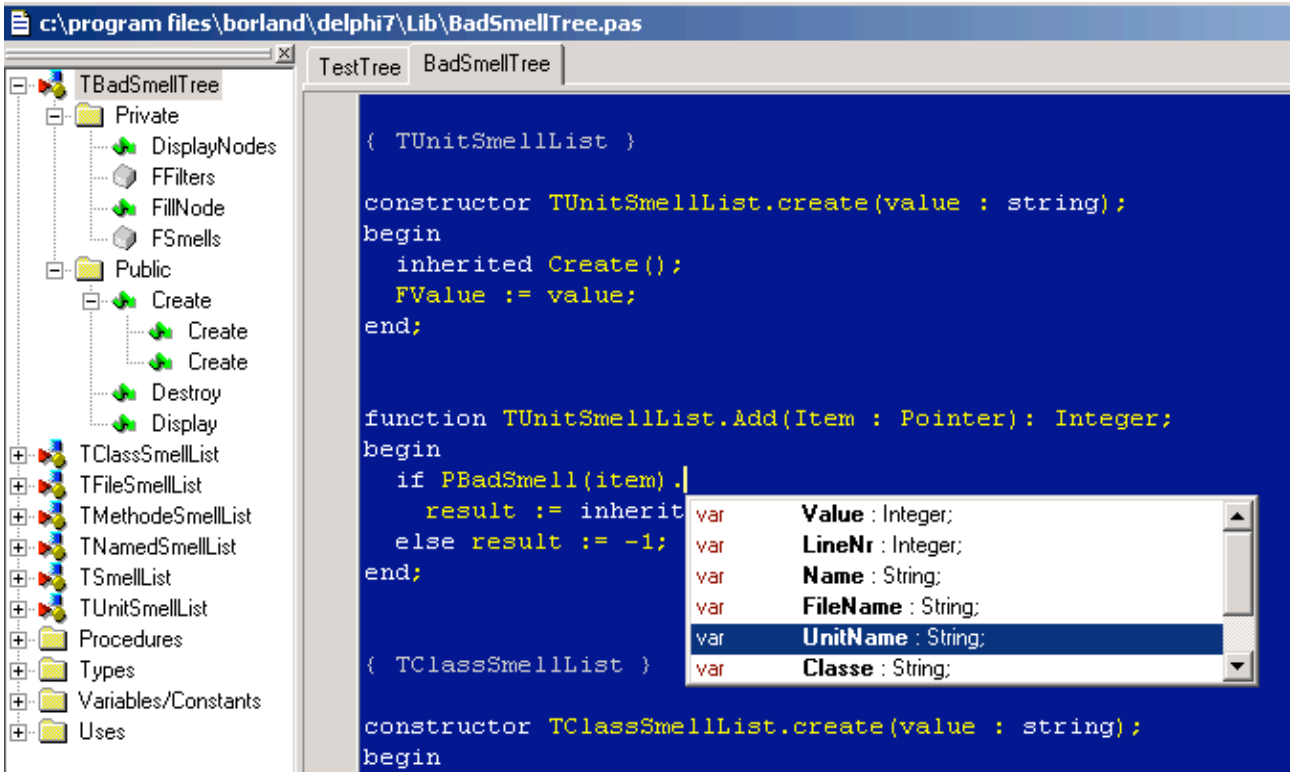
- L'inspecteur d'objet



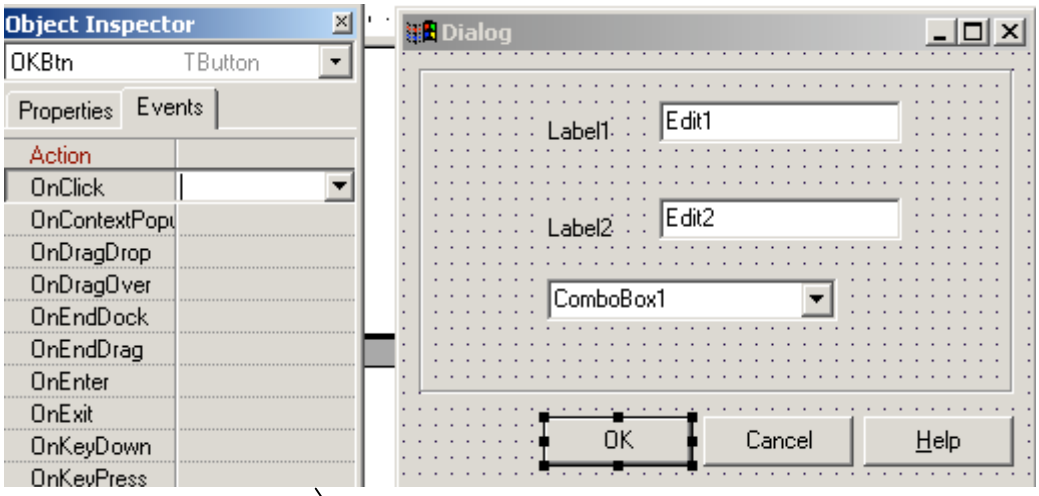
La palette de composants sert à choisir le type de contrôle que l'on veut ajouter à l'interface. Le concepteur de fiche représente la fenêtre que l'on est en train de créer. L'inspecteur d'objet permet d'éditer les propriétés que l'on assigne à chaque contrôle, mais sert aussi à connecter les différents événements que peuvent recevoir ces contrôles avec la méthode à exécuter.

Éditeur de code

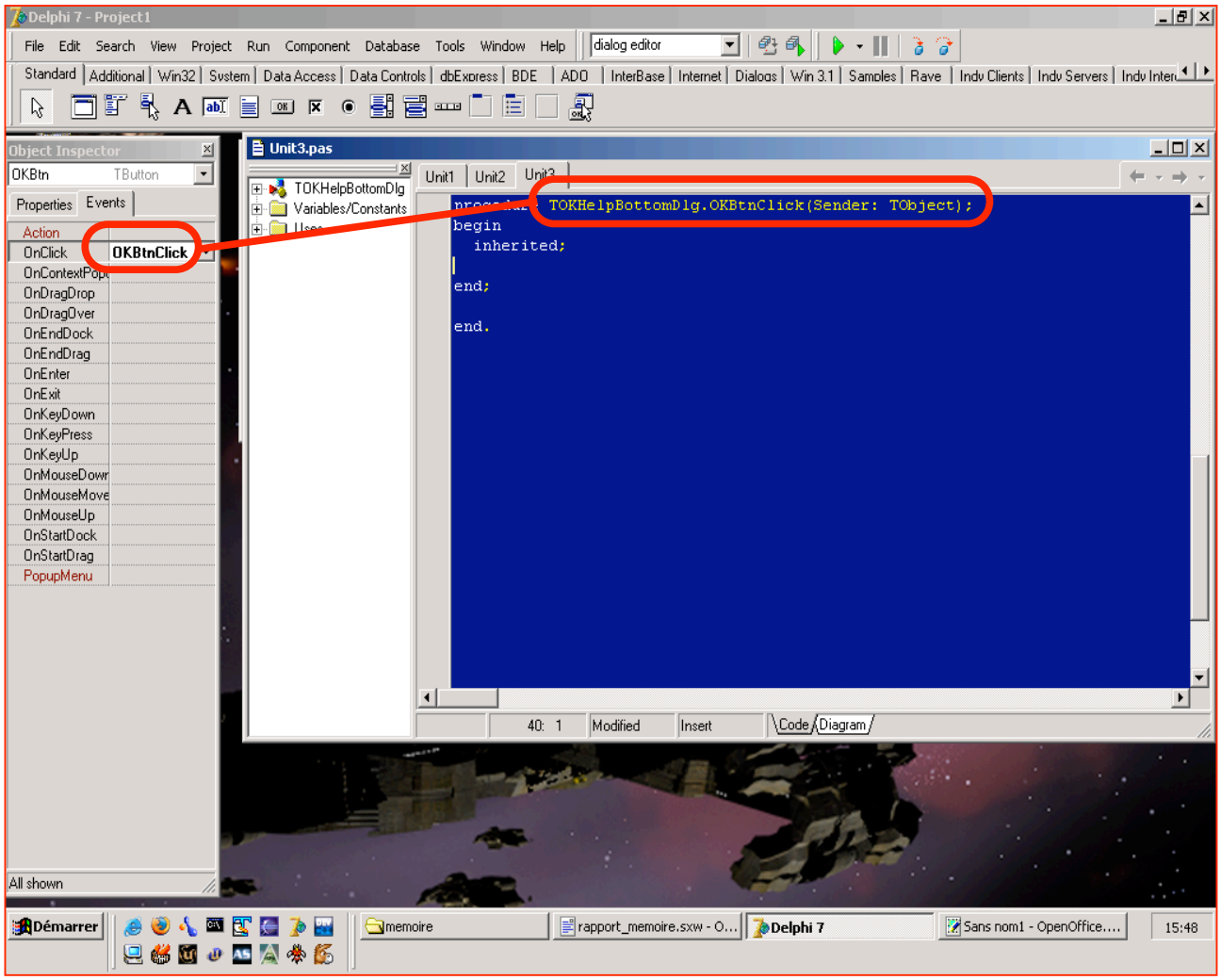
L'éditeur de code Delphi est à la fois assez classique et assez complet. Il supporte (entre autre) la coloration syntaxique automatique du code ou encore l'auto-completion. Il est également possible d'afficher un tas d'outils supplémentaires, comme un explorateur de code.



Une des principales caractéristiques (et avantages) de Delphi est le fait que le concepteur d'interface et l'éditeur de code sont bien intégré ensemble. Par exemple si l'on double-clique sur un évènement dans l' inspecteur d'objet, Delphi génère automatiquement une fonction et vous place dans l'éditeur, de sorte qu'il ne reste plus qu'à entrer le code de votre fonction.



Si je double-clique ici,...



Présentation du langage

Présentation du langage de Delphi : le Pascal Objet

Delphi utilise comme langage de programmation le Pascal Objet.

Le Pascal Objet est un langage qui pourrait être comparé à C++, à savoir qu'il se base sur un langage procédural (le Pascal) et lui rajoute les notions de la programmation orientée objet, comme les classes, la notion d'héritage, de portée des variables (publique, privée ou protégée), ... Il en résulte un langage de programmation hybride, à la fois orienté objet et procédural.

Voici un exemple d'un fichier écrit en Pascal (le Pascal Objet sera vu plus particulièrement plus loin) :

```
program HelloWorld;
procedure Hello(str : String);      // déclaration d'une procédure.
var                                 // déclaration d'une variable locale
   HelloStr : string ;              // Notons que le type String est un type de base en Pascal
begin
   HelloStr := 'Hello ' ;
   writeln(HelloStr + str);  // Affiche Hello + le string passé en paramètre à l'écran
end;
begin                               // point d'entrée du programme
   Hello('World');                  // Affiche à l'écran Hello World
end.
```

Il s'agit d'un simple programme qui se contente d'afficher « Hello World » à l'écran. Dans ce fichier est déclarée une procédure[1] nommée Hello. En ce qui concerne les variables (locales ou non), leur déclaration se fait dans une section spéciale, introduite à l'aide du mot-clef var. Le corps de la méthode se trouve quand à lui entre une paire begin/end ; .

Spécificités du langage

Le Pascal (et forcément le Pascal objet) comporte quelques spécificités qui lui sont propres:

- Il existe 4 types de fichiers sources en Delphi:
    - Les fichiers de type program (comme dans l'exemple ci-dessus). Ce type de fichier contient la déclaration générale d'un programme ainsi que son point d'entrée (équivalent du main () en C)
    - Les fichiers de type library. Ces fichiers sont utilisés pour définir les bibliothèques partagées (DLL sous Windows).
    - Les unités. Ce sont les fichiers classiques dans lesquels sont déclaré les classes du programme.
    - Les fichiers de paquetages. Ce type de fichier est spécifique à l'environnement Delphi et sert uniquement à pouvoir intégrer facilement de nouveaux composants dans l'IDE de Borland. Je n'ai pas tenu compte de ce type de fichier lors de mon travail dans la mesure ou ils sont très petits et ne contiennent pas de code en tant que tel.
- Une autre spécificité est que les fichiers de type unit se composent de différentes parties. Il y a tout d'abord la partie interface, qui contient les déclarations des différentes méthodes, type de données, constantes,… contenues dans ce fichier. (Rôle similaire aux fichiers .h pour le C.) Ensuite il y a la partie implémentation qui contient le corps des méthodes. Enfin il existe également 2 parties optionnelles, à savoir les parties initialisation et finalisation. Ces parties servent à contenir du code qui sera appelé automatiquement au début et à la fin de l'utilisation du fichier.

[1] En Pascal (et donc en Delphi) il existe 2 types de méthodes : les procédures et les fonctions. La différence est que les fonctions ont un type de retour alors que les procédures n'en ont pas.

Le langage pascal OBJET

Comme dit plus haut, le Delphi utilise comme langage un langage dérivé du Pascal en lui ajoutant des notions de programmation orientée objet. Dans cette partie je vais m'attacher à expliquer comment fonctionne et quelles sont les spécificités apportées par cette partie orientée objet du langage.

Tout d'abord le Pascal objet introduit la notion de classe et d'objet. Une classe est considérée comme étant un type de donnée particulier contenant des champs et des méthodes. La création/suppression d'une instance d'une classe se fait via un appel au constructeur / destructeur de la classe. Le fait qu'il y a une notion de destructeur sous-entend qu'en Pascal Objet, il n'y a pas de « garbage collection » comme dans des langages tels que Java ou Smalltalk. Cependant le langage permet également la définition de propriétés, qui sont en quelque sorte des champs virtuels dont il est possible de spécifier une valeur par défaut, la méthode à utiliser pour les accès en lecture et la méthode à utiliser pour les accès en écriture.

La déclaration d'une classe se fait de la façon suivante :

```
Interface          // la déclaration d'une classe se fait généralement dans la partie interface
Type               // indique que l'on s'apprête à définir de nouveau types de données, ici une classe
  MyClass  = class(parent)
      Private          //la portee des membres de la classe fonctionne de façon similaire à Java/C++
        Field1 : int;
        Field2 : string;          //déclaration de 2 champs de la classe
      public
        Constructor create(); //déclaration d'un constructeur
        Destructor destroy(); //déclaration d'un destructeur
        Procedure myProc(a : integer) ;        //déclaration d'une procédure de la classe
        Function myFunc(str : string):string;   //déclaration d'une fonction de la classe
        Property myProp : int read Field1 write myProc;
        //declaration d'une propriété qui est en fait redirigée vers le champ Field1 pour les accès en
        //lecture et vers la méthode myProc pour les accès en écriture
  end ;    // fin de la déclaration de la classe
```

Comme on peut le constater, l'implémentation des méthodes n'est pas présente. En fait elle va se trouver dans la partie implémentation, et ressembler à ceci :

```
Implementation //L'implementation des méthodes se trouve dans la partie implémentation du fichier
Constructor MaClasse.create();
Begin
inherited Create() ;       //appel au constructeur hérité de la classe parente
//…
End;
Destructor MaClasse.destroy();
Begin
//…
inherited destroy() ;      //appel au destructeur hérité de la classe parente (se fait à la fin)
End;
```

```
Procedure MaClasse.myProc(a : integer) ;
Begin
//corps de la méthode
End;
//…
```

Comme on peut le voir, lors de la déclaration d'une classe, il est également possible d'hériter d'une autre classe. Le Pascal Objet ne permet cependant de définir qu'une et une seule classe parente. L'héritage multiple ainsi que l'utilisation d'interfaces n'est donc pas possible. Chaque classe hérite à la fois des champs/méthodes mais aussi des propriétés de sa classe parente. Elle peut alors introduire de nouvelles méthodes, ou bien surcharger celles déjà existantes. Pour se faire il est néanmoins nécessaire d'utiliser certains mots-clefs :

- Virtual/dynamic : Ces 2 mots-clefs ont la même signification. Ils permettent de spécifier le fait que l'appel de la méthode doit se faire dynamiquement. Supposons que j'ai 2 classes A et B. La classe B hérite de la classe A, et surcharge certaines de ses méthodes, dont la méthode Process, laquelle a été déclarée virtual dans la classe A. Le fait que l'appel à cette méthode est dynamique signifie que si je possède un objet (obj) qui a été déclarée comme étant de type A mais qui est en réalité une instance de B (rappelons ici que toute variable déclarée comme étant une instance d'une classe peut contenir une instance de cette clase mais aussi une instance d'une classe dérivée de celle-ci), si jamais je fait appel à la méthode Process à partir de cet objet (obj.Process()) ce sera la méthode Process de la classe B qui sera appelée,et non celle de la classe A.
- Override : Lorsque l'on surcharge une méthode d'une classe parente, il est nécessaire de le signaler au compilateur à l'aide de ce mot-clef.
- Overload : Le mot-clef overload est utilisé lorsque l'on redéfinit une méthode mais en utilisant une signature (entendre par là des paramètres) différents. Notons que la méthode surchargée peut appartenir à la classe parente mais aussi à la classe courante.
- Reintroduce : Lorsqu'une méthode est redéfinie dans une classe dérivée en utilisant ce mot-clef, elle est définie comme si la méthode correspondante dans la classe de base n'existe pas et que la méthode redéfinissante est une nouvelle méthode.
- Inherited : Ce mot-clef est utilisé à l'intérieur de l'implémentation des méthodes pour signifier que l'on accède à une variable/méthode héritée de la classe parente.

Pour déclarer un objet, il suffit de créer une variable dont le type est la classe de l'objet. Par exemple :

```
Var
        A : Maclasse ;
```

Pour créer une instance de Maclasse et l'affecter à A, il faut appeler le constructeur de cette dernière :

```
A := Maclasse.create() ;
```

On peut formuler 2 remarques concernant cet appel au constructeur :

- Il s'agit d'un appel explicite au constructeur. Cela est différent de ce qui se fait dans des langages tels que C++ ou Java, ou l'appel au constructeur se fait via l'opérateur new.
- Le constructeur de la classe est une méthode statique de la classe, ce qui veut dire qu'il n'est pas nécessaire de créer une instance de celle-ci pour appeler le constructeur (sans quoi l'on risquerait d'avoir quelques problèmes …). Cela est certes logique, mais surtout montre qu'il est possible en Delphi de créer des méthodes statiques qui appartiennent et sont appelée sur une classe plutôt que sur une instance de la classe (comme en C++ ou en Java).

Pour libérer la variable, Delphi utilise le principe du destructeur, comme en C++ par exemple si ce n'est que, comme pour le constructeur l'appel à celui-ci est explicite. Cela nous donne :

```
A.Destroy() ;     // Appel explicite qui se fait à partir de l'objet à libérer.
                  // En C++, l'appel est implicite : il faut utiliser l'opérateur Delete.
```

Notons ici que pour le constructeur et le destructeur, j'ai utilisé les noms de méthodes create et destroy. Bien que ces noms soient ceux qui sont le plus souvent employés, il peut être intéressant de

noter que la syntaxe et la sémantique du Pascal Objet permettent d'utiliser n'importe quel nom. En ce qui concerne la visibilité des variables, des méthodes et des propriétés, Delphi permet d'utiliser des déclarations privées, protégées ou publiques (à l'aide des mots-clefs private, protected et public). Leur signification est la même que dans la majorité des langages de programmation orientés objet.

En plus de la notion de classe et d'objet, le Pascal Objet introduit également un mécanisme de gestion d'exceptions. Ce mécanisme permet de capturer les exceptions qui peuvent survenir en encapsulant le code qui la génère entre les mots-clefs try et except ou try et finally. La différence entre les deux est que le code qui se trouve dans une clause finally sera toujours exécuté, alors que le code qui se trouve dans une clause except ne sera exécuté que si une exception est déclenchée.

```
try
…        // code susceptible de générer une exception
except
  on E: EZeroDivide do HandleZeroDivide; //executé si une exception de type EZeroDivide est
                                          //déclenchée (le E: sert à déclarée une variable qui sera
  on E: EOverflow do HandleOverflow;    //initialisée comme étant une instance de l'exception capturée
  on EMathError do HandleMathError;    //Le E: n'est pas obligatoire
else
  HandleAllOthers;    //code par défaut si une exception d'un type non repris au dessus est déclenchée
end ;
```

Pour déclencher une exception, il suffit de créer une instance de la classe exception ou d'une classe dérivée (en Pascal Objet, la classe Exception sert de classe de base pour toutes les exceptions) et de la « lancer » à l'aide du mot-clef raise. Ainsi le code suivant va générer une Exception de type EZeroDivide :

```
if y=0 then
    raise EZeroDivide.create("division par 0") // le raise déclenche une exception à partir de l'instance
else                                           // renvoyée par le constructeur EZeroDivide.create
```

Delphi dans le code

Comme cela a été dit plus haut, Delphi est à la fois un IDE mais aussi contient un concepteur d'interfaces. Cela sous-entend que l'environnement Delphi est capable de lui-même manipuler les sources du projet en cours. Aussi voici un bref aperçu de la façon dont Delphi gère cela.

Tout d'abord, Delphi va générer un fichier .dpr qui va contenir le code du programme. Typiquement il ressemble à cela :

```
program TestBadsmellTree;
uses
  Forms,
  TestTree in 'TestTree.pas' {Form1},
  BadSmllScrptCfg in 'BadSmllScrptCfg.pas' {BadSmllCfg};
{$R *.res}        // directive indiquant au compilateur d'inclure des fichiers de ressource (.res)
begin             // au moment de la création de l'exécutable.
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TBadSmllScrptCfg, BadSmllCfg);
  Application.Run;
end.
```

C'est ce fichier qui va contenir le point d'entrée de l'application. Aussi ce que ce fichier va faire reste limité : il va initialiser le programme, créer les différentes Forms (voir plus loin) puis lancer l'exécution du programme. Pour ce faire on peut voir dans la partie uses qu'il va utiliser les autres fichiers du projet (ceux contenant les déclarations des Form).

