

# **Visualisation des patterns d'héritage à l'aide de l'analyse dynamique**

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

par :

**Michael CANEDO BLANCO**

Conseiller au travail de Bachelor :

**Philippe DUGERDIL, professeur HES**

**Genève, le 23 août 2013**

**Haute École de Gestion de Genève (HEG-GE)**

**Filière Informatique de gestion**

## Déclaration

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

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

Fait à Genève, le 23 août 2013

Michael CANEDO BLANCO

## Remerciements

Je tiens à remercier Monsieur Philippe DUGERDIL pour son soutien et son suivi dans la réalisation de ce travail de Bachelor, spécialement pour sa qualité d'encadrement sans laquelle l'accomplissement de ce travail aurait été plus difficile.

J'aimerais remercier également Monsieur Mihnea NICULESCU qui m'a beaucoup soutenu et aidé tout au long de ce projet. Il m'a apporté ses conseils et a toujours fait preuve d'une grande disponibilité.

Je souhaite également remercier toutes les personnes qui m'ont soutenu tout au long de la réalisation de ce travail, plus particulièrement Monsieur Simon Grandjean et Monsieur Corentin Simon.

## Résumé

La maintenance des logiciels coûte très chère. Les logiciels se voient modifiés au fil du temps pour s'adapter aux besoins métiers qui ne cessent d'évoluer. Pour se faire, des modifications doivent être apportées aux logiciels mis à jour. Au cours du temps, ces modifications risquent d'impacter l'architecture logicielle en augmentant sa complexité. Ceci risque de rendre l'architecture inappropriée pour les développeurs qui auront des difficultés de compréhension.

Le software engineering est devenu l'un des outils les plus utilisés par la maintenance pour comprendre les systèmes maintenus. Les développeurs logiciels font recourt à des analyses statiques, en se concentrant sur l'architecture implémentée. Toutefois, la plupart du temps, ils ne tiennent pas compte de la réelle utilisation de cette architecture et se contentant de l'analyser à l'arrêt.

L'analyse dynamique est une approche qui permet de se concentrer sur l'architecture vraiment requise lors de l'exécution d'une fonctionnalité. La différence avec l'analyse statique, c'est qu'on examine les processus en cours de réalisation pour produire une fonctionnalité demandée au système.

Dans ce travail, j'ai été amené à définir et interpréter, pour la maintenance, une métrique d'héritage dynamique permettant de connaître l'héritage réel utilisé (et donc nécessaire) à l'exécution d'une fonctionnalité métier. J'ai donc commencé par effectuer une analyse de l'état de l'art sur la maintenabilité et l'héritage en étudiant les différentes métriques d'héritages existantes.

Après la proposition de cette métrique, la suite de mon mandat s'est concentré sur la représentation visuelle de celle-ci afin de faciliter son interprétation. Pour finir, ce travail s'est terminé par l'évaluation pratique de cette métrique sur l'architecture dynamique d'ArgoUML, un logiciel open source de taille moyenne.

# Table des matières

Déclaration.....	1
Remerciements .....	2
Résumé .....	3
Liste des tableaux .....	5
Liste des figures.....	6
1. Introduction.....	7
2. Synthèse des travaux analysés.....	8
2.1 Résultats empiriques .....	8
2.2 Visualisations .....	9
2.3 Mesures de l'héritage.....	12
3. Taux d'héritage dynamique (THD).....	32
3.1 Définition du THD.....	32
3.2 Motivation .....	36
3.3 Visualisation du THD .....	38
3.4 Cartes THD .....	42
3.5 Mesurer le THD.....	45
3.6 Interprétations .....	47
4. Mise en œuvre de la métrique THD.....	57
4.1 Evaluation des classes intermédiaires dans une hiérarchie.....	60
5. Analyse pratique sur ArgoUML .....	62
5.1 Scénarios.....	62
5.2 Statistiques générales .....	63
5.3 Analyse des cartes THD (macro).....	65
5.4 Analyse des cartes THD (micro).....	69
5.5 Remarque .....	78
6. Travail futur .....	79
7. Conclusion .....	80
8. Acronymes .....	81
Bibliographie .....	82
Annexes .....	84

## Liste des tableaux

Tableau 1 : Légende de la visualisation « Sunburst ».....	10
Tableau 2 : Evaluations du THD pour la figure 17 .....	33
Tableau 3 : Evaluations du THD pour la figure 18 (1/2) .....	34
Tableau 4 : Evaluations du THD pour la figure 18 (2/2) .....	35
Tableau 5 : Interprétations selon la valeur THD.....	39
Tableau 6 : Evaluations du THD pour la figure 23 .....	45
Tableau 7 : Evaluations du THD pour la figure 24 .....	46
Tableau 8 : Evaluations du THD pour la figure 35 .....	60
Tableau 9 : Scénarios sélectionnés.....	63
Tableau 10 : Informations générales sur les scénarios .....	64
Tableau 11 : Informations générales sur les scénarios .....	64
Tableau 12 : THD des packages utilisant l'héritage dans les différents scénarios .....	66
Tableau 13 : Classes dans le rouge intéressantes .....	71
Tableau 14 : ArgoUML - Evaluation du THD sur la hiérarchie 1.....	73
Tableau 15 : ArgoUML - Evaluation du THD sur la hiérarchie 2.....	75

## Liste des figures

Figure 1 : Exemple de visualisation « Sunburst » .....	9
Figure 2 : Exemple de visualisation « Package Surface Blueprint» (1/2) .....	10
Figure 3 : Exemple de visualisation « Package Surface Blueprint» (2/2) .....	11
Figure 4 : Exemple de hiérarchie DIT et NOC .....	13
Figure 5 : ambiguïtés du DIT .....	13
Figure 6 : Application du PRED() et SUCC() sur une hiérarchie .....	15
Figure 7 : Application de l'AU sur une hiérarchie .....	16
Figure 8 : Application de l'AU et AM sur une hiérarchie .....	17
Figure 9 : AU et AM inutile.....	18
Figure 10 : Hiérarchie avec une bonne abstraction .....	19
Figure 11 : Hiérarchie avec une mauvaise abstraction .....	20
Figure 12 : MIF similaire sur des hiérarchies différentes.....	23
Figure 13 : Application du MRPIR sur une hiérarchie .....	25
Figure 14 : MRPIR similaire sur des hiérarchies différentes .....	26
Figure 15 : Niveaux d'agréations .....	29
Figure 16 : Exemple de polymorphisme .....	30
Figure 17 : Application du THD sur une hiérarchie (1/2) .....	33
Figure 18 : Application du THD sur une hiérarchie (2/2) .....	34
Figure 19 : Comparaison du THD et MIF .....	37
Figure 20 : Code couleur des cartes THD .....	38
Figure 21 : Exemples de carte THD .....	42
Figure 22 : Exemple de visualisation THD et diagramme d'une hiérarchie .....	44
Figure 23 : Exemple de visualisation et évaluation du THD (1/2).....	45
Figure 24 : Exemple de visualisation et évaluation du THD (2/2).....	46
Figure 25 : Avant / Après une maintenance avec un modèle de Figures (1/2) .....	48
Figure 26 : Code des méthodes pour la figure 27 .....	49
Figure 27 : Avant / Après une maintenance avec un modèle de Figures (2/2) .....	50
Figure 28 : Avant / Après une maintenance avec un modèle de Véhicules (1/3) .....	51
Figure 29 : Avant / Après une maintenance avec un modèle de Véhicules (2/3) .....	52
Figure 30 : Avant / Après une maintenance avec un modèle de Véhicules (3/3) .....	53
Figure 31 : Avant / Après une maintenance avec un modèle de Sports (1/2).....	54
Figure 32 : Avant / Après une maintenance avec un modèle de Sports (2/2).....	55
Figure 33 : Exemple de trace pour un appel de méthode .....	58
Figure 34 : Marche à suivre - Evaluer et visualiser le THD sur un programme java .....	59
Figure 35 : Evaluation des sous-structures intermédiaires dans une hiérarchie.....	61
Figure 36 : ArgoUML – Hiérarchie 1 abrégée .....	72
Figure 37 : ArgoUML – Hiérarchie 2 abrégée .....	74

# 1. Introduction

De nos jours, l'architecture est construite en utilisant des tactiques d'architectures permettant d'implémenter les attributs qualités (AQ) requis lors de la construction du système. L'un des AQ phares est la modifiabilité, implémentée pour faciliter la maintenance. En effet, cet objectif est tout à fait compréhensible quand on sait, d'après une étude de Forrester en 2009 [22], que 60 à 77% du budget des départements IT est dépensé dans les maintenances. De plus, environ 40 à 60% du temps de la maintenance est consacré à la simple compréhension du code, ce qui en fait un facteur clef pour réduire le coût des maintenances [23].

Les mesures quantitatives sont essentielles dans toutes les sciences. Les praticiens et théoriciens des sciences informatiques tentent des approches similaires pour le développement de logiciels. Le but étant d'obtenir des mesures objectives sur des propriétés techniques ou fonctionnelles d'un logiciel ceci à titre d'indication de la qualité de conception.

Au vu des nombreux langages de programmation majeurs reposant sur la POO (Java, C++, etc.), il est correct de dire qu'il s'agit du paradigme de programmation le plus utilisé. Parmi les nombreuses caractéristiques de la POO, l'héritage fait partie des plus couramment utilisées dans le développement logiciel. Afin de pouvoir en faire usage, il est nécessaire d'implanter des hiérarchies de classes dans l'architecture du logiciel développé. La structure de ces hiérarchies doit être conçue afin d'éviter d'obscurcir la compréhension et ainsi éviter les statistiques de Forrester. Il est donc intéressant de se focaliser sur l'héritage comme facteur d'analyse afin de pouvoir trouver un outil qui puisse contribuer à la compréhension de l'utilisation des architectures. Notamment celles contenant des hiérarchies d'héritage.

En effet, il y a un grand nombre de hiérarchies d'héritages dans les architectures POO. Cependant, dû aux modifications inévitables à travers les mise à jour et maintenances éprouvées au fil du temps, il serait naturel de voir l'utilisation de cette hiérarchie se détériorer.

C'est pourquoi l'objectif de ce travail est de pouvoir mesurer l'héritage réellement utilisé pour produire les fonctionnalités demandées à l'exécution du programme. Une métrique de cet augure trouverait sa place en tant qu'indicateur lors de maintenances sur les hiérarchies d'héritage.



## 2. Synthèse des travaux analysés

Vous trouverez ci-dessous le résultat des recherches sur l'état de l'art concernant les évaluations de la qualité de l'héritage dans les architectures logicielles. Mes recherches se sont portées dans un premier temps sur l'identification d'articles concernant l'héritage. Puis dans un second temps sur des métriques d'héritage.

### 2.1 Résultats empiriques

En 1995, l'université de Strathclyde au Royaume-Uni a entrepris une expérience afin d'évaluer l'impact de l'héritage sur la maintenance d'un logiciel, en réponse à une enquête dont 55% des participants avaient répondu que l'héritage augmentait la complexité d'une architecture logicielle. L'objectif était le suivant : montrer que l'utilisation d'une hiérarchie à trois niveaux est meilleure en termes de maintenabilité que de ne pas utiliser d'héritage. [2]

L'expérience se déroula ainsi : 31 étudiants, répartis en deux groupes, étaient confrontés à un logiciel inconnu et devaient y apporter certaines modifications en deux heures. Le premier groupe devait travailler sur un code sans héritage, tandis que l'autre groupe devait travailler sur un code utilisant trois niveaux d'héritage.

À la fin de l'expérience, les chercheurs ont constaté que les tâches de maintenance demandées avaient été plus rapidement effectuées sur le code utilisant l'héritage (environ 20% plus rapidement) et en ont donc conclu que l'utilisation de l'héritage rend le code plus simple à maintenir.

Toutefois l'expérience n'étant pas excessivement complexe, il serait intéressant de mener une expérience similaire sur un système plus grand et plus complexe. De plus, leur définition de maintenabilité s'arrête d'après l'expérience à la phase d'implémentation et uniquement en termes de vitesse de réalisation. Il serait également judicieux d'examiner d'autres catégories de maintenances et d'évaluer la qualité de l'implémentation en plus du temps requis pour l'effectuer.

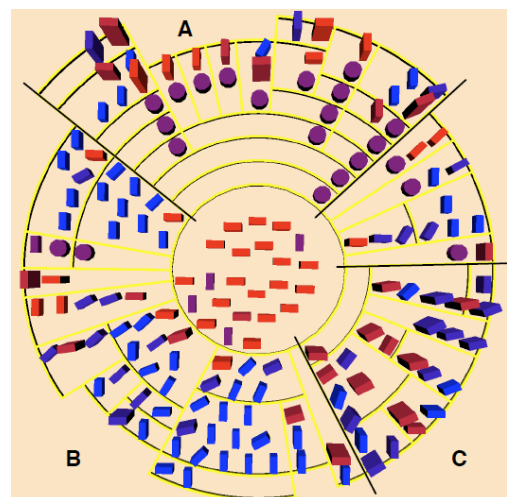
Une expérience similaire fut entreprise dans une université belge en 2001 [5]. À la différence de la précédente qui examinait le temps requis pour l'application des modifications, celle-ci examinait la justesse et complétude de ces modifications. Cette fois, le code utilisant l'héritage a été déclaré plus difficile à maintenir car les modifications étaient plus soignées sur le code sans héritage.

Bien entendu, ces expériences présentent quelques lacunes (notamment le nombre de participants). Mais également plusieurs différences, la première expérience n'évalue que le temps effectué et ne se focalise que sur l'héritage de 3 niveaux ou aucun et ne prenaient en compte que les étudiants ayant réussi dans le temps imparti, les autres n'étaient pas pris en compte dans les statistiques. La deuxième expérience prend en compte même les étudiants n'ayant pas eu le temps de compléter leurs tâches en évaluant les niveaux 0,1,2 et 3 d'héritage. En notant également que le niveau de difficulté de la première expérience est relativement moins complexe que la deuxième et surtout que ces expériences évaluent des choses différentes mais qui sont complémentaires en termes de maintenabilité.

## 2.2 Visualisations

Figure 1 : Exemple de visualisation « Sunburst »

En termes de visualisation de l'héritage, Houari Sahraoui et Simon Denier proposent un concept nommé « Sunburst » [9]. Cette représentation en forme de cercle visualise l'architecture, contenant au centre toutes les classes de couche inférieure au niveau du système du logiciel. La figure sépare en tranches les hiérarchies dans l'architecture. Afin de pouvoir différencier les types de classes un code couleurs est appliqué par type. Le but de cette visualisation est



d'améliorer la compréhension initiale d'un système. Elle permet d'identifier rapidement les différentes parties du système ainsi que l'utilisation de l'héritage au sein du système.

**Tableau 1** : Légende de la visualisation « Sunburst »

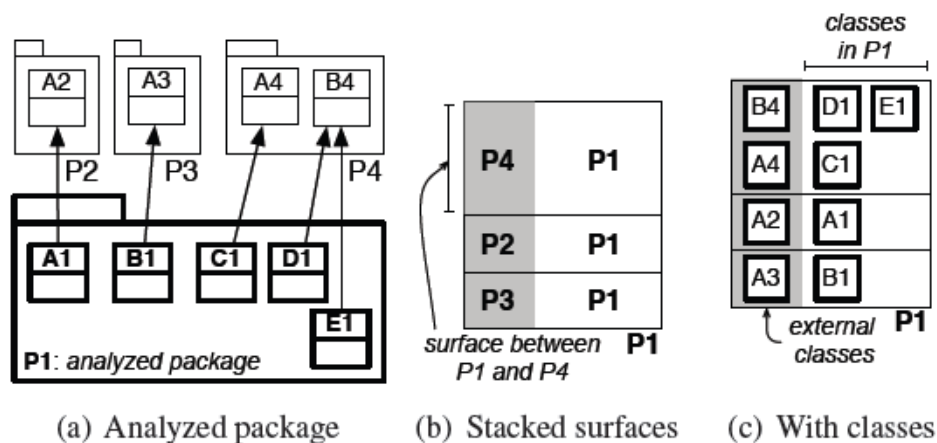
Category	Definition	Colour	Sample Class
Pure Extender	Adding only new methods	Bright red	A
Extender	Adding more methods than overriding	Red	C
Other	No method defined	Purple	X
Overrider	Overriding more methods than adding	Blue	Y
Pure Overrider	Only overriding methods	Bright blue	B, D, Z

La principale lacune de cette représentation visuelle concerne l'héritage multiple qui ne peut être visualisé. Les interfaces ne peuvent être représentées dans la figure. Pourtant la mise en œuvre des interfaces est un élément majeur du langage de programmation Java, elles prennent en charge une grande partie des sous-typages d'héritage. De plus, dans un grand programme avec un grand nombre de petites hiérarchies rend la visualisation moins lisible sans fournir beaucoup d'informations.

En France, l'université de Savoie [7] propose le Package Surface Blueprint, une approche de visualisation différente, se concentrant sur les packages de l'architecture. L'objectif est de pouvoir palier aux problèmes de compréhension concernant les relations entre packages. Grâce à cette représentation, il est facile de repérer les packages mal implémentés dans l'architecture.

Il y a deux vues spécifiques, une soulignant les références faites par un packages et l'autre montrant la structure d'héritage entre packages.

**Figure 2** : Exemple de visualisation « Package Surface Blueprint» (1/2)

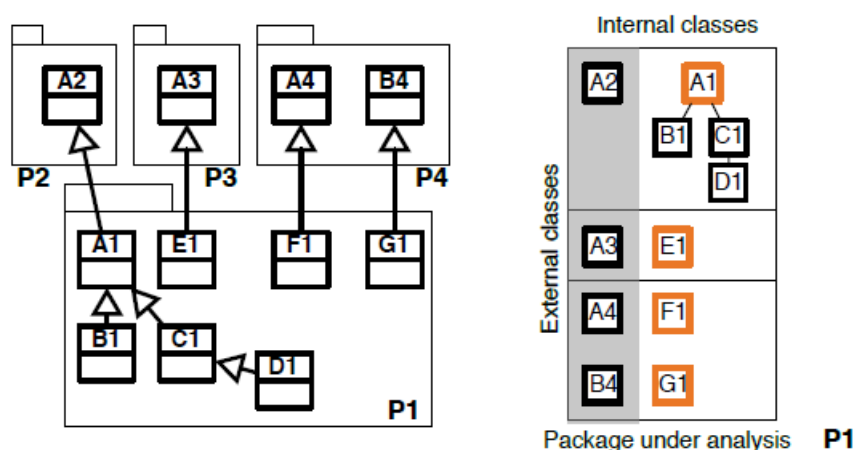


Voici la première vue en exemple. Le package P1 référence les packages P2, P3 et P4. Le package surface blueprint représente par surfaces des appels à ces packages. Plus la surface est grande et plus les appels y sont conséquents. En d'autres termes, on remarque que P4 est plus grand en surface que P2 et P3 car il y a plus d'appels à ce package en comparaison.

Ceci est un exemple du principe, cependant un certain nombre de détails vont être ajoutés à la vue pour rajouter des informations utiles. Telles que les références internes entre les classes dans le package analysé, La position des classes organisés par colonnes pour donner une échelle d'utilisation des références et des couleurs pour l'intensité du nombre de références.

La deuxième vue implémente l'héritage et se présente de la manière qui suit. Les sous-classes directes sont entourées en orange tandis que les sous-classes indirectes en noir. La couleur cyan est attribuée aux objets racine d'une hiérarchie tels que la classe Object par exemple, et les classes en bleu sont les classes abstraites. (Pour plus d'informations cf. article [7]).

Figure 3 : Exemple de visualisation « Package Surface Blueprint» (2/2)



Il faut relever qu'il n'y a pas de prise en compte des sous-packages ce qui est un point négatif pour ce type de représentation.

## 2.3 Mesures de l'héritage

En 1996, J. Mayrand, F. Guay et E. Merlo procèdent à une analyse qui les conduit aux conclusions suivantes [4]. Pour pouvoir simplifier la vue global de l'architecture sous l'angle de l'héritage il faut dans un premier temps grouper les classes en clusters qui représente chacun une hiérarchie. Et dans un deuxième temps, enlever l'héritage multiple qui complexifie les hiérarchies. Ces deux techniques permettent de baisser la complexité des graphs d'héritage.

A l'époque, il était conseillé de simplifier en relocalisant les services d'une deuxième classe mère dans une classe commune à toutes les classes dérivantes, ceci dans le but d'éliminer l'héritage multiple. Toutefois, ce n'était pas la meilleure méthode en termes de cohésion et surtout il est difficile de pouvoir trouver de telles classes communes. De plus, leur algorithme ne prenait pas en compte les interfaces ce qui en faisait leur principal défaut.

D'autres proposent des mesures des systèmes informatiques. Chidamber and Kemerer (CK) en 1994 ont conçu les métriques précurseurs en terme d'héritage suivantes :

- "Depth of Inerithance Tree" (DIT)
- "Number of Childrens" (NOC)

Le DIT est le niveau de profondeur de la hiérarchie en partant de la classe analysée jusqu'à la classe racine (la classe mère de la hiérarchie en question). Théoriquement, cette métrique permet de connaître le nombre de classes parentes pouvant potentiellement affecter la classe analysée. (Une classe parente est une classe à un niveau supérieur dans la hiérarchie par rapport à une classe inférieur à celle-ci.) CK interprètent le DIT de la façon suivante, plus la valeur du DIT est grande plus le degré des méthodes héritées est grand, ce qui rend plus difficile à prédire le comportement. Plus une hiérarchie est profonde, plus elle est complexe au niveau du design impliquant un plus grand nombre de classes et méthodes.

Le NOC est le nombre de classes enfants directes de la classe analysée. La base théorique consiste à pouvoir mesurer le nombre de sous-classes héritant des méthodes de la classe mère. Grâce au NOC d'après CK, il serait possible d'évaluer le

potentiel de réutilisation, la probabilité d'une mauvaise abstraction ainsi que l'influence potentielle d'une classe sur l'architecture du système. Plus la valeur du NOC est grande, plus ces points de vues sont probables.

La figure ci-dessous montre une hiérarchie évaluée avec les valeurs du DIT et NOC :

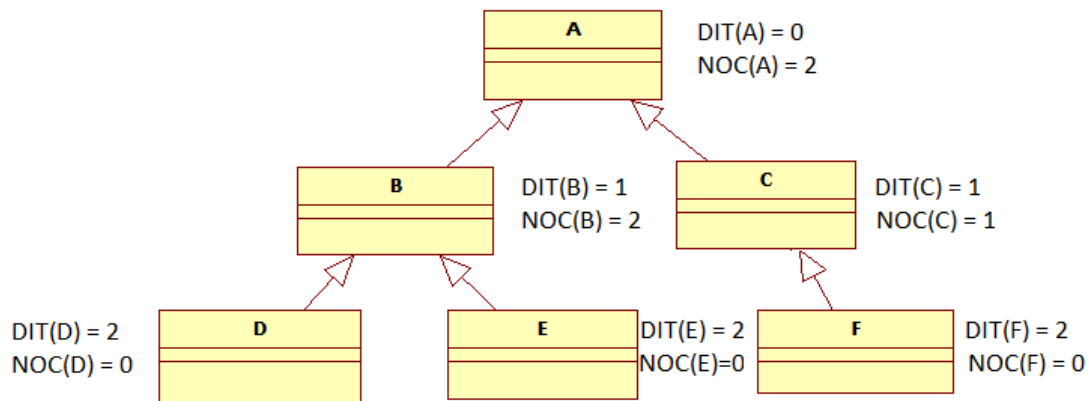


Figure 4 : Exemple de hiérarchie DIT et NOC

En 1998, Li après avoir analysé les travaux de CK se rend compte que lorsqu'il y a de l'héritage multiple, le DIT devient ambiguë. Pour commencer, la définition même du DIT n'est pas applicable car il peut y avoir plusieurs classes racines dans une hiérarchie. De plus, il y a un conflit entre la définition et la base théorique du DIT. La base théorique stipule que le DIT est une mesure du nombre de classes parentes qui peuvent potentiellement affecter la classe analysée. Toutefois, la définition du DIT se concentre sur la longueur de la profondeur de la hiérarchie jusqu'à remonter à la classe racine. Ceci est une contradiction avec la base théorique du DIT visible uniquement lorsque l'héritage multiple est présent. La figure ci-dessous exprime ces ambiguïtés :

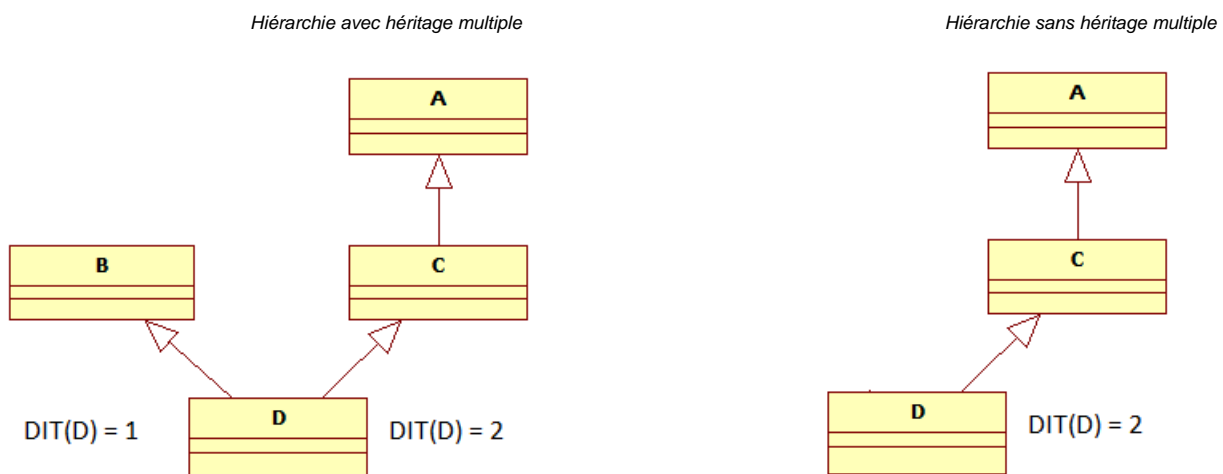


Figure 5 : ambiguïtés du DIT

On remarque que lorsqu'il y a de l'héritage multiple la définition et la base théorique sont contradictoires. En effet, dans l'exemple-ci-dessus, la classe D comporte trois classes parentes. Pourtant le DIT n'est pas de trois.