Parallèlement au fichier dpr, Delphi va générer pour chaque Form que l'on va créer un fichier .pas. Une Form est en fait une fenêtre qui va s'afficher sur l'écran. Il peut s'agir de la fenêtre principale comme d'une boite de dialogue. La première qui est créée est la form principale et sera affichée au lancement de l'application. Les autres devront être affichées explicitement. Typiquement, un fichier contenant une Form ressemble à ceci :

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}        // Inclue cette fois-ci des fichiers .dfm qui contienne la déclaration des formulaires.

procedure TForm1.Button1Click(Sender: TObject);

begin

Application.MessageBox('Hello World!', 'HelloDlg');

end;

end.
```

Il y a plusieurs choses à observer dans ce fichier. Tout d'abord il ne s'agit plus d'un fichier de type program mais bien d'une unité. Ensuite cette unité déclare essentiellement 2 choses : une classe TForm1 qui dérive de la classe TForm et une variable globale Form1 de type TForm1.

Le type TForm1 est en fait une spécialisation de la classe TForm correspondant à la fenêtre que l'on est en train de créer. Dans cet exemple ma fenêtre est très simple puisqu' elle ne contient qu'un bouton, lequel affiche une boite de dialogue avec écrit Hello World lorsque l'on clique dessus. L'environnement a donc créé 2 choses : 1 champ Button1 contenant mon bouton, et une méthode. Cette méthode est appelée dès que je clique sur le bouton. Son prototype et le fait qu'elle reçoit un paramètre de type TObject est également déterminé automatiquement par l'environnement (Pour être exact, il s'agit en fait de la façon dont a été déclarée la classe TButton qui détermine cela, mais pour des raisons de simplicité considérons qu'il s'agit de l'environnement).

La variable Form1 sera quand à elle utilisée dans le fichier .dpr (vu plus haut) afin de créer la Form au lancement de l'application. Elle peut également être utilisée par d'autres fichiers du projet, lesquels

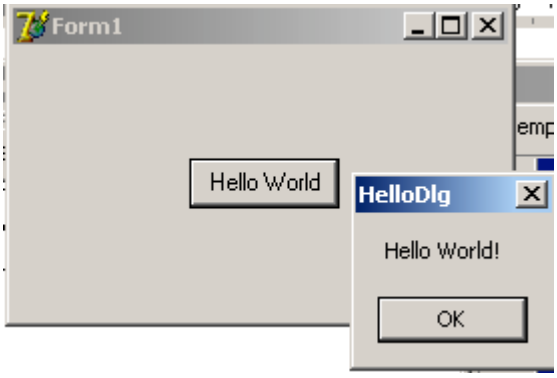
pourrait très bien avoir besoin de cette référence vers la Form afin de pouvoir par exemple en contrôler l'affichage.

Enfin il est une dernière chose importante à remarquer : il s'agit de la ligne {$R *.dfm}. On peut en effet se poser la question de savoir à quoi peut bien servir cette ligne. En fait ce que fait cette ligne, c'est inclure le fichier .dfm contenant la déclaration complète de la forme dans le fichier .pas. Si je reprends mon exemple, l'environnement a déclaré pour ma Form le fichier dfm suivant :

```
object Form1: TForm1
  Left = 192
  Top = 107
  Width = 423
  Height = 240
  Caption = 'HelloWorld'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
  object Button1: TButton
    Left = 96
    Top = 64
    Width = 75
    Height = 25
    Caption = 'Hello World'
    TabOrder = 0
    OnClick = Button1Click
  end
end
```

Dans ce fichier apparaît un tas d'informations. Parmi ces informations il y a la position de ma fenêtre sur l'écran, sa taille ainsi que d'autres propriétés. Mais il y a aussi le fait que ma form contient un bouton. Les différentes propriétés de mon bouton se retrouvent aussi dans ce fichier. On peut même y voir figurer le fait que la fonction à appeler lorsque je clique sur le bouton est la fonction Button1Click. (Dernière ligne).

Autrement dit, ce fichier contient toutes les informations utiles concernant les différentes forms de mon projet. Cela sous-entend qu'il est inutile de rechercher un endroit dans le code ou sont initialisé les différentes options de mes forms ainsi que les méthodes à appeler et à relier aux différents évènements qui peuvent survenir pendant l'exécution du programme, puisque cette initialisation a lieu dans ce fichier.



Résultat lors de l'exécution du programme

b) Présentation de RainCode

Qu'est ce que RainCode?

Pour pouvoir rechercher la présence de bads smells dans du code Delphi, il faut être capable d'analyser ce code. Pour ce faire, plusieurs approches sont possibles. La première consiste en la réécriture d'un parser Delphi, capable d'extraire les informations utiles depuis le code Delphi puis écrire un programme de détection de bads smells sur base des informations extraites. Cette solution n'a pas été retenue car elle présente l'inconvénient de demander une grande quantité de travail, lequel ne présente qu'un faible intérêt dans la mesure où de tel parseurs existent depuis un certain temps.

Une autre solution aurait pu être d'utiliser le parseur intégré à l'environnement de Borland. Le problème est que Borland ne fournit aucune aide et aucune documentation quand à la façon d'utiliser le parseur intégré de Delphi. Aussi il semble que pour pouvoir avoir accès à ces informations, il faille devenir un partenaire privilégié de Borland[7] (ce qui bien entendu n'est pas gratuit).

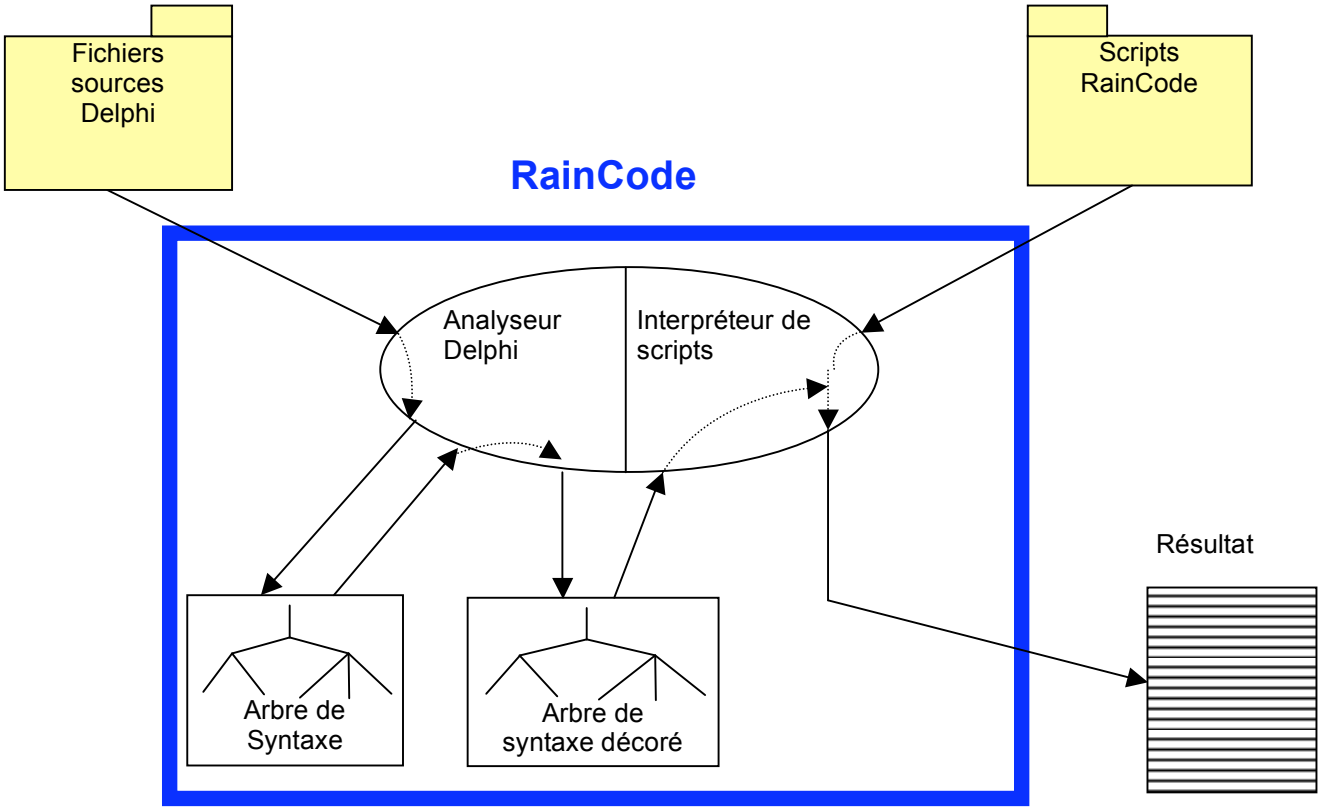
Dès lors il ne reste plus qu'une solution : se tourner vers un outil qui aurait été développé par une société tierce. C'est le cas de RainCode. Le RainCode Engine est un outil commercial développé par la société RainCode[8]. Cependant, l'université a pu obtenir une licence d'utilisation valable au sein du service informatique, dans le cadre du présent mémoire. Le RainCode Engine est capable d'analyser le code source d'un programme et de générer l'arbre de syntaxe correspondant afin de pouvoir en mesurer divers aspects. Mais RainCode[1] va plus loin que cela. Il permet d'exécuter des scripts sur l'arbre qu'il a généré, autorisant par exemple de créer en sortie un fichier au format XML. Il permet également de le manipuler le code source même en créant des patches.

Cet outil existe en plusieurs versions dont une est prévue pour fonctionner avec Delphi (et le Pascal Objet), laquelle sera donc utilisée.

[1] A partir de maintenant, lorsque j'emploierai le terme RainCode, je parlerai du programme RainCode Engine et non de la société RainCode (sauf indication contraire).

Principe de fonctionnement

Voici un diagramme qui explique le principe de fonctionnement de RainCode:



Tout d'abord RainCode recevra en entrée les fichiers sources Delphi que l'ont veut analyser. A partir de ces fichiers, il va générer l'arbre syntaxique correspondant. Ensuite, il va décorer l'arbre afin d'associer à chaque noeud de ce dernier certaines valeurs correspondantes. Ces valeurs pourront ensuite être utilisées par les scripts qui seront exécutés. Une fois l'arbre généré et décoré, RainCode va appeler les scripts que l'on a créé et les exécuter sur cet arbre. En ce qui concerne le résultat en sortie, RainCode ne va rien créer de façon automatique. C'est à l'utilisateur de RainCode que revient la tâche de créer quelque chose en sortie à l'aide de ces scripts (un fichier par exemple).

Programmation de scripts en RainCode

Principe :

La programmation en RainCode se fait via l'utilisation d'un langage de scripting qui lui est propre[9]. Ce langage sert à manipuler l'arbre de syntaxe qui est représenté sous forme d'objets représentant les noeuds. La programmation s'y fait de façon procédurale, mais aussi de façon linéaire. J'entend par là que les scripts reçoivent l'arbre à considérer, s'exécute une fois dessus puis renvoient le résultat. Notons qu'il est possible de créer ses propres fonctions, mais pas ses propres objets.

Types de données :

RainCode définit différents types de données de base que voici :

- Les entiers (signés, codés sur 32 bits)
- Les réels (codés sur 64 bits)
- Les booléens (TRUE ou FALSE)
- Les chaînes de caractère

En sus de ces types de base, RainCode défini également des types de données plus avancés :

- Les noeuds non terminaux. Il s'agit en fait de chacun des noeuds dans l'arbre de syntaxe. Chaque nœud est en fait un objet, et possède un certain nombre d'attributs. Certains d'entre eux (les attributs) sont communs à tous les nœuds (comme l'attribut ClassName qui contient le nom de la classe à laquelle appartient le nœud), tandis que d'autres sont typiques de certains types de nœuds. Ces derniers représentent généralement une partie des sous nœuds de niveau 1 du nœud courant. Exemple : Dans un nœud de type *WhileStatement* (boucle while … do …), on retrouve les attributs *Condition* et *Statement*, lesquels représentent respectivement l'expression à évaluer pour savoir si la boucle doit être exécutée, et l'ensemble des instructions appartenant à cette boucle.

- Les listes. Une liste est donc un ensemble d'objet de n'importe quel type (y compris des listes), possédant une relation d'ordre. Les listes sont couramment utilisées dans RainCode pour représenter un ensemble de nœuds (par exemple dans l'exemple ci-dessus la propriété *Statement* du *WhileStatement* peut dans certains cas n'être qu'une liste de nœuds de type *Statement*). De même certaines structures de contrôle en RainCode ont été spécialement pensées pour être utilisée avec des listes (voir ci-dessous).

Il faut également notez que le système de typage est dynamique, ce qui sous-entend qu'aucun type n'est affecté à une variable lors de sa déclaration mais que c'est le moteur de scripts qui se charge de le déterminer pendant l'exécution du script.

Structures de contrôle :

Il existe essentiellement 2 types de structures de contrôle : les branchements conditionnels et les boucles.

- Les branchements conditionnels : En RainCode seul l'instruction IF existe, éventuellement avec une clause reprise dans un ELSE. Il est également possible d'utiliser l'instruction ELSIF lorsque différents IF sont imbriqués.
- Les boucles : Les structures de boucle en RainCode sont tout à fait particulières. Elles prennent en paramètre une liste est servent à appliquer les instructions qu'elles contiennent sur chacun des éléments de la liste.

Exemple :

Le programme suivant permet d'afficher le nom de la classe de chaque nœud situé dans l'arbre.

```
PROCEDURE INIT ; --Comme dit plus haut, INIT est une des fonctions à surcharger et qui sert de
BEGIN                    -- point d'entrée à RainCode
--ROOT est une variable prédéfini de RainCode qui représente la racine de l'arbre de syntaxe.
FOR IN ROOT.SubNodes DO          --X est une variable déclarée implicitement dans la boucle.
        OUT.WriteLn(X.ClassName) ;  -- Elle représente le nœud courant et prend successivement la
        END ;                    -- valeur de chaque élément de ROOT.SubNodes.
END ;                            -- Il est possible d'utiliser un nom particulier en le spécifiant
                                 --entre le FOR et le IN : FOR myVar IN ROOT.SubNodes DO
```

- Les filtres : Les filtres, comme les boucles prennent une liste comme argument. Ils se contentent de renvoyer une liste qui contient tous les éléments de la liste de départ qui satisfont à une expression donnée. Ils sont souvent utilisés pour sélectionner les nœuds à traiter. Par exemple l'expression suivante permet de récupérer tous les nœuds de l'arbre de type WhileStatement.

```
--…
FOR IN ROOT.SubNodes | X IS WhileStatement DO
        --…traitement
```

Les quantifieurs : Les quantifieurs prennent également une liste en paramètre, ainsi que des instructions à exécuter sur chacun des éléments de la liste. Ces instructions doivent au final renvoyer TRUE ou FALSE. Le quantifieur en lui-même renverra soit TRUE soit FALSE si un des éléments renvoie TRUE, ou si tous les éléments renvoient TRUE, ou encore le nombre d'éléments qui renvoient TRUE, en fonction du quantifieur utilisé.

```
--…
IF THERE_IS IN ROOT.SubNodes :- X IS WhileStatement
        --…traitement
```

RMQ : Les annotations :

Une autre facilité de RainCode est le fait que l'on peut annoter à tout moment une variable (quelque soit son type). Cela est très utile lorsque l'on veut par exemple retenir certaines caractéristiques d'un nœud de l'arbre.

Par exemple, la boucle suivante me permet de construire une liste contenant tous les fichiers utilisés dans mon fichier courant et d'affecter cette liste à une propriété que j'ai nommée « unit » de la racine de mon arbre.

```
-- . . .
FOR IN ROOT.InterfaceSection.UsesClause.Units DO
        ROOT["unit"] = ROOT["unit"] & {X.Data};
        END; -- Le & sert à concaténer ROOT["unit"] avec une liste contenant 1 seul élément : X.Data.
            -- X.Data contient successivement le nom de chaque unité référencée dans la clause
            -- USE du fichier courant.
-- . . .
```

Les annotations sont quelque chose de très utilisé en RainCode. D'ailleurs ce dernier possède un type de données nommé table associative qui représente une variable vide, à savoir qu'elle ne contient rien, si ce n'est qu'elle peut être annoté. J'expliquerai plus loin à quoi peut servir un tel type de données (Chapitre V, B).

c) Détection de bads smells dans Delphi à l'aide de RainCode

Liste des bads smells à détecter

Comme cela a été dit dans le chapitre concernant les bads smells, la liste de Fowler n'est pas un absolu en soi. Aussi j'ai rajouté un certain nombre de bads smells à la liste de Fowler.

Les métriques du logiciel

On pourrait également placer dans cette catégorie toutes les mesures existantes issues des études portant sur les métriques du logiciel. Cependant il a fallu se limiter à quelques unes d'entre elles. Voici donc les métriques qui ont été utilisées:

- La complexité cyclomatique (Thomas McCabe 1976):

La complexité cyclomatique d'une fonction peut être définie comme étant le nombre de chemins d'exécution indépendants que peut emprunter cette fonction.

Elle peut être calculée à partir du graphe de flux de commande à l'aide de la formule suivante:

$C = N_{arc} – N_{noeud} +2$.

Ce bad smell a été ajouté en raison de sa pertinence, à savoir qu'il permet en général de se forger une bonne idée quand à la qualité d'une méthode. En effet, s'il existe un grand nombre de chemins d'exécution indépendants, cela sous-entend que la fonction contient un grand nombre de conditions et de boucles, et peut dès lors être considérée comme plus complexe.

- Cohésion dans une classe

Si l'on regarde la liste de M. Fowler, on constate que certains bads smells qu'il propose se ramène à la métrique de couplage (la façon de calculer le couplage sera précisée plus loin). Par contre, aucune ne se ramène à celle de cohésion, laquelle lui est généralement associée. J'ai donc rajouté un bad smell nommé « Cohésion insuffisante » faisant référence à la métrique de cohésion au sein d'une classe.

Nomenclature adaptée

Bien que Fowler n'aie pas repris le moindre smell concernant le nom qu'il faut attribuer aux différentes variables et/ou méthodes que l'on crée, il n'en demeure pas moins qu'il insiste tout au long de son livre sur l'utilité d'utiliser des noms le plus communicatif possible, ce afin de faciliter la compréhension du code. Si cette idée est tout à fait juste, elle reste néanmoins insuffisante quand il s'agit de trouver quels sont les noms à employer (ou dans le cadre des bads smells à ne pas employer) dans le code source. C'est pourquoi je vais introduire ici quelques smells afin d'aider le programmeur dans cette tâche:

- respect d'une norme de nomenclature:

En effet à partir du moment où l'on travaille à plusieurs, il est bon d'avoir une norme pour la nomenclature à respecter. Un exemple est de considérer que les noms de variables commencent par une minuscule, contrairement à ceux des classes, ou encore qu'il est nécessaire d'utiliser un préfixe pour les noms de classe. L'intérêt ici n'est pas ce que dit la norme en soi, mais bien que tous les programmeurs l'emploient. De cette manière la lecture du code d'un programmeur par un autre sera plus aisée et plus rapide.

- taille des noms employés:

Il va de soi qu'il est difficile de trouver un nom explicite si celui-ci ne fait que 2 à 3 lettres (sauf quelques cas particuliers), de même un nom qui fait plus de 15-20 lettres de long peut être rébarbatif (surtout s'il ne répond à aucune norme qui expliquerait cette longueur). Aussi il ne faut pas rallonger le nom des variables et des méthodes inutilement.

- type incorporé dans le nom[10]:

Une (mauvaise) habitude lorsque l'on crée une classe chargée de stocker un certains nombres d'éléments d'un même type est d'utiliser comme nom de fonction un nom du style

```
Liste.AddSquare (x: Square);
```

Or comme on peut le voir, la fonction AddSquare contient le type de l'objet à stocker, à savoir Square. Le problème est que si jamais l'on décide de remplacer un jour la classe Square par la classe Circle ou d'utiliser cette classe pour stocker des objets d'un autre type, alors le nom AddSquare n'a plus de sens.

Smells typiques à Delphi

Ici, il s'agit de trouver des smells pouvant s'appliquer spécifiquement à Delphi et au langage Pascal objet. Pour ce faire il faut partir des spécificités de ce langage (citée plus haut).

- « Définitions inappropriées »:

Parmi celle-ci, on trouve le fait qu'il existe différents types de fichiers (library, unit, ...). Cela m'amène logiquement à proposer le smell suivant que je nommerai « Définitions inappropriées », à savoir que les classes ne devraient être déclarées que dans les fichiers de type unit. Les fichiers program doivent se contenter de contenir la fonction de départ (point d'entrée) du programme, ainsi qu'éventuellement certaines fonctions utilisées par celle-ci (pour éviter qu'elle ne soit trop importante). Cela sous-entend que si certaines fonctions sont définies dans ce fichier, elles doivent à la fois y être utilisées et n'être utilisée nulle par ailleurs.

Ce smell est à classer dans la catégorie « Non respect des règles de programmation », dans la mesure où il n'est que la conséquence logique de la philosophie issue du Pascal (et dès lors du Pascal Objet).

- Définition interne à une méthode :

Il est aussi un autre smell applicable au langage Delphi qui ne l'est pas pour tous les langages orientés objet. Il s'agit du smell de définition interne à une méthode. En effet Delphi (et le Pascal objet) autorise le fait de définir un type ou une méthode à l'intérieur d'une autre méthode. Cela peut être pratique si l'on désire employer ce type de données dans cette méthode uniquement. Cependant, cela tend à rendre le code plus difficile à lire dans la mesure où il est plus difficile de retrouver où la méthode où le type en question a été défini. De même cela tend à limiter les possibilités de réutilisation du code dans la mesure où ceux-ci (les types et les méthodes définies de façon imbriquées) ne sont accessibles que dans la méthode où ils ont été définis.

- Utilisation du « with »

Le with est un mot-clef en Pacal objet qui permet dans une partie du code de travailler exclusivement sur un objet. Typiquement cela donne des choses du genre :

```
With myForm do
        Posx := 100 ;
        Posy := 200 ;
        Sizex := 320 ;
        Sizey := 240 ;
        Show() ;
        End ;
```

Ces lignes de code auront le même effet que si j'avais fait :

```
myForm.Posx := 100 ;
myForm.Posy := 200 ;
myForm.Sizex := 320 ;
myForm.Sizey := 240 ;
myForm.Show() ;
```

Le fait d'utiliser cet opérateur est considéré comme un bad smell dans la mesure ou si l'on fait référence par exemple à des variables locales de la méthode courante dans un with, il peut devenir difficile de s'y retrouver et de savoir si l'on fait appel à des champs ou à des variables locales. (Pour se convaincre, supposer simplement que dans l'exemple ci-dessus, la méthode contenant le with contienne une variable nommée posx, ou encore mieux que la classe à laquelle appartient cette méthode contienne une méthode show. Qui plus est cela rend le code plus difficile à lire pour quelqu'un devant y incorporer des modifications, qui par exemple ne remarquerait pas au premier coup d'œil la présence de ce with.

Tableau des bads smells ajoutés

	Liste des Bads Smells	Commentaires
Métriques du logiciel	Complexité cyclomatique	Métrique du logiciel introduite par Thomas McCabbe.
	Cohésion dans une classe	Métrique de cohésion, généralement associée avec celle du couplage.
Non respect des règles de programmation (typique à Delphi)	Définitions inapropriées	Le fait de déclarés certaines parties du code (classes par exemple) à certains endroits, bien qu'autorisé par le compilateur est contraire
	Définition interne à une méthode	Ce type de déclaration a tendance à rendre le code plus difficilement lisible, et nuit à la réutilisation de celui-ci.
	Utilisation du with	Tout comme le bad smell précédant, rend le code difficilement lisible. Certains développeurs pensent que son utilisation devrait être tout simplement proscrite.
Nomenclature adaptée	Respect d'une norme	Le fait que tous les développeurs utilisent une même norme permet de faciliter le travail en groupe, chacun comprenant plus facilement le code des autres.
	Taille des noms employés	Utiliser des noms trop longs rend plus difficile la lecture du code.
	Type incorporé dans le nom	Cela rend le nom d'une méthode dépendante de ses paramètres sans pour autant apporter une information supplémentaire (car elle se trouve déjà dans le type du paramètre).

« Détectabilité » des différents bads smells

La question à laquelle il faut trouver une réponse désormais est la suivante:

« Peut-on automatiser la détection de tous les bads smells, et si non lesquelles ne peut-on pas trouver et pourquoi? »

Pour répondre à cette question, je vais traiter le cas de chaque catégorie de smells (définies plus haut) séparément.

Métriques du logiciel

Comme dit plus haut, tous les bads smells présents dans cette catégorie sont issus ou peuvent se ramener à une (éventuellement plusieurs) métriques du logiciel. Or ces dernières étant issues de mesures effectuées dans le code source d'un programme, leur implémentation dans un programme et

l'automatisation de la détection des bads smells de cette catégorie est quelque chose de tout à fait possible. Tout ce qu'il est nécessaire de connaître, c'est tout d'abord une définition précise de la (des) métrique(s) à employer, et ensuite un algorithme capable de la (les) calculer. Aussi pour bon nombre de métriques, cela reste assez trivial (ex : nombre de paramètres, de méthodes par classe, …). Cela dit, pour d'autre cela n'est pas le cas. Je vais donc ici détailler les méthodes employées pour pouvoir calculer la complexité cyclomatique ainsi que les métriques de couplage et de cohésion.

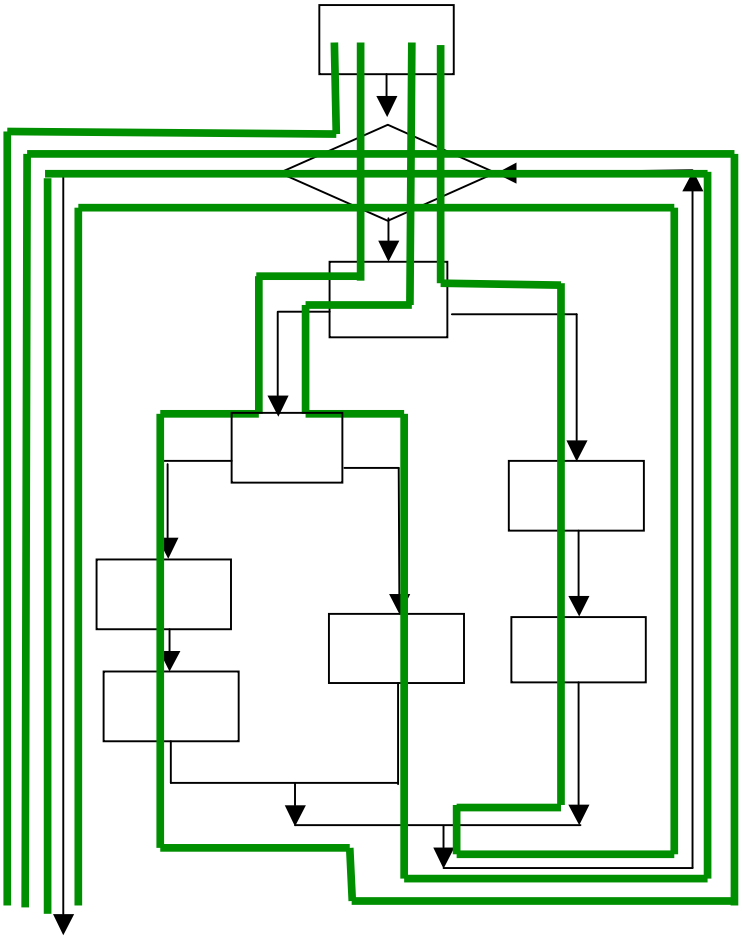
- ***La complexité cyclomatique***

Pour calculer la complexité cyclomatique de McCabe, je suis parti de la définition suivante: « La complexité cyclomatique d'une méthode est égale au nombre de chemins indépendants que peut emprunter cette fonction. »

J'ai ensuite appliqué cette définition au graphe de flux de contrôle pour voir ce qui pouvait influencer ce nombre. Il m'est alors apparu que la complexité cyclomatique d'une fonction dépendait uniquement des instructions de branchements et des instructions conditionnelles telles que les if, les case of, ou encore des instructions telles que les for et les while. Notons également que le mécanisme d'exceptions (try, except,...) doit également être pris en compte.

Exemple:

Voici un diagramme de flux d'un programme :



La complexité cyclomatique de cette fonction vaut 4. En effet il existe 4 chemins indépendants permettant d'exécuter cette fonction.

Si l'on regarde ce graphe, on constate qu'en fait les chemins se séparent lorsque l'on rencontre un losange. On peut supposer dans le cas présent que le premier (celui du haut) correspond à while, tandis que les 2 autres correspondent chacun à un if dans la méthode.

A

En fait, la complexité peut-être calculée de façon récursive, de la manière suivante:

- Lorsque l'on est sur un if: le if divise le flux en 2. Donc pour calculer la complexité cyclomatique, il faut additionner la complexité du flux de droite avec celui du flux de gauche.
- Lorsque l'on est sur un case of: le principe ici est le même que dans le cas du if, si ce n'est que le

case of divise le flux en un nombre variable de sous flux. Il faut donc cette fois additionner les complexités de chacun de ses sous flux.

- Lorsque l'on est sur un for/while/repeat: ces 3 instructions ont en fait le même effet sur la complexité. Au niveau de l'exécution, elles font toutes boucler le programme sur un bloc d'instructions. Au niveau de la complexité, celle-ci vaut la complexité du bloc en question +1.

- Lorsque l'on est sur un try: on peut considérer et ramener le cas du try à celui du case of, dans la mesure où il faut additionner la complexité de chacune des sous parties du try, même si concrètement la syntaxe du Pascal objet rend la gestion des try un peu plus compliquée que celle des case of.

- Sinon: Cela signifie que l'on est sur un autre type d'instruction. Dès lors le programme s'exécute de façon linéaire et dès lors la complexité cyclomatique vaut 1. (1 seule exécution possible).

Sur l'exemple ci-dessus:

On a tout d'abord une boucle, donc la complexité vaut $1+C_{boucle.}$

C_{boucle}: On a 2 instructions conditionnelles imbriquées, donc la complexité vaut

$C_{gauche1} + C_{droite1}$ avec $C_{gauche1} = C_{gauche2} + C_{droite2.}$

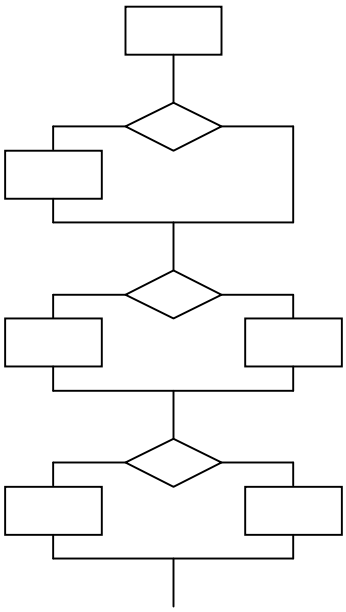
Or $C_{gauche2,}$ $C_{droite2}$ et $C_{droite1}$ ne comprenne aucune boucle et aucune condition. Leur exécution est linéaire et leur complexité vaut 1.

Au total: $C_{total} = 1+C_{boucle} = 1+ C_{gauche1} + C_{droite1} = 1+ C_{gauche2} + C_{droite2} + C_{droite1} = 1 + 1 + 1 + 1 = 4$

Rmq: cas où les instructions if, case,... ne sont pas imbriquées:

Il peut arriver dans une fonction que l'on aie plusieurs instructions de branchements qui se suivent, mais ne sont pas imbriquées. Dans ce cas la complexité totale vaut le produit des complexités de chacune des instructions.

Exemple:



Ce graphe représente un programme composé de 3 instructions conditionnelles non imbriquées. Chacune des branches de chacune de ces instructions a une exécution linéaire, don un complexité cyclomatique de 1. Cela signifie que chaque instruction conditionnelle a une complexité de 2.

La complexité totale vaut le produit de la complexité de chacune de ces instructions soit 2X2X2=8.

En effet, à chaque condition, je peux aller à droite où à gauche, ce choix se représentant 3 fois il y a 2^3 donc 8 exécutions possibles.

Rmq:

Dans certains cas il peut arriver qu'une instruction contienne plusieurs sous-instructions non-imbriquées. La complexité intermédiaire est obtenue en multipliant les complexités de chaque sous-instructions, comme pour le résultat final.

Couplage et cohésion

Les métriques de couplage et cohésion font généralement référence au fait que des entités distinctes (classes différentes) doivent avoir le moins d'interactions possibles entre elles, tandis que des entités proches (des méthodes dans une même classe) peuvent avoir un plus grand nombre d'interactions. Cependant si nous voulons calculer ces valeurs de manière précise, il faut alors employer une méthode de calcul. Aussi, pour calculer ces métriques, j'ai procédé de la façon suivante :

- Pour le couplage : La métrique de couplage est calculée à 2 niveaux différents : au niveau des classes et des méthodes. Pour ce faire je compte les interactions qu'il peut y avoir entre les différentes entités, à savoir que pour les classes, la valeur du couplage d'une classe est égal au

nombre de classes qui interagissent avec la classe courante (parce qu'elles utilisent un champ qui est un objet de cette classe ou qu'elles font appel à l'une ou l'autre méthode de cette classe). Pour les méthodes, le couplage est calculé entre les différentes méthodes d'une même classe. La valeur est calculée de façon similaire à celle des classes, puisqu'elle est égale au nombre de méthodes qui utilisent la méthode courante.

- Pour la cohésion : La cohésion est quand à elle calculée uniquement à l'intérieur d'une classe. Elle l'est comme étant la somme du nombre de champs de la classe accédés par chaque méthode de la classe divisé par le nombre total de champs. Cette façon de calculer la cohésion sert à assurer q'un minimum de travail est réalisé sur chaque champs de la classe. Il existe cependant d'autres méthodes pour calculer la cohésion, notamment certaines qui introduisent des notions comme la similarité entre les méthodes de la classe [11]. L'élément à retenir de telles définitions est le fait que le calcul de la cohésion doit se baser sur les accès aux différents champs de la classe.

Non respect des règles de programmation

En ce qui concerne la détectabilité de ces bads smells, celle-ci dépend beaucoup du smell particulier à considérer.

- Pour ce qui est des problèmes de variables/champs inutilisés, il suffit de regarder pour chaque variable si celle-ci est utilisée quelque part dans l'ensemble des méthodes ou elle est définie. Notons ici qu'il faut également tenir compte pour les champs par exemple qu'il peut exister une variable locale ou un paramètre masquant ce champ au sein de la méthode. De même, pour les variables globales il faut aussi tenir compte du fait qu'une telle variable n'est pas utilisée dans un fichier mais y est déclarée afin que tous les autres fichiers du programme incluant ce fichier puisse faire référence à cette variable et l'utiliser. (Ce cas est typique en Delphi, via le mécanisme utilisé pour créer les Forms, voir chapitre précédant).

- Pour le bad smell de champ temporaire, la méthode aura consisté à regarder pour chaque champ d'une classe si ce dernier était parfois accédé en lecture dans une des méthodes de la classe, ou si elle ne l'était qu'en écriture. (En effet si une variable est accédée en écriture avant de l'être en lecture dans une méthode, cela veut dire que la valeur qu'elle contenait n'est pas utilisée, la variable est donc utilisée comme variable temporaire).

- Pour le bad smell « envie de caractéristique », il suffit de regarder si la méthode considérée accède en majorité aux champs et méthodes de sa classe ou si elle accède en majorité aux champs et méthodes d'une autre classe, et si oui de quelle classe. Malheureusement, la façon dont les scripts ont été écrits ne permet pas de faire une telle analyse, essentiellement parce que chaque fichier est analysé séparément des autres. Une amélioration permettant d'y remédier est néanmoins proposée plus loin.

- Pour le bad smell utilisation du switch (rebaptisé utilisation du case of pour Delphi), il a été considéré que si la variable sur laquelle porte le case n'est pas un paramètre ou une variable locale, c'est qu'il y a quelque chose qui « cloche » et donc ce bad smell est détecté à ce moment là.

- Par contre pour l'utilisation du mot-clef « with », la détection est faite de façon triviale puisque l'on considère que l'on a un bad smell à chaque fois que ce mot-clef est rencontré.

- Le bad smell « classe de donnée » quand à lui est détecté à chaque fois que l'on est en présence d'une classe dont les méthodes ne servent qu'à accéder à ces différents champs (autrement dit servent de « getter/setter »), et que dès lors celle-ci n'effectue aucun travail concret. Une méthode est considérée comme un getter ou un setter sur base de ces paramètres, de sa valeur de retour et des différents accès aux champs de la classe qu'elle effectue.

- Pour le bad smell refus d'héritage, la technique aura consisté en l'examen de la classe de base. (rmq : pour pouvoir détecter ce bad smell, il faut que la déclaration de la classe de base fasse partie du projet). A chaque fois que celle-ci contient une méthode abstraite, il faut vérifier que la classe dérivée redéfinisse bien cette méthode. De même à chaque fois que la classe dérivée redéfinit une méthode, une vérification est faite pour être sûr que la méthode de la classe de base est bien dynamique ou virtuelle.

- Pour le bad smell « homme du milieu », j'ai utilisé la notion de méthode relais. En fait une méthode relais est une méthode qui n'effectue aucun travail par elle-même mais délègue tous à une autre méthode. Je parle de méthode relais car une telle méthode sert de relais entre sa méthode appelante

et sa méthode appelée. Une méthode est considérée comme servant de relais dans le cas où elle accède à une et une seule autre méthode. Une classe est considérée comme étant une instance du bad smell homme du milieu à partir du moment ou la majorité de ses méthodes sont des méthodes relais.

- Enfin et pour terminer les smells de type non respect des règles de programmation, il me reste à évoquer le cas des « data clumps ». Pour détecter ces groupes de données, je me suis limité à comparer les différentes classes entre elles 2 à 2. Pour chaque groupe de 2 classes, je compte le nombre de classes faisant références à ces 2 classes. Si cette valeur atteint un certain nombre, alors le bad smell est détecté. Il faut noter que cette approche a un gros défaut : elle ne me permet que de détecter des groupes de données regroupant 2 classes. S'il fallait comparer toutes les classes entre elles, cela reviendrai à rechercher la présence de chaque sous-ensemble composés d'au moins 2 classes, soit un nombre de sous-ensembles de l'ordre de 2^k, k étant le nombre de classes. Prenons un projet contenant une centaine de classes, il faudrait alors évaluer la présence de $2^{100} \approx 10^{30}$ classes! En annexe est présenté une approche qui devrait si elle était implémentée permettre d'avoir de meilleurs résultats. Cette approche se base sur l'algorithme Apriori utilisé essentiellement dans le Data Mining.

Nomenclature adaptée

- respect d'une norme de nomenclature: En Delphi, l'environnement et les librairies elles-même utilisent certaines normes. Parmi celles-ci, j'ai retenu le fait que chaque type de données (y compris les classes) commence par la lettre T (pour type). De même les champs des classes commencent par la lettre F (field). Evidemment il s'agit d'une implémentation simple voir simpliste du problème, mais si l'on veut vraiment pouvoir spécifier une ou plusieurs normes, alors l'implémentation devient très difficile.
- taille des noms employés: Il suffit de tester la longueur des chaînes de caractère désignant les classes, les méthodes et les variables du projet.
- type incorporé dans le nom : pour détecter ce bad smell, il faut tout d'abord repérer les signatures de toutes les méthodes du projet. Ensuite il est nécessaire, pour chaque paramètre de la méthode de vérifier que la chaîne de caractère désignant son type n'est pas reprise dans le nom de la méthode.

Autres

Les smells présents dans la catégorie autres ne sont pas détectables avec un outil tel que celui qui a été développé, et ce pour différentes raisons (en fonction du smell). Dans certains cas cela est lié au fait qu'il est nécessaire d'analyser plusieurs versions différentes du code d'un programme (changements divergents), dans d'autres cela est lié au fait que le bad smell ne se situe pas à l'intérieur du code source (librairie incomplète).

Principe de détection des bads smells à l'aide de RainCode

Principe général

L'utilisation de RainCode revient en fait à créer des scripts et à les appeler de sorte qu'ils soient appliqués aux sources que l'on veut examiner. Comme dit plus haut, RainCode va automatiquement générer l'arbre de syntaxe à partir des sources. Ce qu'il faut faire dès lors consiste donc à parcourir cet arbre et à trouver dans cet arbre les endroits où se trouvent des structures correspondant à un bad smell. Pour pouvoir détecter les bads smells à partir de l'arbre de syntaxe, j'ai procédé en 2 phases[1] :

- Durant la première phase les scripts parcourent une et une seule fois l'arbre complet. Pendant ce parcours certains nœuds de l'arbre seront annotés/décorés de sorte à retenir certaines informations les concernant.
- Durant la seconde phase les bads smells seront détectés. Pour ce faire, les scripts de détection utilisent les informations récoltées au préalable.

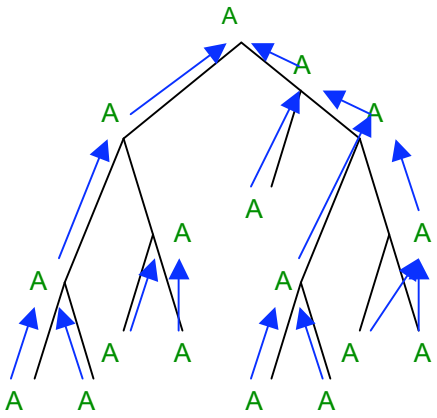
[1] Eva Van Emden et Leon Moonen [3] utilisent comme méthode de détection une approche similaire. Il y est fait référence à des aspects (équivalent à l'annotation des nœuds de l'arbre) à partir desquels les bads smells sont inféres (équivalent à la phase d'analyse).

L'intérêt d'utiliser des scripts séparés pour la détection proprement dite et pour la recherche d'information est double : tout d'abord il ne faut pas à chaque fois réécrire du code afin de naviguer dans l'arbre à la recherche d'une information, ensuite comme l'arbre en lui-même n'est parcouru par les scripts qu'une seule fois, les performances sont meilleures.

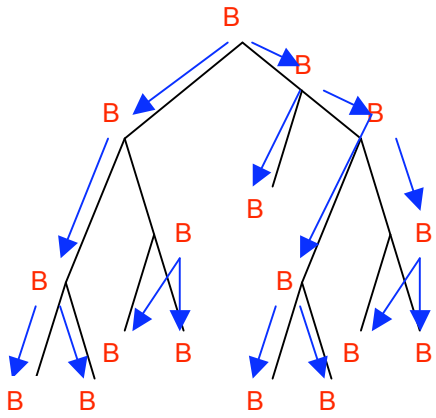
- ***Phase 1 : Recherches d'informations utiles dans l'arbre.***

Le parcours de l'arbre se fait depuis la racine en descendant dans toutes ses sous-branches. La recherche d'informations et l'annotation de différents nœuds ressemble fort à ce que fait un compilateur lors de l'analyse sémantique du code. En effet le but de cette pré-analyse est de retrouver un certain nombre de caractéristiques typiques pour chaque type de nœuds dans l'arbre. Pour un compilateur, ce genre d'informations peut être le type d'une variable, la valeur d'une constante ou d'une expression. Dans le cadre de la recherche de bads smells, ces caractéristiques seront plutôt du style « Quelles variables sont accédées par cette fonction? A quelle classe appartient-elle? Quelles sont les variables globales appartenant à un fichier ? … »[1]

L'analyse et l'attribution des différentes caractéristiques peuvent se faire de manière différente en fonction de la caractéristique proprement dite et des informations dont on a besoin pour la calculer. A ce niveau l'on peut distinguer 2 groupes d'attributs : les attributs hérités et les attributs synthétisés. La différence se situe essentiellement aux nœuds qu'il est nécessaire d'avoir déjà visités pour pouvoir déterminer la valeur de l'attribut en question. Ainsi pour les attributs synthétisés, il est nécessaire d'avoir parcouru tous les sous-nœuds d'un nœud particulier pour pouvoir déterminer la valeur. Pour un attribut hérité par contre, sa valeur est calculée à partir des nœuds parents et « frères » du nœud courant, pas de ses fils. En ce qui concerne le travail de recherche de caractéristiques dans l'arbre, la grande majorité de ces dernières étaient de type synthétisé.



Attribut Synthétisé



Attribut Hérité*

* Dans ce schéma, seules sont représentées les dépendances vis-à-vis des nœuds parents. Mais il peut arriver qu'il y ait aussi des dépendances vis-à-vis des nœuds frères.

Pour les attributs synthétisés, la façon générale de procéder peut être représentée comme suit :

```
PROCEDURE analyse_node(node) ;
BEGIN
FOR IN node.SubNodes DO
        analyse_subnodes(X);
        node["attr_synth_1"] = Process(node["attr_synth_1"], X["attr_synth_1"]);
        -- On peut ici constater l'intéret qu'il y a à utiliser le mécanisme d'annotation de RainCode
        END;
END ;
```

[1] Une liste complète des attributs ainsi que la manière de les calculer, le type de nœud auxquels ils s'appliquent et le type d'information que l'on y trouve est donné en annexe.

Evidemment il s'agit ici d'un exemple simpliste qui est donné, car en réalité la propriété SubNodes n'est pas utilisée (j'utilise d'autres propriétés permettant une analyse plus fine), mais le principe est là. Chaque attribut synthétisé est ainsi calculé à partir des sous-nœuds du nœud courant, et éventuellement de la valeur déjà stockée (pour éviter les doublons par exemple). Il peut également arriver dans certains cas que la valeur de plus d'un attribut soit pris en compte (voir le cas des accès en lecture/écriture décris dans les annexes). On peut également noter que dans le cas de certains attributs, ceux-ci ont directement été affectés à un nœud dans l'arbre sans passer par les nœuds intermédiaires, cela dans le but de limiter le nombre d'attributs à faire remonter à chaque nœud dans l'arbre. Cela n'est cependant pas possible pour tous les attributs, car d'autres demandent de réaliser des calculs à chaque nœud intermédiaire.

Pour les attributs hérités (beaucoup moins nombreux), 2 cas de figures peuvent se présenter :

- Soit l'attribut en question dépend uniquement du père (ou de l'un des ancêtres) du nœud courant. Dans ce cas la fonction servant à analyser le père se chargera d'affecter la propriété en question au fils.
- Soit l'attribut dépend également de ses frères. Ce cas ne s'est présenté qu'une seule fois sur un type de nœuds particulier et a pu être résolu par l'utilisation d'un paramètre indiquant le contexte dans lequel le nœud courant été évalué. Le type de nœud pour lequel ce problème s'est présenté est le type « designator », lequel correspond au nœud représentant un identificateur, un appel de fonction, une variable, …

- ***Phase 2 : détection des bads smells***

La détection des bads smells en elle-même se fait sur base des informations collectées dans la première partie. Pour ce faire les scripts procèdent de la façon suivante : tout d'abord je me place sur un nœud « particulier ». Les nœuds « particuliers » sont en fait les nœuds de type méthode, classe, fichier et la racine principale[1] . Une fois sur l'un de ces nœuds, chaque fonction RainCode permettant de détecter la présence d'un bad smell s'appliquant sur ce nœud sera appelée. Pour pouvoir vérifier la présence d'un bad smell, la fonction appelée n'a pas besoin d'autres informations que le nœud en question et les différentes propriétés de ce nœud qui ont été calculées à l'étape précédente.

Exemple : Je veux vérifier qu'une méthode utilise bien tous ces paramètres. Pour cela, il me suffit de me placer sur le nœud de la méthode en question. Ensuite la fonction de détection du bad smell « paramètre inutilisé » sera appelée. Elle se chargera de vérifier la présence de ce bad smell en regardant pour chaque paramètre de la fonction (les paramètres ont été calculés et son annotés au nœud de la fonction) si la méthode accède à une variable portant ce nom (les variables accédées aussi ont été calculées et son disponibles sous forme d'annotation concernant le nœud de la méthode). Il en résulte le code suivant en ce qui concerne la méthode de détection de ce bad smell :