Concernant le NOC, La base théorique indique qu'il mesure le nombre de sous-classes qui vont hériter des méthodes de la classe parente analysée. Cependant le NOC s'arrête aux sous-classes directes, ce qui n'est pas raisonnable. Une classe peut influencer toutes ses sous-classes et pas uniquement ses sous-classes directes.

Li propose donc deux nouvelles métriques alternatives au DIT et NOC qui sont :

- "Number of Ancestor Classes" (NAC)
- "Number of Descendants Classes" (NDC)

Le NAC compte le nombre de classes parentes de la classe évaluée afin de pallier aux ambiguïtés du DIT en cas d'héritage multiple. Le NDC compte toutes les classes descendantes de la classe évaluée améliorant ainsi le concept du NOC. L'interprétation de ses métriques est similaire à celles faites par CK en 1994, Li avait pour objectif d'améliorer uniquement le DIT et le NOC.

En 2001, F.T. Sheldon et son équipe continuent le travail entrepris jusqu'ici [6]. L'héritage permet la réutilisation des classes déjà conçues lors de la conception d'une nouvelle classe. Cependant, il introduit également une dépendance entre les classes. Si une modification est apportée à la classe parente, le changement pourrait avoir une incidence sur toutes ses classes descendantes. F.T. Sheldon propose deux nouvelles métriques reprenant les principes des métriques de Li qui auront pour objectif d'être spécifiques aux maintenances sur une hiérarchie d'héritage.

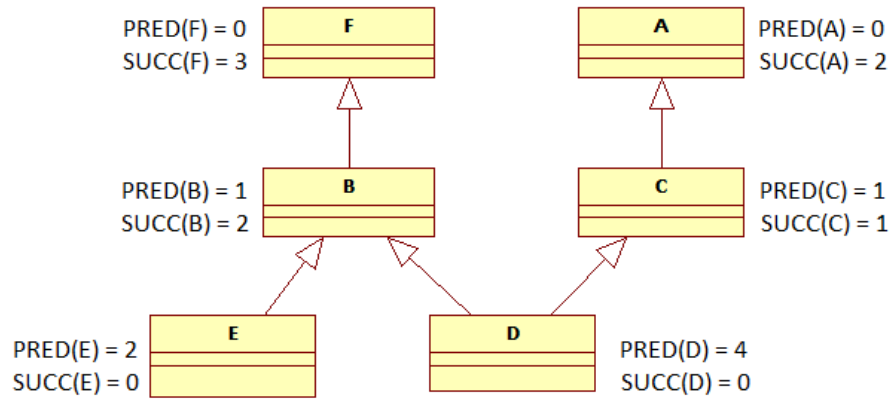
- "Average Understandability" (AU)
- "Average Modifiability" (AM)

Ses métriques sont calculées en utilisant un principe similaire aux métriques de Li. Pour se faire, F.T.Sheldon introduit deux nouvelles fonctions :

- **PRED(j)** : le nombre total de classes prédécesseurs de la classe j.
- **SUCC(j)** : le nombre total de classes successeurs de la classe j.

Voici un exemple d'application de ses fonctions sur la figure ci-dessous :

Figure 6 : Application du PRED() et SUCC() sur une hiérarchie



Il y a deux parties dans la maintenance, la compréhension de l'architecture du système et la modification de celle-ci. Voici comment ils définissent la compréhensibilité et la modifiabilité d'après l'article [6] :

« Understandability is defined as the ease of understanding a program structure or a class inheritance structure and modifiability is defined as the ease with which a change or changes can be made to a program structure or a class inheritance structure. »<sup>1</sup>

La métrique AU est finalement une moyenne du nombre de prédécesseurs pour chaque classe dans la hiérarchie incrémenté de la classe analysée elle-même. Avant d'entreprendre une maintenance sur une hiérarchie il faut premièrement pouvoir la comprendre. Pour cela il faut cerner les classes parentes de la classe qu'on veut modifier. C'est pour cela que dans le calcul d'une classe analysée on prend en compte toutes les classes prédécesseurs (parentes) tout comme le NAC de Li, tout en rajoutant la classe analysée dans le calcul car il faut également la comprendre. Voici la formule pour calculer le niveau de compréhension requis pour une classe analysée :

$$U(C_i) = \text{PRED}(C_i) + 1$$

Une fois qu'on évalue la valeur de U pour chaque classe de la hiérarchie il suffit d'en faire une moyenne pour obtenir l'AU :

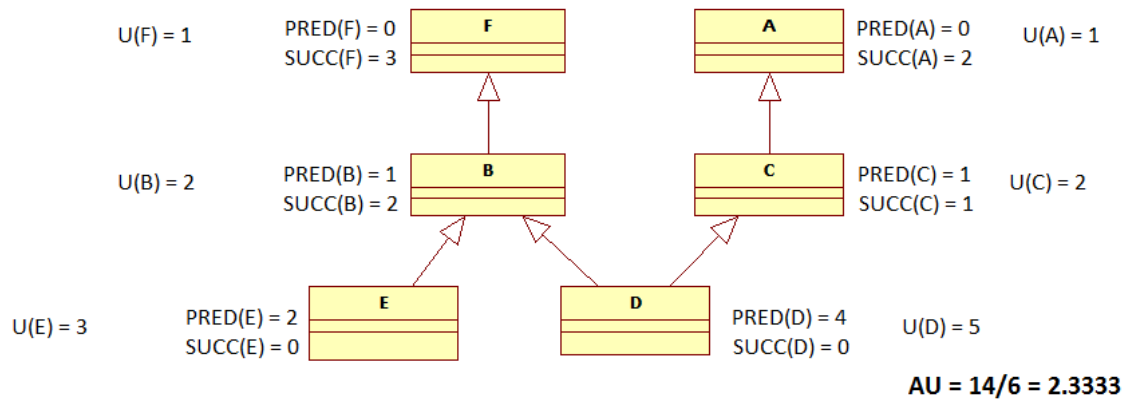
$$\text{AU de la hiérarchie analysée} = \frac{\sum_{i=1}^t (U(C_i))}{t}$$

<sup>1</sup> F.T.Sheldon, K.Jerath, H.Chung, "Metrics for maintainability of class inheritance hierarchies", Journal of Software Maintenance and Evolution: Research and Practice, Vol. 14, 2002, p.7



Dans la formule,  $t$  représente le nombre total de classes dans la hiérarchie analysée. Voici un exemple d'application de cette métrique sur la hiérarchie ci-dessous :

Figure 7 : Application de l'AU sur une hiérarchie



Dans cette hiérarchie, on obtient la valeur AU de 2.3333 qui représente le degré de compréhensibilité. Plus la valeur est faible plus le degré de compréhension est élevée. Ceci reflète le degré de dépendance entre les classes de la hiérarchie.

La métrique AM utilise dans son calcul l'AU. En effet, avant de pouvoir modifier une classe il faut d'abord comprendre la hiérarchie en question mais également les classes descendantes de la classe à modifier afin de pouvoir évaluer l'impact des modifications sur les classes descendantes. Après il suffit de calculer cela pour chaque classe de l'architecture analysée en divisant par le nombre total de classes afin d'obtenir les moyennes qui seront les valeurs des deux métriques. Voici la formule pour calculer la modifiabilité pour une classe analysée :

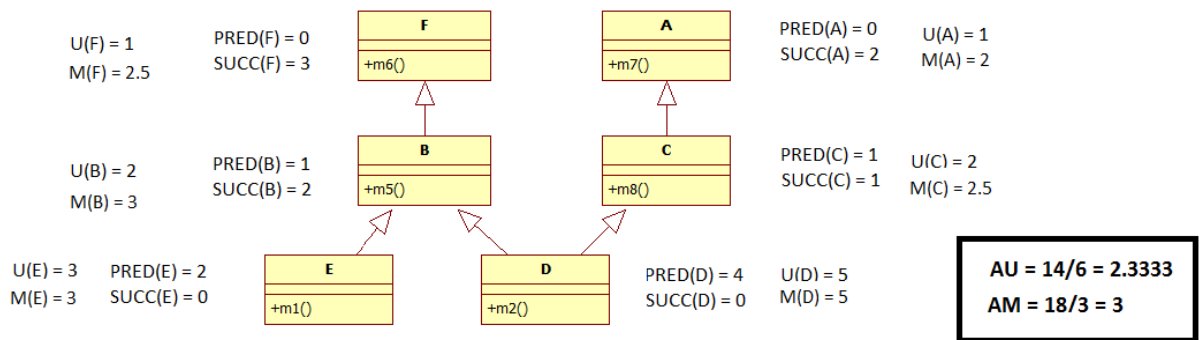
$$M(C_i) = (U(C_i) + SUCC(C_i)) / 2$$

Une fois qu'on évaluée la valeur de  $M$  pour chaque classe de la hiérarchie il suffit d'en faire une moyenne pour obtenir l'AM :

$$AM \text{ de la hiérarchie analysée} = AU + \left( \sum_{i=1}^t (SUCC(C_i)/2) \right) / t$$

Dans la formule,  $t$  représente le nombre total de classes dans la hiérarchie analysée. Voici un exemple d'application de cette métrique sur la hiérarchie ci-dessous :

Figure 8 : Application de l'AU et AM sur une hiérarchie



L'AM représente le degré de modifiabilité d'une hiérarchie de classes. Plus la valeur est faible, plus la modifiabilité est accentuée. Plus une hiérarchie est profonde, plus la réutilisation est accentuée au détriment de la compréhension et modifiabilité. Du point de vue de la maintenance, il est recommandé de diviser une hiérarchie trop profonde afin d'obtenir plusieurs hiérarchies moins profondes.

Au final, ces métriques utilisées lors de la maintenance d'une hiérarchie permettent d'évaluer le compromis entre l'augmentation de la réutilisation par l'héritage et la facilité de maintenance avec une hiérarchie d'héritage moins complexe. Les designers et managers peuvent grâce à ces métriques vérifier la qualité de leur travail mais également estimer, planifier et contrôler les activités de conception lors de la maintenance du projet.

D'un point de vue critique, L'AU essaye de représenter le degré de compréhension, notamment en calculant le nombre de classes qu'il faut comprendre avant de comprendre la classe analysée elle-même. Ce concept est intéressant mais néanmoins incomplet. En effet, l'AU ne s'arrête qu'à l'interdépendance des classes dans une hiérarchie. Il faudrait également intégrer au calcul les classes externes à la hiérarchie qui sont couplées avec la classe analysée. Ces classes externes sont nécessaires à la compréhension de la fonctionnalité délivrée par la classe.

De plus, dans la majorité des maintenances, la hiérarchie n'est pas modifiée entièrement. En effet, l'objectif est dans la plupart des cas, la modification d'une fonctionnalité métier du système. Dans ce cas, l'AU n'étant qu'une moyenne globale sur la hiérarchie ne sera pas très utile. Cependant si on est amené à modifier une

classe de la hiérarchie la valeur U de la classe en question pourrait nous indiquer le nombre de classes à lire avant de la modifier. Dans l'exemple ci-dessous on n'aura qu'à lire et comprendre les classes F, B et E :

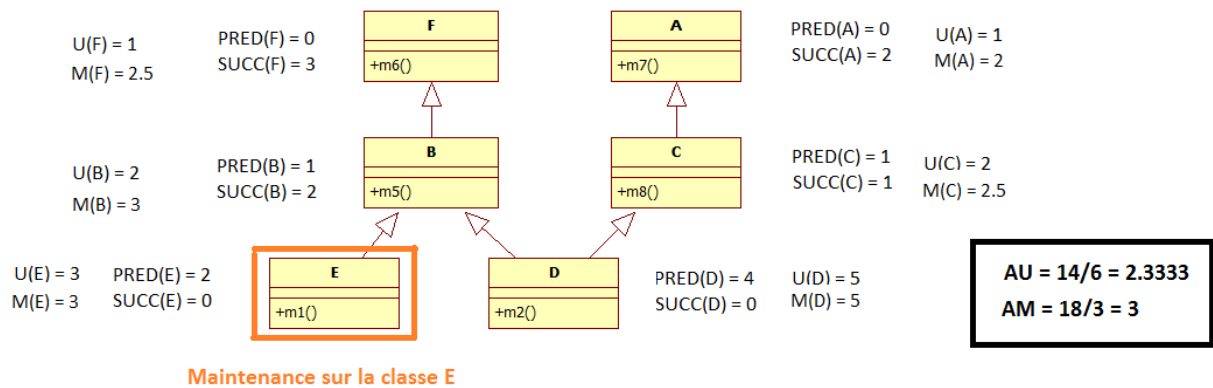


Figure 9 : AU et AM inutile

Cependant, dans le cas où il ne faudrait modifier seulement les méthodes non héritées de la classe E, la métrique AU n'a plus aucune utilité. Si les modifications se portaient uniquement sur la méthode m1(), il ne serait pas nécessaire de comprendre les classes F et B (à moins que m1() utilise les méthodes héritées). On peut donc conclure, que la métrique AU est intéressante dans la phase de conception de la hiérarchie afin comme le dit F.T. Sheldon que les développeurs puissent vérifier la qualité de leur travail. (Les auteurs n'indiquent pas ce qu'ils entendent par qualité).

L'AU et L'AM permettent de pouvoir évaluer le degré d'interdépendance entre les classes ce qui permet d'après la définition de F.T.Sheldon sur la compréhension et modifiabilité, de dire si une hiérarchie va être plus ou moins facile à comprendre et à modifier. Si on ne remet pas en question ces définitions, ces métriques répondent à ce qu'ils voulaient faire. Toutefois, leur définition est un peu vague et ne prend en compte que le fait de lire des classes parentes ou successeurs ce qui n'est pas complet comme mentionné plus haut.

Pour finir, devoir inspecter toutes les classes parentes afin de comprendre une classe n'est pas raisonnable. Imaginons que votre système utilise un Framework. Il n'est pas vraisemblable d'aller analyser toutes les classes de la hiérarchie du Framework. En effet, en plus d'hériter des méthodes et attributs, une bonne hiérarchie permet de pouvoir apporter grâce à son abstraction, un plus en termes de compréhension. Ce qui explique également pourquoi ils affirment qu'une hiérarchie trop profonde est moins compréhensible, ce qui est normal sans la prise en compte des abstractions.

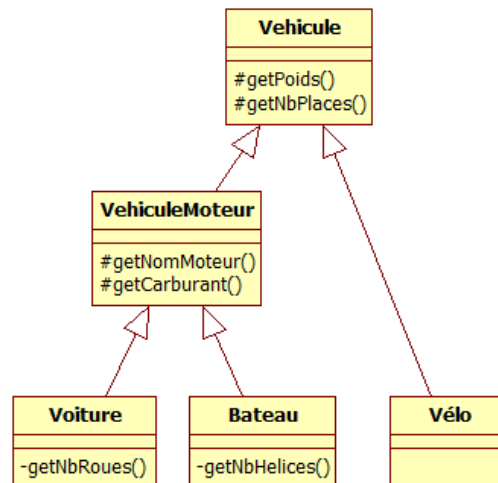


Figure 10 : Hiérarchie avec une bonne abstraction

Dans cette hiérarchie, il n'est pas nécessaire de remonter jusqu'à la classe racine Véhicule pour comprendre une classe comme Voiture ou Bateau. En effet, VehiculeMoteur est une classe représentant une bonne abstraction, favorisant la compréhension. On le remarque au nom de la classe qui permet de comprendre directement de quoi il s'agit par analogie.

#### Le raisonnement par analogie [24]

C'est une similitude entre le nom de l'objet et ce qu'il représente. Le nom porte une sémantique issue du domaine métier et de l'expérience de projets passés. Il faut choisir le nom avec soin. Il conduit de l'information sur ce que l'objet peut faire sa structure ou son rôle dans le système. Le nom permet de « s'attendre à trouver » des propriétés.

Par exemple, l'objet nommé Voiture permet de déduire qu'il modélise le concept d'une voiture. On s'attend donc à retrouver des attributs et méthodes en lien avec son concept. Lorsqu'on observe la méthode `getNbRoues()` on peut tout de suite conclure, toujours par analogie, qu'il s'agit d'une méthode qui retourne le nombre de roues. Si on avait nommé la classe et la méthode autrement il aurait été difficile de faire le même rapprochement et ceci aurait atténué la compréhensibilité. C'est typiquement le cas représenté sur la figure 11. L'abstraction `VehiculeMoteurPesantAVitres` est un nom étrange qui assombri la compréhension.

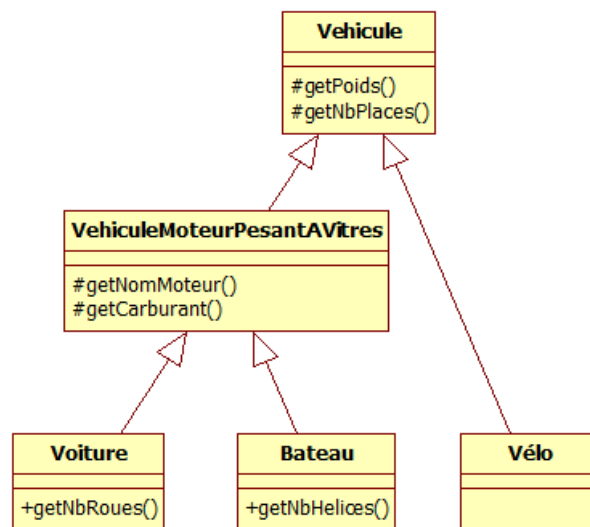


Figure 11 : Hiérarchie avec une mauvaise abstraction

L'AM est également concerné par ces constatations étant donné qu'il a besoin de l'AU pour se calculer. Toutefois, le principe de devoir analyser les classes successeurs à une classe avant de la modifier trouve son sens étant donné qu'elle aura un impact sur les classes successeurs.

Pour conclure sur ces deux métriques, on obtient plutôt des évaluations du degré de dépendance entre les classes de la hiérarchie. Ceci n'est pas suffisant, comme mentionné plus haut (cf. p.18), pour pouvoir conclure à une métrique sur la compréhension ou modifiabilité.

En 1995, à part le DIT, NOC et leurs métriques dérivées, F. Brito e Abreu fondent les six métriques MOOD (Metrics Object-Oriented Design). Ces métriques permettant d'évaluer la qualité d'un système sous plusieurs axes, sont maintenant connues par la communauté du software engineering. Même si ces métriques ont pour habitude d'être utilisées ensemble, je n'analyserai que les métriques concernant l'héritage dans le cadre de mon travail, pour plus d'information (cf. articles [1] [3]).

- « Method Inheritance Factor » (MIF)
- « Attribute Inheritance Factor » (AIF)

Les métriques MIF et AIF permettent de mesurer l'héritage d'un système. Autrement dit, ce sont des moyennes sur l'ensemble du système qui expriment le ratio de méthodes et d'attributs hérités.

Concernant le calcul du MIF, Le numérateur est la somme de toutes les méthodes héritées dans chaque classe du système. Le dénominateur est le total de toutes les méthodes disponibles (définies et héritées) dans chaque classe du système.

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

$$\text{where } M_a(C_i) = M_d(C_i) + M_i(C_i)$$

**M<sub>i</sub> (C<sub>i</sub>)** est le nombre de méthodes qui sont héritées (pas les méthodes redéfinies).

**M<sub>d</sub> (C<sub>i</sub>)** est le nombre de méthodes non abstraites déclarées dans C<sub>i</sub>.

**M<sub>a</sub> (C<sub>i</sub>)** est le nombre de méthodes qui peuvent être appelées dans une instance de la classe C<sub>i</sub>.

**TC** est le nombre total de classes dans le système / package analysé.

L'AIF est similaire au MIF mais se concentre sur les attributs. Le numérateur est la somme de tous les attributs hérités dans chaque classe du système. Le dénominateur est le total de tous les attributs disponibles (définies et héritées) dans chaque classe du système.

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

**Ai (Ci)** est le nombre d'attributs qui sont hérités (pas les attributs redéfinis).

**Ad (Ci)** est le nombre d'attributs déclarés dans Ci.

**Aa (Ci)** est le nombre d'attributs qui peuvent être utilisés dans une instance de la classe Ci.

**TC** est le nombre total de classes dans le système / package analysé.

Le but de ces métriques est de résumer la qualité globale d'une architecture d'un système orienté objet. Le MIF et AIF permettent d'avoir une moyenne globale sur l'héritage implanté dans l'architecture du système analysé.

Plus la valeur du MIF est haute, plus il y a d'héritage entre les classes dans l'architecture du système analysé. Si la valeur du MIF correspond à 0 c'est qu'il n'y a pas de méthodes héritées (pouvant être dû au niveau de protection des méthodes de la hiérarchie qui empêcherait l'héritage des méthodes). C'est également le cas pour l'AIF concernant les attributs hérités. A noter qu'on s'intéresse moins au AIF car dans la plupart des logiciels, les attributs sont en « Private », ce qui empêche l'héritage et donne des valeurs faibles à l'AIF. (On peut supposer qu'à l'époque, aux débuts de Java, la visibilité n'était pas complètement mise au point. Notamment sur le langage de programmation orienté objet Smalltalk, créé en 1972, les attributs étaient obligatoirement « Protected ». Ce qui peut justifier ce genre de métrique.)

L'évaluation de qualité proposée par ces métriques ne s'applique que d'un point de vue statique. De plus, le MIF et AIF ne sont pas précis. Ce ne sont que des moyennes sur l'ensemble de la hiérarchie évaluée. L'utilité de leurs valeurs est à remettre en question étant donné que la plupart du temps les maintenances sont spécifiques et ne s'intéressent pas à l'ensemble du système, (à moins que la maintenance remette en question toute l'architecture du système).

La figure 11 présente deux hiérarchies de classes avec une structure différente en termes d'architecture. Pourtant, il s'agit d'un cas où le MIF donne la même valeur pour les deux hiérarchies.

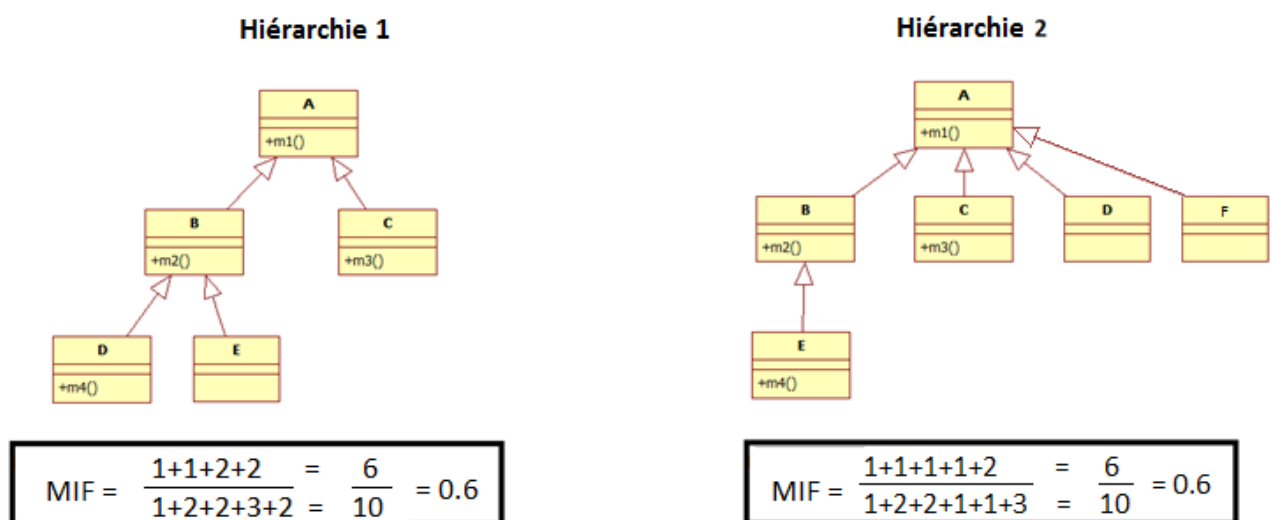


Figure 12 : MIF similaire sur des hiérarchies différentes

Nous émettons donc quelques doutes sur l'utilité pratique du MIF (même chose pour le AIF) qui ne donne que des moyennes sur l'architecture statique d'un système et ne distingue pas des architectures très différentes.

En 1996, F. Brito e Abreu et W.Melo vont évaluer l'impact des six MOOD. Ils ont analysé que plus le MIF a tendance à être élevé, plus les bugs et l'effort dépensé pour corriger les bugs auront tendance à diminuer. Ces résultats montrent comment l'héritage semble être une technique appropriée pour réduire le nombre de bugs et de maintenances correctives, lorsqu'il est utilisé avec modération. Des valeurs très élevées de MIF (au-dessus de la fourchette de 70% à 80%) sont suspectées d'inverser



cet effet bénéfique, cependant d'après l'article, cette hypothèse n'a pas encore été validée par une étude.

En d'autres termes, si une erreur de code est commise, elle ne sera implantée qu'à un endroit dans le code, étant donné qu'il n'y a pas (en principe) de code dupliqué grâce à la réutilisation des méthodes dans les classes de la hiérarchie d'héritage. Cependant si une hiérarchie est trop grande elle aura tendance à être plus complexe. Néanmoins, les corrélations entre le MIF et les bugs et maintenances correctives ne sont pas à sens unique. Même en ayant un MIF élevé, il se peut que d'autres facteurs provoquent des erreurs dans le code. Ce n'est donc pas une valeur sûre.

Nasib S. Gill propose quant à lui en 2011, cinq nouvelles métriques en abordant l'aspect de la réutilisabilité dans l'architecture logicielle. En rapport avec mon travail je n'analyserai que les métriques suivantes : (pour plus d'informations cf. article [10].)

- « Method Reuse Per Inheritance Relation » (MRPIR)
- « Attribute Reuse Per Inheritance Relation » (ARPIR)

Le MRPIR calcule le nombre total de méthodes « réutilisées » au travers de la relation d'héritage dans la hiérarchie d'héritage. Cela s'applique à toute la hiérarchie d'héritage dans le système.

$$\text{MRPIR} = \frac{\sum_{k=1}^r MI_k}{r}$$

**MI (k)** est le nombre de méthodes héritées de la relation d'héritage k

(Si une même méthode est héritée par différentes relations d'héritage, alors il est calculé séparément dans chaque relation).