```
PROCEDURE TestUnusedMethodeParam(methode); --Voici la fonction que j'utilise à peine simplifiée
BEGIN
FOR IN methode["params"] DO   --Je teste tous les paramètres
  IF NOT IsUsed(X, methode["used_var"]) THEN   --Je vérifie ici que le paramètre a bien été utilisée
    RegisterSmell("Useless param", X.Data, X);   --Le paramètre n'est pas utilisé
    END;                                          -- → j'enregistre le bad smell
  END;
END TestUnusedMethodeParam;
```

[1] RainCode crée au lancement des scripts un nœud principal qui est en fait une liste contenant les racines des arbres de tous les fichiers parsés.

Tableau de détecttion

Le tableau suivant reprend les différents bads smells évoqués, indique s'ils sont détectés complètement, partiellement ou pas du tout et indique en commentaire la méthode utilisé ou encore la raison pour laquelle un bad smell n'est pas détecté.

Bad smell	**Détecté**	**Commentaires**
Code Dupliqué	Oui	Réalisé par un autre étudiant [6]
Méthode longue	Oui	Détecté à partir de plusieurs paramètres, comme la taille de cette méthode ou le nombre de ses variables locales.
Large classe	Oui	Détecté à partir de différents paramètres. Outre le nombre de champs, on peut compter par exemple le nombre de méthodes qu'elle comprend.
Longue liste de paramètres	Oui	Comptage du nombre de paramètres de chaque méthode.
Changements divergents	Partiellement	Se base sur une mesure du nombre de classes utilisées par la classe courante. Idéalement, il faudrait se baser sur une analyse de différentes versions du code.
Shotgun Surgery	Partiellement	Même principe que pour le smells précédant. La détection se base sur une mesure du couplage.
Envie de caractéristique	Non	Nécessite une analyse de type plus poussée
Data clumps	Partiellement	Oui pour les groupes regroupant 2 classes. L'utilisation d'un algorithme tel que l'algorithme Apriori peut apporter une solution.
Obsession de primitives	Non	Trop complexe.
Utilisation de switch	Partiellement	Oui dans le cas ou le switch est effectué sur un champ.
Hiérarchie d'héritage parallèle	Non	Trop complexe. Peut nécessiter d'avoir différentes versions du code.
Classes fainéantes	Oui	Se base sur la complexité moyenne des méthodes de la classe et sur le nombre de champs de celle-ci.
Généralité spéculative	Partiellement	Détection des variables, champs, paramètres inutilisés.
Champ temporaire	Oui	Vérification des accès en lecture pour chaque champ.
Chaîne de messages	Non	Nécessite une analyse un peu plus fine du code en certains endroits.
Homme du milieu	Oui	Détection des méthodes servant de « relais ».
Intimité inappropriée	Partiellement	Dans certains cas, entre une classe et sa classe parente.
Classes alternatives avec interface différentes	Non	Impossible à partir du code.
Librairie incomplète	Non	Impossible à partir du code.
Classe de données	Oui	Détection des classes ne contenant que des méthodes faisant office de getter/setter.
Refus d'héritage	Oui	Vérification de l'utilisation des mots-clefs ad hoc dans le cadre de l'héritage.
Commentaires	Non	Impossible de vérifier la pertinence d'un commentaire à partir du code.
Complexité Cyclomatique	Oui	Mesure de la complexité pour chaque fonction.
Cohésion insuffisante	Oui	Mesure de la cohésion pour chaque fonction.

		Aurait pu se baser sur un calcul de la similarité entre méthodes (plus complexe)
Nom trop long	Oui	Mesure de la longueur des identifiants employés. Possibilité de spécifier des tailles différentes pour les variables, méthodes, classes,…
Respect d'une norme de nomenclature	Partiellement	Détection de certaines normes utilisées en Delphi. Non paramétrable.
Type embarqué dans le nom	Non	Nécessite une résolution de type plus poussée lors de la phase de décoration.
Utilisation du with	Oui	Détecté dès qu'un with est rencontré.
Définition récursive de type	Oui	Vérifie la déclaration de méthodes imbriquées et celle des types propres à une méthode.
Définition inappropriée	Partiellement	Vérifie qu'aucune classe n'est déclarée ailleurs que dans les Unit.

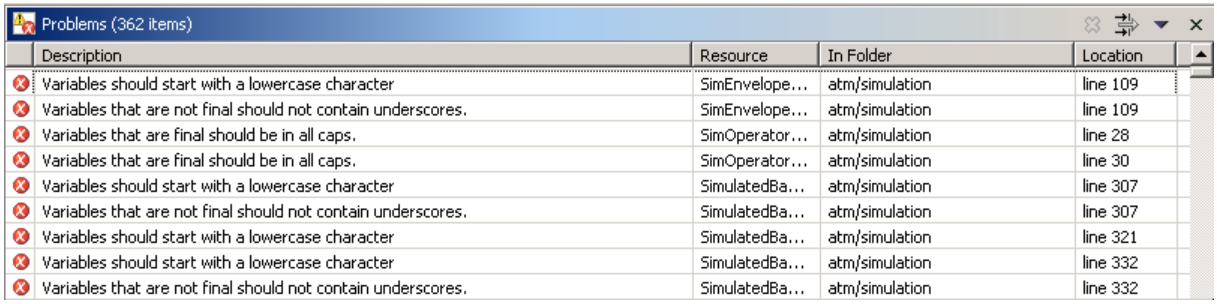
IV. Creation d'un outil de détection

a) *Outils déjà existant*

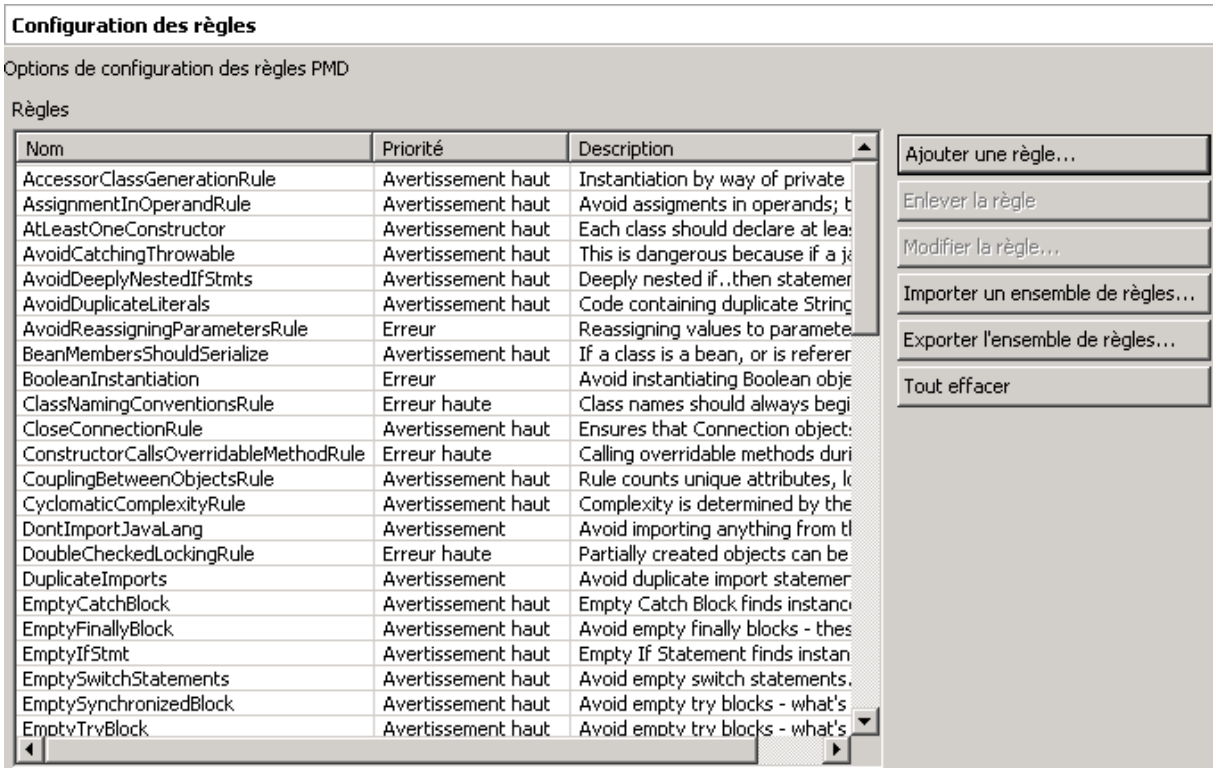
Comme je l'ai dit dans l'introduction, il existe déjà bon nombre d'outil similaires pour d'autres langages de programmation tels que java[5] ou Smalltalk. Aussi peut-il être intéressant de se pencher sur ces outils et sur leur interface avant de développer sa propre interface. En ce qui concerne Delphi, je n'est cependant trouvé aucun outil d'analyse de code.

PMD pour Java

PMD est un outil pour JAVA qui recherche dans le code source des problèmes tels que des variables inutilisées, des instructions switch vides, des fonctions trop complexes,... Cet outil est redistribué sous forme de plugins. Il en existe différentes versions pour différents IDE (Eclipse, JEdit, JBuilder, ...). En ce qui concerne la version pour Eclipse (qui est celle que je connais), ce plugin s'intègre parfaitement dans l'IDE. Il suffit de lui demander d'analyser le code à l'aide de PMD, et les problèmes que ce dernier détecte viennent s'ajouter aux messages que renvoie le compilateur java.

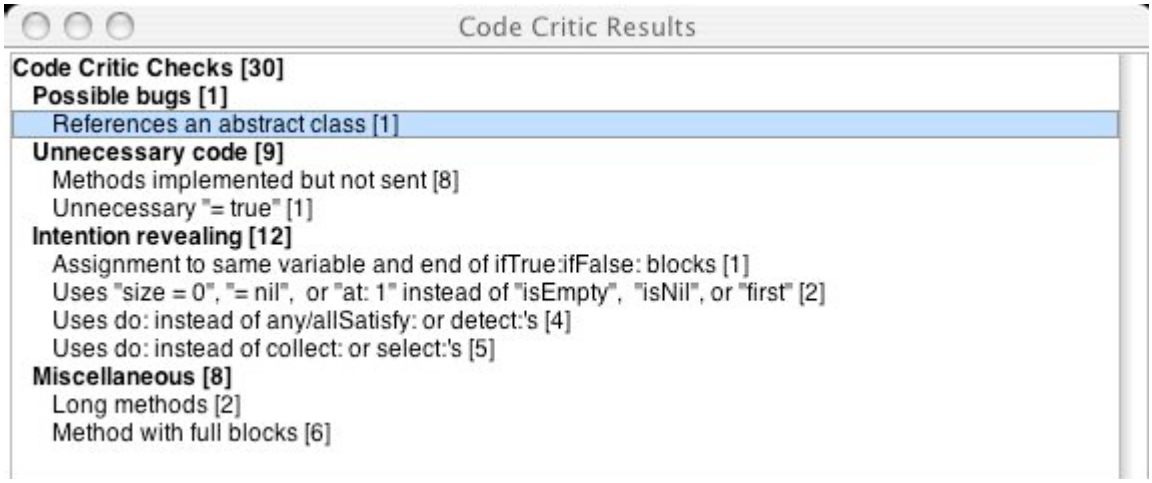


Il possède également une boite de dialogue qui permet de configurer la détection de chaque smell, voir éventuellement de déterminer de nouveaux smells à détecter.



Code critics pour SmallTalk

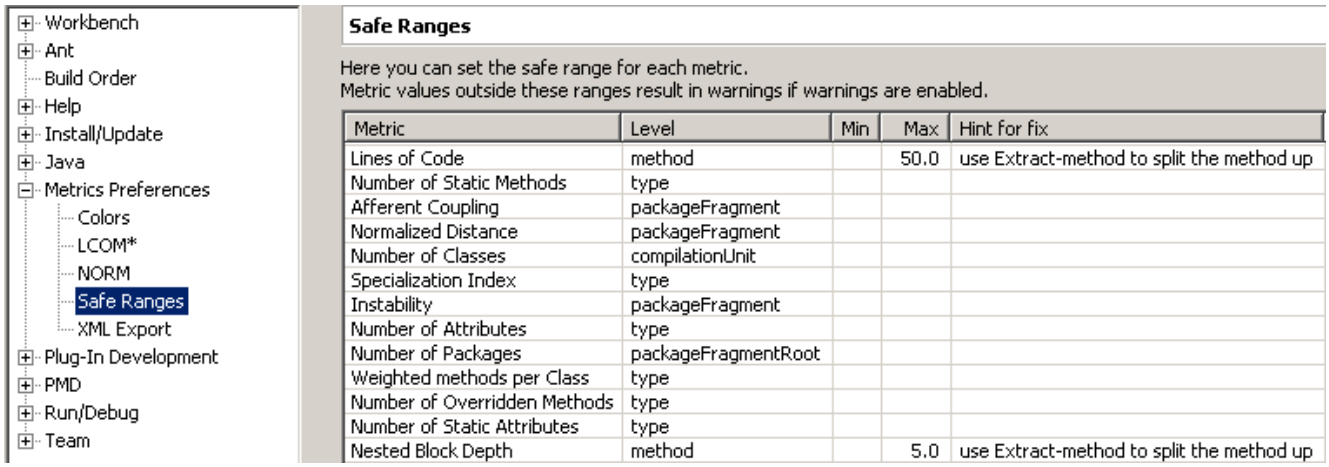
Code critics est un outil dévellopé ayant comme objet la détection des bads smells. Il est dévelloppé pour le langage SmallTalk. Une de ces principales caractéristiques par rapport à PMD est qu'il possède un browser un peu plus avancé pour pouvoir afficher les bads smells:



Comme on le voit sur l'image ci-dessus, les smells sont affichés par groupe, rendant la lecture de ceux-ci plus claire.

Metrics pour Java

Le plugin metrics est un plugin créé pour l'environnement de développement Eclipse pour Java. Ce plugin porte tout son intérêt sur les métriques du logiciel et en calcule un certain nombre. Comme PMD, il offre la possibilité de configurer certaines valeurs limites pour chaque métrique. Cependant, ce plugin ne permet pas de définir ses propres métriques.

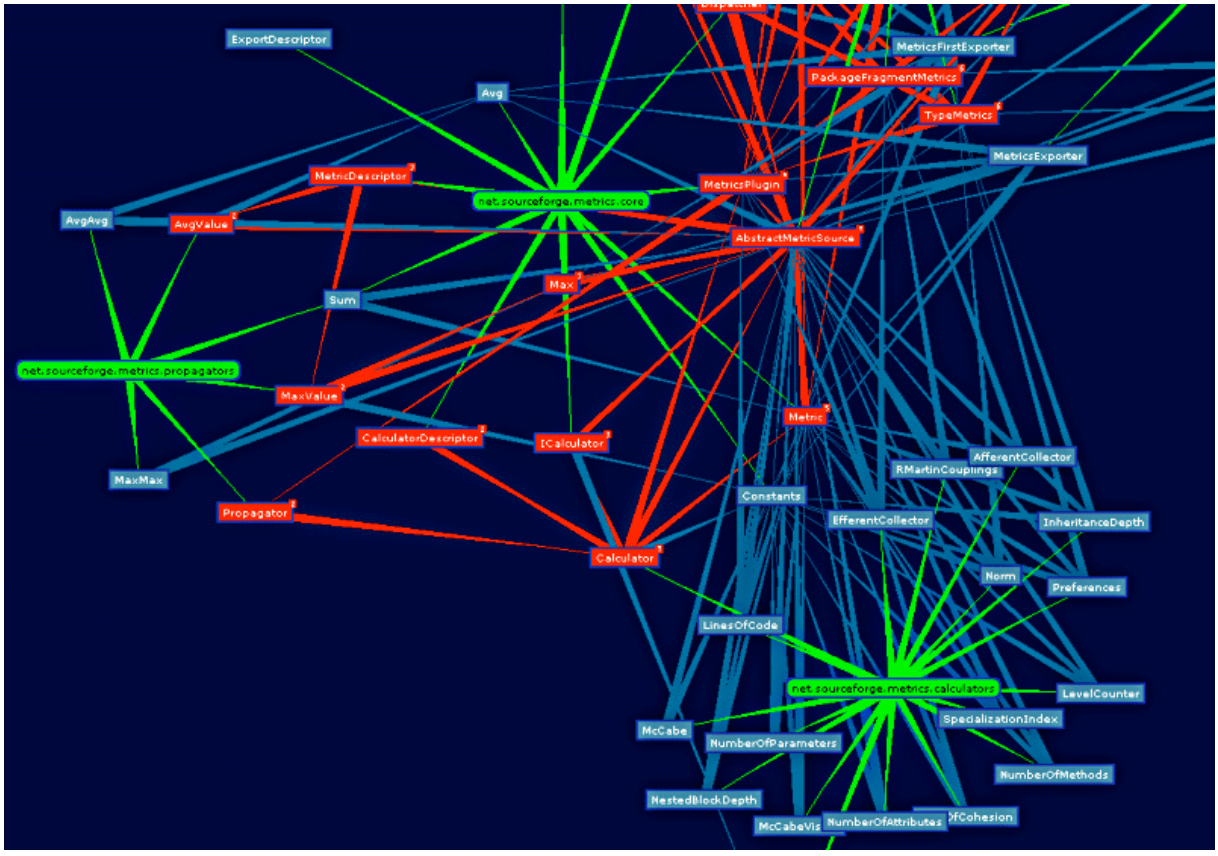


Pour la représentation des résultats, metrics propose 2 modes de visualisations. Le premier affiche les résultats sous forme de messages parmi ceux du compilateur lorsqu'une métrique trop importante est détectée. Les résultats y sont classés un peu comme avec code critics en fonction de leur métrique, mais aussi du package affecté.

Metrics - /net.sourceforge.metrics

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
Number of Packages	16					
Number of Methods (avg/max per type)	1310	6.65	8.553	76	/net.sourceforge.metrics/tgsrc/com/touchgrap...	
tgsrc	489	7.191	11.544	76	/net.sourceforge.metrics/tgsrc/com/touchgrap...	
src	761	6.238	6.553	45	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.core.sources	108	15.429	12.129	45	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.ui	77	9.625	10.111	33	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.core	198	6.6	7.093	27	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.ui.preferences	52	6.5	7.467	26	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.ui.dependencies	95	5.588	3.727	15	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.persistence	18	4.5	4.33	12	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.prevayler.implementa...	54	5.4	2.871	10	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.xml	41	4.1	2.022	9	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.calculators	79	4.158	2.254	8	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.propagators	31	5.167	1.067	7	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.tests	8	2.667	1.886	4	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.prevayler	0	0	0			
classycle	60	8.571	2.556	13	/net.sourceforge.metrics/classycle/classycle/g...	
Lines of Code (avg/max per type)	6593	33.467	49.02	339	/net.sourceforge.metrics/tgsrc/com/touchgrap...	
Number of Interfaces (avg/max per packageFragment)	16	1	1.414	4	/net.sourceforge.metrics/src/net/sourceforge/...	
Lines of Code (avg/max per method)	6593	4.812	7.355	69	/net.sourceforge.metrics/classycle/classycle/g...	calculateAttributes
classycle	324	5.4	9.94	69	/net.sourceforge.metrics/classycle/classycle/g...	calculateAttributes
tgsrc	2321	4.661	8.278	59	/net.sourceforge.metrics/tgsrc/com/touchgrap...	scrollSelectPanel
src	3948	4.862	6.473	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
net.sourceforge.metrics.ui	544	6.8	8.707	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
MetricsTable.java	194	10.778	13.831	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
MetricsTable	194	10.778	13.831	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
setMetrics	52					

Le second mode de visualisation consiste en un graphe des dépendances du système. La vue générée à partir des métriques peut être retournée, zoomé, … Il est également possible d'utiliser des couleurs pour représenter différentes parties du graphe, de n'afficher qu'un sous graphe du graphe complet, …



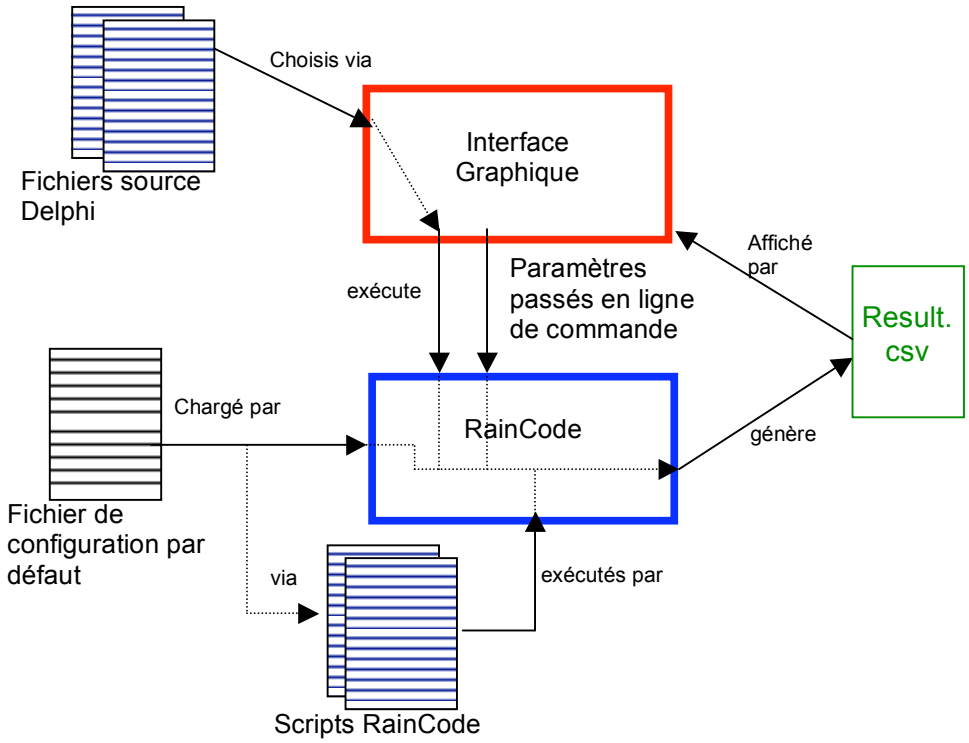
b) Présentation de l'outil développé

Sur base de ce qui se trouve juste au-dessus, l'outil développé comporte essentiellement deux parties :

- Il consiste en une phase de détection des bads smells. Cette dernière repose uniquement sur RainCode et l'utilisation de scripts. Les scripts se chargent de retrouver les bads smells présents dans le code source et vont enregistrer le résultat dans un fichier csv.

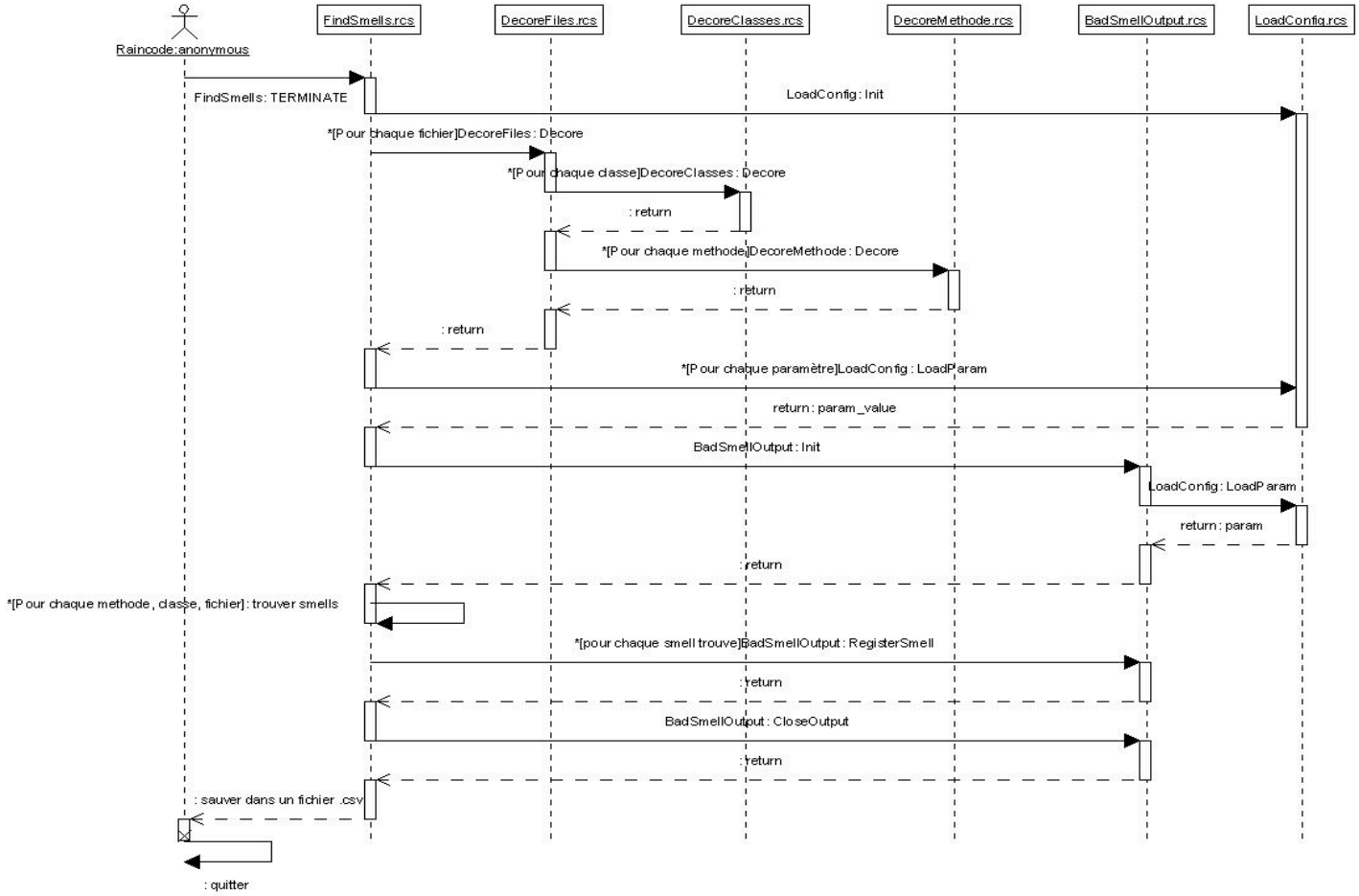
- L'outil possède également sa propre interface graphique, chargée de lancer les scripts RainCode sur les fichiers sources renseignés par l'utilisateur. Ensuite l'interface propose différentes vues à l'utilisateur lui présentant les smells détectés. En ce qui concerne la façon de représenter les bads smells, il a été décidé d'utiliser une interface proche de celle de code critics, plutôt qu'un graphe comme pour le plugin metrics.

Schématiquement:



c) Implémentation des scripts RainCode

Le but de cette section est de décrire comment ont été structurés les scripts en RainCode utilisés pour la phase de détection des bads smells. Ceux-ci sont organisés en différents fichiers, certains faisant appel à d'autres. Voici un schéma représentant les interactions entre eux:



Le fonctionnement est le suivant: le script appelé est le script Findsmells.rcs. La première chose qu'il fait est d'envoyer les paramètres qu'il a reçu au fichier LoadConfig, lequel va se charger de traiter et de stocker les différents paramètres reçu. Eventuellement, si un fichier de configuration est présent le script LoadConfig en tiendra compte pour les valeurs non spécifiée en ligne de commande. Une fois cela fait, la main repasse au script FindSmells. Ce dernier va alors lancer la première phase qui consiste en l'analyse du code et l'annotation de certains noeuds. Pour ce faire, il va appeler le script DecoreDelphiFiles sur chaque fichier source traité. Ce dernier va alors commencer à analyser le code. Le fichier DecoreDelphiFiles fait appel aux scripts DecoreClasses et DecoreMethodes pour les parties correspondantes du code. Une fois que ces différents scripts ont finis d'analyser leurs parties de code respective, cela signifie que la phase de décoration est terminée. La main revient au fichier FindSmells.rcs et ce dernier commence alors la phase de détection des bads smells. Il prend lui même cette phase complètement en charge en procédant de la façon suivante:

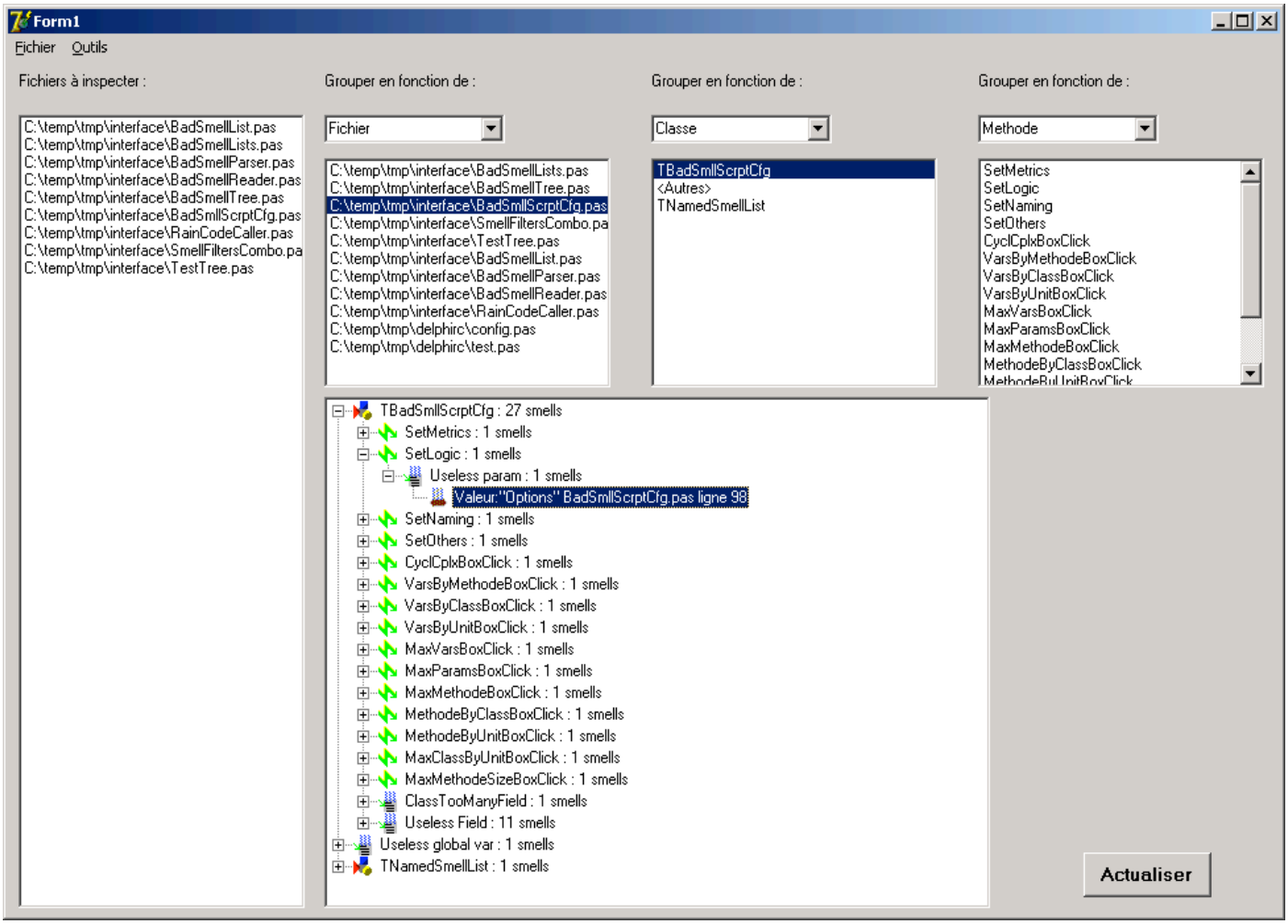
- Il se place sur un noeud "intéressant" du code (lequel a été décoré).
- Il recherche la présence des bads smells détectables à partir de ce noeud.

A chaque fois qu'un bad smell est détecté, le fichier FindSmells va appeler le fichier Bd_Smell_Output, lequel va se charger d'enregister celui-ci. Une fois la détection des bads smells terminée, le fichier FindSmells va en informer le fichier Bd_Smell_Output, lequel va alors écrire tous les bads smells détectés dans un fichier au format csv.

d) Implémentation de l'interface

Présentation générale

L' interface a été développé en utilisant l'environnement Delphi lui-même. Elle se compose essentiellement de composants Delphi (réutilisables par n'importe qui utilise cet IDE) qui ont été dévellopés puis placé sur une Form. Il ne constitue pas un plugin pour l'environnement de Borland comme cela aurait pu être idéalement le cas (le fait d'en faire un plugin ne se justifiant pas dans le cadre de ce travail, le but n'étant pas de créer un outil professionnel). Il n'est en fait qu'un outil indépendant à utiliser en collaboration avec Deplhi. Le resultat final est le suivant:



L'interface développée comporte plusieurs parties. On trouve tout d'abord à gauche la liste des fichiers qui sont analysés. Ensuite en haut de la fenêtre se trouve différentes listes qui sont en fait des regroupement des bads smells détectés. Comme sur l'image par exemple, la première liste affiche l'ensemble des smells détectés triés par fichiers. La seconde trie les bads smells par classe, tandis que la troisième les trie par méthode. On peut noter que le tri à effectuer pour chacune de ces listes est configurable via la combobox qui se situe juste au dessus. De même il est possible de trier les bads smells en fonction du smell en lui-même, ou de ne pas les regrouper et d'afficher les smells sous forme d'une simple liste.

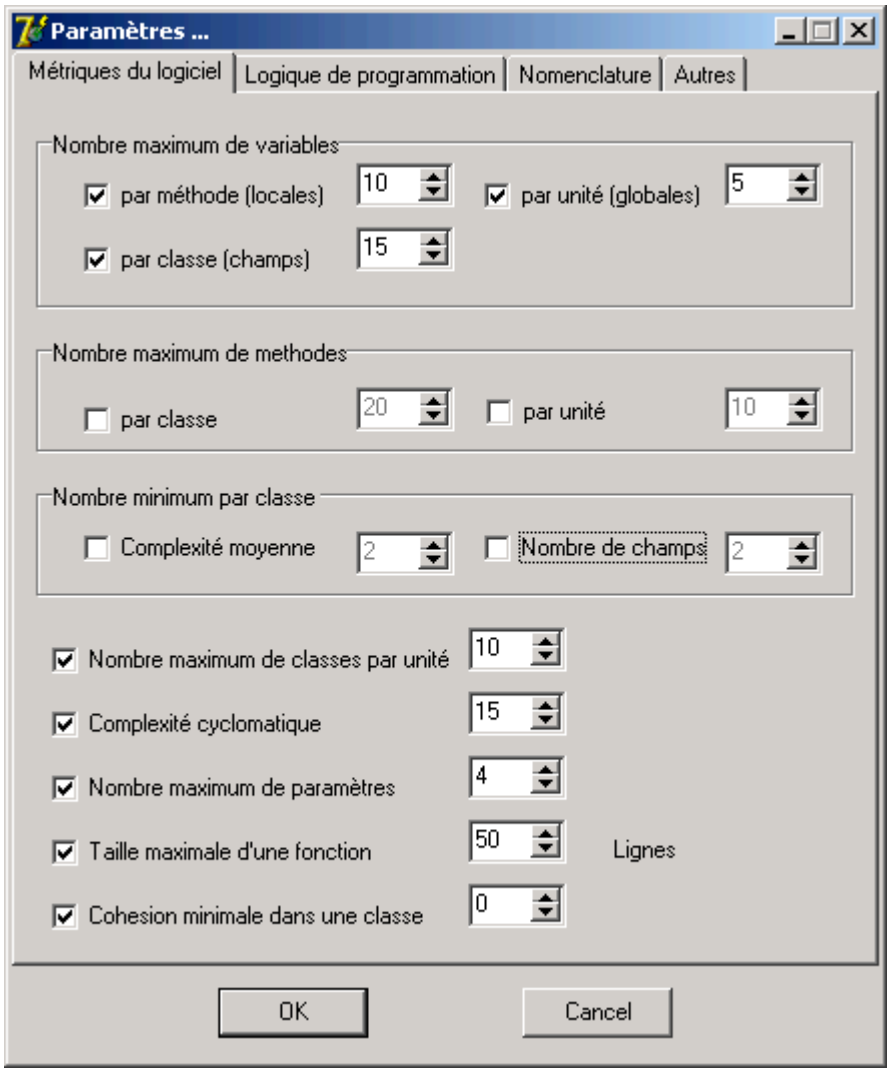
Dans le bas de l'image, on trouve une autre représentation des bads smells, cette fois-ci sous forme d'un arbre. Cela permet d'avoir une vue rapide de tous les bads smells qui ont été détectés et de la façon dont ils sont regroupés. De même cette vue en arbre est rafraichie automatiquement en fonction de l'élément sélectionné dans les différentes liste, de sorte à n'afficher que les bads smells correspondants. Cet arbre est également capable de lancer une commande lorsque l'on double clique sur un bad smell particulier, typiquement pour lancer un éditeur, voir l'environnement Delphi.

Le bouton actualiser à droite sert à réeffectuer la recherche de bads smells en réappelant RainCode et en exécutant les scripts, si par exemple les fichiers sources avait été modifiés entre temps. L'actuallisation des bads smells doit en effet se faire manuellement, tout d'abord parce que cette interface n'est pas capable de détecter quand est ce qu'un fichier a été modifié, et aussi parce que la recherche de bads smells peut prendre un certain temps s'il y a beaucoup de fichiers (de l'ordre d'une minute). Il n'est malheureusement pas possible de diminuer efficacement ce temps, par exemple en gardant en mémoire les informations utiles dans la mesure où c'est le moteur RainCode qui sert à effectuer ce travail, et que dès lors un appel à celui-ci est inévitable. Une autre conséquence du fait qu'il n'est pas possible de garder ces informations en mémoire est le fait qu'il est à chaque fois nécessaire d'analyser tous les fichiers sources, même ceux qui n'ont pas été modifiés.

Capacités de l'interface

L'interface développée n'est pas seulement capable d'affciher les bads smells concernant un ou plusieurs fichiers. Elle offre également la possibilité de sauvegarder les différents parmètres du projet (à savoir fichiers sources à utiliser, bads smells détectés et configuration de la détection, voir ci après). De même il est possible d'exporter la liste de tous les bads smells détectés dans un fichier .csv. L'intéret de la chose est que cela permet de récupérer ces données dans un autre programme (un tableur par exemple) afin de les analyser plus en détail (un exemple de ce type de démarche sera donné plus loin).

Une autre possibilité offerte par cette interface est le fait qu'il est possible de configurer la détection des bads smells en elle-même. Par exemple on peut activer/desactiver la détection d'un bad smell, de même il est possible pour certains d'entre eux (ceux de la catégorie "métrique du logiciel") de définir une valeur limite à partir de laquelle le bad smell est détecté.



Par contre, il est impossible pour différentes raisons d'ajouter et de définir ses propres bads smells sans modifier manuellement les scripts RainCode et la partie de l'interface chargée de la configuration de leur détection.

V. Évaluation du logiciel développé

a) Cas d'étude

Le programme a été testé au sein de la société IDLink. Pour ce faire, divers projets de différentes tailles ont étés analysés.
La partie observation se base sur l'analyse d'un projet (appelons le projet A) considéré comme étant bien structuré. Celui-ci comporte un peu plus de 200 classes. Le but est de mettre en évidence certains cas où la détection de bads smells pouvait être améliorée, par exemple en l'adaptant avec la façon dont travaille l'environnement Delphi.
Pour la partie analyse statistique, 2 autres projets (B et C) ont été utilisés.
Le projet B est assez bien structuré, mais comporte cependant quelques erreurs. Par exemple, une partie de la logique du proramme se trouve dans la même partie de code que celle se chargeant de l'interface graphique.
Le projet C ne comporte qu'une vingtaine de classes, mais est cependant considéré comme étant mal structuré.
Le fait que certains projets soient mieux structurés que d'autres permettra de voir si cette « mauvaise structuration » peut être mise en évidence grâce aux bads smells.

b) Observations

Champs inutilisés

Il s'agit du cas le plus marquant ou la détection de bads smells est en fait biaisée, à cause de la façon dont Delphi travaille. Typiquement, l'analyse du code source via les scripts développés va retourner un grand nombre de champs qui ne sont pas utilisés, surtout dans les classes servant au niveau de l'interface graphique. Cela est du au fait que Delphi crée des variables pour chaque composant qui appartient à une fiche. Mais ces variables ne sont jamais initialisées dans le code. En fait, elles le sont dans les fichiers .dfm dont il a été question dans le chapitre III.a (partie "Delphi dans le code"). Dès lors ce bad smell est détecté de façon automatique, alors qu'il ne devrait pas l'être toujours.
Pour pouvoir éviter de détecter celui-ci, la solution est cependant assez simple (du moins dans sa formulation, mais pas dans sa réalisation !), il suffit de prendre en compte les fichiers .dfm utilisés et de les analyser comme s'il s'agissait du code source de l'application.

Paramètres inutilisés

Ce problème n'est pas tellement lié à la façon dont Delphi travaille, mais plutôt à l'API utilisée par Delphi. Cette API a été pensée de façon à être la plus complète possible. De ce fait, lorsque l'on assigne une fonction pour la gestion de l'un ou l'autre évènement (clic de souris, frappe au clavier, ...), cette fonction se voit recevoir automatiquement un certain nombre de paramètres liés à l'évènement. Cependant ces paramètres ne sont pas toujours utilisés (en fait il le sont rarement tous), ce qui a pour conséquence le fait qu'il sont détecté par les scripts comme étant des paramètres inutilisés. Seulement dans le cas présent, il est impossible de ne pas recevoir ces paramètres, et qui plus est l'API de Delphi se doit de les envoyer, même s'ils ne sont employés que rarement car elle se doit d'être la plus généraliste possible.
Comme pour le cas des champs inutilisés, une solution ici serait d'également analyser les fichiers .dfm. Ceux-ci contiennent en effet la déclaration des méthodes utilisées pour la gestion des évènements (du moins si celles-ci sont affectées de façon statique). Cela permettrait alors de désactiver la détection des paramètres inutilisés pour ces méthodes.

Fichiers générés

En effet, il se trouve que dans certains cas, les programmeurs de chez IDlink ont recours à des générateurs de code. Or il se trouve que le code généré par ceux-ci comprend souvent un grand nombre de bads smells. Par exemple beaucoup de ces fichiers déclarés des méthodes ayant une grande complexité cyclomatique. Cependant, cette complexité est souvent liée au fait que cette méthode est prévue pour effectuer un travail relativement important (mettre à jour certaines données

en fonction de ce qui a été entré dans un boite de dialogue par exemple). Il en va de même pour le bad smell concernant des méthodes trop grandes, lequel est également souvent détecté. Notons aussi le fait que ces fichiers contiennent souvent l'une ou l'autre variable inutilisée. Cela est du au fait que suivant le fichier qui va être généré, celui-ci aura parfois besoin d'une variable à un endroit donné, parfois pas. Aussi la générer à chaque fois se révèle beaucoup plus simple pour le programmeur que de devoir programmer la logique permettant de savoir si cette variable sera utilisée ou non, raison pour laquelle elle l'est toujours.
Ce qui est important de noter ici, c'est que ces bads smells bien qu'intéressants à relever ne sont pas si importants, les fichiers issus de générateurs de code n'étant jamais modifiés à la main. De plus certains de ces fichiers sont en fait inclus dans les fichiers Delphi via l'utilisation de directives de précompilation. Dès lors il n'est pas possible de les ignorer durant la phase de détection de bads smells.
La seule solution que je vois ici serait de désactiver la détection de bads smells pour les fichiers qui sont ou qui englobent des fichiers inclus à l'aide de la directive include. Seulement cette solution se heurte à un autre problème. En effet si cette solution peut éventuellement convenir dans le cadre de la société IDlink, il se peut qu'une autre société utilise également la directive include, mais dans un tout autre but et dès lors la solution proposée ici devient tout à fait inadéquate.

Classes tampons

Les programmeurs de chez IDlink travaillent souvent avec ce que l'on pourrait appeler des classes « tampons ». En fait ce sont des classes qui servent essentiellement de base à d'autres classes. Du coup elles contiennent souvent des méthodes qui n'effectuent pas ou peu de travail, des champs inutilisés ou qui sont juste initialisés, la valeur qui leur est affecté n'étant utilisée que dans les classes dérivées. Ces classes sont très souvent sujettes à de bads smells tels que classe fainéante, champ temporaire ou encore cohésion insuffisante.
Faut-il chercher une solution visant à ne plus détecter de bads smells dans le cadre de ses classes ? En l'occurrence il s'agit d'un choix fait par les programmeurs que de travailler de la sorte. Aussi cela pourrait probablement être sujet à discussion, dès lors il est inutile de chercher une telle soluion. Il convient par ailleurs de rappeler ici que l'interprétation et l'importance qu'il faut accorder à chaque bad smell est toujours du ressort du programmeur et qu'aucune méthode empirique n'existe en la matière.

c) Analyse statistique des résultats

Méthode et outil utilisés

Pour rappel, le but de cette analyse est double :

- Montrer que les bads smells peuvent être utilisés pour surveiller et assurer une certaine qualité au niveau du code source des logiciels développés. Dès lors cette section va tenter, via une approche statistique, de montrer les corrélations qu'il peut y avoir entre la détection d'un ou différents bads smells dans le code source d'un projet, et les problèmes structurels que l'on peut y retrouver.
- Montrer que l'outil développé fournit suffisamment d'informations est peut-être utilisé afin d'exécuter ce type de tâches.

En ce qui concerne les outils utilisés pour faire cette analyse, aucune interface n'a été développée en ce sens. A la place, j'ai préféré utiliser le logiciel R. Il s'agit d'un logiciel d'analyse statistique Open Source, à la fois très puissant (il est capable de faire des analyses complexes très simplement) mais aussi très complet (il existe de nombreux packages différents, chacun ayant ces spécificités) [12]. L'utilisation de ce logiciel consiste à taper des commandes au clavier dans une fenêtre de type terminal. Il existe des interfaces plus « évoluées », mais aucune d'entre elles n'a été utilisée.

En ce qui concerne la méthode utilisée, celle-ci peut être divisée en différents points :

- Préparation des données
- Analyse de correspondances
- Classification

La phase de préparation des données est liée au fait que celle-ci ont beau être disponibles dans un fichier *.csv (en faisant outils→exporter smells dans l'interface), le format dans lequel elles se présentent ne convient pas quand il s'agit de faire une analyse statistique. En fait, on pourrait

représenter ces données sous la forme du tableau suivant :

Smell_name	Value	Line	Unit	Classe	Methode
Method Name Too Long	17	148	BadSmellList	TBadSmellList	GetSelectedSmells
Cyclomatique Complexity	6	157	BadSmellList	TBadSmellList	Display
Insufficient cohesion	3	11	BadSmellList	TBadSmellList	
Cyclomatique Complexity	22	116	BadSmellLists	TSmellList	HeavyFilter
Temporary Field	FFilter	34	BadSmellLists	TSmellList	
Insufficient cohesion	2	31	BadSmellLists	TSmellList	
Inappropriate Intimacy	1	31	BadSmellLists	TSmellList	
Lazy Class	0 fields	51	BadSmellLists	TClassSmellList	
Inappropriate Intimacy	1	51	BadSmellLists	TClassSmellList	
Lazy Class	0 fields	59	BadSmellLists	TFileSmellList	
Lazy Class	0 fields	68	BadSmellLists	TNamedSmellList	
Inappropriate Intimacy	1	68	BadSmellLists	TNamedSmellList	
…	…	…	…	…	…

Aussi ce que l'on veut faire ici est d'analyser les liens qu'il peut y avoir entre différents bads smells ou savoir si certains fichiers/certaines classes contiennent l'un ou l'autre bad smell particulier. C'est pourquoi, ce tableau doit être retravaillé de sorte que les informations soient représentées comme suit :

Fichier	smell1	smell2	smell3	smell4	smell5	,,,
file1.pas	10	3	6	12	0	
file2.pas	12	27	11	2	3	
file3.pas	2	5	13	8	1	
,,,						

Dans ce tableau chaque nombre représente le nombre de fois qu'un bad smell de ce type a été détecté dans ce fichier. Un 0 signifie que le fichier ne contient aucun bad smell de ce type. Le fait de représenter les bads smells de cette façon devrait permettre de détecter plus facilement les liens qui existent entre eux et les fichiers. Un même tableau mais reprenant cette fois-ci chaque classe plutôt que chaque fichier peut être obtenu de façon analogue. Cependant, il faut aussi ce méfier car cette représentation introduit aussi une légère perte d'informations. Par exemple, si l'on détecte dans un fichier une complexité cyclomatique de 40 et une de 4000, les deux seront représentés dans ce tableau de la même façon, alors que concrètement elle n'ont pas la même importance (on pourrait essayer de rectifier cela en pondérant chaque occurrence en fonction de sa valeur. Cependant, en première approche je me contente de compter leur nombre et ne pondère rien.) De même certaines variables telles que la taille de chaque fichier (en nombre de lignes) ne sont pas représentées ici, alors que l'on peut imaginer que celle–ci entre en ligne de compte dans la mesure où un fichier plus grand devrait logiquement comporter plus de bads smells.
L'analyse de correspondances[1] est une méthode utilisée en statistique pour déterminer si un facteur ou un ensemble de facteurs sont capables de caractériser un ensemble de données. Ensuite, il est possible de représenter les données de départ dans un plan en utilisant ces 2 facteurs. Cette représentation permet d'aider à identifier un lien existant entre certains bads smells et certains types de fichiers.
Les méthodes de classification servent essentiellement à regrouper les données en un certain nombre de sous-ensembles (clusters) de l'ensemble de données de départ. Il sera par exemple intéressant de voir ici si des fichiers ayant des rôles similaires se retrouvent dans un même groupe ou non.

Analyse en elle-même

Dans cette partie, je vais tout d'abord exposer la manière et les étapes suivies pour réaliser l'analyse statistique dans le cadre du projet A (en reprenant entre autre les commandes utilisées au sein de l'environnement R). Ensuite je vais montrer les résultats obtenus dans le cadre des projets B et C, la méthode utilisée étant strictement la même.

[1] Un rappel concernant les principes de base de l'analyse de correspondances et sur les méthodes de classification se trouve en annexe.

La première chose à faire est de charger les données issues de l'analyse dans l'environnement R.