**r** est le nombre total relations d'héritage

$$ARPIR = \frac{\sum_{k=1}^r AI_k}{r}$$

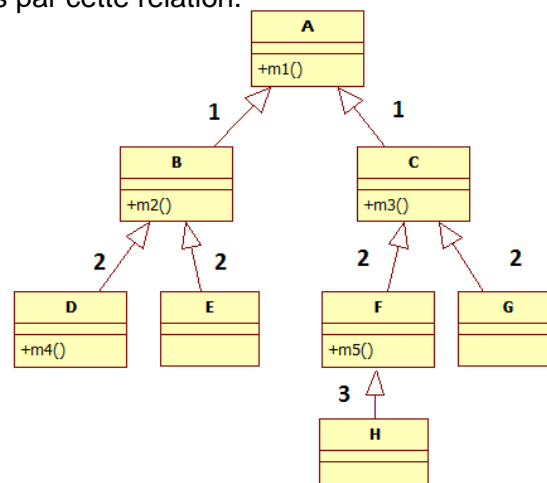
**AI (k)** est le nombre d'attributs hérités de la relation d'héritage k

**r** est le nombre total relations d'héritage.

ARPIR procède de la même façon mais cette fois en calculant les attributs des classes. (En partant du principe qu'il existe des attributs hérités. Dans le cas contraire, où tous les attributs seraient « Private » cette métrique est inutile.)

Ces métriques sont très similaires aux MIF et AIF. Cependant, elles ne s'appliquent pas comme une moyenne globale sur l'ensemble du système. Le MRPIR et ARPIR s'applique à une hiérarchie. De plus dans leur calcul, le MIF et AIF prennent en compte les méthodes ou attributs définies et héritées. Tandis que le MRPIR et ARPIR ne fait attention qu'à ceux héritées et aux relations d'héritage.

Voici un exemple ci-dessous d'application du MRPIR à une hiérarchie d'héritage. On remarque que chaque chiffre à côté d'une relation d'héritage représente le nombre de méthodes héritées par cette relation.



$$MRPIR = 1 + 2 + 2 + 1 + 2 + 2 + 3 = 13 / 7 = 1.85714285$$

Figure 13 : Application du MRPIR sur une hiérarchie

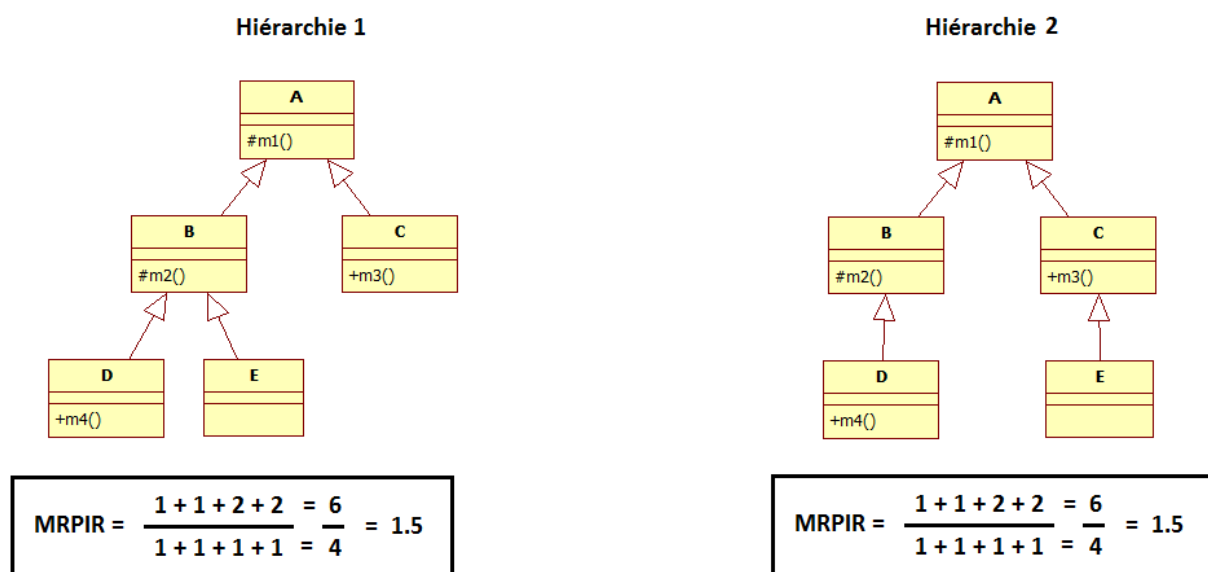
Plus la valeur du MRPIR est haute plus il y a des méthodes héritées. Le MRPIR calcule en fait le nombre moyen de méthodes héritées et donc réutilisables par relation d'héritage dans la hiérarchie.

Au final, l'intérêt de ces métriques d'après leur auteur, serait de pouvoir comparer deux hiérarchies d'héritage différentes. En effet, lors de la conception d'une hiérarchie il se pourrait qu'il y ait plusieurs propositions pour la structure de la hiérarchie. Dans ce cas, le MRPIR et ARPIR pourraient être intéressants afin de pouvoir comparer le nombre de méthodes héritées dans une hiérarchie en question et aider au choix de la hiérarchie.

Cependant, choisir la structure d'une hiérarchie par la simple évaluation d'une moyenne statique n'est pas judicieux et ne prend pas en compte l'héritage effectif. Il serait plus intéressant de comparer des structures de hiérarchies différentes avec des métriques dynamiques basées sur les fonctionnalités métiers. Dans le but de pouvoir évaluer l'héritage réellement nécessaire lors de l'exécution d'une fonctionnalité métier et ainsi choisir l'organisation de la hiérarchie la plus adéquate.

La figure ci-dessous représente deux hiérarchies de classes avec une structure différente. Le MRPIR donne la même valeur pour les deux hiérarchies.

**Figure 14 : MRPIR similaire sur des hiérarchies différentes**



Le MRPIR (ainsi que le ARPIR) ne sont que des moyennes tout comme le MIF et AIF, ces métriques sont susceptibles de ne pas différencier des architectures différentes.

Les critiques prononcées contre le MIF et AIF sont donc valables pour le MRPIR et ARPIR. Même en obtenant une valeur qui permet de se faire une idée plus précise du nombre effectif de méthodes héritées d'une classe à une autre avec le MRPIR, on n'a tout de même pas cette notion de l'utilisation réelle de l'héritage. Même si en moyenne 1.857 méthodes sont héritées par lien d'héritage, rien ne nous confirme qu'elles sont réellement utilisées en réponse à la réalisation d'un scénario donné. (De plus cela reste une mesure globale, donc peu informative.)

Au lieu de se concentrer uniquement sur l'architecture des systèmes globalement, il serait intéressant de pouvoir se concentrer sur les différents niveaux d'agrégations (classes, packages) de l'architecture concernée par le scénario utilisé, chose difficile avec les métriques statiques mesurant l'ensemble de l'architecture au niveau des classes seulement.

On entend par scénario, la mise en œuvre d'un « use-case » du logiciel analysé. En d'autres termes l'exécution d'un scénario signifie d'appliquer les actions du « use-case » correspondant sur le logiciel. Ceci va déclencher des instanciations de classes qui vont réaliser la fonctionnalité désirée. On pourra ainsi pour un scénario donné connaître les classes et packages nécessaires à la réalisation de ce scénario et par conséquent n'analyser que les composants utilisées. Il y a en général plusieurs « use-cases » pour un logiciel.

En se concentrant sur l'analyse par scénario, on pourrait avoir des évaluations par niveaux d'agrégations contrairement au MIF qui n'évalue que le système en moyenne, et ceci pour un scénario donné. Des sous-structures peuvent contenir d'autres sous structures et définissent ainsi différents niveaux d'agrégations.

Définition d'une sous-structure [11] :

Une sous-structure d'un programme ou système est un ensemble de déclaration dans le langage de programmation qui :

- représente un regroupement syntaxique d'éléments de programme, quel que soit leur niveau au-dessus de la déclaration de méthode/procédure/fonction
- et
- obéit à une relation de composition.

Cette définition permet de prendre en compte les packages en Java et leurs équivalences dans les autres langages de programmation. Par exemple, une sous-structure pour le langage Cobol correspond à une section.

L'évaluation peut se faire aussi bien à un niveau plus abstrait (macro) comme par exemple celui des packages, qu'au niveau intermédiaire des sous-packages ou encore au niveau plus détaillé (micro) des classes. Ceci, en plus d'être plus complet (ne s'arrêtant pas qu'à l'évaluation des classes, mais également des packages), permettrait de discerner les situations où un package entier est dépendant en termes d'héritage d'un autre package. On pourrait également se retrouver dans une situation où l'évaluation micro d'un package démontrerait que ces classes ont besoin de l'héritage pour répondre à la fonctionnalité qui leur ai demandée, alors que le package en question n'a pas besoin d'héritage pour fonctionner. (Cela reviendrait à dire que la hiérarchie entière se trouve dans le même package.)

Contrairement au MIF, nous voulons pouvoir évaluer un niveau d'héritage effectif pour chaque niveau d'agrégation de l'architecture et ceci pour un scénario donné et non pas une moyenne statique sur l'ensemble de l'architecture.

Dans l'exemple ci-dessous, on remarque qu'il y a plusieurs sous-structures. Ce que nous aimerions réaliser, c'est pouvoir évaluer chaque niveau d'agrégation, partant du point de vue macro, il y a premièrement le système dans son ensemble. Ensuite, Le package 1 et le package 2. Pour finir, chaque classe peut être analysée indépendamment.

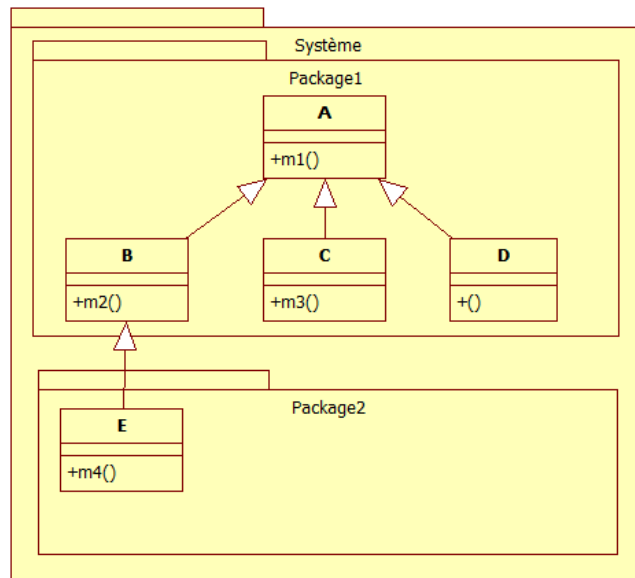


Figure 15 : Niveaux d'agrégations

Dans l'exemple du système ci-dessus, on pourrait par exemple, avoir décelé une basse utilisation de l'héritage au niveau de la classe C pour un scénario donné. Puis pour un autre scénario, il se pourrait que ce soit la classe B qui se retrouve dans ce cas. Au final, on pourrait s'apercevoir en se focalisant à un niveau d'agrégation plus abstrait, que le Package 1 est dans la plupart des scénarios, évalué comme n'utilisant que très peu l'héritage dans sa hiérarchie. Les méthodes de la classe mère A étant rarement réutilisées dans la plupart des scénarios du logiciel amèneraient à conclure à une mauvaise implémentation de cette hiérarchie. (Chose difficile à voir si l'évaluation s'arrêterait au niveau de l'ensemble du système ou uniquement des classes.)

On pourrait également discerner un cas où les classes B, C et D n'utilisent que la méthode héritée m1() par la classe parente A et n'apporte rien de plus en termes de fonctionnalité. (Autrement dit, on ne fait appel à B, C et D que pour appeler m1() et rien de plus en termes de code.) Ce qui reviendrait à dire que la fonctionnalité nécessaire se trouve dans la classe parente A et remettrait en question les classes B, C et D qui n'apportent rien de plus. Finalement on pourrait se demander à quoi servent ces

classes étant donné que la classe A peut très bien réaliser le travail. Ceci amènerait à remettre en cause la structure de la hiérarchie.

Cependant on ne peut pas juger que l'implémentation d'une hiérarchie soit mauvaise sous seul prétexte que l'héritage n'est jamais utilisé. En effet, il est conseillé de créer des hiérarchies de généralisation afin de bénéficier des axes suivants [19]:

- DRY
- Polymorphisme
- Compréhension

## DRY

Le principe DRY (« Don't Repeat Yourself ») consiste à ne pas dupliquer de code dans un programme. Si on ne le respecte pas, lors d'une maintenance sur ce code il faudra modifier à tous les endroits où il a été dupliqué. Ceci va également dupliquer le temps de la maintenance, étant donné qu'on ne modifie plus une fois à un seul endroit.

L'héritage permet de pouvoir facilement respecter le principe DRY. En effet, il suffit d'implanter le code qu'on aimerait réutiliser sans le dupliquer à un niveau supérieur dans une hiérarchie de classes afin que les classes successeurs de la classe possédant le code, héritent du code en question.

## Polymorphisme

Lorsqu'on redéfinit une méthode héritée dans les classes successeurs, il est alors possible d'appeler cette méthode sans se soucier de son type dynamique. Le polymorphisme permet à une classe successeur de pouvoir être attribué à un type statique d'une classe parente. Ainsi on pourrait manipuler une hiérarchie de véhicules sans se soucier des types dynamiques des instances traitées (ex : Voiture, Camion, Bateau, etc.)

```
Vehicule voiture = new Voiture();
Vehicule camion = new Camion();
Vehicule bateau = new Bateau();

public void traiterVehicules(Vehicule vehicule) {
    double poids = vehicule.calculPoids();
    double prix = vehicule.calculPrix();
    System.out.println("Poids du véhicule en KG : " + poids);
    System.out.println("Prix du véhicule en CHF : " + prix);
} //traiterVehicules
```

Figure 16 : Exemple de polymorphisme

La figure 15 montre que la méthode «traiterVehicules(Vehicule vehicule) traite n'importe quel instance de type statique Vehicule. Du moment où les méthodes calculPoids() et calculPrix() est implantées dans la classe parente Vehicule et qu'elles sont redéfinies après avoir été héritées dans les classes successeurs (Voiture, Camion, Bateau) le polymorphisme s'applique. Au runtime lorsque la méthode traiterVehicules sera appelée avec les instances Voiture, Camion et Bateau, ça sera les méthodes redéfinies dans leurs types dynamiques qui vont s'exécuter.

Le polymorphisme a besoin de l'héritage pour fonctionner, il suffit de typer les objets par une classe parente et vous pouvez attribuer un même traitement à tous ces objets.

## Compréhension

Une hiérarchie peut améliorer la compréhension. Lorsqu'il n'y a pas de hiérarchie et que les objets ne sont liés que par la composition, il est souvent plus difficile de mapper les liens qui les regroupent. La figure 10 est un exemple d'une hiérarchie améliorant la compréhension. L'abstraction de la classe VehiculeMoteur permet par analogie de comprendre immédiatement de quoi il s'agit.

Il est conseillé de ne pas mélanger les dimensions à un même niveau hiérarchique mais également de ne conserver que les classes qui se distinguent de leur classe parente par des méthodes ou attributs différents. Lorsqu'on a des classes avec des méthodes communes, il faut essayer de créer une classe parente commune faisant office d'abstraction qui peut représenter un concept pas vraiment courant dans le métier mais utile pour le modèle. Cependant, il faut que cette abstraction permette d'aider à la compréhension par le biais de l'analogie. Dans le cas contraire, la hiérarchie permettra de satisfaire le DRY et le polymorphisme mais va obscurcir la compréhension ce qui n'est pas l'objectif (cf. Figure 11 Hiérarchie avec une mauvaise abstraction). Une hiérarchie doit donc être implémentée lorsqu'elle permet de satisfaire les trois axes (DRY, polymorphisme et compréhension).

Il faut donc les évaluer avant d'arriver à la conclusion qu'une hiérarchie est mal implémentée. Il se peut effectivement qu'on n'appelle pas de méthodes héritées dans cette hiérarchie mais par contre qu'on utilise le polymorphisme (méthodes redéfinies), ce qui justifierait l'implémentation de cette hiérarchie de classes.



### 3. Taux d'héritage dynamique (THD)

Pour pallier aux déficiences des métriques proposées dans la littérature, nous proposons une nouvelle approche : la métrique THD. L'objectif de cette métrique est d'évaluer, dans une hiérarchie de classes, quel est le taux d'héritage utilisé dans un scénario réalisant une fonctionnalité métier. Afin de déterminer si les hiérarchies d'héritage implantées dans l'architecture sont réellement utilisées. L'utilisation effective d'un Framework ou Progiciel intégré au système analysé peut également être évaluée. La métrique proposée, serait une mesure permettant d'évaluer pour un scénario exécuté et pour chaque méthode impliquée, quel est, le rapport entre les méthodes implantées dans une classe et celles dont la classe hérite pour réaliser le comportement de la méthode. Pour ce faire, il faut se concentrer sur les classes d'implantation de chaque méthode : la classe implantant une méthode est celle qui contient le code source de la méthode.

L'interprétation de cette métrique se place dans le contexte de la stricte encapsulation, ce qui explique pourquoi la métrique ne prend pas en compte les attributs qui sont, dans l'état de l'art actuel, déclarés en Private et donc non hérités.

#### 3.1 Définition du THD

Le THD s'applique à chaque sous-structure utilisée pour répondre à la fonctionnalité demandée. Son calcul s'applique comme suit à chaque sous-structure utilisée pour implanter la fonctionnalité du scénario :

$$\text{THD}_{(S)} = \frac{\text{MIE}(s)}{\text{MIE}(s) + \text{MHE}(s)}$$

**MIE(S)**

est le nombre de **méthodes** implantées dans la sous-structure S et **exécutées**, pour un scénario donné.

**MHE(S)**

est le nombre de **méthodes** héritées par la sous-structure S et **exécutées**, pour un scénario donné.

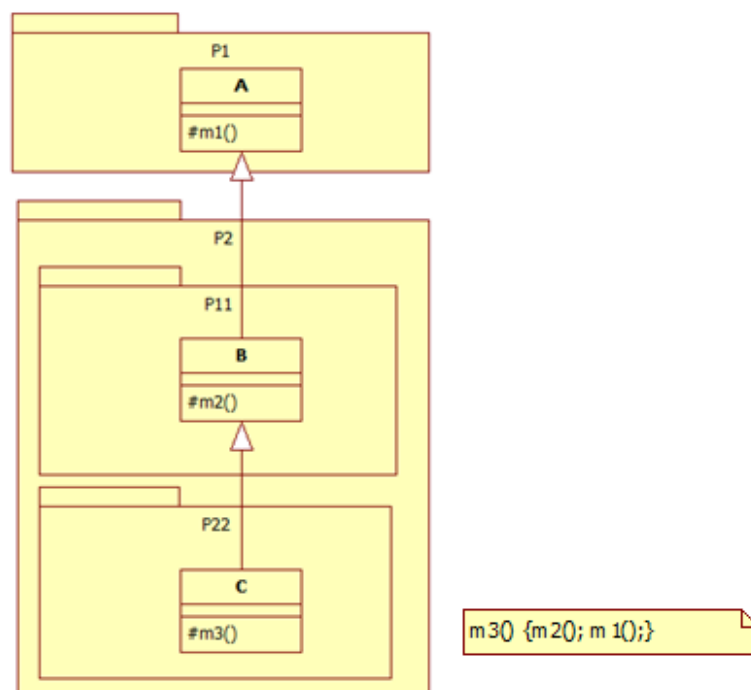
Le THD représente donc **le pourcentage des méthodes implantées et exécutées dans la sous-structure par rapport à toutes les méthodes exécutées (implantés et hérités.)**

Voici un exemple d'application de la métrique pour le scénario donné suivant :

Scénario : Appel de la méthode *m3()* sur une instance de la Classe *C*.

**Tableau 2** : Evaluations du THD pour la figure 17

Niveau d'agrégation	Sous-structure	THD	THD %
<b>Micro</b> (classes)	<b>C</b>	1/3	33.33%
	<b>B</b>	1/1	100%
	<b>A</b>	1/1	100%
<b>Macro</b> (packages)	<b>P22</b>	1/3	33.33%
	<b>P11</b>	1/1	100%
	<b>P2</b>	2/3	66.66%
	<b>P1</b>	1/1	100%



**Figure 17** : Application du THD sur une hiérarchie (1/2)

Ces résultats sont générés par une analyse dynamique des méthodes impliquées dans un scénario et non pas par une analyse statique de l'héritage potentiel entre les classes.

On remarque d'ailleurs qu'en additionnant tous les numérateurs des THD obtenus pour les sous-structures d'un même niveau d'agrégation, on obtient le nombre de méthodes exécutées pour délivrer la fonctionnalité.

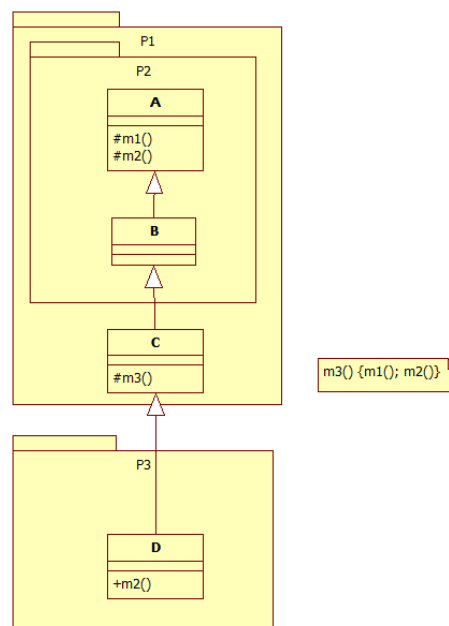
Dans le cas où la sous-structure est un package, la formule s'applique de la même manière. Il faut simplement considérer le périmètre de calcul pour la zone évaluée (le package en question). Toutes les méthodes définies dans les classes contenues dans le package analysé sont considérées comme non héritées, même s'il existe un lien de spécialisation entre ces classes, tandis que les méthodes comptées comme « héritées » sont celles provenant des classes localisées dans d'autres packages que celui analysé.

Voici un autre exemple :

*Scénario : Appel de la méthode m3() sur une instance de la Classe D.*

**Tableau 3** : Evaluations du THD pour la figure 18 (1/2)

Niveau d'agrégation	Sous-structure	THD	THD %
<b>Micro</b> (classes)	<b>D</b>	<b>1/3</b>	<b>33.33%</b>
	<b>C</b>	<b>1/2</b>	<b>50%</b>
	<b>B</b>	<b>N/A</b>	<b>N/A</b>
	<b>A</b>	<b>1/1</b>	<b>100%</b>
<b>Macro</b> (packages)	<b>P2</b>	<b>1/1</b>	<b>100%</b>
	<b>P3</b>	<b>1/3</b>	<b>33.33%</b>
	<b>P1</b>	<b>2/2</b>	<b>100%</b>

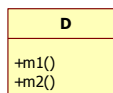


**Figure 18** : Application du THD sur une hiérarchie (2/2)

Concernant la classe D, la méthode m2() ayant été redéfinie, elle sera appelée par le code de m3(). Il faut donc la compter comme méthode implantée et exécutée car elle est impliquée dans le scénario.

La métrique THD permet d'apercevoir d'une part où est localisée la fonctionnalité nécessaire à la réalisation du scénario : on remarque ainsi par les ratios que les bouts de code exécutés se trouvent dans les classes D, C et A. D'autre part, la valeur du THD permet de déceler, pour chaque sous-structure utilisée, le degré de dépendance pour livrer sa fonctionnalité. Autrement dit, les classes A et D avec un THD de 1 sont indépendantes (elles n'ont pas besoin d'utiliser l'héritage pour délivrer la fonctionnalité qu'on leur demande) tandis que la classe C, pour réaliser ce qui lui est demandé, a besoin de faire appel à une classe parente. En l'occurrence ici, la classe A.

Prenons maintenant l'exemple de la classe D qui redéfinit la méthode héritée m1() :



Nous aurions obtenu les valeurs suivantes :

**Tableau 4** : Evaluations du THD pour la figure 18 (2/2)

Niveau d'agrégation	Sous-structure	THD	THD %
<b>Micro</b> (classes)	<b>D</b>	<b>2/3</b>	<b>66.66%</b>
	<b>C</b>	<b>1/1</b>	<b>100%</b>
	<b>B</b>	<b>N/A</b>	<b>N/A</b>
	<b>A</b>	<b>N/A</b>	<b>N/A</b>
<b>Macro</b> (packages)	<b>P2</b>	<b>N/A</b>	<b>N/A</b>
	<b>P3</b>	<b>2/3</b>	<b>66.66%</b>
	<b>P1</b>	<b>1/1</b>	<b>100%</b>

Dans le ce cas où toutes les méthodes héritées seraient redéfinies, la hiérarchie se révélerait inutile. Les classes A et B ne sont pas utilisées (d'ailleurs la sous-structure du niveau d'agrégation intermédiaire le déclare). Seule la méthode héritée m3() reste utile dans la classe C. Dès lors que des méthodes sont redéfinies, il est important d'évaluer la hiérarchie avec la métrique THD pour s'assurer de l'utilisation effective (au runtime) de la hiérarchie d'héritage.

Etant une métrique dynamique, le calcul du THD ne prend en compte que les méthodes utilisées pour répondre à la fonctionnalité du scénario exécuté au runtime. Contrairement à des métriques statiques telles que le MIF qui ne donnent qu'une moyenne sur l'ensemble du système, le THD permet d'être précis pour chaque sous-structure pour chaque scénario.

Si la valeur du THD est de 100 pour une sous-structure, cela signifie qu'il n'y a aucune utilisation de l'héritage pour un scénario donné. A l'inverse, si sa valeur est de 0, alors l'utilisation de l'héritage est totale : il n'y a pas de méthode exécutée qui soit déclarée dans la sous-structure, et la sous-structure est totalement dépendante des classes déclarées dans d'autres sous-structures. Lorsqu'une classe n'est pas utilisée pour délivrer la fonctionnalité, la valeur de la métrique sera 0/N (avec N un entier strictement positif). Grâce à ceci, il sera facile de pouvoir différencier les classes utilisées pour délivrer la fonctionnalité.

## 3.2 Motivation

Les métriques analysées jusqu'ici ont le point commun de mesurer les architectures logicielles d'un point de vue statique. On entend par statique l'analyse du code source du système indépendamment de son exécution. En revanche, l'analyse dynamique s'intéresse aux éléments de programmes exécutés lors d'une utilisation du système.