```
myData <- read.table("ProjetA.csv", header=T, sep=",")
names(myData)
[1] "Smell_name" "Value"     "Line"      "File"      "Unit"      "Classe"    "Methode"
```

Comme on peut le voir, myData contient l'ensemble des données qui étaient dans le fichier csv. Il faut ensuite, comme expliqué plus haut, faire une transformation afin d'avoir un tableau ayant comme lignes les différents fichiers et comme colonnes les différents bads smells, chaque case représentant la fréquence avec laquelle un bad smell se retrouve dans un fichier. Heureusement, en R une seule commande permet de faire cela :

```
myFileTable <- table(myData$File, myData$Smell_name)
myFileTable
```

Classe Name	Too Long	ClassTooManyField	CyclomaticComplexity
BaseComponents.pas	0	0	0
ConfigForm.pas	0	1	0
ConfigManager.pas	0	1	5
ConfigViews.pas	1	4	23
DocGenerator.pas	0	0	0
EvalForm.pas	0	0	0

…

Avant d'aller plus loin, il est nécessaire de transformer le tableau myFileTable en un autre type de tableau car la fonction qui va par la suite réaliser l'analyse en composantes principales en a besoin. Cependant, on notera que cela ne change rien aux données.

```
myFileTableFrame <- as.data.frame(myFileTable)
class(tab <- xtabs(Freq ~ ., myFileTableFrame))
tab
```

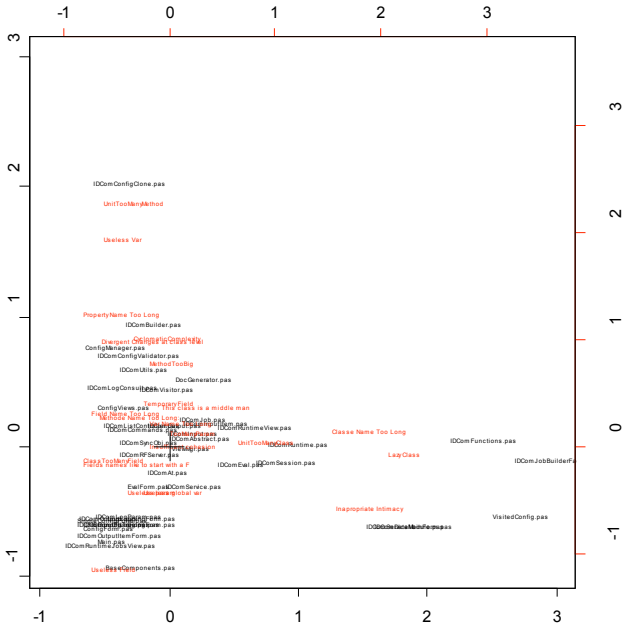
BaseComponents.pas	0	0	0
ConfigForm.pas	0	1	0
ConfigManager.pas	0	1	5
ConfigViews.pas	1	4	23
DocGenerator.pas	0	0	0

…

Maintenant, on peut passer à l'analyse de correspondances en elle-même. Elle se fait via l'utilisation de la fonction corresp. Le paramètres nf = 2 permet de spécifier que l'on va calculer les 2 premiers facteurs.