Il est facile grâce au bon nombre de métriques statiques existantes d'avoir un aperçu des hiérarchies de l'architecture d'un système. Cependant, l'évaluation statique ne présente qu'un niveau théorique de l'utilisation de l'héritage. Lors de l'utilisation du logiciel, il est possible que certaines parties de l'architecture, évaluées comme très intégrées au système par l'évaluation statique, ne soient en réalité que rarement utilisées

De plus, il est difficile de savoir quelles méthodes sont réellement utilisées, notamment à cause des différentes possibilités qu'apporte la programmation orientée objet telles que le polymorphisme, le sous-typage et les méthodes redéfinies (override). On ne peut parfois le savoir qu'au runtime du système. Pour pallier à cette difficulté, une métrique basée sur l'analyse dynamique permettrait de pouvoir les mesurer, contrairement à des métriques statiques.

Sachant les coûts qu'entraînent les maintenances, il est intéressant de relever qu'elles ne peuvent pas être évitées. D'après les lois de Lehman [13], un logiciel utilisé va forcément être amené à évoluer dû à l'évolution de l'entreprise. Autrement dit, il sera amené à être modifié. Il est donc indispensable de passer par des maintenances. De

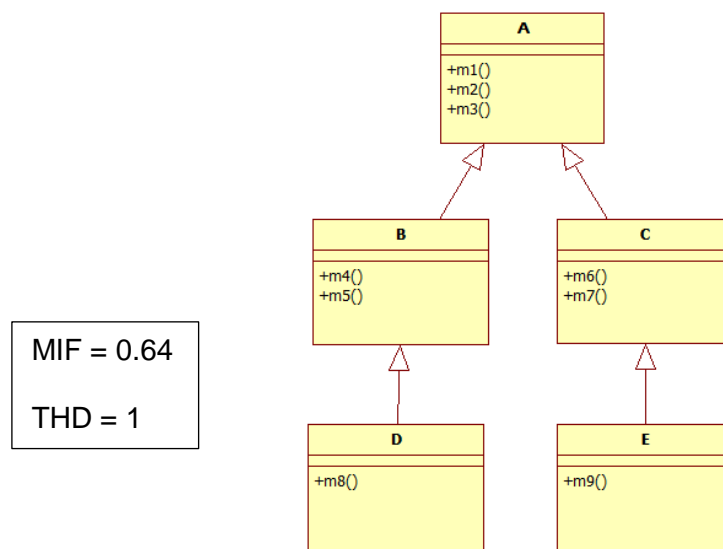
plus, dû à l'évolution du logiciel, sa structure aura tendance à se complexifier. Il faudra donc prendre des mesures pour inverser sa dégradation.

Voici les facteurs qui influencent la complexité [14] :

- **La taille du système** – Le nombre de classes dans un système.
- **La diversité** – Le nombre de classes dans un système.
- **La connectivité** – Les liens entre les classes (le couplage).

Lors d'ajouts de nouvelles fonctionnalités, la taille du système et la connectivité sont forcément impliqués étant donné que de nouveaux composants vont être conçus et doivent être couplés avec les composants déjà présents dans le système. Ceci suggère que la complexité des systèmes aura tendance à augmenter. Toutefois, avec une architecture adaptée au système et en surveillant cette augmentation de complexité, on peut s'en apercevoir et prendre des mesures adéquates pour arrêter sa croissance.

Cette métrique permettrait de se rendre compte de l'usage réel de l'héritage implémenté dans l'architecture, en d'autres termes l'héritage utilisé à l'exécution du système, et ainsi de savoir quels sont les classes vraiment impliquées dans la livraison de la fonctionnalité. En effet, il est possible qu'une hiérarchie soit présente mais il se peut qu'elle ne soit pas forcément utilisée. Prenons un exemple sur la hiérarchie ci-dessous.



Le THD est appliqué sur les instances des classes « D » et « E » du au scénario fictif envisagé.

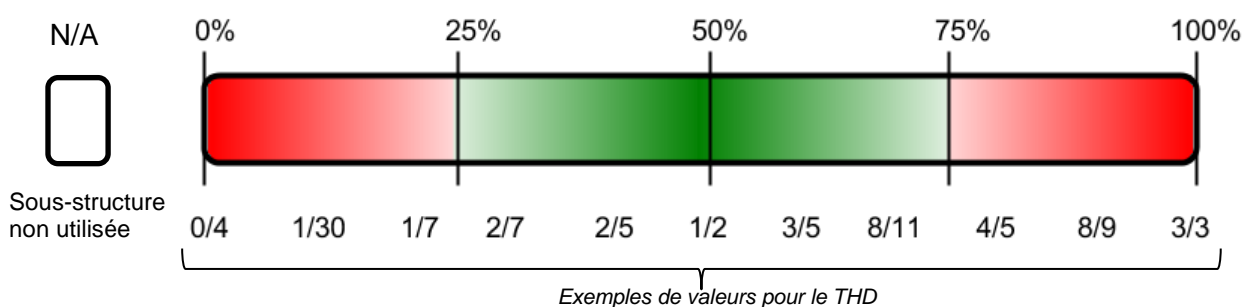
Figure 19 : Comparaison du THD et MIF

On remarque un ratio du MIF élevé qui indique que, dans ce système, il y a un héritage conséquent de 64%. Toutefois, en imaginant que les scénarios de ce système n'utilisent que les méthodes m8() et m9() des instances de classes « D » et « E » on aurait ici un DIF de 1 étant donné qu'à l'exécution du programme l'héritage n'est pas utilisé. Dans cet exemple simpliste, la hiérarchie instaurée n'étant jamais utilisée ne sert donc à rien.

### 3.3 Visualisation du THD

La représentation visuelle du THD va se faire sous forme de carte. Il y aura une carte par scénario. Chaque sous-structure sera représentée sous forme de boîte. Pour chaque carte, la sous-structure sera localisée au même endroit afin de pouvoir améliorer la facilité des comparaisons entre scénarios. Un code couleur va permettre de distinguer les différentes valeurs attribuées au THD. Ceci va permettre d'améliorer l'interprétation des résultats du THD sur l'architecture du système. La couleur verte est attribuée aux résultats du THD positifs en termes d'héritage utilisé pour délivrer la fonctionnalité requise. Le rouge permet de discerner les cas extrêmes (l'utilisation abusive de l'héritage et la non-utilisation de l'héritage pour fournir la fonctionnalité demandée). Le dégradé de ces deux couleurs permet de discerner la différence entre les valeurs obtenues (plus la couleur tend vers le vert, plus le THD est équilibré).

Figure 20 : Code couleur des cartes THD



Si l'héritage effectif utilisé par la sous-structure analysée s'approche des 50%, sa couleur aura tendance à devenir verte. Partant du principe qu'une sous-structure utilisée doit contribuer à la fonctionnalité demandée (autrement dit, elle doit apporter un plus en termes de fonctionnalité) et du fait qu'elle puisse faire appel aux fonctionnalités héritées grâce à la hiérarchie mise en place, le taux de 50% s'avère être une bonne valeur en terme d'équilibre.

Au contraire, une valeur de 100% montrerait qu'il n'y a pas d'héritage utilisé pour produire la fonctionnalité. Si la sous-structure évaluée se trouve dans une hiérarchie, cela voudrait dire qu'elle n'est pas utilisée (on remarque que les classes racine d'une hiérarchie auront forcément cette valeur, de même que les sous-structures ne faisant pas partie d'une hiérarchie). La couleur des sous-structures proches de 100% aura tendance à devenir rouge.

Les sous-structures s'approchant d'un THD de 0% montrent une dépendance à l'héritage importante. La couleur aura également tendance à tendre vers le rouge. On se rapproche du cas où la sous-structure en question n'apporte plus de fonctionnalité d'elle-même. En effet, une valeur de 0/N (avec N un entier strictement positif) montre que la sous-structure est totalement dépendante en termes d'héritage et n'apporte rien de nouveau en termes de fonctionnalité.

La valeur du THD pour chaque sous-structure peut être interprétée de la manière suivante :

**Tableau 5** : Interprétations selon la valeur THD

Interprétation	Degré	THD	THD %
Dépendante	Totale (0/N) (avec N un entier strictement positif)	0/9	N/A
		0/5	N/A
		0/1	N/A
	Elevée (>0% et < 25%)	1/20	5%
		3/33	9.09%
		1/5	20%
	Moyen (>=25% et <= 75%)	1/3	33.33%
		1/2	50%
		2/3	66.66%
	Faible (> 75% et < 100%)	4/5	80%
8/9		88.88%	
9/10		90%	
Indépendante	N/A (100%)	1/1	100%
		5/5	100%
		10/10	100%
Non utilisée	N/A (pas appelé)	N/A	N/A



Voici les significations ainsi que les vérifications à entreprendre sur la sous-structure évaluée pour chaque degré d'interprétation :

## **Dépendante**

### **Totale**

La sous-structure évaluée est totalement dépendante de l'héritage pour produire la fonctionnalité du scénario exécuté. En d'autres termes, toutes les méthodes utilisées sont héritées. Elle n'apporte rien en plus dans la fonctionnalité.

### **Vérifications**

Lorsqu'une sous-structure dépend totalement de ses classes parentes, il est important de vérifier si cette sous-structure est réellement nécessaire, car la classe parente peut réaliser le même travail demandé.

### **Elevée, Moyen, Faible**

La sous-structure évaluée est dépendante de l'héritage pour produire la fonctionnalité du scénario exécuté. Toutefois, contrairement à une dépendance totale, une ou plusieurs fonctionnalités internes à la sous-structure sont nécessaires pour réaliser la fonctionnalité. Le degré de dépendance permet de se rendre compte à quel point la fonctionnalité interne de la sous-structure est importante par rapport aux fonctionnalités héritées.

### **Vérifications**

Si le degré de dépendance est élevé, il est important de vérifier la pertinence de la fonctionnalité interne à la sous-structure nécessaire à la réalisation du scénario. Dans le cas négatif, il serait simple d'implanter la fonctionnalité dans la sous-structure parente qui réalise déjà en grande partie le scénario exécuté.

Cependant, si le degré de dépendance est faible, il faut vérifier que le peu de fonctionnalités héritées nécessaires à la réalisation du scénario soient justifiées. Autrement, il suffirait de redéfinir les fonctionnalités dans la sous-structure analysée afin de ne pas avoir recourt à l'héritage et pouvoir remettre en question la hiérarchie.

## **Indépendante**

La sous-structure évaluée n'as pas besoin d'héritage pour produire la fonctionnalité du scénario exécuté.

## **Vérifications**

Dans le cas d'une hiérarchie, il est important de vérifier que les méthodes de la sous-structure analysée ne soient pas des méthodes héritées et redéfinies, ce qui signifierait que la hiérarchie d'héritage a été dénigrée par la redéfinition des méthodes. Il faut que ce soit justifié, et dans ce cas, la hiérarchie n'étant pas utilisée est à remettre en question. Cependant, si ce n'est pas justifié, des erreurs d'implémentation peuvent être décelées.

## **Non utilisée**

La sous-structure évaluée n'est pas utilisée pour produire la fonctionnalité du scénario exécuté.

## **Remarque générale**

Toutes les constatations faites dans les vérifications de chaque degré d'indépendance sont à prendre avec précaution. Il faut pour cela avoir évalué un THD similaire sur plusieurs scénarios. En effet, l'évaluation du THD s'effectue pour un scénario donné. Il faut donc, avant d'amorcer un quelconque changement dans l'architecture ou l'implantation de méthodes du système, analyser le système avec tous les scénarios du logiciel. Autrement dit, il faudrait évaluer toutes les fonctionnalités du logiciel utilisées dans le métier (celles qui ont de la valeur pour l'entreprise). De plus, il faut également vérifier que la hiérarchie évaluée inutilisée par le THD ne soit déployée afin de pouvoir satisfaire les trois axes (DRY, polymorphisme et compréhension), ce qui justifiait son implémentation.

Cependant, il ne faudrait pas non plus que cette hiérarchie ne soit implémentée que pour bénéficier de l'un de ces trois axes. Par exemple, en concevant une classe parente commune uniquement pour pouvoir bénéficier du DRY ou du polymorphisme sur un ensemble d'objets alors que cette abstraction brouille la compressibilité. (cf. Figure 11 Hiérarchie avec une mauvaise abstraction). Une hiérarchie d'héritage est justifiée dans le cas où les trois axes sont réalisés en même temps. Dans le cas contraire, il faut la remettre en question.

### 3.4 Cartes THD

Voici une visualisation de la distribution du THD sur l'architecture d'une application java développée pour l'exemple (un gestionnaire de véhicules). Vous pouvez consulter la hiérarchie de classes du projet (cf. annexe 1).

*Après analyse du scénario 1)*



*Après analyse du scénario 2)*



Figure 21 : Exemples de carte THD

Fonctionnalités du projet java GestionnaireVehicules :

- Afficher le gestionnaire (la liste des véhicules)
- Sélectionner un véhicule dans la liste ouvre une fenêtre contenant ses détails.
- Modifier les détails du véhicule sélectionné à partir de sa fenêtre de détail.

Chaque sous-structure est représentée par une boîte. Une sous-structure qui contient d'autres sous-structures est forcément un package.

(ex : « gestionnaireVehiculesMOD », « VehiculesSpecifiques » ou « TypesVehicules »)  
Tandis qu'une sous-structure qui ne contient rien est forcément une classe. Ces cartes sont basées sur la technique de treemapping, qui permet de produire des cartes graphiques montrant une structure hiérarchique. Le titre de la boîte contient le nom de la sous-structure, suivie de sa valeur THD évaluée pour le scénario analysé. Si il n'y a pas eu d'évaluation dans ce scénario d'une sous-structure mais qu'elle a été utilisée dans un autre scénario analysé, à ce moment la sous-structure est de couleur blanche.

Dans cet exemple, les classes finissant par le nom « View » sont les classes qui permettent d'afficher la fenêtre contenant les détails du véhicule en question.

Les classes « CamionView » et « BateauView » apparaissent en blanc dans le scénario 2. Tout simplement car dans le scénario deux il n'y a que l'objet « Voiture » qui a été sélectionné dans la liste des véhicules.

Dans le scénario 1, on remarque que les objets « Voiture », « Camion » et « Bateau » ont été sélectionnés. Tandis que l'objet « HelicoptereView » n'apparaît pas dans les différents scénarios. Ceci montre que l'objet « Helicoptere » n'a jamais été sélectionné (dans aucun des deux scénarios, il n'y a donc aucun intérêt à qu'il soit représenté).

Notre représentation visuelle ne permet pas de représenter simultanément les métriques et les liens d'héritage entre classes. Cependant je conseille fortement d'effectuer les vues des hiérarchies à part, afin de pouvoir visualiser les liens d'héritages entres classes et sous-structures ce qui permettra d'établir des relations. Par exemple, des sous-structures se retrouvant souvent dans le même cas (ex : 0/N rouge) se retrouvent d'après la vue de la hiérarchie à toujours hérité d'un certain Framework ou bien des sous-structures toujours évaluées avec un THD de 100%, justifié par le fait qu'elles ne sont pas implantées dans une hiérarchie d'héritage.

Exemple : Scénario 2 : Appel de la méthode `m2()` sur une instance de la Classe `C1` .

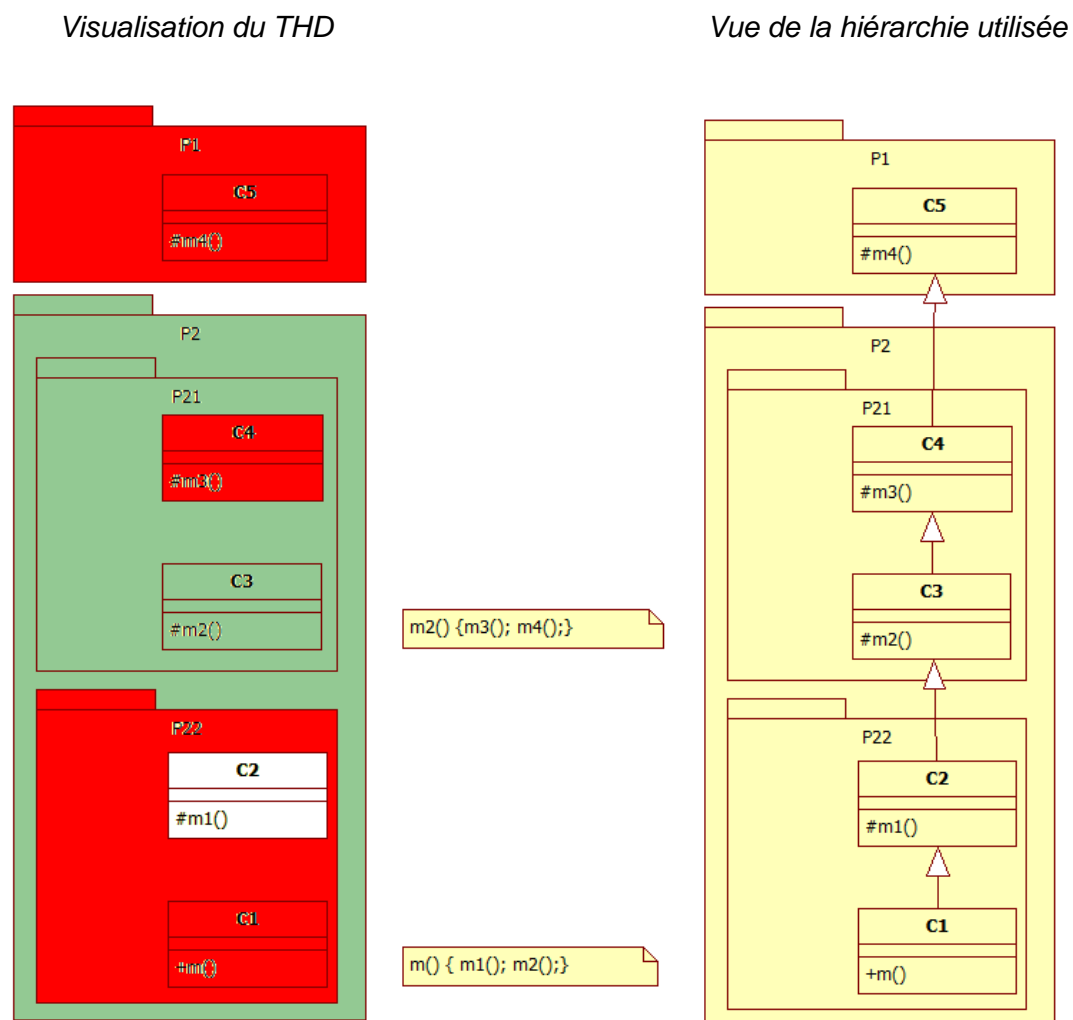


Figure 22 : Exemple de visualisation THD et diagramme d'une hiérarchie

### 3.5 Mesurer le THD

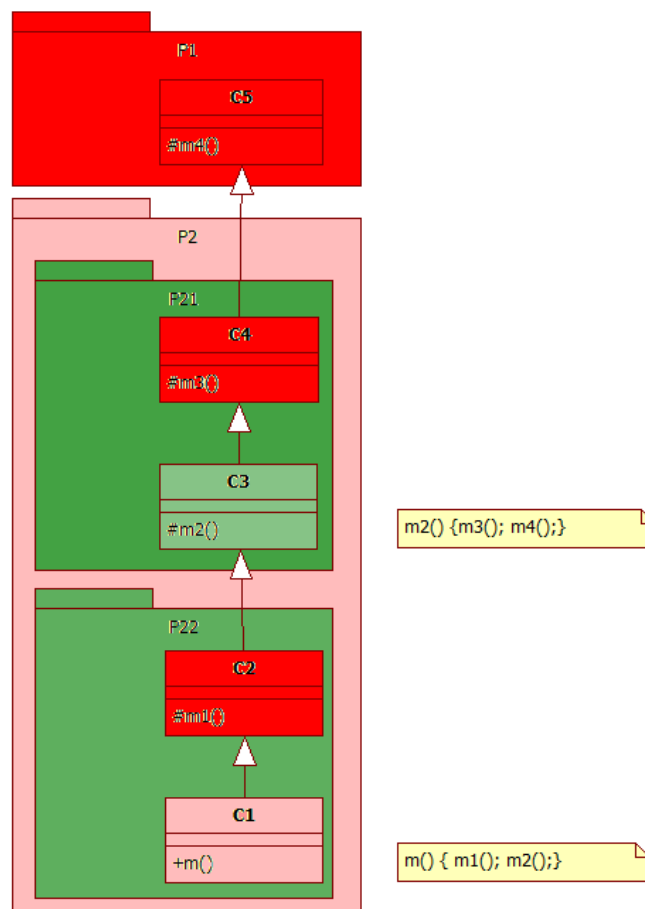
Voici un exemple de calculs de THD ainsi qu'une visualisation sur l'appel d'une fonctionnalité appelant la méthode m :

Scénario 1 : Appel de la méthode m() sur une instance de la Classe C1.

Tableau 6 : Evaluations du THD pour la figure 23

Niveau d'agrégation	Sous-structure	THD	THD %
<b>Micro</b> (classes)	<b>C1</b>	1/5	20%
	<b>C2</b>	1/1	100%
	<b>C3</b>	1/3	33.33%
	<b>C4</b>	1/1	100%
	<b>C5</b>	1/1	100%
<b>Macro</b> (packages)	<b>P22</b>	2/5	40%
	<b>P21</b>	2/3	66.66%
	<b>P2</b>	4/5	80%
	<b>P1</b>	1/1	100%

Figure 23 : Exemple de visualisation et évaluation du THD (1/2)



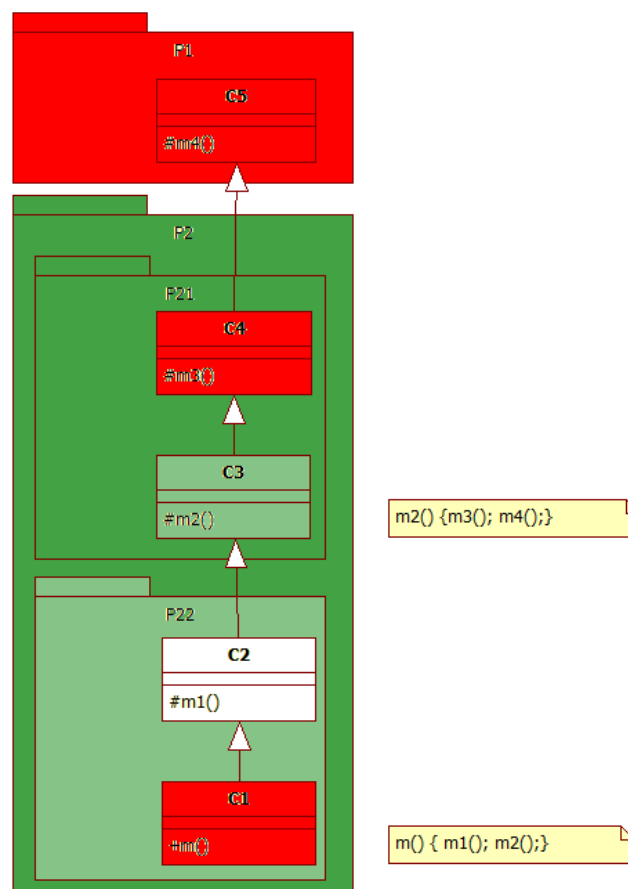
Autre exemple appelant la méthode m2().

Scénario 2 : Appel de la méthode m2() sur une instance de la Classe C1 .

Tableau 7 : Evaluations du THD pour la figure 24

Niveau d'agrégation	Sous-structure	THD	THD %
<b>Micro</b> (classes)	<b>C1</b>	0/3	0%
	<b>C2</b>	N/A	N/A
	<b>C3</b>	1/3	33.33%
	<b>C4</b>	1/1	100%
	<b>C5</b>	1/1	100%
<b>Macro</b> (packages)	<b>P22</b>	0/3	0%
	<b>P21</b>	2/3	66.66%
	<b>P2</b>	2/3	66.66%
	<b>P1</b>	1/1	100%

Figure 24 : Exemple de visualisation et évaluation du THD (2/2)



### 3.6 Interprétations

Le THD permet de déceler des Framework mal ou non utilisés dans un système. Prenons l'exemple d'un système utilisant le Framework Hibernate.

Hibernate est un Framework open source gérant la persistance des objets en base de données relationnelle. Très souple, ce Framework peut être utilisé dans tous types de développement (client lourd, environnement web, J2EE). Il facilite la vie des développeurs grâce à des interfaces basiques. Il remplace l'accès à la base de données par des appels à des méthodes de haut niveau. Pour sauvegarder un objet il suffit de quelques appels de méthodes.

Toutefois, d'après le travail de recherche de D.Goman [12], environs 400 classes sont utilisées pour mapper cinq classes et effectuer quelques opérations CRUD. C'est un nombre important à regard de la simplicité du travail demandé à Hibernate. La complexité de ce Framework est donc un élément à prendre en compte lorsqu'on décide d'utiliser cet API. D'après A.Patricio [15], « pour un développeur moyennement expérimenté, la courbe d'apprentissage <...> est généralement estimée de quatre à six mois pour en maîtriser les 80% de fonctionnalités les plus utilisées. » Grâce au THD il serait possible d'évaluer l'utilisation d'héritage utilisé à l'exécution du programme et de se rendre compte grâce aux niveaux d'agréations si le Framework est réellement utilisé.

Si on découvre que dans la majorité des scénarios, les méthodes héritées par ce Framework ne sont quasiment jamais utilisées, on pourrait remettre en question son utilisation. Implémenter une API complexe qui n'est jamais utilisée est en soi un problème à éviter. Il se pourrait aussi que le Framework soit mal configuré ou que les développeurs ne respectent pas les patterns imposés par le Framework.