```
myFileTableCor <- corresp(tab, nf=2)
biplot(myFileTableCor, cex=0.7, arrow.len=0.01)
```

La fonction biplot nous renvoie le graphe suivant :



Ce graphe nous permet de voir clairement que certains fichiers mais aussi certains bads smells ont tendances à suivre soit l'axe des X, soit l'axe des Y. Par exemple on voit que les bads smells lazy class et UnitTooManyMethod prennent des directions opposées. On voit aussi par exemple que le fichier IDComConfigClone.pas est à part des autres et suit la direction indiquée par des bads smells tels que UnitTooManyMethod mais aussi Useless var ou encore Cyclomatic Complexity. Si l'on regarde le code de ce fichier, on s'aperçoit que celui-ci utilise des fichiers générés (comme expliqué plus haut) et qu'il semble ne contenire que des méthodes. On constate aussi qu'un certain nombre de fichiers se retrouvent en bas à gauche et sont très proches les uns des autres. En fait, il s'agit essentiellement de fichiers chargés de s'occuper de l'interface graphique.

Après l'analyse de correspondances, on peut aussi effectuer une analyse en utilisant les méthodes de classification. Pour ce faire, j'utilise une méthode dite de classification hiérarchique, qui consiste à regrouper chaque sous-groupe 2 à 2 en vue d'en former un seul, cela jusqu 'à ce que tous les éléments de départ soient réunis dans 1 seul.
Avant de créer un ensemble de cluster hiérarchique, il me faut d'abord calculer la matrice des distances. La fonction dist permet de calculer cette matrice entre les différentes lignes du tableau de départ. Celle-ci est calculée en utilisant la distance euclidienne classique.

myFileTable.dist <- dist(myFileTable)

Cela nous donne :

myFileTable.dist

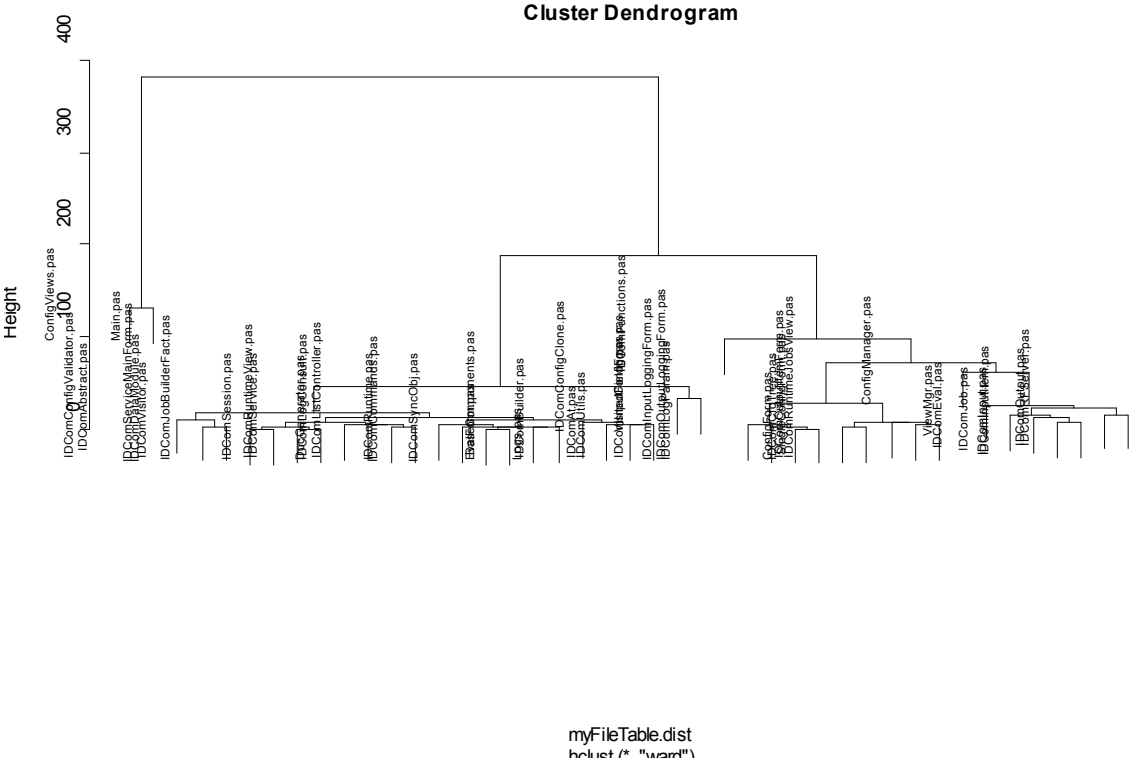
	BaseComponents.pas	*ConfigForm.pas*	*ConfigManager.pas*	*ConfigViews.pas*
ConfigForm.pas	*19.157244*			
ConfigManager.pas	*50.328918*	*42.708313*		
ConfigViews.pas	*186.598499*	*171.889499*	*157.974682*	
DocGenerator.pas	*7.937254*	*23.916521*	*50.655701*	*190.099974*
EvalForm.pas	*5.567764*	*23.194827*	*51.458721*	*188.679623*
IDComAbstract.pas	*9.273618*	*18.411953*	*45.022217*	*181.887328*
IDComAt.pas	*6.480741*	*20.856654*	*49.487372*	*186.169278*

…

Comme on peut le voir :
-Cette matrice est triangulaire. En fait, elle contient toutes les valeurs utiles, simplement il s'agit de distances, aussi la distance de A à B étant égale à la distance de B à A, celle-ci n'est reprise qu'une seule fois dans le tableau. C'est d'ailleurs pour cela que le fichier BaseComponents.pas n'est repris que dans les colonnes. Toutes les distances de ce fichier par rapport aux autres y étant repris, il est inutile d'ajouter une ligne pour ce fichier car elle serait vide.
-Cette matrice reprend les distances des fichiers entre eux. Cela veut dire que si l'on désire analyser les bads smells entre eux, il faudra alors calculer une autre matrice reprenant la distance de chaque vis-à-vis des autres bads smells.

Maintenant que l'on a une matrice reprenant les distances entre les différents fichiers, on est en mesure de créer une classification hiérarchique en regroupant 2 à 2 chaque élément. Ce regroupement peut se faire de différente manière, par exemple en regroupant systématiquement les 2 ensembles les plus proches. Dans notre contexte, j'ai utilisé différents critères pour effectuer ce clustering, aussi je ne reprendrai ici que le graphe obtenu avec celui de ward car c'est celui qui met le mieux en évidence les relations entre les fichiers.

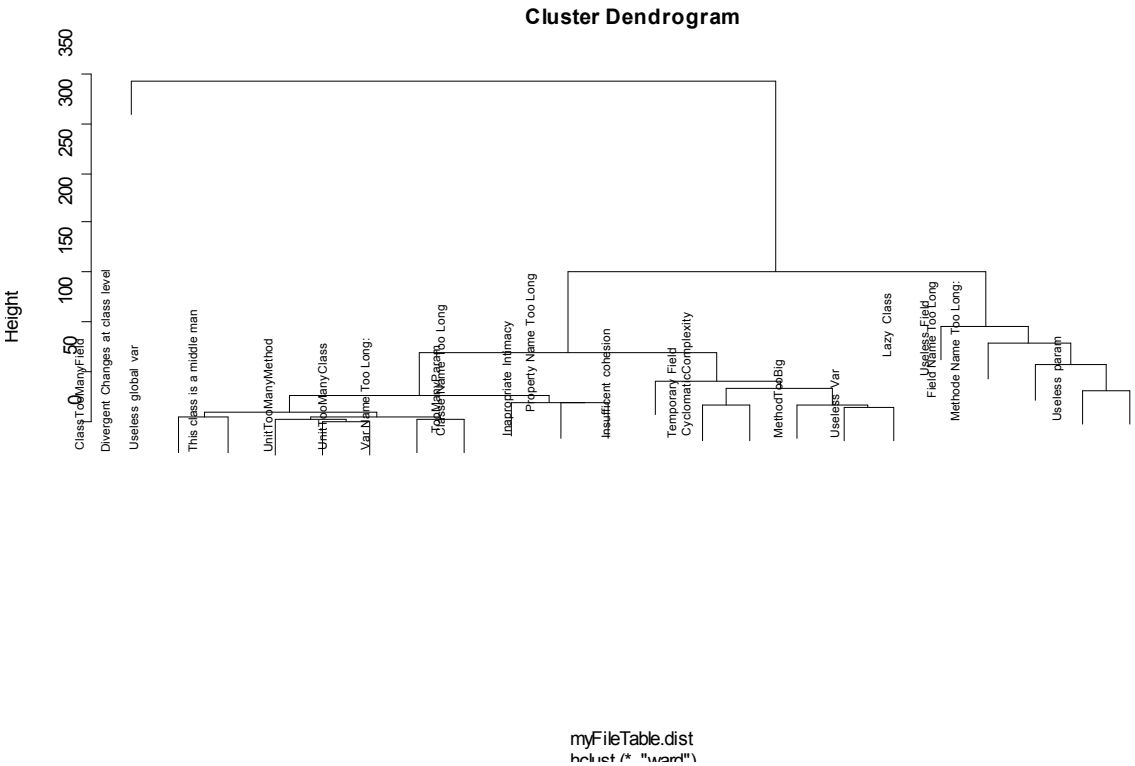
myFileTable.hclust <- hclust(myFileTable.dist, "ward")
plot(myFileTable.hclust,cex=0.7)



Ce dendrogramme permet d'identifier clairement le fait qu'il existe 3 groupes de fichiers différents. En fait, après avoir parlé avec les membres de la société IDLink, il apparaît que ces 3 groupes de fichiers correspondent à des rôles différents dans le projet.
Ainsi, le premier groupe (celui le plus à gauche) correspond à des fichiers contenant des classes dites techniques. Nombre d'entre elles sont là entre autre de par le fait que la conception de ce projet a été faite en respectant des design pattern.
Parmi les fichiers plus à droite, celles du groupe de gauche (de IDComFunctions à IDComRuntimeJobsView) correspondent à la partie interface graphique. Ce groupe de fichiers avait déjà pu être mis en évidence grâce à l'analyse de correspondance. Enfin, le groupe de fichiers le plus à droite correspond au core system.

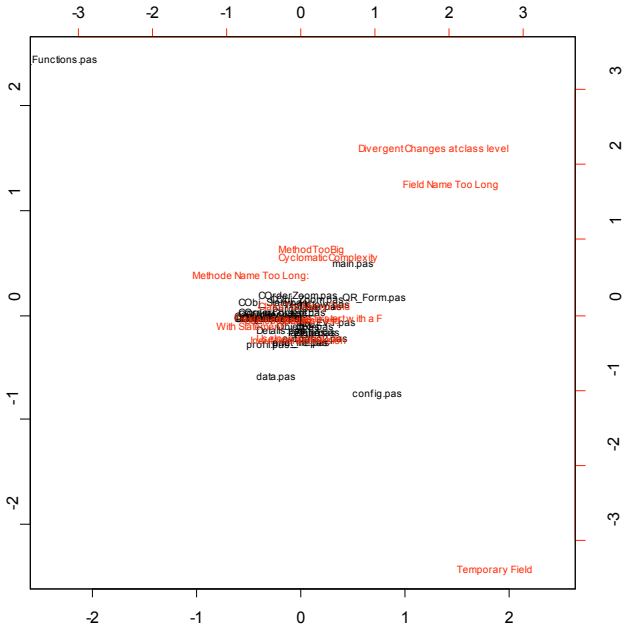
Maintenant, effectuons la classification non plus sur les fichiers mais sur les bads smells. Pour ce faire, il suffit de partir de la transposée de la matrice contenant les fréquences et de recommencer :

myFileTable <- table(myData$Smell_name, myData$File)
myFileTable.dist <- dist(myFileTable)
myFileTable.hclust <- hclust(myFileTable.dist, "ward")
plot(myFileTable.hclust,cex=0.7)



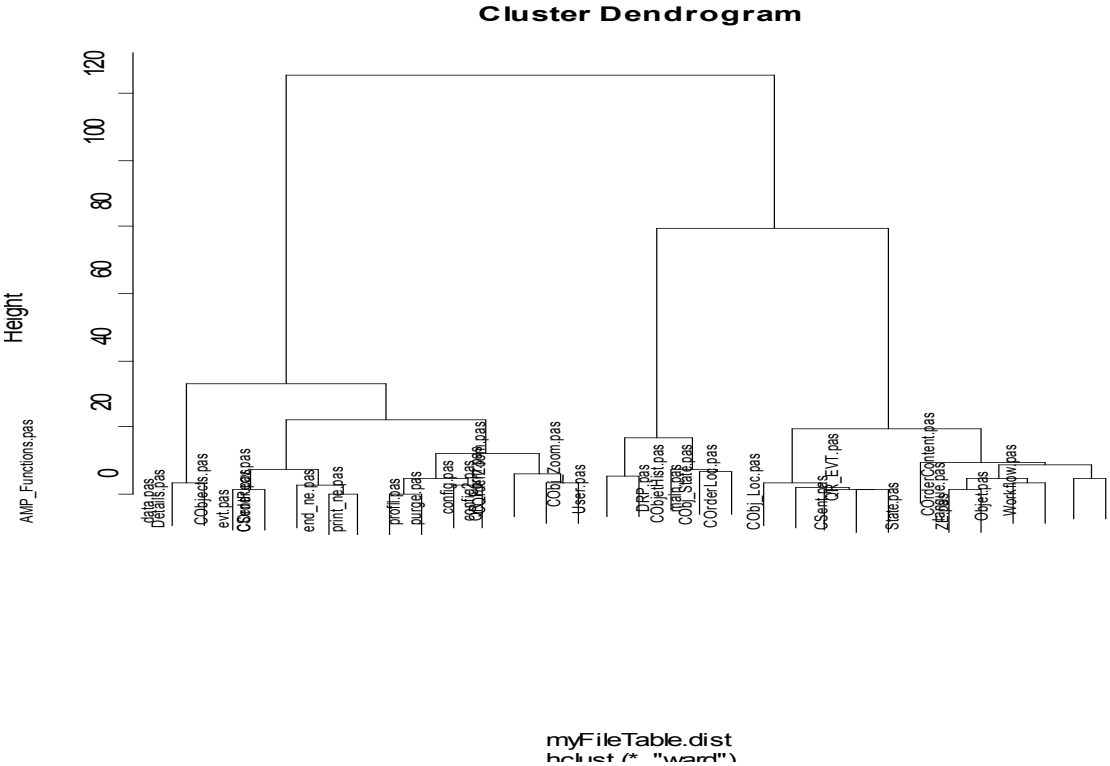
Comme on peut le voir, il semble qu'il y aie 2 voir 3 groupes distincts de bads smells (celui concernant la nomenclature des champs faisant bande à part). Parmi ce que l'on peut voir, on constate que le bad smell lazy class est souvent lié aux bads smells de champs et de paramètres inutilisés. De même, on voit que les bads smells complexité cyclomatique et méthodes trop grandes se retrouvent l'un à coté de l'autre, ce qui semble logique. Par contre, les variables locales inutilisées se retrouvent assez de celles qui sont globales, suggérant le fait que ces 2 bads smells assez ressemblant dans leur principe sont en fait issus de facteurs différents.
On pourrait encore faire un tas d'autres observations, cependant elles sont encore insuffisantes pour dire d'en tirer des conclusions, et il faudra au minimum comparer ces résultats avec ceux d'autres projets.

On peut dès lors passer à l'analyse des projets B et C. Aussi, voici les graphes que j'ai obtenus pour le projet B :
Analyse de correspondances :



Contrairement au premier projet, tous les fichiers mais aussi la grande majorité des bads smells se retrouvent concentrés au centre du graphe. Il est dès lors difficile de trouver une information utile.

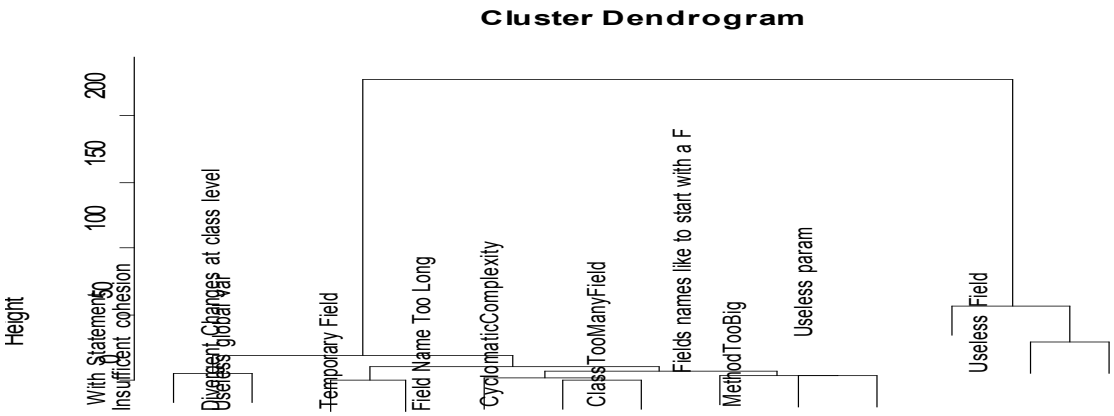
Essayons maintenant de voir les liens entre les fichiers :



Comme pour le projet A, on voit clairement apparaître 3 groupes de fichiers distincts.
Malheureusement, la personne de chez IDLink avec qui je me suis entretenu ne connaît pas

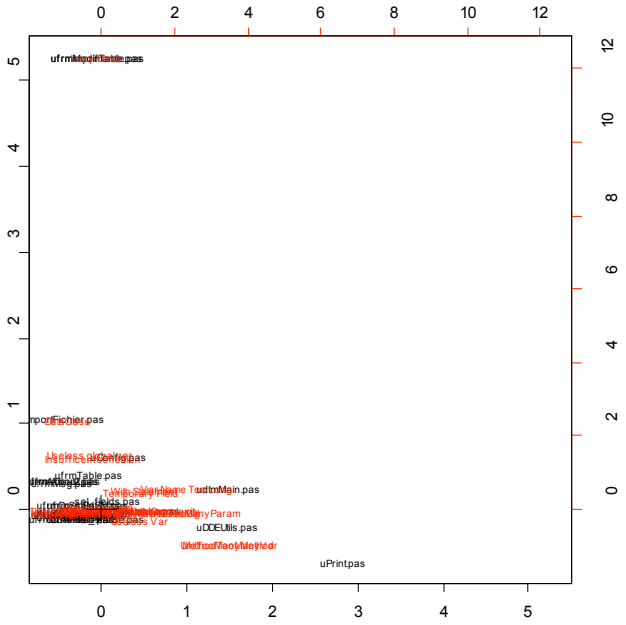
suffisemment ce projet pour pouvoir dire si ces groupes correspondent à des entités cohérentes.

Maintenant en ce qui concerne les bads smells :

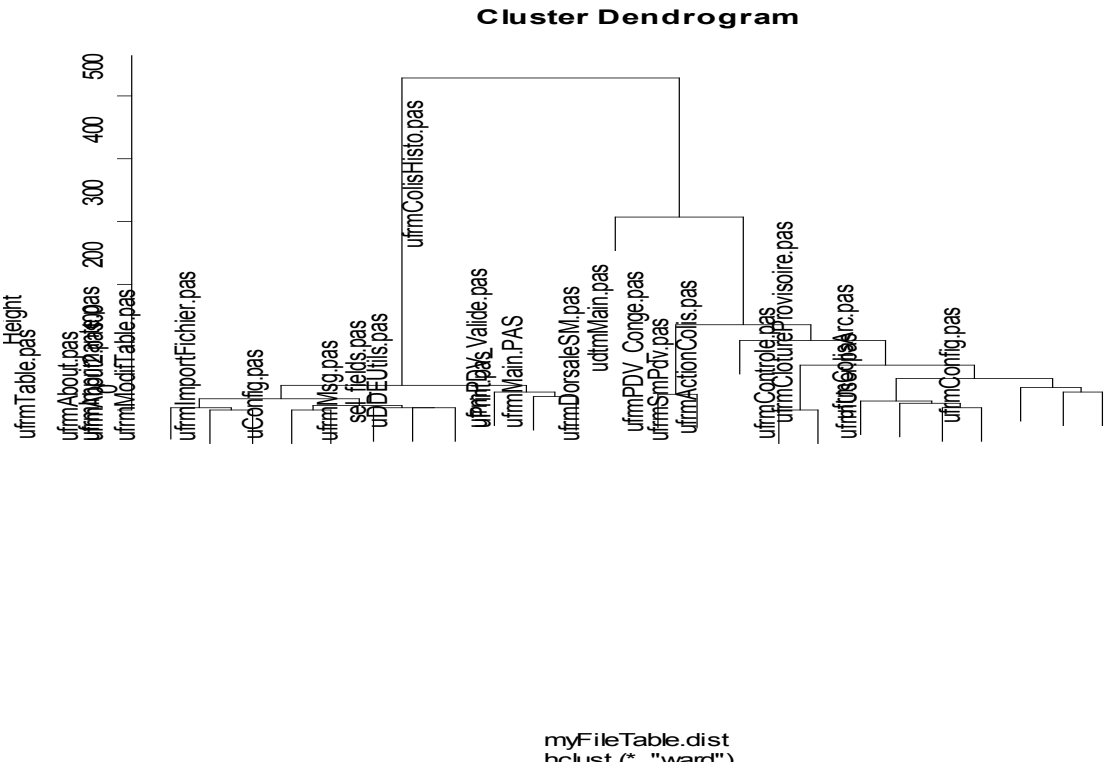


Tout comme pour le projet A, on note qu'il y a 2 groupes qui se forment, lesquels sont assez semblables. Par exemple, les bads smells paramètre inutilisé et champ inutilisé se retrouvent ensemble dans un groupe à part (le bad smell classe fainéante n'a cette fois-ci pas été détecté). Par contre ceux concernant la nomenclature qui se retrouvait dans ce même groupe tantôt sont ici dans celui contenant la majorité des bads smells.

Si maintenant l'on recommence la procédure pour le projet C:
Analyse de correspondaces :

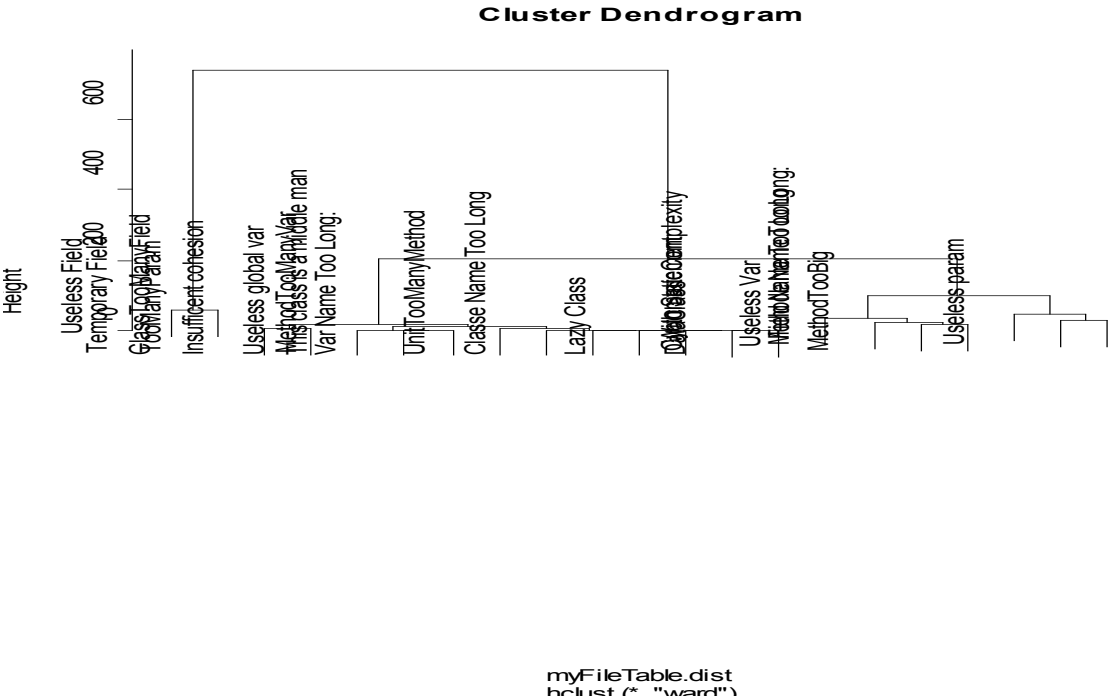


Comme pour le projet B et contrairement au projet A, la majorité des bads smells et des fichiers se retrouvent proches du centre. On ne peut donc tirer aucune information concernant le projet dans sa globalité.
Liens entre les fichiers :



Comme pour le projet précedant, la personne avec qui je me suis entretenu ne connaisait pas suffisement ce projet pour interpréter ce graphe. Cependant on constate quand même que celui-ci semble indiquer qu'il existe essentiellement 2 types de fichiers dans ce projet.

Liens entre les bads smells :



On voit toujours apparaître 2 groupes distincts de bads smells (éventuellement 3), le bad smell champ inutilisé étant toujours à part. Mais cette fois-ci celui concernant les paramètres inutilisés n'est plus avec, et celui concernant les classes fainéantes se retrouve en plein milieu. Les bads smells méthode trop grande et complexité cyclomatique continuent quand à eux d'être ensemble. En allant un peu plus loin, on peut également constater que le bad smells variable globale inutilisée et trop de champs dans la classe sont assez proches.

Observations générales

On constate en ce qui cocnerne l'analyse de corespondances, hormis dans le cas du projet A ou certaines choses étaient observables, celle-ci n'a pas été très efficace. En effet, dans les cas des projets B et C, tout ou presque était rassemblé autour du centre du dessin, et aucune véritable tendance n'était observable. Le fait que cela est différent dans le cas du projet A par rapport aux projets B et C est-il un signe permettant de voir si un projet est bien structuré? Dans la mesure où seulement 3 projets ont été testés, il est encore trop tôt pour tirer une telle conclusion.
En ce qui concerne la classification hiérarchique, elle a permit quand à elle de montrer qu'à chaque fois se forme 2 ou 3 groupes de fichiers. Si l'on se referre au projet A, il semble également que cette classification soit à même d'identidier les différentes parties de chaque projet.
En ce qui concerne la classification des bads smells, ici aussi on voit souvent se former 2 ou 3 groupes. Certains bads smells semblent être plus ou moins liés, cependant ici aussi il faudrait faire d'autres tests pour en savoir plus.

D'une manière générale, les graphes obtenus sont très semblables d'un projet à l'autre. Aussi se réferer à eux pour savoir si un projet est bien structuré semble inefficace. En fait, la seule mesure qui permet de quantifier cela revient à calculer le nombre de bads smells par fichiers.
Dans le cas du projet A, on obtient 1820 bads smells pour 43 fichiers soit une moyenne de 43,32.
Dans le cas du projet B, on obtient 1220 bads smells pour 31 fichiers soit une moyenne de 39,35
Dans le cas du projet C, on obtient 2485 bads smells pour 24 fichiers soit une moyenne de 103,54.
Si on regarde les classes au lieu des fichiers, on obtient respectivement une moyenne de 8,66, de 40,6 et de 108,04 bads smells/classe.

VI. Conclusion et travaux futurs

a) Contributions

Notion de Bad Smell

Pour rappel, le premier objectif de ce mémoire était d'introduire et d'expliciter la notion de bad smell. Cela a été fait en donnant tout d'abord une définition et en expliquant dans quel domaine cette notion est utilisée. Ensuite j'ai présenté la suite décrite par M. Fowler, laquelle a ensuite été analysée puis étendue un peu plus loin. Enfin un bref aperçu de la façon dont les informations concernant les bads smells peuvent être recueillis, présentés et utilisés a été démontrée via la création d'un outil de detection ainsi que via une analyse statistique des résultats.

Adaptation à Delphi

Un des aspects novateurs de ce travail est le fait qu'il s'intéresse au langage Pascal Objet (utilisé par Delphi), lequel n'a jusqu'ici suscité que peu d'attention de la part de la majorité des personnes s'intéressant aux bads smells et plus généralement à l'évolution des logiciels en général. Dans ce contexte, le but était avant tout de voir quels bads smells pouvaient être adaptés à ce langage. Or il se trouve que le Pascal Objet étant un langage de programmation orienté objet et partageant un grand nombre de points en commun avec Java (je parle ici au niveau de la façon d'organiser le code sans tenir compte de la syntaxe), la plupart des bads smells ont pu être repris tel quel.

Ensuite, ce travail introduit d'autres bads smells spécifiques au Pascal Objet. Ces derniers peuvent certes être débattus, ils me permettent de montrer la possibilité de trouver des bads smells typiques à ce langage.

Suite à cela, s'en suit une présentation de la démarche qui a été adoptée afin de pouvoir détecter ces bads smells. Dans un premier temps, il a fallu étudier lesquels sont détectables, comment, lesquels ne le sont pas et pourquoi. Puis une manière d'effectuer ce travail en analysant le code source a été présentée avant d'être implémentée, via l'utilisation du RainCode Engine.

Résultats concrets

Les tests sur divers projets de la société IDlink ont permis de mettre en évidence certains cas concrets où la détection devrait être adaptée afin de prendre en compte non seulement le langage Pascal objet mais aussi la façon dont travaille Delphi, en analysant par exemple les fichiers dfm.
En ce qui concerne l'analyse statistique, l'analyse de correspondances n'a pas permis de mettre en évidence des liens entre les fichiers et les bads smells que dans le cadre du projet A.
En ce qui concerne la classification, lorsqu'elle est utilisée pour classer les fichiers, elle semble permet tre de mettre en évidence les différentes parties de chaque projet. Il seble aussi que la classification permet de montrer des liens entre certains bads smells. Seulement les résultats en ce qui concerne la classification des bads smells sont moins nets et demandent de tester un plus grand nombre de projet pour être confimés.

b) Travaux Futurs

Amélioration de la détection

Bien que l'outil développé soit capable de détecter un grand nombre de bads smells, cette détection reste loin d'être optimale, ce pour différentes raisons. Aussi voici un bref aperçu des améliorations et des modifications qu'il y aurait lieu de faire si quelqu'un en venait à reprendre ce projet.

Au niveau de la détection

La première amélioration à apporter serait d'adapter la détection à l'environnement Delphi. Cela peut se faire en intégrant une analyse des fichiers dfm. Cependant, il se peut qu'une telle analyse demande aussi certaines modifications au niveau de RainCode afin d'intégrer les données issues de

ces fichiers dans l'arbre de syntaxe.

En ce qui concerne l'analyse en elle-même, pour le moment elle se déroule en 2 phases : la décoration et la détection. Or pour la décoration, le processus consiste à décorer tous les fichiers à la suite, un fichier à la fois. Une amélioration possible serait de ne plus analyser chaque fichier un par un, mais de les analyser simultanément. Plus concrètement, il s'agirait en fait d'analyser la partie interface de chacun d'entre eux, puis ensuite seulement leur partie implémentation. L'avantage d'un tel procédé est que cela permettrait d'avoir au moment de l'analyse de l'implémetation des méthodes un nombre d'informations plus important que celui disponible actuellement, notament en ce qui concerne la résolution des différents types.

En allant plus loin, on pourrait imaginer un système permettant de trier l'ordre dans lequel chaque fichier doit être analysé, ce en fonction des inclusions et inter-dépendances qui peuvent exister entre ceux-ci. Cela permettrait de disposer de toutes les informations nécessaires à la fois lors de l'analyse de la partie implémentation, mais aussi au moment de l'analyse de la partie interface.

Au niveau de la configurabilité/extensibilité

Comme il a été dit plus haut, les bads smells sont quelque chose d'assez subjectif et sujet à interprétations. Aussi en première approche l'outil réalisé permet de spécifier quels bads smells sont à détecter, et pour certains d'entre eux il est possible d'entrer une valeur comme paramètre. Une approche supérieure serait de considérer que l'utilisateur doit avoir un contrôle total sur les bads smells qu'il cherche à détecter. Dès lors cela implique qu'il doit être capable d'avoir accès au code chargé de la détection, qu'il doit être capable de le modifier mais également d'ajouter du code en vue de détecter de nouveaux bads smells.

Seulement une telle exigence demande de repenser à la fois la façon dont s'effectue la phase de détection, mais aussi comment est programmée l'interface chargée d'appeler les scripts RainCode.

Au niveau des scripts, 2 solutions sont envisageables. La première consiste en la création d'un script capable de prendre en paramètres l'arbre de syntaxe enrichit durant la phase d'analyse, ainsi qu'un fichier contenant les bads smells définis par l'utilisateur décris soit dans un langage déjà existant (XML par exemple. Un étudiant de l'université a par ailleur réaliser un travail sur ce sujet **Error! Style not defined.**[13]), soit dans un meta-langage dédié. Une autre solution serait de permettre à l'utilisateur d'entrer et de créer ses propres scripts en RainCode directement, en partant du principe que le langage de script RainCode est déjà en lui-même un langage dédié. Cela implique de prévoir la possibilité de pouvoir appeler différents scripts sur les fichiers à analyser.

Au niveau de l'interface, il faut avant tout prévoir une boite de dialogue permettant à l'utilisateur de spécifier ses propres définitions de bads smells dans le langage ou le meta-langage choisi. Ensuite, il faut aussi prévoir que la boite de dialogue permettant de choisir et de paramétrer les différents bads smells ne doit dès lors plus être créée de façon statique, à savoir en dur dans le code comme c'est le cas actuellement, mais être créée de façon dynamique en fonction des bads smells qui ont été définis.

Au niveau de l'utilisation

Actuellement, il existe 2 contraintes au niveau de l'utilisation de l'outil développé. La première est le fait que si l'on veut que l'analyse soit complète, alors il est nécessaire d'analyser tous les fichiers du projet, certains bads smells pouvant dépendre de plusieurs fichiers sources. La seconde est le fait qu'il est impossible de garder en mémoire l'arbre de syntaxe de chaque fichier, et que sès lors il est nécessaire de ré-analyser tous les fichiers à chaque fois que l'on veut re-détecter les bads smells. La conséquence de cela est que cette recherche ne peut se faire en temps réel et que c'est à l'utilisateur de manuellement cliquer sur un boutton pour re-lancer la détection, laquelle si elle n'est pas excessivement longue n'est cependant pas instantanée.

Cependant, il se peut que ces 2 problèmes puissent trouver une solution commune. En effet, il se trouve qu'il existe une version du RainCode engine créé pour ce type de besoins spécifiques. En fait il s'agit d'une version qui, plutôt que de se présenter sous la forme d'un exécutable que l'on lance se présente sous la forme d'une dll (accessible via des objets COM). L'avantage de cette version de RainCode est qu'elle est persistente, ce qui veut dire qu'elle ne perd pas les informations qui sont contenues dans l'arbre. De même, il est possible de lui demander recharger et de re-parser un fichier indépendemment des autres si jamais il a été modifié entre temps, ce qui permet de gagner pas mal de temps. En ce qui concerne les bads smells qui ont été détectés, à partir du moment où l'arbre est

les données utilisées par RainCode sont persistentes, il devient alors également possible de garder les résultats en mémoire, permettant dès lors de ne plus ré-analyser le code que pour les bads smells où cela est nécessaire. Cependant, le fait d'utiliser cette version de RainCode risque d'impliquer un nombre important de changements au niveau de l'interface développée. De même, le fait de garder en mémoire les résultats des analyses précédantes et de ne mettre à jour que les informations nécessaires implique également beaucoup de changements au niveau des scripts de détection.

Une autre avancée en ce qui conerne l'interface touche quand à elle la partie du code chargée d'afficher les bads smells détectés. Actuellement, celle-ci affiche tous les bads smells. Une conséquence de cela est que si l'on ne désire afficher qu'une partie des bads smells détectés, alors il est nécessaire de faire une nouvelle recherche qui ne prend en compte que les bads smells spécifiés. Si l'interface était capable de faire elle-même le tri, alors il ne serait plus nécessaire de faire une nouvelle recherche à chaque fois que l'on veut n'afficher qu'un sous–ensemble des bads smells détectés.

Autres améliorations

Une des amélioration possible et des plus naturelles du projet serait une intégration dans l'environnement Delphi. Cependant, commme pour le parseur, si l'on veut avoir les ressources (documentation et autres) permettant de faire cela, il faut alors devenir partenaire avec Borland, ce qui n'est pas envisageable ici[7].

Une autre amélioration consisterait en l'intégration d'une interface permettant de faciliter la phase d'analyse des bads smells détectés. En effet, dans ce travail j'effectue une analyse statistique des résultats obtenus. Cependant cette analyse a été faite à l'aide d'un programme externe (en l'occurence R). Le problème est que l'utilisation de ce type de logiciel prend un certain temps et que bon nombre d'étapes pourrait être automatisées dans la mesure où le type de données utilisées pour effectuer cette analyse est connu d'avance. Pour ce faire, deux solutions sont envisageables:

La première serait d'intégrer un outil de visualisation, similaire à ce qui se fait déjà dans d'autres outils comme le plugin Metrics pour Eclipse (voir chapitre III), ou proposer un autre type de visualisation. Cela suposerait de trouver un moyen de représenter toutes les entités (classes, fichiers, bads smells, methodes,...) en un seul graphe et de permettre de le manipuler.

Une autre solution serait de créer un outil capable de faire une analyse statistique de façon automatique ou semi-automatique (entendre par la configurable par l'utilisateur) sur base des bads smells détectés. Cet outil serait capable de par lui même effectué des analyses telles que l'analyse en composantes principale ou encore une analyse en composantes multiples,... et ce sur toutes les données mais aussi sur un ensemble restreint de ces dernières.

Enfin une autre amélioration concerne elle l'analyse statistique en elle-même. Comme je l'ai indiqué plus haut, celle-ci se base sur un tableau reprenant la fréquence de chaque bad smell au sein de chaque fichier. Une avancée serait de voir si cette analyse ne pourrait pas être plus pertinente en pondérant chaque occurence de chaque bad smell. La question qui se pose alors est de savoir en fonction de quoi il faut les pondérer. La seule chose permettant cependant de pondérer une occurence d'un bad smell est en fait la valeur associée à ce bad smell. En effet, des aspects comme la taille d'un fichier étant déjà pris en compte par certains bads smells, les utiliser pour en pondérer d'autres serait une erreur. Cependant, il faut encore voir comment les pondérer, car chaque bad smell peut être pondéré de façon différente. Par exemple dans le cas de la complexité cyclomatique, on pourrait prendre le logarithme néperien de la complexité détectée. Par contre dans le cas des paramètres en trop, on pourrait par exemple prendre le nombre de paramètres – la valeur limite utilisée.

Autres travaux

Si l'on se replace dans le contexte de recherche, le travail et plus spécifiquement l'outil réalisé peut être utilisé pour effectuer des recherches dans le domaine des bads smells. En effet, il reste encore bon nombre de sujets pour lesquels le nombre de travaux effectués reste insuffisant.
Le premier d'entre eux concerne les problèmes d'interprétation des bads smells. Il existe pour le moment encore peu d'études permettant de démontrer leur intérêt, en quoi ils peuvent aider les programmeurs, quels bads smells sont les plus représentatifs et quels liens il peut exister entre la présence de bads smells et les problèmes d' évolution des logiciels. L'interprétation de ceux-ci et leur pertinence est encore quelque chose d'intuitif et est de ce fait laissé à la charge du programmeur.

L'analyse statistique faite sur les projets de la société IDlink est un premier pas en ce sens, mais pour avoir des résultats qui soient vraiment exploitables il faudrait effectuer ce type de tâche sur un plus grand nombre de projets et le faire pendant un labs de temps plus long afin de pouvoir comparer différentes version de chaque projet.
De même une autre voie qui peut être suivie est celle de l'intégration des bads smells dans le processus de refactoring. Pour rappel Fowler avait introduit sa liste de bads smells en vue de localiser les endroits où effectuer un refactoring. Dans son livre, il va même plus loin que cela car il propose pour chacun d'entre eux quels refactoring peut être appliqué pour supprimer ce bad smell. Dans un premier temps ce qui peut alors être fait serait de mettre au point un outil capable non seulement de détecter les bads smells mais aussi de proposer les refactorings à effectuer [14]. En allant plus loin, on pourrait tenter de mettre au point un outil capable d'effectuer ces refactorings de façon automatique, en choisissant par exemple lorsque plusieurs refactorings sont applicables celui qui semble le plus adéquat, la tâche consistant à trouvé le bad smell le plus adéquat pouvant être un sujet de recherche en soi.
Toujours en lien avec le refactoring, on pense aujourd'hui que le fait de faire du refactoring permet de diminuer le nombre de bads smells. Aussi il peut être intéressant de faire une étude permettant de montrer, sur base de cas concrets, quel refactoring permet de faire diminuer quel type de bad smell? De même, on peut se poser la question de savoir si un refactoring particulier, s'il permet d'éliminer un ou plusieurs types de bads smells n'est pas susceptible d'en introduire de nouveaux.
Enfin une dernière voie serait d'utiliser le programme développé plus dans le temps afin de surveiller par exemple l'évolution d'un logiciel afin de voir si on peut constater une augmentation du nombre de bads smells au cours du temps, et voir si cela est ou n'est pas lié à certaines phases du projet. De même, on pourrait également surveiller non pas l'évolution d'un programme mais l'évolution de différents programmeurs afin de voir si certains bads smells sont typiques des débutants (lesquels?), alors que d'autres seraient typiques de certains programmeurs (sorte d'empreinte du programmeur).

VII. References

[1] Martin Fowler, **Refactoring: Improving the design of existing code** ; Addison Wesley, 1999, chapitre 3:*Bads smells in code*.

[2] M. M. Lehman, L. Belady. **Program Evolution: Processes of Software Change**. Academic Press, London, 1985.

[3] Eva Van Emden et Leon Moonen, **Java quality assurance by detecting code smells**, 2002. Voir l'adresse suivante: http://homepages.cwi.nl/~leon/papers/wcre2002/

[4] Voir exemple énonés par Mika Mantyla, **Bads smells in software – a taxonomy and an Empirical study**, Helsinky University of Technologie, 2003, chapitre 3 (p.18-21): *Measuring Maintenability.*

[5] Une liste d'outils de ce type est disponible à l'adresse suivante: http://pmd.sourceforge.net/similar-projects.html.

[6] Jimmy Gilles, **rapport de stage : Détection de code dupliqué**, 2004, Université Mons-Hainault.

[7] Voir http://www.borland.fr/partners/index.html

[8] Pour plus d'informations, voir le site officiel de RainCode : http://www.raincode.com

[9] La syntaxe complète est disponible ici : http://www.raincode.com/rcdoc/online.html

[10] Il est fait mention de ce bad smell à l'adresse suivante : http://wiki.java.net/bin/view/People/SmellsToRefactorings

[11] Shyam R. Chidamber et Chris F. Kemerer, **A metrics suite for object oriented design**, IEEE, 1994.

[12] R homepage : http://www.r-project.org/

Error! Style not defined.[13] Van Regen Olivier, **Détecter les bads smells avec XQuery et XSLT**, Université Mons-Hainaut, 2005.

[14] Cette approche a été utilisée dans le travail suivant : Francisca Munoz Bravo, **A Logic Meta-Programming Framework for Supporting the Refactoring Process**, Vrije Universiteit Brussel, 2003

[15] Un travail concernant l'algorithme Apriori avait déjà été soumis au sein de l'université sur base du document suivant : Paulo J Azevedo, **CAREN – A java based Apriorori Implementation for Classification Purposes** ; 2003.

VIII. Annexes

a) *Grammaire Delphi utilisée par RainCode*

Voici la grammaire utilisée par RainCode pour effectuer le parsing des sources Delphi. Ce paragraphe reprend le nom de chaque type de nœud ainsi qu'une description de sa composition. Les différents types de nœuds sont repris par ordre alphabétique. Le nœud racine de l'arbre de parsing est de type DelphiFile.

ArrayConstant

ArrayConstant ::= ['**>**'] '**(**' { TypedConstant } + BY '**,**' '**)**'