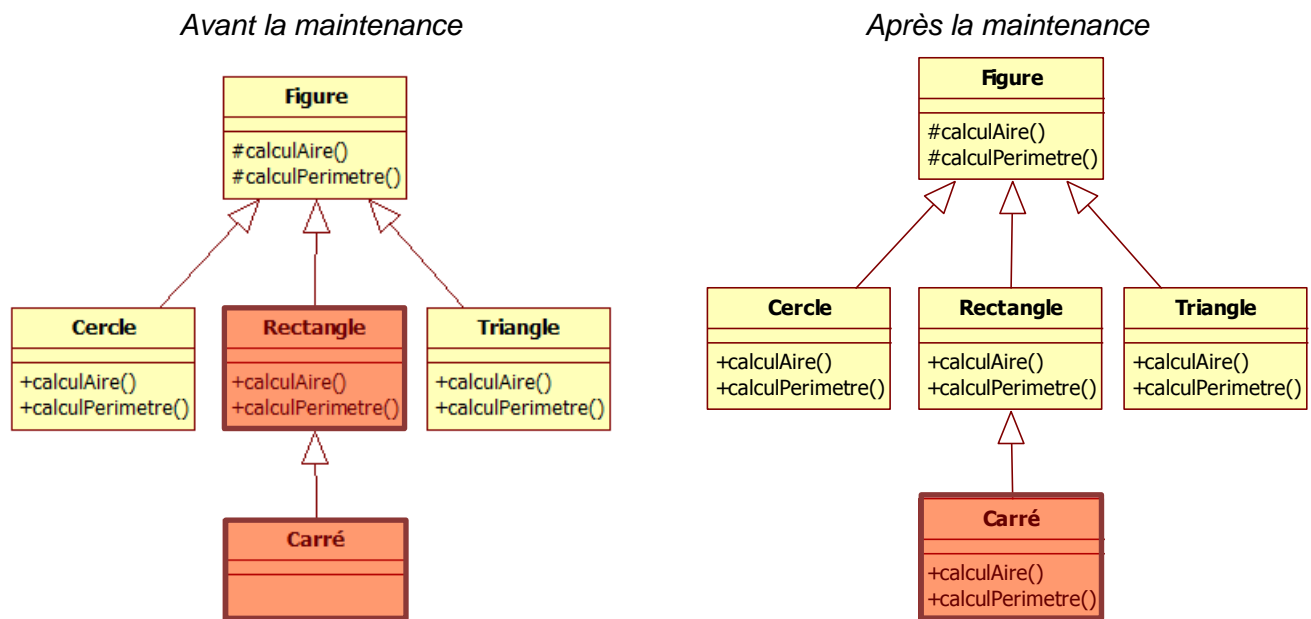
On pourrait également imaginer de voir une corrélation entre le nombre de maintenances effectuées et les valeurs du THD pour chaque scénario analysé. Ceci permettrait d'inspecter la qualité des maintenances. A savoir est-ce que les



développeurs prennent le temps d'analyser la hiérarchie afin d'en faire un usage optimum (réutilisation) dans l'implantation des modifications.

Prenons l'exemple d'un logiciel et ses maintenances de modifications d'une fonctionnalité.

**Figure 25** : Avant / Après une maintenance avec un modèle de Figures (1/2)



Avant la maintenance, un scénario fictif ayant pour objectif de mesurer un carré utilisait les méthodes calculAire() et calculPerimetre() de l'instance de la classe Carré. Après la maintenance, le scénario n'a pas changé, sauf que le développeur a redéfini la méthode calculAire() et calculPerimetre() dans la classe Carré. On obtient donc pour ce scénario les valeurs du THD suivantes :

**Scénario : Affichage des caractéristiques d'une instance de Carré.**

*Appel de la méthode calculAire() et calculPerimetre() sur une instance de Carré.*

*Avant la maintenance, THD de la classe Carré = 0/2,*

**THD de la classe Rectangle = 2/2**

*Après la maintenance, THD de la classe Carré = 2/2,*

**THD de la classe Rectangle = N/A**

Dans cet exemple simple, on remarque que le THD d'un scénario évolue après une maintenance. Cet exemple n'est pas anodin. En effet, dû au temps restreint d'implémentation des maintenances, il est tout à fait logique de penser que la qualité de la maintenance ne soit pas optimale.

Dans cet exemple, le développeur a préféré redéfinir la méthode `calculAire()` et `calculPerimetre()` de la classe Carré pour ajouter une spécification. Toutefois, dans cet exemple, les calculs sont applicables de la même façon pour un Rectangle ou un Carré, ou encore n'importe quelle classe dérivant d'un Rectangle. Dans ce cas-là, il aurait dû modifier ces méthodes dans la classe Rectangle afin de ne pas restreindre l'héritage implanté.

Cependant, cet exemple montre également qu'avant la maintenance la classe Carré n'apporte rien de plus en termes de fonctionnalité comparé à sa classe parente Rectangle. Autrement dit, il aurait suffi d'appeler une instance de la classe Rectangle qui aurait très bien pu faire le travail toute seule. On remarque donc par la valeur 0/2 obtenu de la classe Carré qu'elle est inutile en termes de fonctionnalité réalisée. C'est la classe Rectangle qui est indispensable pour réaliser la fonctionnalité demandée. Ceci permet donc de dire qu'avant la maintenance la classe Carré pour ce scénario n'était pas utile, mais après la maintenance c'est la classe Rectangle qui n'est plus utilisée et la classe Carrée est indispensable. De plus les ratios de 1 évalués pour la classe Rectangle avant la maintenance et la classe Carré après la maintenance permettent d'affirmer que ces classes sont indépendantes pour délivrer leurs fonctionnalités elles n'ont pas recourt à l'héritage.

Le THD serait utilisé comme mesure pour éviter la dégradation d'un système dû à son évolution qui aura tendance à le complexifier. Les personnes chargées de la maintenance pourront vérifier la qualité du travail effectué. On pourrait même planifier un THD à respecter en se calquant sur la maintenance précédente. Par exemple, un des objectifs pourrait être de ne pas diminuer le THD évalué par sous-structure avant la maintenance. Ce qui permettrait de contrôler si le travail effectué a su préserver la qualité de la hiérarchie actuelle. Dès lors qu'on modifie, rajoute ou supprime une fonctionnalité, le THD serait intéressant. Ceci englobe donc tous les types de maintenances, que ce soit les maintenances correctives, préventives mais surtout évolutives.

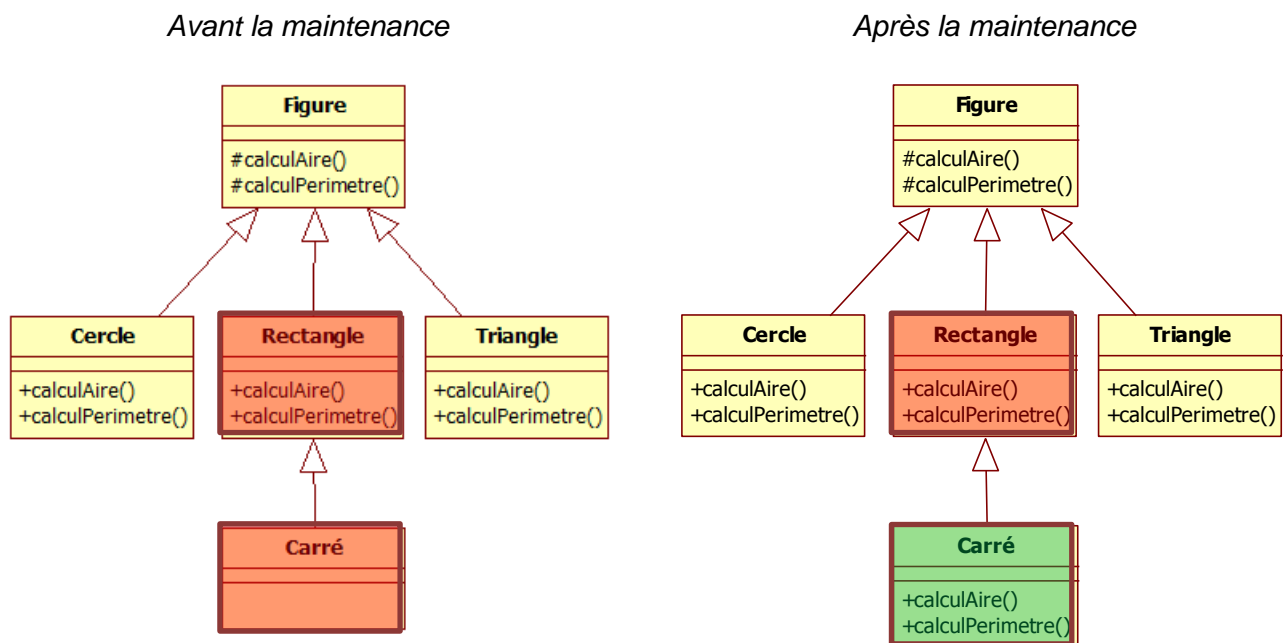
```
calculAire() {super.calculAire();}  
calculPerimetre() {super.calculPerimetre();}
```

Figure 26 : Code des méthodes pour la figure 27

Voici un exemple similaire, avec les mêmes scénarios fictifs. La différence réside dans le code des méthodes `calculAire()` et `calculPerimetre()`. Cette fois ci ces méthodes sont redéfinies mais font appel au code implanté dans la méthode héritée dans la classe parente en appelant la méthode avec le `super` en Java.

L'utilisation du `super` va remonter dans la hiérarchie jusqu'à retrouver la classe qui implante ou redéfinit la méthode `calculAire()` et `calculPerimetre()`.

**Figure 27** : Avant / Après une maintenance avec un modèle de Figures (2/2)



On obtient donc pour le même scénario les valeurs du THD suivantes :

**Scénario : Affichage des caractéristiques d'une instance de Carré.**

*Appel de la méthode `calculAire()` et `calculPerimetre()` sur une instance de Carré.*

*Avant la maintenance, THD de la classe Carré = 0/2,*

**THD de la classe Rectangle = 2/2**

*Après la maintenance, THD de la classe Carré = 2/4,*

**THD de la classe Rectangle = 2/2**

Même si les méthodes ont été redéfinies, le `super` utilise l'héritage en faisant appel aux méthodes de la classe Rectangle. Pour que le scénario soit réalisé il faut exécuter les deux méthodes appelées dans l'instance de la classe Carré (`calculAire()` et `calculPerimetre()`). Puis le `super` appelle les méthodes de la classe Rectangle qui sont

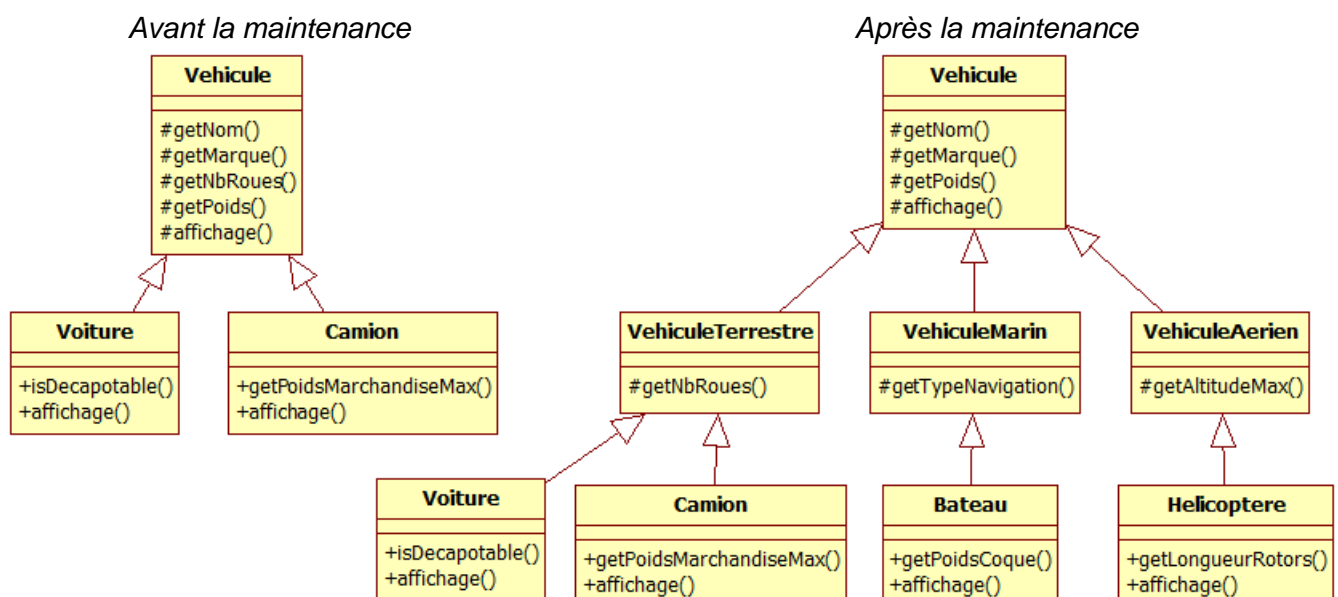
également nécessaire à la réalisation du scénario (`calculAire()` et `calculPerimetre()`). Il y a donc une différence d'évaluation du THD sur la classe Carré de 50% comparé au premier exemple. En effet, dans celui-ci, il suffisait d'utiliser les deux méthodes implantées dans la classe Rectangle héritées par la classe Carré. Cependant ici, nous avons besoin des méthodes redéfinies dans la classe Carré également. (En effet, l'objectif de cette implémentation est de réaliser la fonctionnalité héritée grâce au super ainsi que le code écrit après le super.)

Cet exemple vient d'être appliqué sur une hiérarchie simple utilisant des figures afin d'améliorer sa compréhension. Cependant il est facile de s'imaginer le même type de maintenance sur un logiciel plus grand utilisant un Framework ou encore un Progiciel dont après une maintenance on se retrouve avec des méthodes redéfinies au lieu d'utiliser le Framework par l'héritage.

Il est donc intéressant d'utiliser le THD avant et après une modification, rajout ou suppressions de méthodes dans une hiérarchie de classes. Ainsi, on peut s'assurer que la hiérarchie mise en place est toujours réellement utilisée à l'exécution des fonctionnalités du système.

C'est également le cas pour un changement dans la structure de la hiérarchie de classes. Prenons l'exemple d'un logiciel qui voit son architecture changée après une maintenance afin d'améliorer ses performances due à des rajouts de classes dans la hiérarchie.

**Figure 28** : Avant / Après une maintenance avec un modèle de Véhicules (1/3)



On remarque dans la hiérarchie avant la maintenance qu'il n'y avait qu'une profondeur de deux niveaux d'héritage. La maintenance a trouvé judicieux de rajouter un niveau servant d'abstraction améliorant ainsi la compréhension de la hiérarchie actuelle due aux nouvelles classes implémentées (Bateau et Hélicoptère). En effet, l'entreprise fictive prévoit d'obtenir de nouveaux types de véhicules dans un futur proche, ce qui argumente la mise en œuvre de cette nouvelle hiérarchie de classes.

Il est très intéressant de pouvoir comparer les valeurs obtenues par le THD sur ces deux architectures différentes pour un même scénario donné afin de vérifier qu'on utilise toujours l'héritage de la hiérarchie. En focalisant le scénario fictif permettant de visualiser la fiche du véhicule Voiture on obtient les valeurs suivantes :

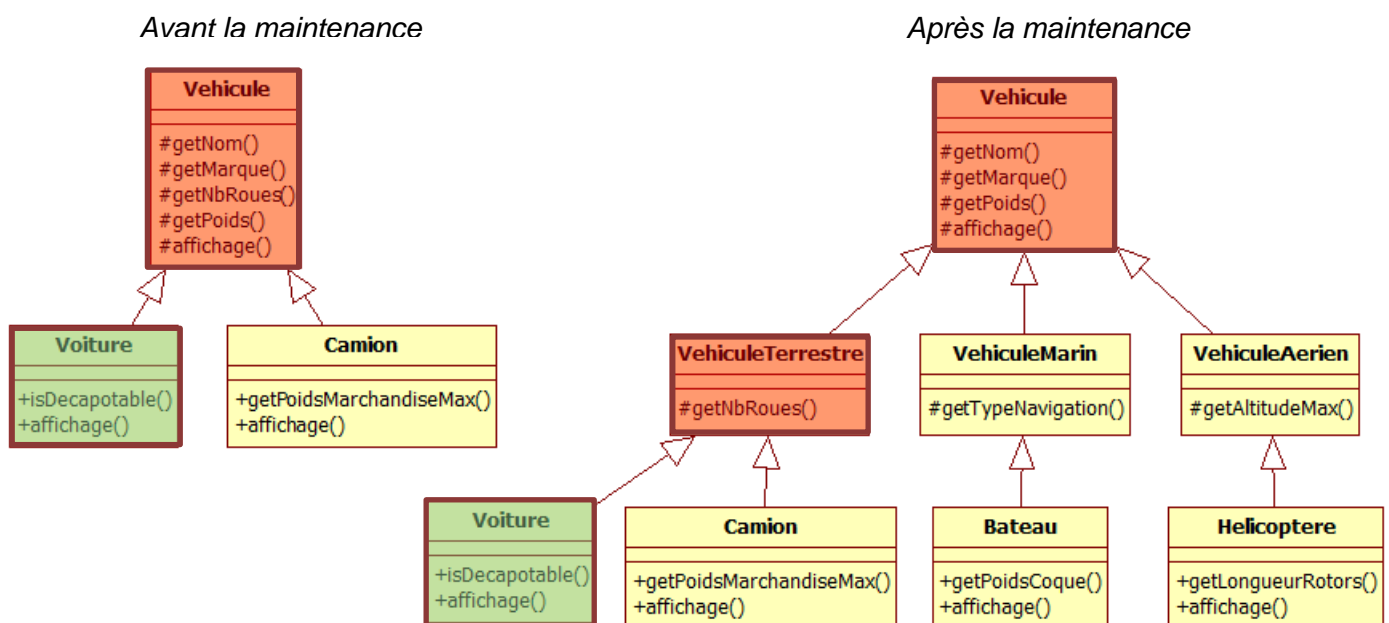


Figure 29 : Avant / Après une maintenance avec un modèle de Véhicules (2/3)

**Scénario : Affichage d'une instance de Voiture**  
*Appel de la méthode affichage() sur une instance de Voiture.*

```
affichage() { isDecapotable(); getNbRoues(); getNom(); getMarque(); getPoids(); }
```

*Avant la maintenance*, THD de la classe Voiture = **2/6**,  
 THD de la classe Vehicule = **4/4**

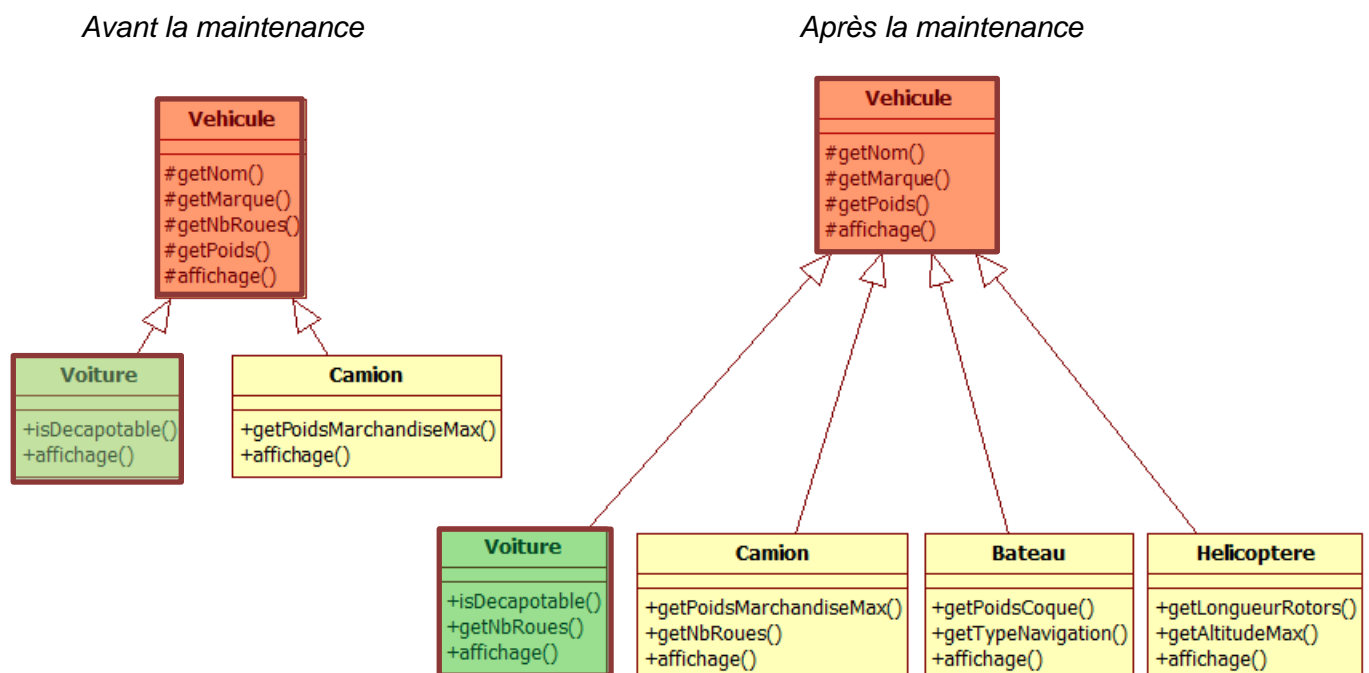
*Après la maintenance*, THD de la classe Voiture = **2/6**,  
 THD de la classe VehiculeTerrestre = **1/1**,  
 THD de la classe Vehicule = **3/3**

Après l'évaluation des valeurs du THD, on remarque que le THD reste inchangé avant et après la maintenance pour la classe Voiture. En effet, il n'y a pas eu de redéfinitions des méthodes héritées dans la classe Voiture. Autrement dit, la classe voiture a besoin de la fonctionnalité implantée dans ses classes parentes, ce qui est un bon point. On a besoin des méthodes implantées dans les classes VehiculeTerrestre et Vehicule. D'ailleurs la visualisation permet de voir que même la classe racine Vehicule est nécessaire. On remarque le changement d'implantation de la méthode « getNbRoues() » de la classe Vehicule vers la classe VehiculeTerrestre qui est correctement fait par le biais de l'abstraction. En effet, cette méthode s'applique uniquement aux véhicules terrestres.

On peut donc conclure après l'application de cette métrique que la restructuration de la hiérarchie est un succès. En effet, il n'y a pas eu de perte d'utilité de la hiérarchie car son héritage est toujours nécessaire pour la réalisation du scénario.

Prenons maintenant l'exemple inverse où la hiérarchie restructurée est moins utile que la hiérarchie avant la maintenance.

Figure 30 : Avant / Après une maintenance avec un modèle de Véhicules (3/3)



### Scénario : Affichage d'une instance de Voiture

Appel de la méthode `affichage()` sur une instance de Voiture.

```
affichage() { isDecapotable(); getNbRoues(); getNom(); getMarque(); getPoids(); }
```

Avant la maintenance, **THD** de la classe Voiture = **2/6**,

**THD** de la classe Vehicule = **4/4**

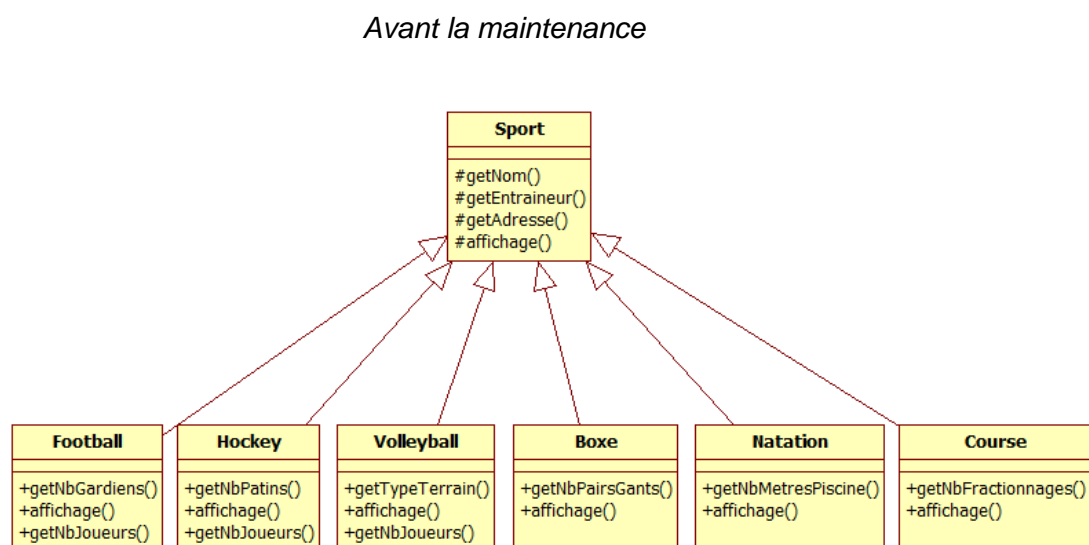
Après la maintenance, **THD** de la classe Voiture = **3/6**,

**THD** de la classe Vehicule = **3/3**

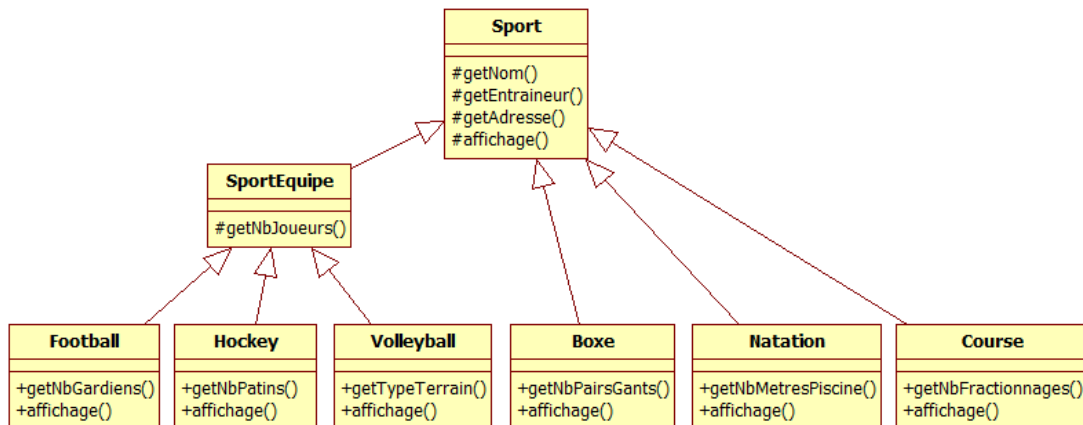
Ici on constate que la classe Voiture n'as pas besoin d'autant de fonctionnalités de sa classe parente. En effet, étant donné qu'il n'y a pas de niveau d'abstraction dans cette hiérarchie, les développeurs ont été obligés de redéfinir les méthodes spécifiques aux types de véhicules dans les classes concernées (par exemple, `getNbRoues()`). La même méthode est implantée dans la classe Voiture et Camion ce qui ne respecte pas le principe DRY. En effet, lors d'une modification de cette méthode, il faudra l'implémenter dans chaque classe où la méthode est implantée.

Voici maintenant un exemple différent qui montre la mise en place d'une abstraction lors d'une maintenance.

**Figure 31** : Avant / Après une maintenance avec un modèle de Sports (1/2)



### Après la maintenance

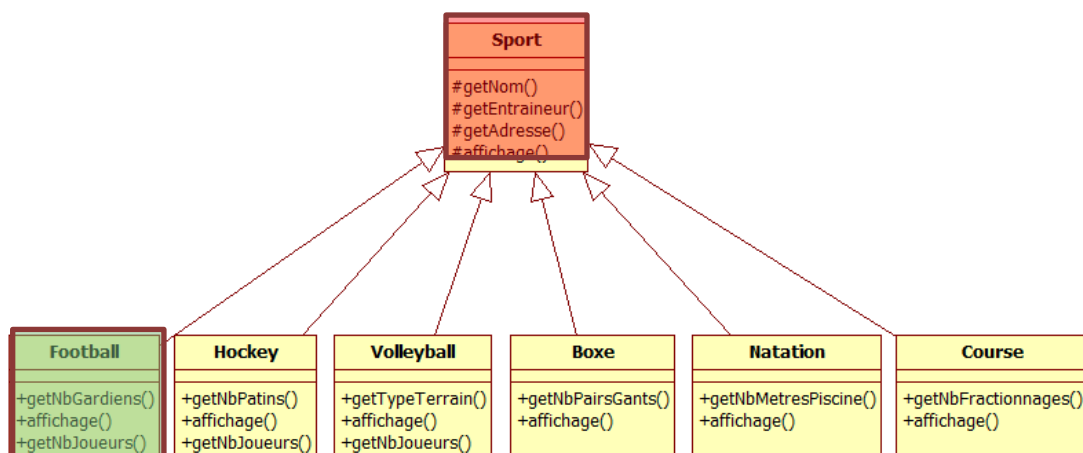


Les développeurs de la maintenance ont remarqué que la méthode `getNbJoueurs()` avait été répétée dans les classes `Football`, `Hockey` et `Volleyball` et ne respecte donc pas le principe DRY. Afin de palier à ce problème, ils ont inséré une classe parente permettant ainsi l'héritage de cette méthode. Leur choix paraît à première vue très adéquat.

Cependant, on remarque que la méthode `getNbJoueurs()` n'est utilisée dans aucun des scénarios de l'application. Il avait sûrement été développé à la première version de l'application en prévoyant son besoin futur. Cependant l'entreprise utilisant ce logiciel n'en a jamais eu besoin et aucune fonctionnalité du système ne fait appel à cette méthode.

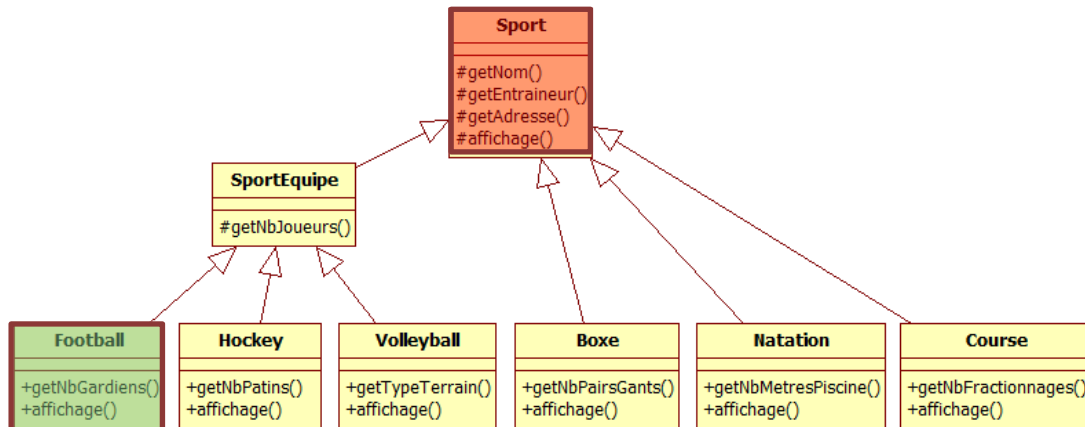
Figure 32 : Avant / Après une maintenance avec un modèle de Sports (2/2)

### Avant la maintenance





*Après la maintenance*



### Scénario : Affichage d'une instance de Football

*Appel de la méthode affichage() sur une instance de Football.*

```
affichage() { getNbGardiens(); getNom(); getEntraîneur(); getAdresse(); }
```

*Avant la maintenance*, THD de la classe Football = **2/5**,

THD de la classe Sport = **3/3**

*Après la maintenance*, THD de la classe Football = **2/5**,

THD de la classe SportEquipe = **N/A**,

THD de la classe Sport = **3/3**

Grâce au THD, on réalise que la fonctionnalité de la classe SportEquipe n'est jamais appelée par l'héritage dans aucun scénario. Le THD permet ainsi de vérification d'un choix de la maintenance sur la modification de structure d'une hiérarchie. C'est d'ailleurs dans ce type de cas, où l'on croit que le choix de modification est évident que le THD permettra de déceler si cette modification apporte réellement un plus en termes de fonctionnalités héritées utilisées.

## 4. Mise en œuvre de la métrique THD.

La mise en pratique est générée en deux parties :

1. Génération de traces d'exécution
2. Utilisation des traces obtenues par les outils d'analyse de traces.

L'analyse dynamique d'un système de logiciel est basée sur la trace d'exécution du système obtenue lors de l'exécution du système après utilisation d'un scénario. La première étape de l'analyse dynamique est de produire la trace d'exécution. Ensuite, nous avons utilisé un plugin Eclipse et une bibliothèque d'exécution pour générer la trace de tout programme Java. La trace est générée écrit dans un fichier texte. Ensuite, la trace doit être chargée dans une base de données pour traitement.

On peut utiliser n'importe quelle base de données. Les fichiers texte de trace peuvent être assez importants, leur taille peut varier de quelques Mo à plusieurs Go. Par conséquent, la base de données doit prendre en charge un tel volume de données.

Pour tester ou pour charger de petits fichiers, l'utilisateur peut utiliser une base de données ODBC tels que MS Access. Toutefois, pour une meilleure performance et ainsi pouvoir utiliser de grands fichiers de trace, il est conseillé de travailler avec Oracle.

Pour être en mesure de reproduire la hiérarchie d'appel de méthode, nous enregistrons un événement à l'exécution d'un scénario lorsqu'on rentre dans une méthode et quand on en sort. La trace d'exécution est donc représentée par un ensemble d'événements ayant le format suivant [18] :

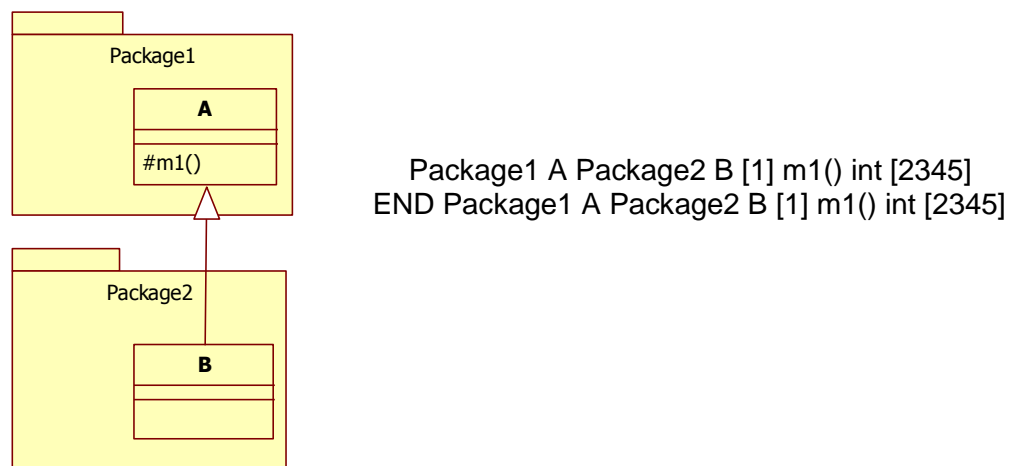
1. [SP] [SC] [DP] [DC] '[' [TN] ']' [MS] [RT] '[' [TS] ']' [PV]
- or
2. 'END' [SP] [SC] [DP] [DC] '[' [TN] ']' [MS] [RT] '[' [TS] ']'

Where:

- [SP] : full package name of [SC] ("static" package)
- [SC] : class where the called method is defined ("static" class)
- [DP] : full package name of [DC] ("dynamic" package)
- [DC] : class of the instance that received the message ("dynamic" class)
- [TN] : thread number
- [MS] : signature of the called method
- [RT] : returned type of the called method
- [TS] : time stamp of the call
- [PV] : parameter values of the called method (printable parameters only)

Prenons un exemple pratique sur la hiérarchie représentée sur la figure 33. Pour un scénario fictif utilisant une instance de la classe B qui appellera la méthode m1() nous aurions l'événement d'exécution suivant :

Figure 33 : Exemple de trace pour un appel de méthode

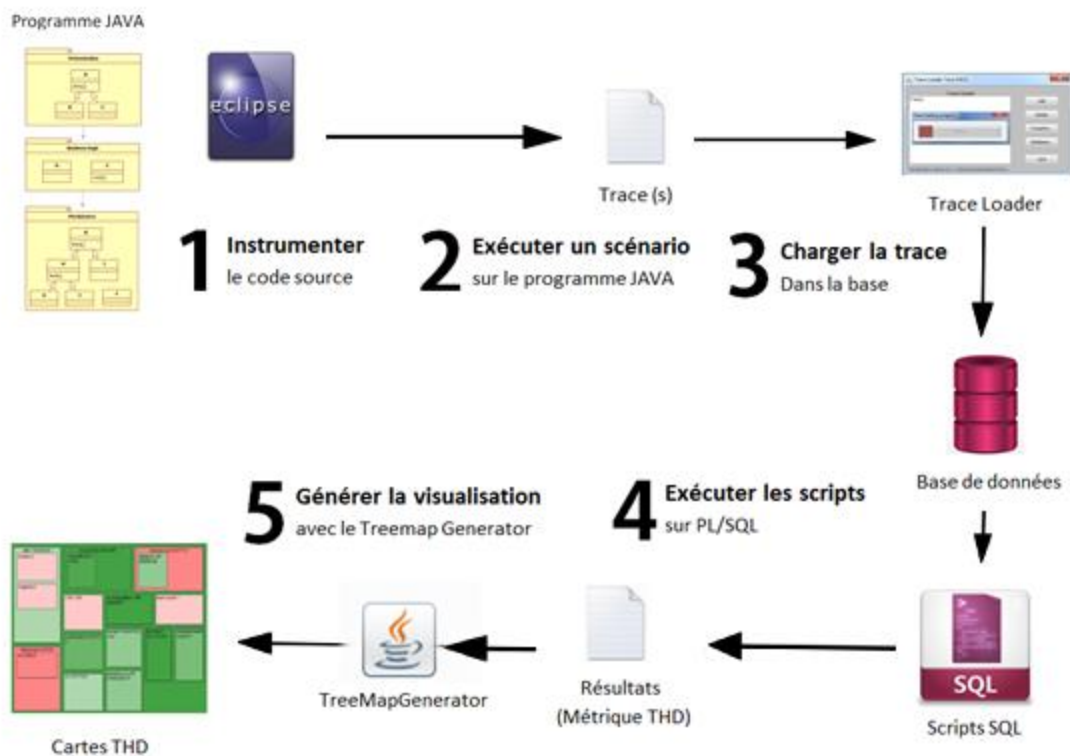


Il faut ensuite charger la trace d'exécution dans une base de données. Pour ce faire il faut utiliser le TraceLoader (programme conçu par Ph. Dugerdil [18]). Une fois tous les fichiers traces chargés (un fichier par scénario), il faut exécuter des scripts SQL permettant de calculer la métrique THD sur l'ensemble des scénarios tracés ce qui va nous produire dans une table SQL les résultats pour chaque scénario et chaque sous-structure. Par la suite en utilisant le Teemap Generator (programme conçu par M.Niculescu dans le cadre de son travail de Bachelor [21]), on va pouvoir créer un fichier par scénario contenant le format adéquat pouvant être lu par Microsoft Treemappper. Ce dernier logiciel permet de visualiser des cartes contenant les sous-structures du système avec les valeurs obtenues pour le THD ainsi que les couleurs permettant d'accentuer la compréhension.

C'est ainsi qu'on peut analyser dynamiquement un logiciel. En effet, à l'inverse d'analyser simplement son code source dans son ensemble, on va pouvoir enregistrer les événements provoqués par l'utilisateur produisant le scénario souhaité. Ceci va permettre de se focaliser uniquement sur le code exécuté pour produire les fonctionnalités du scénario. D'ailleurs en supposant qu'un composant ne soit pas utilisé (c.-à-d. aucun de ses services n'est demandé par l'ensemble des scénarios analysés.) Il n'apparaîtra pas dans les cartes THD étant donné qu'il n'y aura aucune trace le

concernant. Tandis que si il est utilisé ne serait-ce que dans un seul des scénarios évalués il y aura au moins une trace le concernant et il sera présent dans toutes les cartes THD. (Il sera de couleur blanche dans les scénarios où il n'est pas utilisé.) Ceci afin de pouvoir se rendre compte dans quel scénario cette sous-structure est nécessaire pour produire la fonctionnalité demandée mais également pour pouvoir comparer les différentes valeurs obtenues par la sous-structure en question dans chaque scénario. (D'ailleurs, chaque sous-structure garde le même emplacement pour chaque carte THD afin de pouvoir faciliter l'analyse visuelle.)

**Figure 34** : Marche à suivre - Evaluer et visualiser le THD sur un programme java



Pour plus de détails concernant l'installation et l'utilisation des outils permettant la génération de cartes THD à partir d'un programme java, un guide d'installation et d'utilisation est disponible (cf. annexes 4 et 5).

## 4.1 Evaluation des classes intermédiaires dans une hiérarchie

Lorsqu'une classe successeur fait appel à une méthode héritée par une classe parente dont elle n'hérite pas directement, ceci provoque un appel en chaîne. Il y a donc des classes intermédiaires dans la hiérarchie d'héritage qui sont indispensables pour maintenir les liens d'héritage. Dans la figure 35 ci-dessous, la classe D peut hériter des méthodes m1() et redéfinir la méthode m2() grâce à la hiérarchie implémentée. En effet, la classe D n'hérite pas directement de la classe A, mais grâce aux classes intermédiaires de la hiérarchie (les classes B et C) elle hérite indirectement des fonctionnalités de la classe racine qui en devient une classe parente également (une classe parente éloignée).

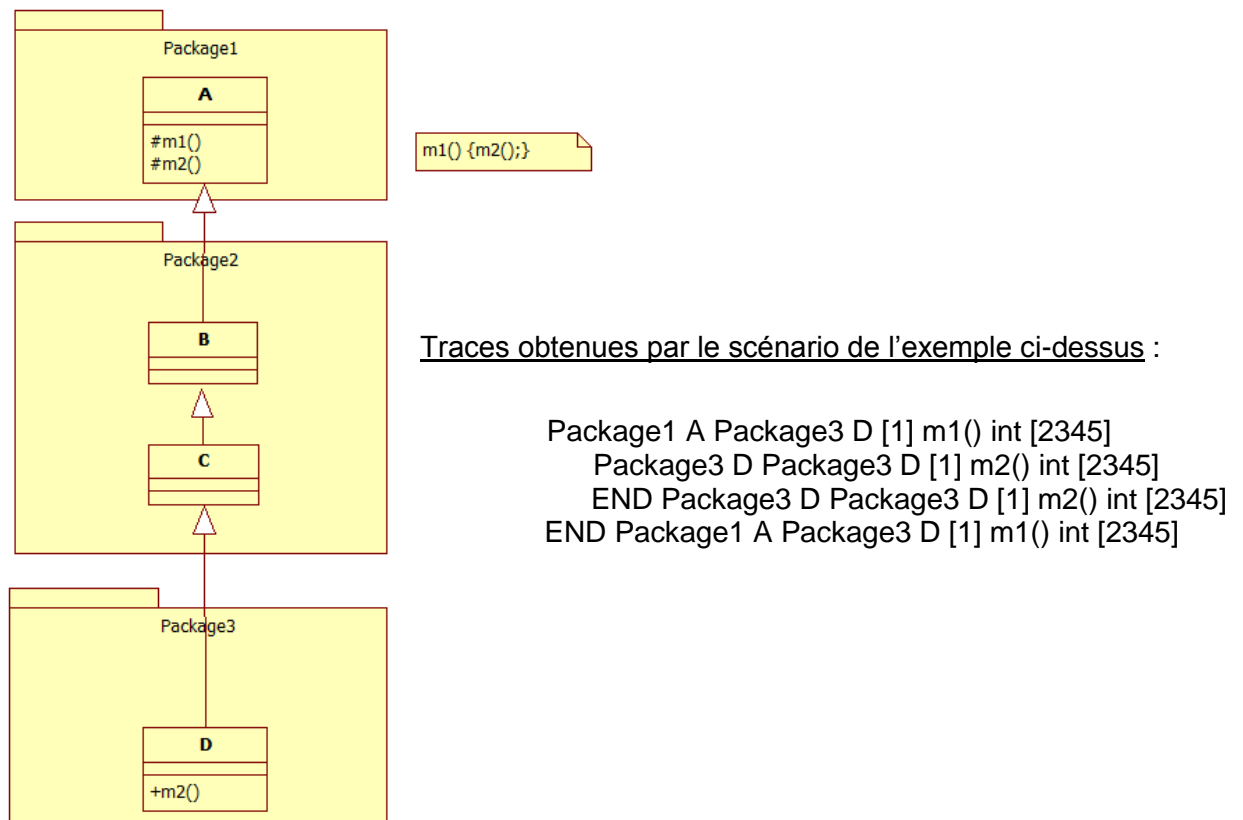
Etant donné que toutes les classes intermédiaires donnent forcément la valeur 0/N je ne m'en préoccupe pas. D'ailleurs il n'est pas possible de pouvoir calculer la valeur exacte du THD pour les classes intermédiaires avec les outils utilisés dans le cadre de ce travail. En effet, les traces générées lors de la réalisation d'un scénario sur le programme analysé ne permettent pas de distinguer les classes intermédiaires. Voici un exemple sur la figure 35 :

*Scénario : Appel de la méthode m1() sur une instance de la Classe D.*

**Tableau 8** : Evaluations du THD pour la figure 35

Niveau d'agrégation	Sous-structure	THD Théorique	THD calculé	THD %
<b>Micro</b> (classes)	<b>A</b>	1/1	1/1	100%
	<b>B</b>	0/1	N/A	N/A
	<b>C</b>	0/1	N/A	N/A
	<b>D</b>	1/2	1/2	50%
<b>Macro</b> (packages)	<b>Package 1</b>	1/1	1/1	100%
	<b>Package 2</b>	0/1	N/A	N/A
	<b>Package 3</b>	1/2	1/2	50%

Figure 35 : Evaluation des sous-structures intermédiaires dans une hiérarchie



Les seules sous-structures visualisées par les traces sont le package 3 et sa classe D ainsi que le package 1 et sa classe A. Il n'y a donc aucun moyen de pouvoir calculer pour les classes intermédiaires la valeur effective du THD.

Cependant c'est un mal pour un bien. Actuellement, lorsqu'une sous-structure est évaluée 0/N, il est certain que :

- la classe concernée est instanciée
- la sous-structure n'apporte rien en termes de fonctionnalité

Etant donné que sont MIE a une valeur de 0. En d'autres termes elle utilise totalement l'héritage pour produire la fonctionnalité demandée. Si nous avons pu évaluer les sous-structures intermédiaires, elles auraient obtenues la valeur similaire de 0/N. Dans ce cas, nous ne pourrions avoir ces certitudes. Car cela pourrait être des classes intermédiaires dans la hiérarchie qui ne sont donc pas instanciées. Toutefois, il est toujours certain qu'elles n'apporteraient rien en termes de fonctionnalité propre.

## 5. Analyse pratique sur ArgoUML

Nous allons maintenant analyser ArgoUML. Ce logiciel open source permet de concevoir des diagrammes UML. Programmé en Java, cela nous permet de pouvoir l'ouvrir par le biais d'Eclipse afin de pouvoir l'instrumenter et l'évaluer avec notre métrique THD. Suite à l'instrumentation de son code, six traces ont été générées, reprenant différentes fonctionnalités.

Dans un premier temps, nous allons expliciter les six scénarios exécutés pour réaliser ces traces. Suite à quoi, nous allons examiner les données dans leur ensemble. Nous étudierons les composants d'ArgoUML ainsi que le nombre de sous-structures totales, le nombre de sous-structures utilisées pour fournir les fonctionnalités demandées. Par la suite, nous allons nous intéresser aux différences évaluées par le THD pour les mêmes sous-structures dans des scénarios différents.

Dans un deuxième temps, nous allons nous intéresser à l'analyse d'un niveau d'agrégation macro en ne focalisant que les packages afin d'avoir un aperçu sur l'héritage effectif utilisé.

Pour finir, nous passerons à l'analyse micro en nous intéressant aux classes. Nous examinerons certains cas évalués par le THD avec des valeurs extrêmes sur l'ensemble de nos scénarios et nous terminerons par conclure sur l'usage effectif de l'héritage d'ArgoUML.

### 5.1 Scénarios

Les scénarios analysés permettent de couvrir un maximum de fonctionnalités d'ArgoUML. Cependant, certaines fonctionnalités comme la génération de code à partir d'un diagramme ou encore la génération de diagramme à partir du code java n'ont pas été analysées pour des raisons pratiques. En effet, la majorité du code source permettant de réaliser ces fonctionnalités provient du système et n'est donc pas instrumentée.

Les scénarios suivants ont été sélectionnés par rapport aux fonctionnalités qu'ils englobent. Pour les quatre premiers, la couleur orange démarquera leur type de

scénario visant à concevoir un diagramme UML. Pour les deux derniers scénarios la couleur bleu sera attribué à leur type (output), à savoir produire une sortie (soit XML soit une impression). Ces fonctionnalités sont supposées être utilisées fréquemment :

**Tableau 9** : Scénarios sélectionnés

<b>Scénario 1</b>	<b>Création de diagrammes de classes</b>
Diagramme UML	Crée un diagramme de classes et le sauvegarde
<b>Scénario 2</b>	<b>Création de diagrammes de séquences</b>
Diagramme UML	Crée un diagramme de séquence et le sauvegarde
<b>Scénario 3</b>	<b>Création de diagrammes d'état</b>
Diagramme UML	Crée un diagramme d'état et le sauvegarde
<b>Scénario 4</b>	<b>Création de diagrammes de use-case</b>
Diagramme UML	Crée un diagramme de classe et le sauvegarde
<b>Scénario 5</b>	<b>Exportation de diagrammes</b>
Output	Ouvre et exporte un diagramme de classes en fichier XML
<b>Scénario 6</b>	<b>Impression de diagrammes</b>
Output	Ouvre et imprime un diagramme de classes



## 5.2 Statistiques générales

Pour réaliser un scénario, un certain nombre de sous-structures vont travailler au sein du programme pour fournir les fonctionnalités demandées. Le tableau 10 ci-dessous nous informe du détail et du nombre des sous-structures utilisées pour chaque scénario exécuté. Le tableau 11 nous détaille le nombre total de sous-structures présentes dans le projet java. (Le tableau 10 a été réalisé grâce à des requêtes sur les









tables résultats de la base de données. Quant au tableau 11, il est le fruit d'un plug-in sur Eclipse permettant de calculer pour un projet java son nombre de classes et packages.)

**Tableau 10** : Informations générales sur les scénarios

 	Nombre de packages utilisés	Nombre de classes utilisés	Nombre de sous-structures utilisés
<b>Scénario 1</b>	76	696	<b>772</b>
<b>Scénario 2</b>	79	600	<b>679</b>
<b>Scénario 3</b>	79	606	<b>685</b>
<b>Scénario 4</b>	76	627	<b>703</b>
<b>Scénario 5</b>	74	566	<b>640</b>
<b>Scénario 6</b>	74	564	<b>638</b>
<b>Moyenne ~</b>	<b>77</b>	<b>610</b>	<b>687</b>

**Tableau 11** : Informations générales sur les scénarios

<b>Projets</b>	 argouml-app	 argouml-core-diagrams-sequence2	 argouml-core-model	 argouml-core-model-mdr	 argouml-core-transformer	 argouml-core-umlpropertypanels
<b>Nb Packages</b>	121	6	2	12	1	3
<b>Nb Classes</b>	1493	32	32	60	11	278
<b>Nb sous-structures</b>	1614	38	34	72	13	301

ArgoUML possède au total :

**145** packages

+ 1906 classes

**2051** sous-structures

ArgoUML utilise en moyenne par scénario :

**77** packages

+ 610 classes

**687** sous-structures

On constate donc qu'en règle général ArgoUML utilise environ 33.6% de ses sous-structures pour réaliser les fonctionnalités demandées par un scénario. Soit, environ un tiers de son architecture. Il est important de spécifier qu'on utilise environ 53% des packages d'ArgoUML mais nous ne pouvons pas connaître leur degré d'utilisation. En effet, nous ne pouvons pas confirmer à quel point ces packages sont impliqués. Pour un package utilisé, il n'y a peut-être que quelques classes impliquées ou, à l'inverse, la totalité des classes du package. On ne peut pas non plus préciser la fréquence d'utilisation du package, étant donné qu'on ne traite pas les doublons dans les traces engendrées par un scénario (en effet, pour le calcul du THD il ne serait pas approprié de compter les doublons d'appel de méthodes. Ceci fausserait les résultats en y incluant le niveau de fréquence des appels de méthodes qui ne rentre pas en compte dans le cadre de ce travail). On est cependant certain qu'à un certain moment, lors de l'exécution des fonctionnalités pour un scénario, ArgoUML fait appel à une méthode d'une classe se trouvant dans un des packages de couleur rouge ou verte (rappel : la couleur blanche est appliquée aux sous-structures non utilisées dans le scénario visualisé). On peut donc maintenant rectifier en déclarant qu'en moyenne, la moitié des packages d'ArgoUML et 32% de ses classes sont impliquées dans la réalisation de ces scénarios.

### 5.3 Analyse des cartes THD (macro)

Cette métrique a l'avantage de s'appliquer aussi bien aux classes qu'aux packages. Nous allons commencer cette analyse d'un point de vue macro sur l'architecture et nous concentrer sur les packages utilisant l'héritage. Par la suite nous procéderons à l'analyse micro en nous concentrant sur les classes utilisant l'héritage.