ArrayType

ArrayType ::= ['**PACKED**'] '**ARRAY**' ['**[**' { OrdinalType | TypeId } + BY '**,**' '**]**'] '**OF**' Type

AsmBlock

AsmBlock ::= '**ASM**' { AsmWord } + EndClause

AsmBlockStmt

AsmBlockStmt ::= AsmBlock '**;**'

AsmStatement

AsmStatement ::= '**ASM**' { AsmWord } +

AsmWord

AsmWord ::= '**ABSOLUTE**' .. '**ELSE**'

AssignStatement

AssignStatement ::= Designator '**:=**' Expression

BinaryExpression

BinaryExpression ::= Expression '**=**' .. '**<=**' Expression
BinaryExpression ::= Expression { '**AND**' | '**DIV**' | '**MOD**' | '**SHL**' | '**SHR**' | '*****' | '**/**' } Expression
BinaryExpression ::= Expression { '**OR**' | '**XOR**' | '**+**' | '**-**' } Expression

Block

Block ::= AsmBlock
Block ::= ExternalBlock
Block ::= RegularBlock

BooleanLiteral

BooleanLiteral ::= { '**TRUE**' | '**FALSE**' }

BracketDesignator

BracketDesignator ::= Designator '**[**' { Expression } + BY ',' '**]**'

CaretDesignator

CaretDesignator ::= '**^**' Designator ['**>**']
CaretDesignator ::= Designator '**^**'

CaseLabel

CaseLabel ::= Expression '**..**' Expression
CaseLabel ::= Expression

CaseSelector

CaseSelector ::= { CaseLabel } + BY ',' ':' [Statement]

CaseStatement

CaseStatement ::= '**CASE**' Expression '**OF**' { CaseSelector } + BY ';' [[';'] '**ELSE**' [StatementList]] [';'] '**END**'

Cast

Cast ::= Expression '**AS**' QualId

ClassFieldList

ClassFieldList ::= ObjFieldList

ClassHeritage

ClassHeritage ::= '**(**' { QualId } + BY ',' '**)**'

ClassMember

ClassMember ::= ClassVisibility { ClassFieldList | ClassMethodList | ClassPropertyList } *
ClassMember ::= { ClassFieldList | ClassMethodList | ClassPropertyList } +

ClassMethodList

ClassMethodList ::= { MethodList } + BY ';' ';'

ClassPropertyList

ClassPropertyList ::= { PropertyList } + BY ';' ';'

ClassRefType

ClassRefType ::= '**CLASS**' ['**OF**' TypeId]

ClassType

ClassType ::= '**CLASS**' [ClassHeritage] { ClassMember } * [EndClause]

ClassVisibility

ClassVisibility ::= { '**PRIVATE**' | '**PROTECTED**' | '**PUBLIC**' | '**PUBLISHED**' }

CompoundStmt

CompoundStmt ::= '**BEGIN**' { ';' } * [StatementList] '**END**'

ConditionalStmt

ConditionalStmt ::= CaseStatement
ConditionalStmt ::= IfStatement

ConstExpr

ConstExpr ::= Expression

ConstSection

ConstSection ::= '**CONST**' { ConstantDecl } + BY ';' ';'

ConstantDecl

ConstantDecl ::= Ident '**:**' Type '**=**' TypedConstant
ConstantDecl ::= Ident '**=**' TypedConstant

ConstructorHeading

ConstructorHeading ::= '**CONSTRUCTOR**' QualId [FormalParameters] [';' { Directive } + BY ';']

ContainsClause

ContainsClause ::= '**CONTAINS**' { Ident } + BY ',' ';'

DeclSection

DeclSection ::= ConstSection
DeclSection ::= LabelDeclSection

DeclSection ::= ProcedureDeclSection
DeclSection ::= ResourceStringSection
DeclSection ::= TypeSection
DeclSection ::= VarSection

DelphiFile

DelphiFile ::= Library
DelphiFile ::= Package
DelphiFile ::= Program
DelphiFile ::= Unit

Designator

Designator ::= BracketDesignator
Designator ::= CaretDesignator
Designator ::= Cast
Designator ::= DotDesignator
Designator ::= FunctionCall
Designator ::= InheritedCall
Designator ::= PExpression
Designator ::= QualId
Designator ::= WriteLnCall

DestructorHeading

DestructorHeading ::= '**DESTRUCTOR**' QualId [FormalParameters] [';' { Directive } + BY ';']

Directive

Directive ::= DispIdDirective
Directive ::= ExternalDirective
Directive ::= MessageDirective
Directive ::= PlainDirective

DispIdDirective

DispIdDirective ::= '**DISPID**' Expression

DotDesignator

DotDesignator ::= Designator '**.**' Ident

EndClause

EndClause ::= '**END**'

EnumeratedType

EnumeratedType ::= '**(**' { EnumeratedTypeElem } + BY ',' '**)**'

EnumeratedTypeElem

EnumeratedTypeElem ::= Ident ['**=**' Expression]

ExceptionBlock

ExceptionBlock ::= StatementList
ExceptionBlock ::= { ExceptionHandler } + BY ';' [';' '**ELSE**' [StatementList]]

ExceptionHandler

ExceptionHandler ::= '**ON**' [Ident '**:**'] Ident '**DO**' [Statement]

ExitStatement

ExitStatement ::= '**EXIT**'

ExportedHeading

ExportedHeading ::= { ProcedureHeading | FunctionHeading } ';' { Directive } * BY ';'

Expression

Expression ::= BinaryExpression
Expression ::= Designator
Expression ::= Literal
Expression ::= SetConstructor
Expression ::= TypeInfo
Expression ::= TypedConstExpr
Expression ::= UnaryExpression

ExternalBlock

ExternalBlock ::= '**EXTERNAL**' { StringLiteral | Ident } ['**INDEX**' Expression]

ExternalDirective

ExternalDirective ::= '**EXTERNAL**' { StringLiteral | Ident } Ident Expression

FieldDecl

FieldDecl ::= { Ident } + BY ',' '**:**' Type

FieldList

FieldList ::= VariantSection '**;**'
FieldList ::= { FieldDecl } + BY ';' [';'] [VariantSection '**;**']

FileType

FileType ::= ['**PACKED**'] '**FILE**' ['**OF**' TypeId]

ForStatement

ForStatement ::= **'FOR'** QualId **':='** Expression { **'DOWNTO'** | **'TO'** } Expression **'DO'** Statement

FormalParameters

FormalParameters ::= **'('** { FormalParm } * BY **';'** **')'**

FormalParm

FormalParm ::= [**'CONST'** | **'OUT'** | **'VAR'**] Parameter

FullQualId

FullQualId ::= Ident **'.'** Ident

FunctionCall

FunctionCall ::= Designator **'('** { Expression } * BY **','** **')'**

FunctionHeading

FunctionHeading ::= [**'CLASS'**] **'FUNCTION'** QualId [FormalParameters] [**':'** FunctionReturnType] [[';'] { Directive } + BY ';']

FunctionReturnType

FunctionReturnType ::= Type
FunctionReturnType ::= **'STRING'**

FunctionTypeHeading

FunctionTypeHeading ::= **'FUNCTION'** [QualId] [FormalParameters] **':'** FunctionReturnType [[';'] { Directive } + BY **';'**]

GotoStatement

GotoStatement ::= **'GOTO'** LabelId

Heritage

Heritage ::= ClassHeritage
Heritage ::= InterfaceHeritage
Heritage ::= ObjHeritage

Ident

Ident ::= **DataIdentToken**

IfStatement

IfStatement ::= '**IF**' Expression '**THEN**' [Statement] ['**ELSE**' [Statement]]

ImplementationSection

ImplementationSection ::= '**IMPLEMENTATION**' [UsesClause] { DeclSection } *

InheritedCall

InheritedCall ::= '**INHERITED**' [Designator]

InitSection

InitSection ::= '**BEGIN**' [StatementList] EndClause
InitSection ::= '**INITIALIZATION**' [StatementList] ['**FINALIZATION**' StatementList] EndClause
InitSection ::= EndClause

InlineElement

InlineElement ::= Ident { InlineOffset } +
InlineElement ::= Literal
InlineElement ::= { '**>**' | '**<**' } Literal

InlineOffset

InlineOffset ::= { '**+**' | '**-**' } Literal

InlineStatement

InlineStatement ::= '**INLINE**' '**(**' { InlineElement } + BY '/' '**)**'

InterfaceDecl

InterfaceDecl ::= { ConstSection | TypeSection | VarSection | ResourceStringSection | ExportedHeading }

InterfaceHeritage

InterfaceHeritage ::= '**(**' { QualId } + BY ',' '**)**'

InterfaceSection

InterfaceSection ::= '**INTERFACE**' [UsesClause] { InterfaceDecl } *

InterfaceType

InterfaceType ::= { '**INTERFACE**' | '**DISPINTERFACE**' } [InterfaceHeritage] ['**['{[^}]+}']**'] [ClassMethodList] [ClassPropertyList] ['**END**']

LabelDeclSection

LabelDeclSection ::= '**LABEL**' { LabelId } + BY ',' ';'

LabelId

LabelId ::= { **DataIdentToken** | **LabelIdentToken** }

LabeledStatement

LabeledStatement ::= LabelId ':' [Statement]

Library

Library ::= '**LIBRARY**' Ident ';' ProgramBlock '.'

Literal

Literal ::= BooleanLiteral
Literal ::= Nil
Literal ::= Number
Literal ::= StringLiteral

LoopStmt

LoopStmt ::= ForStatement
LoopStmt ::= RepeatStatement
LoopStmt ::= WhileStatement

MessageDirective

MessageDirective ::= '**MESSAGE**' Expression

MethodHeading

MethodHeading ::= ConstructorHeading
MethodHeading ::= DestructorHeading
MethodHeading ::= FunctionHeading
MethodHeading ::= ProcedureHeading

MethodList

MethodList ::= { MethodHeading } + BY ';'

Nil

Nil ::= '**NIL**'

Number

Number ::= **AsmNumber**
Number ::= **NumberToken**
Number ::= **RealNumberToken**

ObjFieldList

ObjFieldList ::= { ObjFieldListElem } +

ObjFieldListElem

ObjFieldListElem ::= { Ident } + BY ',' ':' Type ';'

ObjHeritage

ObjHeritage ::= '**(**' { QualId } + BY ',' '**)**'

ObjectType

ObjectType ::= '**OBJECT**' [ObjHeritage] [ObjFieldList] [MethodList] EndClause

OrdIdent

OrdIdent ::= { '**BOOLEAN**' | **ByteKeyword** | '**CHAR**' | '**INT64**' | '**LONGINT**' | '**LONGWORD**' | '**PCHAR**' | '**SHORTINT**' | '**SMALLINT**' | '**WIDECHAR**' | '**WORD**' | '**INTEGER**' }

OrdinalType

OrdinalType ::= EnumeratedType
OrdinalType ::= OrdIdent
OrdinalType ::= SubrangeType

PExpression

PExpression ::= '**(**' Expression '**)**'

Package

Package ::= '**PACKAGE**' Ident '**;**' [RequiresClause] [ContainsClause] EndClause '**.**'

Parameter

Parameter ::= { Ident } + BY ',' '**:**' ParameterType '**=**' Expression
Parameter ::= { Ident } + BY ',' ['**:**' ParameterType]

ParameterType

ParameterType ::= Type
ParameterType ::= '**ARRAY**' '**OF**' SimpleType
ParameterType ::= '**ARRAY**' '**OF**' '**CONST**'

ParameterType ::= { '**FILE**' | '**STRING**' }

PlainDirective

PlainDirective ::= { '**ABSTRACT**' | '**ASSEMBLER**' | '**CDECL**' | '**DYNAMIC**' | '**EXPORT**' | '**FAR**' | '**FORWARD**' | '**MESSAGE**' | '**OVERRIDE**' | '**OVERLOAD**' | '**PASCAL**' | '**REGISTER**' | '**REINTRODUCE**' | '**SAFECALL**' | '**STDCALL**' | '**VIRTUAL**' }

PlainStatement

PlainStatement ::= Designator

PointerType

PointerType ::= '**^**' TypeId

ProcedureDecl

ProcedureDecl ::= MethodHeading '**;**' Block '**;**'

ProcedureDeclSection

ProcedureDeclSection ::= ProcedureDecl

ProcedureHeading

ProcedureHeading ::= ['**CLASS**'] '**PROCEDURE**' QualId [FormalParameters] [[';'] { Directive } + BY ';']

ProcedureType

ProcedureType ::= { ProcedureTypeHeading | FunctionTypeHeading } ['**OF**' '**OBJECT**'] [[';'] { Directive } + BY '**;**']

ProcedureTypeHeading

ProcedureTypeHeading ::= '**PROCEDURE**' [QualId] [FormalParameters] [[';'] { Directive } + BY ';']

Program

Program ::= ['**PROGRAM**' Ident ['**(**' { Ident } + BY ',' '**)**'] ';'] ProgramBlock '**.**'

ProgramBlock

ProgramBlock ::= [UsesClause] Block

PropertyInterface

PropertyInterface ::= [PropertyParameterList] **':'** { StringType | TypeId }

PropertyList

PropertyList ::= **'PROPERTY'** Ident [PropertyInterface] [PropertySpecifiers]

PropertyParameterList

PropertyParameterList ::= **'['** { PropertyParameterListElem } + BY **';'** **']'**

PropertyParameterListElem

PropertyParameterListElem ::= [**'CONST'**] { Ident } + BY **','** **':'** TypeId

PropertySpecifiers

PropertySpecifiers ::= { **'INDEX'** Expression | **'READ'** QualId | **'WRITE'** QualId | **'STORED'** [Ident | Expression] | **';'** **'DEFAULT'** | **'DEFAULT'** Expression | **'NODEFAULT'** | **'READONLY'** | **'WRITEONLY'** | **'IMPLEMENTS'** TypeId | **'DISPID'** Expression } +

ProtectedBlock

ProtectedBlock ::= **'TRY'** [StatementList] [**';'**] **'EXCEPT'** [ExceptionBlock] [**';'**] **'END'**
ProtectedBlock ::= **'TRY'** [StatementList] [**';'**] **'FINALLY'** [StatementList] [**';'**] **'END'**

QualId

QualId ::= FullQualId
QualId ::= Ident

RaiseStatement

RaiseStatement ::= **'RAISE'** [Designator]

RealType

RealType ::= { **'COMP'** | **'CURRENCY'** | **'DOUBLE'** | **'EXTENDED'** | **'REAL'** | **'REAL48'** | **'SINGLE'** }

RecType

RecType ::= [**'PACKED'**] **'RECORD'** [FieldList] **'END'**

RecVariant

RecVariant ::= { Expression } + BY **','** **':'** **'('** [FieldList] **')'**

RecordConstant

RecordConstant ::= '**(**' { RecordFieldConstant } + BY '**;**' '**)**'

RecordFieldConstant

RecordFieldConstant ::= Ident '**:**' TypedConstant

RegularBlock

RegularBlock ::= { DeclSection } * CompoundStmt

RepeatStatement

RepeatStatement ::= '**REPEAT**' StatementList '**UNTIL**' Expression

RequiresClause

RequiresClause ::= '**REQUIRES**' { Ident } + BY '**,**' '**;**'

ResourceStringSection

ResourceStringSection ::= '**RESOURCESTRING**' { ConstantDecl } + BY '**;**' '**;**'

RestrictedType

RestrictedType ::= ClassType
RestrictedType ::= InterfaceType
RestrictedType ::= ObjectType

SetConstructor

SetConstructor ::= '**[**' { SetElement } * BY '**,**' '**]**'

SetElement

SetElement ::= Expression ['**..**' Expression]

SetType

SetType ::= ['**PACKED**'] '**SET**' '**OF**' OrdinalType
SetType ::= ['**PACKED**'] '**SET**' '**OF**' TypeId

SimpleStatement

SimpleStatement ::= AsmStatement
SimpleStatement ::= AssignStatement
SimpleStatement ::= ExitStatement
SimpleStatement ::= GotoStatement
SimpleStatement ::= RaiseStatement

SimpleType

SimpleType ::= OrdinalType
SimpleType ::= RealType

Statement

Statement ::= AsmBlockStmt
Statement ::= InlineStatement
Statement ::= LabeledStatement
Statement ::= PlainStatement
Statement ::= SimpleStatement
Statement ::= StructStmt

StatementList

StatementList ::= { Statement } + BY { ';' } + { ';' } *

StringLiteral

StringLiteral ::= **StringToken**

StringType

StringType ::= ['**TYPE**'] { '**ANSISTRING**' | '**STRING**' | '**WIDESTRING**' } ['**[**' Expression '**]**']

StructStmt

StructStmt ::= CompoundStmt
StructStmt ::= ConditionalStmt
StructStmt ::= LoopStmt
StructStmt ::= ProtectedBlock
StructStmt ::= WithStmt

StructType

StructType ::= ArrayType
StructType ::= FileType
StructType ::= RecType
StructType ::= SetType

SubrangeType

SubrangeType ::= Expression '**..**' Expression

Type

Type ::= ClassRefType
Type ::= PointerType
Type ::= ProcedureType
Type ::= SimpleType
Type ::= StringType
Type ::= StructType

Type ::= TypeId
Type ::= VariantType

TypeDecl

TypeDecl ::= Ident '**=**' ['**TYPE**'] { Type | RestrictedType }

TypeId

TypeId ::= [Ident '**.**'] Ident

TypeInfo

TypeInfo ::= '**TYPEINFO**' '**(**' { TypeId | StringType } '**)**'

TypeSection

TypeSection ::= '**TYPE**' { TypeDecl } * BY ';' ';'

TypedConstExpr

TypedConstExpr ::= { ArrayConstant | RecordConstant }

TypedConstant

TypedConstant ::= ArrayConstant
TypedConstant ::= ConstExpr
TypedConstant ::= RecordConstant

UnaryExpression

UnaryExpression ::= { '**NOT**' | '**+**' | '**-**' | '**@**' } Expression

Unit

Unit ::= '**UNIT**' Ident '**;**' InterfaceSection ImplementationSection InitSection '**.**'

UsesClause

UsesClause ::= '**USES**' { Ident } + BY ',' ';'

VarDecl

VarDecl ::= { Ident } + BY ',' '**:**' Type ['**ABSOLUTE**' { Ident | Expression } | '**=**' TypedConstant]

VarSection

VarSection ::= { '**VAR**' | '**THREADVAR**' } { VarDecl } + BY '**;**' ';'

VariantSection

VariantSection ::= '**CASE**' [Ident '**:**'] TypeId '**OF**' { RecVariant } + BY '**;**'

VariantType
VariantType ::= { '**OLEVARIANT**' | '**VARIANT**' }

WhileStatement

WhileStatement ::= '**WHILE**' Expression '**DO**' [Statement]

WithStmt

WithStmt ::= '**WITH**' { Designator } + BY '**,**' '**DO**' Statement

WriteArgument

WriteArgument ::= Expression ['**:**' Expression ['**:**' Expression]]

WriteLnCall

WriteLnCall ::= { '**WRITE**' | '**WRITELN**' } '**(**' { WriteArgument } * BY '**,**' '**)**'

b) Liste des propriétés affectées aux nœuds lors de la phase d'analyse

Voici la liste des différentes propriétés qui sont affectées aux nœuds durant la phase d'analyse. Je les aie regroupées en fonctions du fichier ou elles sont affectées. Pour chaque propriété, je vais donner le nom de l'annotation à utiliser, le type de nœud concerné (en utilisant le nom de la classe à laquelle appartient le nœud utilisé par RainCode) ainsi que le type d'information que l'on trouve à l'intérieur (toujours en utilisant le nom du type utilisé par RainCode).

DecoreDelphiFiles.rcs

["uses"]

type de noeud affecté : DelphiFile

type de l'information contenue : Liste d'ident.

Description : Ensemble des fichiers présents dans la clause uses de chaque fichier, que ce soit dans la partie interface ou dans la partie implémentation.

["methodDecls"]

type de nœud affecté : DelphiFile

type de l'information contenue : Liste d' ExportedHeading.

Description : Liste des fonctions déclarées dans la partie interface. Ne sont reprises que les méthodes n'appartenant pas à une classe.

["classe"]

type de nœud affecté : DelphiFile

type de l'information contenue : Liste des classes déclarées dans la partie interface.

Description : Liste de TypeDecl dont le type est un ClassType.

["classeDecls"]

type de nœud affecté : DelphiFile

type de l'information contenue : Liste de TypeDecl dont le type est un ClassType.

Description : Ensemble des déclarations de classe de type *a = class; b = class(c);...*

["methodImpl"]

type de nœud affecté : DelphiFile

type de l'information contenue : Liste de ProcedureDeclSection.

Description : Liste de toutes les fonctions définies dans la partie implémentation. Les fonctions appartenant à des classes sont comprises ici.

["vars"]

type de nœud affecté : DelphiFile

type de l'information contenue : Liste d'ident.

Description : Liste des variables globales définies dans la partie implémentation ou dans la partie interface.

DecoreClasses.rcs

["parent"]

type de nœud affecté : TypeDecl dont le type est un ClassType.

type de l'information contenue : 1 elelment de type QualId

Description : Classe parente de la classe en cours.

["interfaces"]

type de nœud affecté : TypeDecl dont le type est un ClassType.

type de l'information contenue : Liste de Qualid.

Description : Interfaces incluses dans la classe en cours.

["methodImpl"]

type de nœud affecté : TypeDecl dont le type est un ClassType.

type de l'information contenue : Liste de ProcedureDecl.

Description : Ensemble des méthodes de la classe de la partie implémentation. Cela comprend toutes les méthodes dont le UnitId est le nom de la classe.

["fields"]

type de nœud affecté : TypeDecl dont le type est un ClassType.

type de l'information contenue : Liste d'id.

Description : Champs déclarés dans la partie interface de la classe.

["methodDecls"]

type de nœud affecté : TypeDecl dont le type est un ClassType.

type de l'information contenue : Liste de MethodHeading.

Description : Listes des méthodes des classes déclarées dans la partie interface.

["properties"]

type de nœud affecté : TypeDecl dont le type est un ClassType.

type de l'information contenue : Liste de PropertyList.

Description : Propriétés de la classe.

["type_ids"]

type de nœud affecté : TypeDecl dont le type est un ClassType.

type de l'information contenue : Liste d'id

Description : Ensemble des types rencontrés dans la classe, que ce soit dans la déclaration ou l'implémentation des variables et/ou des champs.

["proc_types"]

type de nœud affecté : TypeDecl dont le type est un ClassType.

type de l'information contenue :

Description : Ensemble des déclarations de types de fonctions.

["record_type"]

type de nœud affecté : TypeDecl dont le type est un ClassType.

type de l'information contenue :

Description : Ensemble des déclarations rencontrées de type enregistrement.

["class_parent_call"]

type de nœud affecté : TypeDecl dont le type est un ClassType.

type de l'information contenue : Liste d' ident.

Description : Appel des méthodes héritées de la classe parente. Ne vaut que si la fonction appartient à une classe, laquelle doit hériter d'une autre classe, cette dernière étant présente dans les fichiers analysés.

["class_extern_call"]

type de nœud affecté : TypeDecl dont le type est un ClassType.

type de l'information contenue :

Description : Nom des autres classes accédées. La classe parente n'est prise en compte que si son nom est mentionné explicitement.

["extern_calls"]

type de nœud affecté : TypeDecl dont le type est un ClassType.

type de l'information contenue :

Description : Noms des fonctions accédées qui n'appartiennent pas à une classe.

["symbols"]

type de nœud affecté : TypeDecl dont le type est un ClassType.

type de l’information contenue :

Description : Ensembles des symboles rencontrés inconnus au moment de l'analyse. Il peut s'agir de types (classe, enregistrement ou autre), de constantes, ...

DecoreMethodes.rcs

["cycl_cplx"]

type de nœud affecté : ProcedureDecl.

type de l’information contenue : Nombre entier.

Description : Complexité cyclomatique de la fonction.

["readvar"]

type de nœud affecté : ProcedureDecl.

type de l’information contenue : Liste d'Id.

Description : Noms des variables susceptibles d'être lus dans la fonction courante.

["writevar"]

type de nœud affecté : ProcedureDecl.

type de l’information contenue : Liste d'Id.

Description : Variables qui sont dans tous les cas accédées en écriture avant d'être accédées en lecture.

["params"]

type de nœud affecté : ProcedureDecl.

type de l’information contenue : Liste d'Id.

Description : Liste des paramètres de la fonction.

["var"]

type de nœud affecté : ProcedureDecl.

type de l’information contenue : Liste d'Id.

Description : Ensemble des variables locales de la fonction.

["container"]

type de nœud affecté : ProcedureDecl.

type de l’information contenue : Nœud de type ProcedureDecl

Description : Méthode dans laquelle est déclarée la méthode courante (dans le cas de méthodes imbriquées).

["declared_methods"]

type de nœud affecté : ProcedureDecl.

type de l’information contenue : Liste de ProcedureDecl

Description : Liste des methodes imbriquées dans la méthode courante.

["declared_types"]

type de nœud affecté : ProcedureDecl.

type de l’information contenue : Liste de TypeDecl.

Description : Liste des types imbriqués dans la méthode courante.

["with"]

type de nœud affecté : ProcedureDecl.

type de l'information contenue : Liste de WithStatement

Description : Endroits où est utilisé le mot-clef « with ».

["case"]

type de nœud affecté : ProcedureDecl.

type de l'information contenue : Liste de CaseStatement.

Description : Endroit où l'on retrouve des case .. of ... .

["type_ids"]

type de nœud affecté : ProcedureDecl.

type de l'information contenue : Liste d'id.

Description : Ensemble des types rencontrés dans l'implémentation de la méthode.

["proc_types"]

type de nœud affecté : ProcedureDecl.

type de l'information contenue : Liste de ProcedureType.

Description : Type de procédure rencontrés/déclarés dans la méthode (pas utilisé).

["record_type"]

type de nœud affecté : ProcedureDecl.

type de l'information contenue : Liste de RecType.

Description : Types enregistrement rencontrés dans la méthode. (pas utilisé)

["return_type"]

type de nœud affecté : ProcedureDecl.

type de l'information contenue : TypeId.

Description : Type de retour de la fonction. Rmq: si le type de retour n'est pas un TypeId , celui-ci n'est pas pris en compte.

["class_intern_call"]

type de nœud affecté : ProcedureDecl.

type de l'information contenue : Liste d' Id.

Description : Ensemble des fonctions appartenant à la classe de la méthode appelée à partir de la méthode courante.

["class_parent_call"]

type de nœud affecté : ProcedureDecl.

type de l'information contenue : Liste d'Id.

Description : Ensemble des fonctions de la classe parente utilisées à l'aide du mot-clef inherited.

["class_extern_call"]

type de nœud affecté : ProcedureDecl.

type de l’information contenue :

Description : Ensemble des appels vers une autre classe. Non implémenté pour le moment (nécessite résolution de type)

["extern_calls"]

type de nœud affecté : ProcedureDecl.

type de l’information contenue :

Description : Méthodes appelées à partir de la méthode courante, lesquelles n'appartiennent à aucune classe.

["read_symbol"]

type de nœud affecté : ProcedureDecl.

type de l’information contenue : Liste d'Id.

Description : Symboles inconnus rencontrés dans la méthode. Il peut s'agir de variables déclarées dans un autre fichier, ou encore de contantes,... Ceux-ci sont accédés en lecture.

["write_symbol"]

type de nœud affecté : ProcedureDecl.

type de l’information contenue : Liste d'Id.

Description : Symboles inconnus rencontrés dans la méthode. Il peut s'agir de variables déclarées dans un autre fichier, ou encore de contantes,... Ceux-ci sont accédés en écriture avant d'être accédé en lecture.

["is_getter"]

type de nœud affecté : ProcedureDecl.

type de l’information contenue : Booléen (TRUE/FALSE).

Description : Indique si la méthode sert de getter.

["is_setter"]

type de nœud affecté : ProcedureDecl.

type de l’information contenue : Booléen (TRUE/FALSE).

Description : Indique si la méthode sert de setter.

["is_relay"]

type de nœud affecté : ProcedureDecl.

type de l’information contenue : Booléen (TRUE/FALSE).

Description : Indique si la méthode se contente d'appeler une autre méthode qui fait le travail à sa place.

["classe"]

type de nœud affecté : ProcedureDecl.

type de l’information contenue : TypeDecl dont le type est un ClassType.

Description : Refercence vers la classe dans laquelle est déclarée la méthode.

["declarator"]

type de nœud affecté : ProcedureDecl.

type de l’information contenue : MethodHeading.

Description : Reference vers l'endroit ou la méthode est déclarée dans la classe.

FindSmells.rcs

classe["type_link"]

type de nœud affecté : TypeDecl dont le type est un ClassType

type de l’information contenue : Liste de TypeDecl dont le type est un ClassType.

Description : Ensemble des autres classes auxquelles fait référence la classe courante.

classe["externdecls"]

type de nœud affecté : TypeDecl dont le type est un ClassType

type de l’information contenue : Liste de TypeDecl dont le type est un ClassType.

Description :Ensemble des autres classes qui font références à la classe courante.

classe["abstract"]

type de nœud affecté : TypeDecl dont le type est un ClassType

type de l’information contenue : Liste de MethodHeading.

Description : Ensemble des fonctions abstraites de la classe.

classe["virtual"]

type de nœud affecté : TypeDecl dont le type est un ClassType

type de l’information contenue : Liste de MethodHeading.

Description : Ensemble des fonctions virtuelles/dynamiques de la classe.

classe["overriders"]

type de nœud affecté : TypeDecl dont le type est un ClassType

type de l’information contenue : Liste de MethodHeading.

Description : Ensemble des fonctions de la classe marquées par la directive override.

method["interndecls"]

type de nœud affecté : ProcedureDecl

type de l’information contenue : Liste de ProcedureDeclSection.

Description : Ensemble des méthodes appartenant à la même classe que la méthode courante qui font référence à cette méthode courante.

racine["unused_var"]

type de nœud affecté : DelphiFile.

type de l’information contenue : Liste d'id.

Description : Variables globales inutilisées dans le fichier courant (sont peut-être utilisée ailleurs).

c) Utilisation de l'algorithme Apriori pour le Data Clump.

L'algorithme Apriori est un algorithme généralement utilisé pour faire du data mining, et plus précisément dans les problèmes d'itemset mining[15]. Ce type de problèmes revient à rechercher par exemple parmi un ensemble d'articles de magasins lesquels sont souvent achetés ensemble. Pour trouver la solution, l'algorithme Apriori utilise la notion d'ensembles fréquents (qui sont en fait les articles achetés ensemble), et cherche à chaque itération à partir des ensembles fréquents de taille n à créer des ensembles de taille n+1. Ensuite il y a une phase de sélection qui sert à déterminer à partir des ensembles de taille n+1 lesquels sont fréquents. Dans le cas des articles de magasins par exemple, on regarde dans chaque transaction quels sont les ensembles qui apparaissent. Les autres sont mis de coté. Une fois que cela est fait, une nouvelle itération commence afin de trouver les ensembles fréquents de taille (n+1) +1 = n+2.Pour pouvoir fonctionner, cet algorithme a besoin des éléments suivants :

- Un ensemble d'items dont on désire savoir lesquels sont corrélés.
- Un ensemble de transactions opérant sur ces différents items.

Si je reprends l'exemple des articles en magasins, les items sont les différents articles, tandis que les transactions sont les achats effectués par chaque client.

Maintenant revenons à notre problème. Ce que l'on veut c'est trouver quelles classes sont souvent utilisées ensembles. Il en découle que dans notre cas les items seront les différentes classes du projet. Reste à déterminer quels ensembles faut-il prendre comme transactions. Il apparaît fort logiquement que les transactions seront prises comme étant les différentes classes accédées à partir de chacune d'entre elles. Si l'on essaie d'écrire un tel algorithme, on obtient alors quelque chose comme ceci :