**Tableau 12** : THD des packages utilisant l'héritage dans les différents scénarios

Valeurs du THD en %	Scénarios					
Packages avec un MHE non nulle	1	2	3	4	5	6
org.argouml.cognitive	95	96	96	96	95	95
org.argouml.cognitive.ui	66	66	67	66	65	66
org.argouml.cognitive.checklist.ui	76	73	78	78	73	73
org.argouml.kernel	98	98	98	98	98	98
org.argouml.model.mdr	96	97	97	96	98	98
org.argouml.notation.providers.uml	62	71	59	46	52	52
org.argouml.pattern.cognitive.critics	16	15	15	15	16	16
org.argouml.profile.internal	88	87	88	91	87	87
org.argouml.profile.internal.ocl	44	43	45	49	43	44
org.argouml.profile.internal.ocl.uml	50	33	60	75	33	50
org.argouml.sequence2	100	85	100	100	100	100
org.argouml.sequence2.diagram	N/A	37	N/A	N/A	N/A	N/A
org.argouml.ui	96	97	97	96	99	99
org.argouml.ui.cmd	73	72	72	72	72	74
org.argouml.ui.explorer	98	98	98	98	98	98
org.argouml.uml.diagram	99	96	99	100	99	99
org.argouml.uml.diagram.state	100	100	92	100	100	100
org.argouml.uml.diagram.state.ui	100	100	43	100	100	100
org.argouml.uml.diagram.static_structure	86	89	89	89	90	90
org.argouml.uml.diagram.static_structure.ui	32	32	39	39	32	32
org.argouml.uml.diagram.ui	72	63	62	69	72	72
org.argouml.uml.diagram.use_case	78	78	78	83	78	78
org.argouml.uml.diagram.use_case.ui	33	35	35	34	29	30
org.argouml.uml.ui	84	84	86	83	84	84
org.argouml.uml.ui.foundation.core	80	100	100	80	100	100
org.argouml.uml.cognitive	92	90	92	90	92	92
org.argouml.uml.cognitive.checklist	33	33	33	33	33	33
org.argouml.uml.cognitive.critics	72	66	72	65	72	72

Le tableau 12 est un récapitulatif des cartes THD macros des scénarios. Vous pouvez consulter toutes les cartes en annexe 2.

On remarque que sur une moyenne de 77 packages utilisés pour fournir les fonctionnalités d'un des six scénarios exécutés, seulement 28 d'entre eux utilisent l'héritage. On observe donc un rapport moyen d'environ 37% d'héritage nécessaire aux packages utilisés. La métrique THD permet de connaître l'héritage effectif utilisé. Par contre elle ne nous indique pas l'héritage potentiel utilisable. Il serait donc intéressant de pouvoir mesurer sur les 77 packages utilisés, le taux d'héritage potentiel (le nombre de packages faisant partie d'une hiérarchie d'héritage). Ainsi nous pourrions évaluer le niveau d'utilisation de ces hiérarchies. Lorsque le MHE du THD est supérieur à zéro, cela implique que l'héritage est utilisé (Autrement dit lorsque la valeur du THD est différente de 100%). Grâce à cette particularité nous pouvons déceler sur l'ensemble des scénarios quels sont les packages utilisant des hiérarchies d'héritages. Par contre, nous ne pouvons pas être sûrs que ce sont les seuls packages implantés dans des hiérarchies d'héritage. Il se pourrait qu'il y en ait d'autres qui ne fassent pas appel à l'héritage dans le cadre de ces six scénarios. Toutefois, l'intérêt d'une métrique dynamique est de se concentrer sur le code exécuté pour fournir les fonctionnalités analysées. Lors d'une maintenance sur une fonctionnalité, ce qui nous intéresse c'est de voir l'héritage réellement utilisé pour la produire, ce que réalise parfaitement le THD.

Le niveau macro ne montre pas de grandes différences entre les scénarios distincts, même au niveau du type de scénario. (Les scénarios 5 et 6 étant des scénarios générant des sorties (exportation XML et impression) ne se démarquent pas par rapport aux scénarios 1 – 4 (créations de diagrammes).) Cela signifie que les mêmes hiérarchies d'héritages sont utilisées pour fournir des fonctionnalités différentes. Ceci peut montrer à première vue une forme de stabilité sur l'architecture d'ArgoUML. Si, pour produire des diagrammes UML de différents types, nous avons remarqué que des composants très différents faisaient appel à l'héritage, nous n'aurions pas pu conclure de la sorte. Cependant, étant donné le niveau d'agrégation actuel, rien ne prouve que ce soient les mêmes hiérarchies d'héritages qui sont appelées à chaque fois. En effet, étant donné que le lien d'héritage se fait au niveau des classes, il se pourrait très bien qu'entre deux scénarios, qui à première vue ont attribué un THD similaire à un package donné, ce ne soient pas les mêmes classes qui fassent appel à l'héritage. Il se pourrait également que ces classes fassent parties de hiérarchies

d'héritages différentes. Autrement dit, au niveau macro, nous ne pouvons pas être certains que ce sont les mêmes hiérarchies d'héritages qui sont utilisées.

Il y a tout de même certaines différences. Premièrement au niveau du package `org.argouml.sequence2` qui n'utilise pas d'héritage à part dans le scénario deux (création et sauvegarde de diagramme de séquence) et de son sous-package `org.argouml.sequence2.diagram` qui lui est utilisé exclusivement dans le scénario 2. De plus, ce dernier a également besoin de l'héritage pour fonctionner. (On visualise très bien ces différences grâce aux cartes THD (cf : annexe 2).

Deuxièmement, les packages :

`org.argouml.uml.diagram.state` et `org.argouml.uml.diagram.state.ui` qui ont besoin de l'héritage pour le scénario 3 (création et sauvegarde d'un diagramme d'état).

Troisièmement, on distingue pour le package `org.argouml.uml.ui.foundation.core` que l'utilisation de l'héritage n'est nécessaire que pour le scénario 1 et 4.

Les deux premiers cas peuvent s'expliquer de la même façon : lorsque l'héritage est utilisé c'est pour répondre à une fonctionnalité demandée non présente dans les autres scénarios. En effet, dans le premier cas, le package « `org.argouml.sequence2` » comme son nom l'indique à un rapport avec les diagrammes de séquences. Et c'est justement lorsqu'on en crée un que lui et son sous-package ont besoin d'utiliser l'héritage. Le diagramme de séquence nécessite une fonctionnalité détenue dans la hiérarchie d'héritage de ce package. C'est également le cas sur le package `org.argouml.uml.diagram.state` et son sous-package pour une fonctionnalité nécessaire à la génération d'un diagramme d'état.

La métrique THD nous a permis de pouvoir visualiser différents niveaux d'agrégations (actuellement : packages et sous-packages). Ainsi il a été facile de pouvoir discerner les dépendances d'héritage entre les packages. Grâce à la valeur de la métrique et à l'aide des cartes, nous avons pu repérer les packages utilisant effectivement l'héritage pour réaliser les différents scénarios analysés. En plus d'évaluer les packages, cette métrique permet de rajouter le niveau micro afin d'affiner l'analyse. Et c'est ce que nous allons faire dans le chapitre qui suit.

## 5.4 Analyse des cartes THD (micro)

La vue macro nous a permis de remarquer les packages nécessitant l'héritage au runtime des scénarios. Toutefois, sans avoir analysé le niveau micro, il est difficile d'établir d'autres types d'observations. Cependant la vue macro a permis de voir l'héritage impliqué entre les grandes sous-structures de l'architecture pour chaque scénario, ce qui permet de se faire une idée sur la localisation des hiérarchies de classes (en effet, un package nécessitant l'héritage contient forcément les classes faisant ce lien d'héritage en question). Regardons donc au niveau des classes l'héritage réellement utilisé pour produire les fonctionnalités des scénarios (cf. : annexe 2).

A première vue, certains packages n'ayant pas besoin d'héritage (évalués 100% THD, et n'ayant pas été distingués dans l'analyse macro) possèdent des classes qui utilisent l'héritage pour délivrer les fonctionnalités. Ceci permet d'affirmer que la hiérarchie d'héritage des classes d'un de ces packages est interne. Autrement dit, les liens d'héritage ne se font qu'entre les classes internes du package. Il n'y a donc pas de classe héritant d'une classe localisée dans un autre package. C'est typiquement le cas des packages suivant :

- org.argouml.core.
- org.argouml.core.propertypanels
- org.argouml.core.propertypanels.ui
- org.argouml.persistence
- org.argouml.application.events
- org.argouml.profile
- org.argouml.configuration
- org.argouml.notation.providers

A l'aide des interprétations visuelles sur les cartes THD (cf. : annexe 2), on décèle dans la plupart des scénarios un grand nombre de classes faisant appel à l'héritage localisées dans les packages. Les packages suivants sont ceux qu'on remarque le plus visuellement sur les cartes THD en termes de classes faisant usage de l'héritage.

- org.argouml.uml.diagram.ui
- org.argouml.uml.diagram.static\_structure
- org.argouml.uml.diagram.state
- org.argouml.uml.diagram.use\_case

Ces packages avaient déjà été remarqués dans l'analyse macro par la valeur de leur taux d'héritage dynamique. Il est normal de remarquer une forte utilisation de l'héritage étant donné qu'on se trouve dans les packages spécifiques à la génération des différents diagrammes créés dans la plupart de nos scénarios. D'ailleurs le nom du package le fait sous-entendre par analogie. On peut donc supposer que l'héritage des classes de ces packages varie en fonction des scénarios, comme relaté dans l'analyse macro entre le package `*.diagram.state` et le scénario 3, créant un diagramme d'état, ou encore `*.diagram.use_case` et le scénario 4, lié au diagramme de use-case. De plus, le package `*.diagram.ui` sollicite fortement l'héritage de ses classes dans tous les cas. On peut donc présumer que ce sont les classes de base pour tout type de diagrammes UML.

- `*.diagram.ui` : sert à tous les diagrammes
- `*.diagram.static_structure` : sert au diagramme de classes
- `*.diagram.state` : sert au diagramme d'état
- `*.diagram.use_case` : sert au diagramme use-case

Au niveau des classes, on remarque grâce au dégradé de couleurs l'équilibre entre les fonctionnalités propres apportées et celles héritées. Les sous-structures en vert obtiennent une évaluation équilibrée entre ces deux types de fonctionnalités. C'est donc les classes dans le rouge qui nous intéressent, et en particulier les cas extrêmes afin de s'assurer que ce ne sont pas des fautes d'implémentations.

Voici pour commencer quelques classes évaluées comme totalement dépendantes de l'héritage dans la plupart des scénarios analysés :

Tableau 13 : Classes dans le rouge intéressantes

Valeurs du THD en %	Scénarios					
Classes	1	2	3	4	5	6
org.argouml.uml.diagram.static_structure. <b>ClassDiagramGraphModel</b>	N/A	0/3	0/3	0/3	0/6	0/6
org.argouml.uml.diagram.static_structure.ui. <b>FigInterface</b>	4/74	N/A	N/A	N/A	0/53	0/52
org.argouml.uml.diagram.static_structure.ui. <b>FigPackage&amp;FigPackageFigText</b>	0/6	N/A	N/A	N/A	0/6	0/6
org.argouml.uml.diagram.static_structure.ui. <b>FigPackage&amp;PackageBackground</b>	0/6	N/A	N/A	N/A	0/2	0/2
org.argouml.uml.diagram.static_structure.ui. <b>PropPanelUMLClassDiagram</b>	0/13	0/13	0/13	0/13	0/11	0/11
org.argouml.model. <b>AddAssociationEvent</b>	0/1	0/1	0/1	0/1	N/A	N/A
org.argouml.model. <b>RemoveAssociationEvent</b>	0/1	0/1	0/1	N/A	N/A	N/A
org.argouml.uml.diagram.ui <b>FigCompartment\$FigSeparator</b>	0/2	N/A	N/A	0/1	0/2	0/2
org.argouml.uml.diagram.ui <b>FigNodeModelElement\$SelectionDefaultClarifiers</b>	0/5	N/A	N/A	N/A	N/A	N/A
org.argouml.uml.diagram.ui <b>FigCompartment\$FigPort</b>	0/4	N/A	N/A	0/3	0/3	0/3
org.argouml.uml.diagram.ui <b>SPFigEdgeModelElement</b>	0/17	0/17	0/17	0/16	N/A	N/A
org.argouml.core.propertypanels.ui <b>UMLSearchableComboBox</b>	0/2	0/2	N/A	0/2	N/A	N/A

Ces classes ont la particularité de, soit utiliser totalement l'héritage, soit ne pas être appelées pour réaliser répondre aux demande d'un scénario. On remarque selon le type de scénario des différences flagrantes. Plus particulièrement, soit on utilise cette classe en l'instanciant et en faisant appel à l'héritage (avec une valeur de 0/N), soit on ne l'utilise pas du tout.

Cependant, ces différences ne sont dues qu'aux manipulations engendrées lors de l'exécution du scénario. Par exemple, une différence de valeur pour le scénario 4 (la création et sauvegarde d'un diagramme use-case) correspond dans ce scénario au fait que la classe `RemoveAssociationEvent` n'est pas utilisée. Ceci s'explique car à l'exécution du scénario, je n'ai pas enlevé d'associations lors de ma création de diagramme sur le use-case, contrairement aux autres diagrammes. On remarque



d'ailleurs ici une différence entre les types de scénarios. Pour les sorties de fichier XML ou imprimés, je n'ai fait qu'ouvrir les diagrammes pour produire les sorties des scénarios. Je n'ai donc pas ajouté ni enlevé d'associations. Cette classe d'événement n'a donc pas été instanciée dans ce cas.

On observe une logique liée au type de package utilisé : lorsqu'une classe du package `org.argouml.uml.diagram.static_structure.ui` est utilisée, cela ne concerne que les scénarios 1, 5 et 6 dans la majorité des cas. En effet, ce package concerne la création des objets du diagramme de classe, ce qui justifie l'utilisation du package dans le premier scénario. Et pour les scénarios de types outputs, cela s'explique par le fait que j'ai imprimé et exporté en XML un diagramme de classe à chaque fois. Voilà pourquoi le package est également utilisé. Autrement dit, si j'avais choisi un diagramme UML différent pour l'impression et l'exportation, le package `static_structure` n'aurait pas été utilisé dans les scénarios 5 et 6. D'ailleurs, même constat si j'avais par exemple imprimé un diagramme d'état et exporté un diagramme de classes, il n'y aurait pas autant de similitudes entre ces deux derniers scénarios.

La classe `FigInterface` attire particulièrement mon attention quant à sa valeur obtenue dans le premier scénario. C'est la seule classe du tableau à obtenir une valeur différente de 100% THD lorsqu'elle est utilisée. Analysons de plus près sa hiérarchie (cf. : annexe 3.1).

Figure 36 : ArgoUML – Hiérarchie 1 abrégée

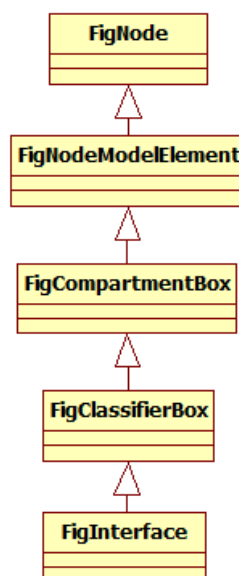


Tableau 14 : ArgoUML - Evaluation du THD sur la hiérarchie 1

Valeurs du THD en %	Scénarios					
Classes	1	2	3	4	5	6
org.tigris.gef.presentation. <b>FigNode</b>	NON INSTRUMENTE					
org.argouml.uml.diagram.ui <b>FigNodeModelElement</b>	61/68	43/43	52/57	56/60	39/44	38/43
org.argouml.uml.diagram.ui <b>FigCompartmentBox</b>	22/32	N/A	N/A	11/21	19/27	19/27
org.argouml.uml.diagram.static_structure.ui. <b>FigClassifierBox</b>	4/16	N/A	N/A	N/A	4/10	4/10
org.argouml.uml.diagram.static_structure.ui. <b>FigInterface</b>	4/74	N/A	N/A	N/A	0/53	0/52

FigInterface utilise quatre de ses méthodes implantées lors de la création du diagramme de classes (dans le premier scénario). C'est l'unique différence constatée entre le premier scénario et les scénarios 5 et 6. En effet, dans ces derniers scénarios je n'ai pas créé le diagramme de classe, je l'ai uniquement ouvert et imprimer ou exporter.

Les quatre méthodes implantées et utilisées par la classe FigInterface sont :

- makeSelection() : Selection
- setEnclosingFig(Fig) : Void
- classNameAndBounds() : String
- updateListeners(Object, Object) : Void

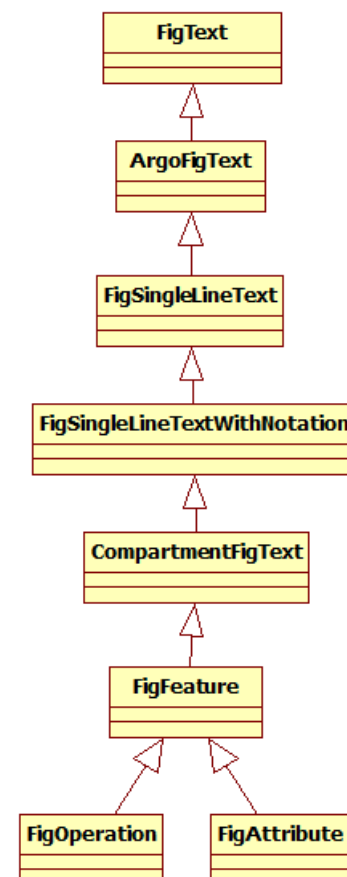
La classe FigInterface utilise donc sa propre fonctionnalité lors de la création d'un diagramme de classe sur ArgoUML. Cette classe prend donc un intérêt pour réaliser une fonctionnalité du premier scénario. On remarque grâce à cet exemple qu'une classe évaluée (même souvent) comme étant totalement dépendante en termes de hiérarchie (0/N THD) n'est pas forcément une classe inutile. Il se peut comme montré ci-dessus, que pour une fonctionnalité particulière ses méthodes implantées soient utilisées.

On suppose donc que toutes les classes du tableau 13 sont dans ce cas, en considérant le fait que les six scénarios analysés soient loin d'être complets pour couvrir l'entièreté des fonctionnalités d'ArgoUML. De plus, ces scénarios ont été réalisés en n'utilisant que des fonctionnalités réduites. Par exemple, je n'ai pas utilisé toutes les sortes de figures possibles pour concevoir mes diagrammes.

Lorsqu'on rencontre des classes évaluées similairement aux classes du tableau 13, il faut tout d'abord comparer les évaluations de tous les scénarios ayant une valeur métier. Si, dans tous les scénarios, une classe n'apporte jamais de fonctionnalité propre (autrement dit, elle obtient toujours un MIE nulle) cela signifie qu'elle pourrait être remplacée par une classe parente. Il faudrait, à ce moment, la remettre en question.

Concernant les classes ayant obtenu une valeur de 100% THD (coloriées en rouge également), cela signifie qu'elles sont appelées car on a besoin de leur fonctionnalité interne. Mais elles ne font en aucun cas appel à l'héritage pour la produire. Cela concerne également des classes qui ne sont pas dans une hiérarchie d'héritage ou qui se situent à la racine d'une hiérarchie, mais il arrive également que certaines classes implantées dans une hiérarchie sans en être la racine soient concernées.

Prenons maintenant un exemple concret sur une hiérarchie de figures. Le tableau ci-dessous va nous indiquer pour chaque classe de la hiérarchie abrégée les valeurs obtenues du THD pour chaque scénario (la hiérarchie ci-dessous est abrégée pour nous concentrer sur les classes spécifiques FigOperation et FigAttribute dans l'hypothèse d'une éventuelle maintenance sur celles-ci : (Vous trouverez en annexe 3 la hiérarchie complète).



**Figure 37** : ArgoUML – Hiérarchie 2 abrégée

Tableau 15 : ArgoUML - Evaluation du THD sur la hiérarchie 2

Valeurs du THD en %	Scénarios					
Classes	1	2	3	4	5	6
org.tigris.gef.presentation. <b>FigText</b>	NON INSTRUMENTE					
org.argouml.uml.diagram.ui <b>ArgoFigText</b>	4/4	4/4	4/4	4/4	4/4	4/4
org.argouml.uml.diagram.ui <b>FigSingleLineText</b>	5/7	5/6	5/6	3/4	4/6	4/6
org.argouml.uml.diagram.ui <b>FigSingleLineTextWithNotation</b>	8/10	N/A	N/A	5/6	5/7	5/7
org.argouml.uml.diagram.ui <b>CompartmentFigText</b>	4/4	N/A	N/A	N/A	1/1	1/1
org.argouml.uml.diagram.static_structure.ui. <b>FigFeature</b>	1/1	N/A	N/A	N/A	2/3	2/3
org.argouml.uml.diagram.static_structure.ui. <b>FigOperation</b>	N/A	N/A	N/A	N/A	2/16	2/16
org.argouml.uml.diagram.static_structure.ui. <b>FigAttribute</b>	1/20	N/A	N/A	N/A	1/16	1/16

(Grâce à l'utilisation d'un plug-in sur Eclipse, il m'est possible de générer facilement la hiérarchie de classes, sans quoi l'analyse serait moins commode (cf. : annexe3.2)).

Avant toute chose, il est important de rappeler que le tableau 14 ne comporte pas l'intégralité des classes de cette hiérarchie. En effet, pour simplifier l'analyse de la hiérarchie, nous prenons uniquement les branches menant aux feuilles FigOperation et FigAttribut. Nous prenons l'hypothèse suivante : les modifications à effectuer sur cette hiérarchie ne concernent que ces deux classes.

Tout d'abord, on note que la classe FigFeature a besoin de l'héritage pour le deuxième type de scénario (les outputs XML et impression). Cependant, dans le premier scénario, elle réalise une fonctionnalité sans dépendre de sa hiérarchie. De plus, pour la création et sauvegarde des autres types de diagrammes UML, elle n'est même pas utilisée.

On remarque également que la classe `CompartmentFigText` se retrouve toujours avec un THD de 100% lorsqu'elle est utilisée, ceci même en étant une classe intermédiaire. Il y a deux possibilités : La première c'est qu'une fonctionnalité lui soit demandée en dehors de la hiérarchie. Dans ce cas, il suffirait d'instancier cette classe et de lui demander d'exécuter une de ces méthodes pour obtenir le résultat de 1/1. La deuxième possibilité c'est que l'une ou plusieurs de ses classes successeurs fassent appel à l'héritage pour une méthode implantée chez elle.

Après analyse de la trace des scénarios similaires 5 et 6, on retrouve un bon nombre de fois les événements suivants :

```
org.argouml.uml.diagram.ui CompartmentFigText org.argouml.uml.diagram.static_structure.ui FigAttribute [15] isFilled() AS boolean [1376794106708]  
END org.argouml.uml.diagram.ui CompartmentFigText org.argouml.uml.diagram.static_structure.ui FigAttribute [15] isFilled() AS boolean [1376794106708]
```

```
org.argouml.uml.diagram.ui CompartmentFigText org.argouml.uml.diagram.static_structure.ui FigOperation [15] isFilled() AS boolean [1376794104335]  
END org.argouml.uml.diagram.ui CompartmentFigText org.argouml.uml.diagram.static_structure.ui FigOperation [15] isFilled() AS boolean [1376794104335]
```

C'est donc la deuxième hypothèse qui s'avère être juste. En effet, on peut être sûr que sur les 14 méthodes héritées exécutées d'une instance de la classe `FigOperation` et des 15 méthodes héritées exécutées de la classe `FigAttribute`, une d'elle est la méthode `isFilled()` implantée dans la classe `CompartmentFigText`, car les traces nous révèlent que le type statique de cette méthode est bien `CompartmentFigText`. De plus, dans les traces des scénarios 5 et 6, c'est le seul endroit où `CompartmentFigText` est tracé. Cependant, dû au grand nombre de méthodes héritées nécessaires aux instances des classes `FigAttribute` et `FigOperation`, on remarque que l'ensemble de la hiérarchie est nécessaire pour répondre à la fonctionnalité. C'est en soit une hiérarchie bien utilisée dans les scénarios 5 et 6.

On peut tout de même reprocher certains points. Lorsque les classes `FigAttribute` et `FigOperation` sont utilisées, elles sont très dépendantes de la hiérarchie (THD : 1/20, 1/16 ou encore 2/16). La hiérarchie implémente bien les trois axes (polymorphisme, DRY, compréhension), cependant son grand niveau de profondeur ainsi que son grand nombre de classes (cf. : annexe 3.2) peut obscurcir la compréhension. Lors de modifications dans les classes de niveau supérieur dans la hiérarchie, il faudra faire attention à l'impact engendré sur les classes successeurs se révélant à priori important. En plus d'être une hiérarchie profonde, les classes feuilles étant très dépendantes en termes d'héritage se verront à coup sûr affectées par les modifications.

C'est typiquement dans ce genre de cas qu'on s'attend à retrouver un même type de réaction de la part de la maintenance. En effet, il serait possible qu'un développeur opte plus facilement pour une redéfinition de méthodes (afin de gagner du temps et ne pas se compliquer la tâche), au lieu de prendre du temps pour examiner avec soin la hiérarchie en question, dans le but de préserver l'héritage utilisé.

On finira par la remarque suivante, toujours sur l'exemple du tableau 14 : la classe `ArgoFigText` n'utilise jamais l'héritage dans nos scénarios, ce qui peut vouloir dire que sa classe mère `FigText` n'est peut-être jamais utilisée. Il est dommage que nous ne puissions pas instrumenter la classe `FigText` pour confirmer cette hypothèse. Dans le cas où nous serions dans le juste, cela voudrait dire (toujours en considérant qu'on ait évalué l'ensemble des scénarios métiers envisageables) que la classe `FigText` n'est jamais utilisée, à cause des redéfinitions de méthodes des classes successeurs. Néanmoins, cette hypothèse n'est pas certaine : il se pourrait que d'autres classes successeurs fassent appel aux services hérités de la classe `FigText` ou tout simplement que ces méthodes soient requises pour des fonctionnalités non examinées dans cette analyse. Cette remarque est également applicable au tableau 13, concernant la classe `FigNode`. Cependant, dans ce cas, sa classe enfant directe `FigNodeModelElement` fait souvent appel à l'héritage, ce qui nous assure que `FigNode` est appelée par héritage pour fournir sa propre fonctionnalité.