```
Apriori_Clump
C1 ← toutes les classes
k ← 1
tant que Ck != 0
        pour toutes les classes X                  //Calcule le support de chaque groupe de classes
                pour tous Y Є Ck
                        si Y  C  {classes accédées depuis X}
                                Y.support ← Y.support +1
                                Fin si
                        Fin pour
                Fin pour
        Fk ← { Y Є Ck  tel que Y.support > σ}      //Retient ceux qui ont un support suffisant
        Apriori_gen(Fk)                            //Génère les candidats de taille k+1
        k++                                        //passe au niveau suivant
        fin tant que
retourner tous les Fk

Apriori_gen(Fk)
Pour tout   X, Y  Є Fk tel que X[i] == Y[i] (1≤ i ≤k-1) ET  X[k] != Y[k]      //Pour tous les groupes ne
                                                                              //différant que d'une classe
        I ← X U { Y[k] }                           // On crée un groupe de taille k+1
        Si ( pour tout J C I, |J| =k ➔ J  Є Fk)    // Si tous ses sous-ensembles sont fréquents
                Ck+1 ← Ck+1 U I                    //alors il fait partie des candidats de taille k+1
        Fin si
Fin pour tout
```

d) Rappels concernant les techniques d'analyse statistique.

Introduction

Les techniques statistiques sont utilisées lorsque l'on a à faire à un grand nombre de données et que l'on essaie de voir les relations qu'il y a entre elles. L'analyse de correspondances fait en fait partie de ce que l'on appelle les méthodes factorielles. Le but de ces méthodes est de diminuer le nombre de données à représenter. En fait :
On part d'un tableau contenant les différentes données à analyser :

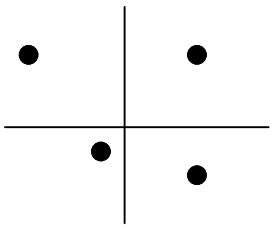
Variables / Individus	1	2	…	j	…	p
1						
…						
i				$\mathbf{x_{ij}}$		
…						
n						

Ces données peuvent être vu de la façon suivante :

- Si l'on prend tous les x_{ij} pour un i donné, alors on obtient un point représentable dans un espace à p dimensions. En répétant l'opération pour tous les i tel que $1 \leq i \leq n$; on parvient alors à représenter l'ensemble des données dans un espace à p dimensions. ➔ R^p (ou espace des variables)
- Si l'on prend tous les x_{ij} pour un j donné, alors on obtient un point représentable dans un espace à n dimensions. En répétant l'opération pour tous les j tel que $1 \leq j \leq p$; on parvient alors à représenter l'ensemble des données dans un espace à n dimensions. ➔ R^n (ou espace des individus)

Analyse de correspondances

L'analyse de correspondances permet d'analyser les relations qu'il y a entre les lignes et les colonnes du tableau. La méthode essaie de construire un tableau en partant de l'hypothèse que les données en lignes et les données en colonnes sont indépendantes. Ensuite, la technique consiste à construire des facteurs en mesurant la distance entre les données théoriques et les données dont l'on dispose. Puis il est possible de représenter les données en ligne et en colonne à partir de ces facteurs.
Au niveau de la représentation, on représente les données sur un graphe munis d'axes x et y.



Ensuite, on y place les différents points. Cependant, il faut faire attention que dans une telle représentation car les échelles utilisées pour les lignes et les colonnes sont différentes. Néanmoins, en regardant la position des différents points, il est possible de voir s'il y a une correspondance entre ces données.

Les techniques de classification

Les techniques de classification fonctionnent d'une manière assez différente de celle des méthodes factorielles. Alors que ces dernières cherchent à représenter l'ensemble des données sous formes de points dans un ensemble à 2 ou 3 dimensions, les méthodes de classification cherchent à les regrouper, chaque groupe contenant des données très proches.
Pour cela, les techniques de classification ne se basent pas sur le tableau reprenant les variables et les individus mais sur une matrice reprenant les distances soit entre toutes les variables, soit entre tous les individus. Cela a comme conséquence qu'il n'est pas possible en utilisant ces techniques de pouvoir représenter sur un même schéma à la fois les variables et les individus.
Il existe 2 grand groupes de techniques de classification: celles consistant à partitionner les différentes données en un certain nombre de groupe (comme k-means par exemple), et celles permettant de former une hiérarchie de groupes représentables sous forme de dendrogrammes.

Dans ce travail, j'ai utilisé une classification hiérarchique dite ascendante. Le principe de ce type de classification est de partir des données de départ. On considère chaque point, chaque élément comme étant un groupe. Ensuite on va appliquer de façon itérative un algorithme qui va regrouper 2 ensemble afin de n'en former plus qu'un seul. En général, cet algorithme va regrouper les 2 ensembles les plus proches selon une métrique donnée. Cette métrique peut varier. On peut par exemple considérer que la distance entre 2 groupes de points est en fait la distance minimale entre 1 point du premier et un point du second groupe (critère de saut minimum : D(AB) = Min(D(x,y)) avec x Є A et y Є B) , ou encore prendre la distance maximale (critère de saut maximal). Parfois l'on décide aussi d'employer comme distance la distance moyenne entre les 2 groupes. D'autres critères plus compliqués existent, notamment le critère de ward généralisé, qui va agréger différents groupes de sorte que la perte d'inertie liée à cette agrégation soit minimale. Cela a pour conséquence de limiter les effets dits de chaînage entre les différents groupes.
Notons qu'il existe d'autres techniques d'agrégation plus « algorithmiques » visant par exemple à obtenir un arbre qui soit le plus petit possible (algorithme de Prim ou encore de Kruskal).