La métrique THD peut être utilisée pour éviter ce genre de situation. Avant une modification sur ce style de hiérarchie, il faudrait dans un premier temps analyser chaque sous-structure avec le THD pour les scénarios voulu, puis dans un deuxième temps, effectuer les modifications avec précaution en prenant garde à préserver l'héritage réellement utilisé au runtime. L'idéal serait de terminer le processus par l'analyse encore une fois de la hiérarchie après avoir terminé ses modifications. Ainsi le THD ferait office de contrôle afin de garantir ou même d'améliorer l'équilibre entre l'héritage utilisé (la dépendance fonctionnelle) et la fonctionnalité propre utilisée (l'indépendance fonctionnelle) des sous-structures de la hiérarchie d'héritage.

Avant une maintenance, il ne faut pas seulement s'intéresser aux classes évaluées dans le rouge, ce sont toutes les classes susceptibles d'être modifiées par la maintenance ainsi que les classes dans sa hiérarchie d'héritage qui se doivent d'être examinées. Ceci dans le but de préserver ou même d'équilibrer l'héritage effectif lors de la réalisation des scénarios métiers du logiciel analysé. Le THD devient utile dès

qu'on envisage des modifications sur une hiérarchie d'héritage (ce qui est fréquent lors de modifications sur un système conçu en POO).

Le taux d'héritage dynamique permet de pouvoir évaluer l'héritage véritablement employé au runtime des scénarios. Ceci permet de pouvoir se concentrer sur des fonctionnalités particulières, ce qui est important pour les maintenances qui, la plupart du temps, s'occupent de rajouter, modifier ou supprimer des fonctionnalités métiers. Dans notre analyse sur ArgoUML, en se focalisant sur les fonctionnalités de nos scénarios (créations de diagrammes, exportations XML et impressions) on a pu se rendre compte des sous-structures utilisant réellement l'héritage aux niveaux macro et micro pour concevoir ces fonctionnalités. Cela permet de pouvoir prendre des précautions d'un point de vue la modifiabilité. Autrement dit, avant d'effectuer une modification dans une sous-structure, il est intéressant de savoir à quel point elle utilise réellement l'héritage afin d'éviter de le détériorer.

Rappelons également que l'une des principales utilisations du THD est de pouvoir comparer un même logiciel avant et après une maintenance avec les mêmes scénarios, ceci dans le but de vérifier la qualité de la maintenance. De plus, cette métrique permettrait de déceler par exemple un Framework non utilisé. Par exemple, on peut imaginer des développeurs mal à l'aise face aux patterns imposés par le Framework, dû à la pression des temps imparti qui les poussent à négliger la qualité et redéfinir les méthodes héritées de celui-ci pour pouvoir développer sans contraintes.

## 5.5 Remarque

Dans le cadre de ce travail, il aurait été intéressant de pouvoir effectuer un exemple pratique en analysant ArgoUML avec une version antérieure incluant des différences dans le code sur les fonctionnalités de nos scénarios analysés. Ainsi il aurait été intéressant de voir pour un même scénario des différences au niveau de l'évaluation du THD entre la version antérieure et celle à jour. Néanmoins, dans le chapitre concernant l'interprétation de la métrique, des exemples théoriques permettent de se faire une idée.

## 6. Travail futur

Pour continuer le travail réalisé, il faudrait tout d'abord améliorer la visualisation actuelle du THD. Elle ne permet pas de voir directement les liens d'héritage. Comme mentionné dans le chapitre sur les visualisations, le diagramme de la hiérarchie est fortement conseillé afin d'améliorer la compréhension visuelle des valeurs obtenues par le THD. Cependant, ce n'est qu'une deuxième vue complémentaire. De plus, il est parfois compliqué de retrouver toutes les classes d'une même hiérarchie étant donné qu'elles ne se trouvent pas forcément dans le même package. L'utilisation d'un plug-in Eclipse m'a permis de faciliter cette tâche qui a tout de même été ardue. L'idéal serait d'améliorer nos cartes THD en exploitant des concepts de visualisations proches de la visualisation « Sunburst » [9]. Elle permet notamment de visualiser les hiérarchies par tranches et permet en plus de se faire une idée sur les classes implantées dans les différentes couches comme on peut le constater sur la figure 1 : Exemple de visualisation « Sunburst ».

Par la suite, il serait intéressant de pouvoir modifier le code couleur. En effet, les paramètres du logiciel Microsoft Treemapper permettant de concevoir les cartes THD ne permettent d'utiliser que deux couleurs (et les nuances entre les deux). Nous avons fait avec les moyens du bord et le résultat est plutôt satisfaisant. Toutefois, nous pourrions tenter d'améliorer ce point en y intégrant une nouvelle couleur afin de différencier visuellement les cas de dépendance ou d'indépendance (différencier les valeurs extrêmes 0/N et 1 du THD) qui sont actuellement représentées par la même couleur (rouge foncé).

Finalement, il serait intéressant de pouvoir comparer les analyses faites par le THD et l'AR [21] sur ArgoUML. Cette métrique permet de connaître le ratio d'autonomie d'une sous-structure. Son calcul est basé sur le couplage (la dépendance fonctionnelle externe) et la cohésion (la dépendance fonctionnelle interne). On pourrait espérer y trouver des corrélations intéressantes, par exemple des sous-structures évaluées comme fortement dépendantes par l'AR pourraient se retrouver avec un THD important (étant donné que le THD indique la dépendance fonctionnelle en termes d'héritage). Cette étude aurait pu faire partie de ce travail, malheureusement par manque de temps il faudrait l'envisager dans le cadre d'un travail futur.



## 7. Conclusion

Rappelons l'objectif de ce travail de diplôme, à savoir la conception d'une métrique permettant de pouvoir évaluer l'héritage effectif utilisé pour un scénario donné à l'exécution du logiciel analysé. Nous avons voulu apporter une mesure servant d'indicateur aux maintenances lors de modifications sur une hiérarchie de classes. En plus de donner une valeur sur l'emploi effectif de l'héritage pour toute sous-structure, cette métrique fait également office de contrôle qualité dans le but de préserver l'utilisation des hiérarchies d'héritage.

Nous avons par la suite donné des exemples permettant de montrer comment cette métrique peut être interprétée, notamment avant et après une maintenance sur un logiciel. Ceci nous a permis de comprendre comment l'utiliser et en tirer des conclusions. Puis, nous avons mis en œuvre une analyse pratique sur le logiciel open source ArgoUML.

Lors de cette analyse, nous avons exploré l'architecture dynamique de six scénarios différents permettant de regrouper un certain nombre de fonctionnalités, sans doute les plus utilisées dans ce logiciel. Puis, nous avons, grâce aux outils permettant de retracer les méthodes utilisées lorsque les scénarios ont été réalisés sur ArgoUML, généré des cartes permettant de visualiser les différentes sous-structures utilisées avec leur taux d'héritage dynamique dont les valeurs ont été soumises à un code couleur permettant de faciliter leur interprétation.

Le champ de notre analyse s'est étendue tout d'abord à un niveau d'agrégation macro, en ne nous souciant que des packages et sous-packages d'ArgoUML. Nous avons par la suite, grâce à la souplesse de notre métrique, approfondi l'analyse à un niveau micro en nous concentrant sur les classes intégrées dans les hiérarchies d'héritage utilisées pour produire les fonctionnalités des scénarios examinés. Pour finir, nous avons exposé certains exemples de classes pouvant paraître à première vue mal implémentées, pour déduire que nous ne pouvions rien affirmer tant que nous n'avons pas analysé l'ensemble des scénarios métiers.

Pour conclure, l'application du THD lors de modifications sur une hiérarchie de classes permettra, je l'espère, d'empêcher les négligences sur la qualité de développement commises par les employées de la maintenance et causées par le stress des délais de livraison, qui doit on l'admettre, se font de plus en plus court.

## 8. Acronymes

API	Application Programming Interface
AR	Autonomy Ratio
CRUD	Create, Read, Update, Delete
CSV	Coma Separated Values
DRY	Don't Repeat Yourself
MHE	Méthodes Héritées Executées
MIE	Méthodes Implantées Executées
MOOD	Metrics Object Oriented Design
ODBC	Open Database Connectivity
POO	Programmation Orientée Objet
SQL	Sequence Query Language
THD	Taux d'Héritage Dynamique
UML	Unified Modeling Language

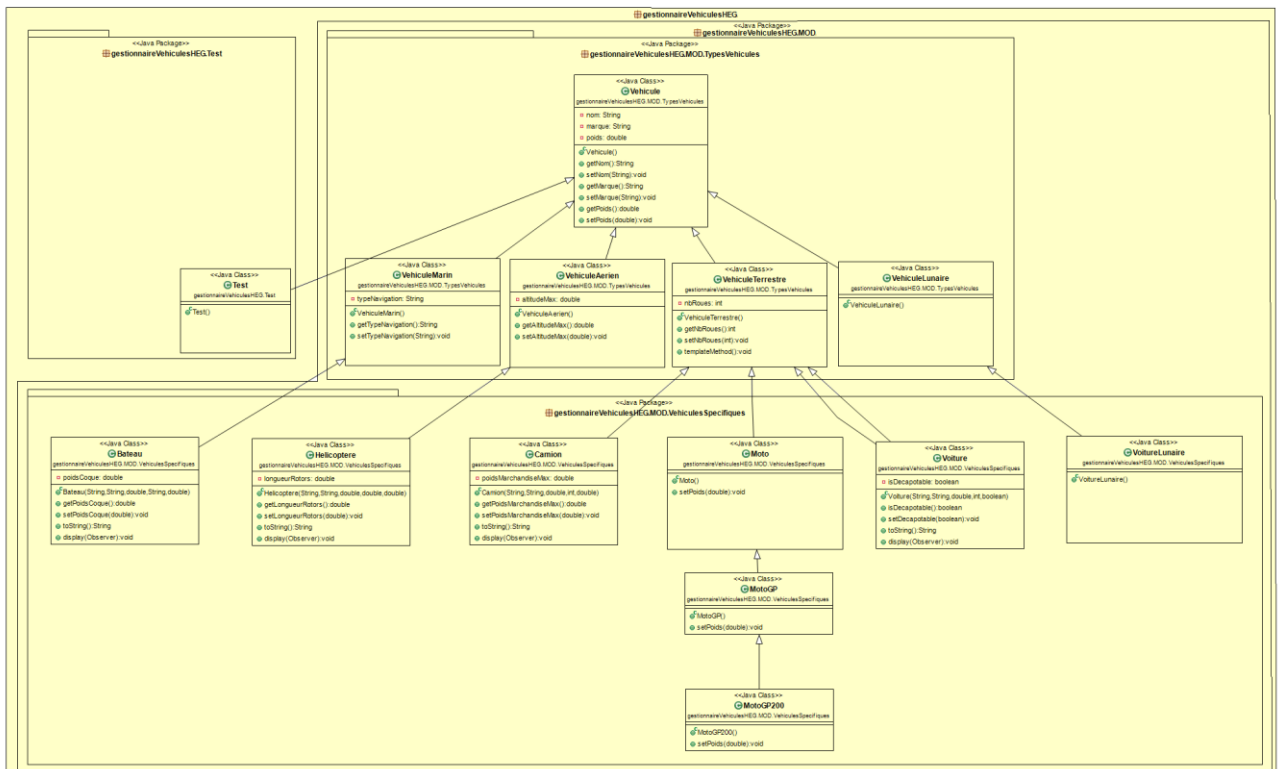
## Bibliographie

1. F. Brito e Abreu, "The MOOD Metrics Set," Proc. ECOOP'95 Workshop on Metrics, 1995
2. J. Daly, J. Miller, A. Brooks, M. Roper, M. Wood, "The Effect of Inheritance on the Maintainability of Object-Oriented Software: An Empirical Study". In Proc. of the IEEE International Conf. on Software Maintenance, 1995, pp. 20-29.
3. F. Brito e Abreu, Melo, W., "Evaluating the Impact of Object Oriented Design on Software Quality", Proceedings of Third International Software Metrics Symposium, pp. 90-99, 1996.
4. J. Mayrand, F. Guay, E. Merlo, "Inheritance Graph Assessment Using Metrics". Proc. Third Int. Software Metrics Symposium (Metrics'96), 1996.
5. G. Poels, G. Dedene, "Evaluating the Effect of Inheritance on the Modifiability of Object-Oriented Business Domain Models", in: Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001), Lisbon, Portugal, 2001, pp. 20–28.
6. F.T.Sheldon, K.Jerath, H.Chung, "Metrics for maintainability of class inheritance hierarchies", Journal of Software Maintenance and Evolution: Research and Practice, Vol. 14, pp. 1-14, 2002
7. S.Ducasse, D.Pollet, M.Suen, H.Abdeen, I.Alloui, "Package Surface Blueprints: Visually Supporting the Understanding of Package Relationships", ICSM 2007, 2007
8. K. M. Breesam, "Metrics for Object-Oriented Design Focusing on Class Inheritance Metrics", In DepCoS-RELCOMEX, pp. 231–237. IEEE Computer Society, 2007.
9. S. Denier, H.A. Sahraoui, "Understanding the Use of Inheritance with Visual Patterns", in: Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement, ESEM 2009, USA, October 2009, pp.79–88.
10. N. S. Gill, S. Sikka, "Inheritance Hierarchy Based Reuse & Reusability Metrics in OOSD", International Journal on Computer Science and Engineering (IJCSE), vol.3, June 2011, pp.2300-2309.

11. Ph.Dugerdil, "Assessing Legacy Software Architecture with the Autonomy Ratio Metric", In Software Engineering and International Journal (SEIJ). Vol. 1, No. 1, Sept. 2011
12. D.Goman, "Analyse Dynamique de l'architecture de Hibernate en lien avec les strategies de mapping", Travail de diplôme réalisé en vue de l'obtention du diplôme HEG, filière informatique de gestion, Haute école de gestion de Genève, 2012
13. L.Belady, M.Lehman, "A model of large program development". IBM Systems Journal 15, 3 (1976), 225-251.
14. J.A.McDermid, "Complexity: Concept, Causes and Control". Proc. IEEE Int. Conf. On Complex Computer Systems, 2000.
15. A.Patricio, "*Java Persistence et Hibernate*." Paris : Eyrolles, 2008, p.365.
16. S.Adolph, P.Bramble, A.Cockburn, A.Pols, "Patterns for Effective Use Cases", 2002.
17. A.Cockburn, "Writing Effective Use Cases", ISBN 0-201-70225-8, 2000.
18. Ph. Dugerdil, "Trace Loader. User Manual", 2013.
19. Ph. Dugerdil, "Modélisation Object & UML Module 623.1", 04.10.2011.
20. Ph. Dugerdil, "Instrumentation\_plugin\_user\_manual.pdf", 2012.
21. M.Niculescu, "Evaluating the efficiency of using the Autonomy Ratio Metric for assessing ArgoUML architecture", Travail de diplôme réalisé en vue de l'obtention du diplôme HES, filière informatique de gestion, Haute école de gestion de Genève, 2012.
22. Murphy Ph, "Beyond Benchmarks: Estimate Your Application Maintenance Costs Using Internal Data", 2009.
23. Ph. Dugerdil, "Génie logiciel Module 625-1", 25.09.2012.
24. Ph. Dugerdil, "Modélisation Objet & UML Module 623.1", 04.10.2011

# Annexes

## Annexe 1 : Hiérarchie de classes du projet java GestionnaireVehicules.

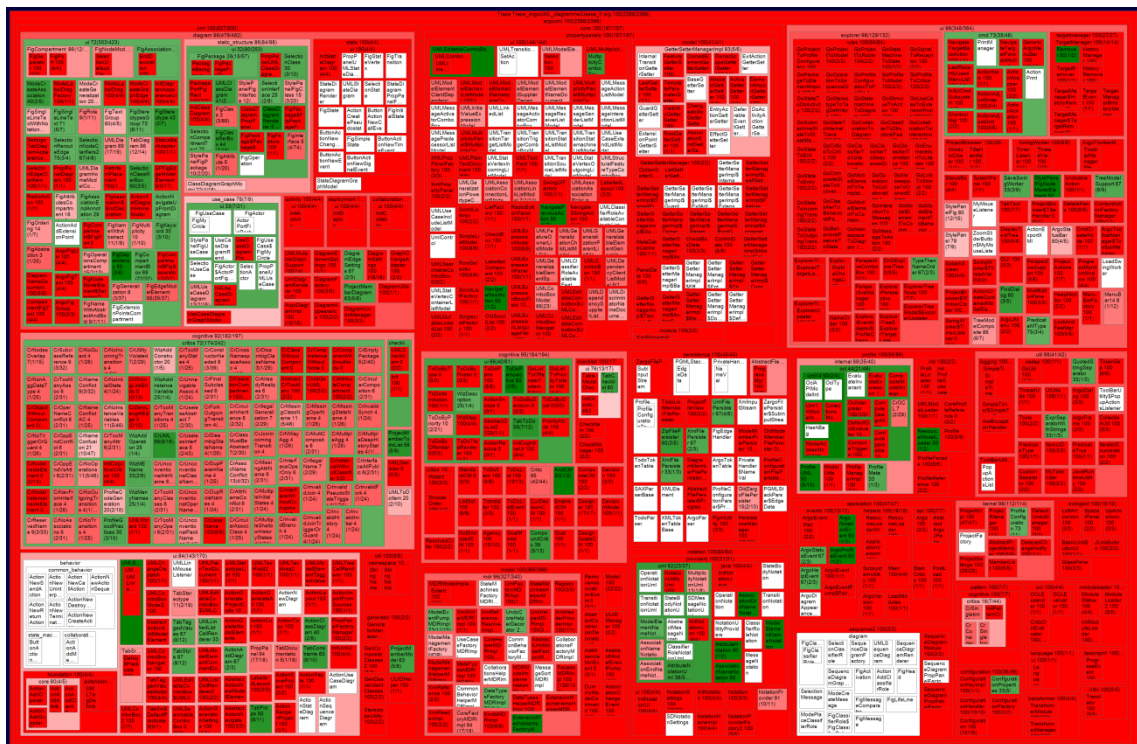


## Annexe 2 : cartes THD

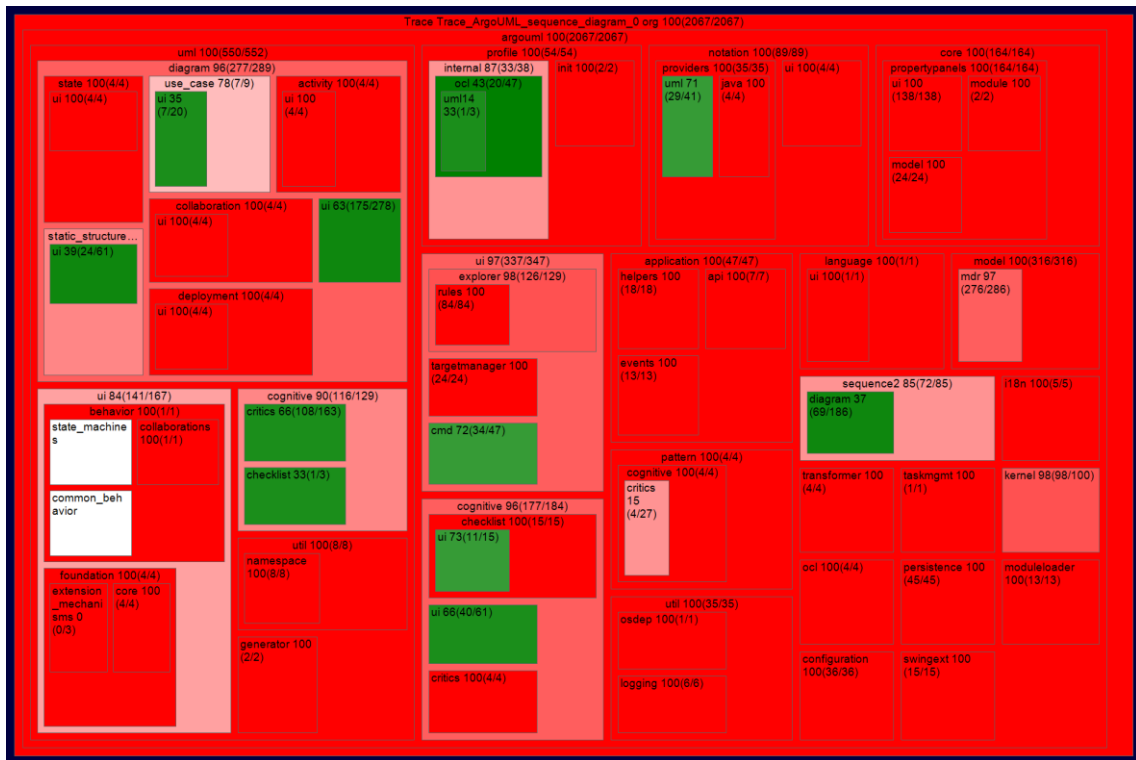
### Annexe 2.1 : Scénario – Création d'un diagramme de classe et sauvegarde



## Annexe 2.1 : Scénario – Création d'un diagramme de classe et sauvegarde

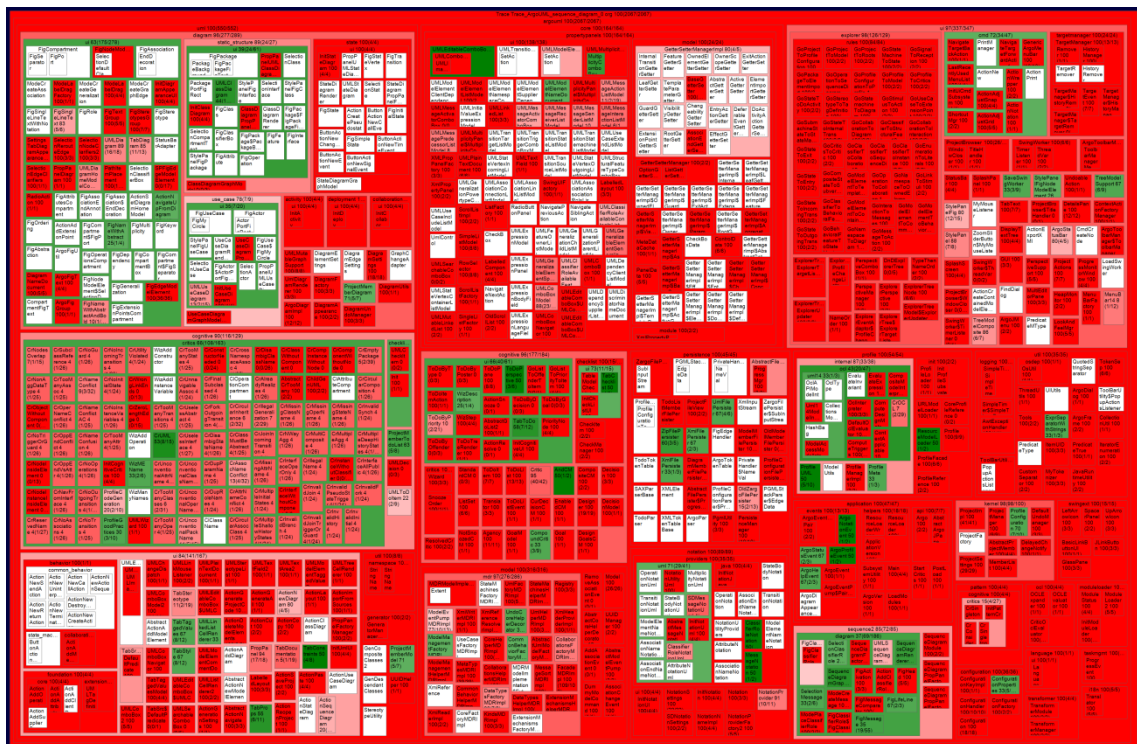


## Annexe 2.2 : Scénario – Création d'un diagramme de séquence et sauvegarde





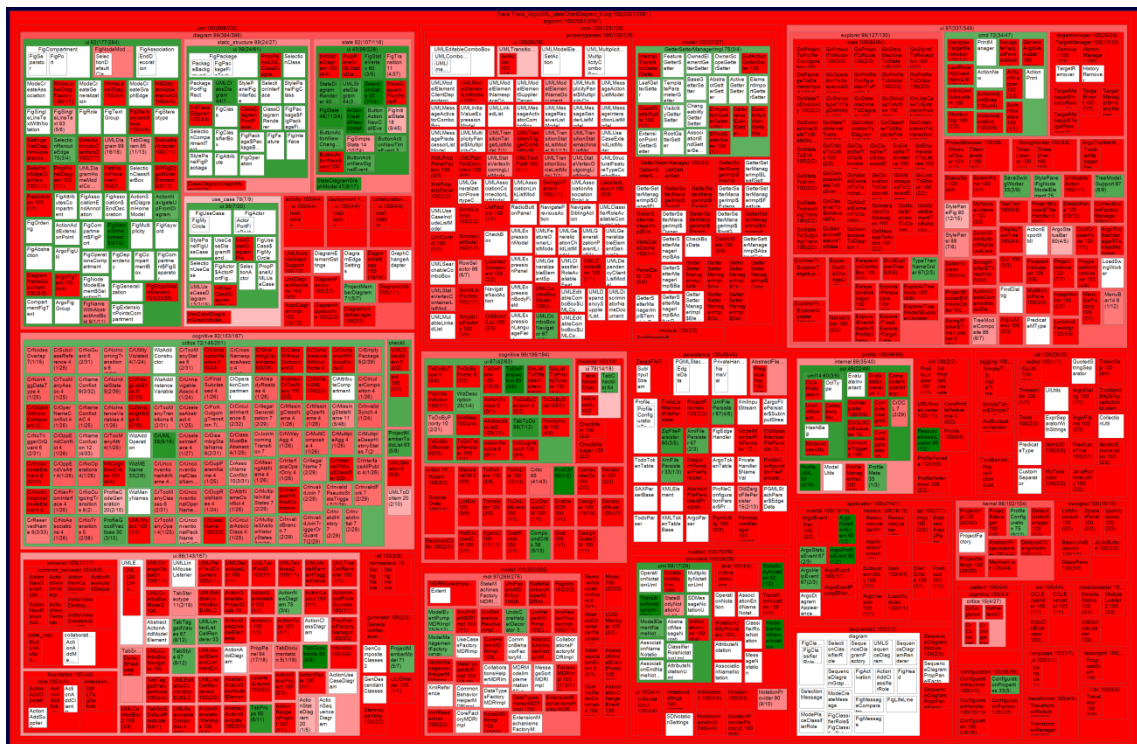
## Annexe 2.2 : Scénario – Création d'un diagramme de séquence et sauvegarde



### Annexe 2.3 : Scénario – Création d'un diagramme d'états et sauvegarde



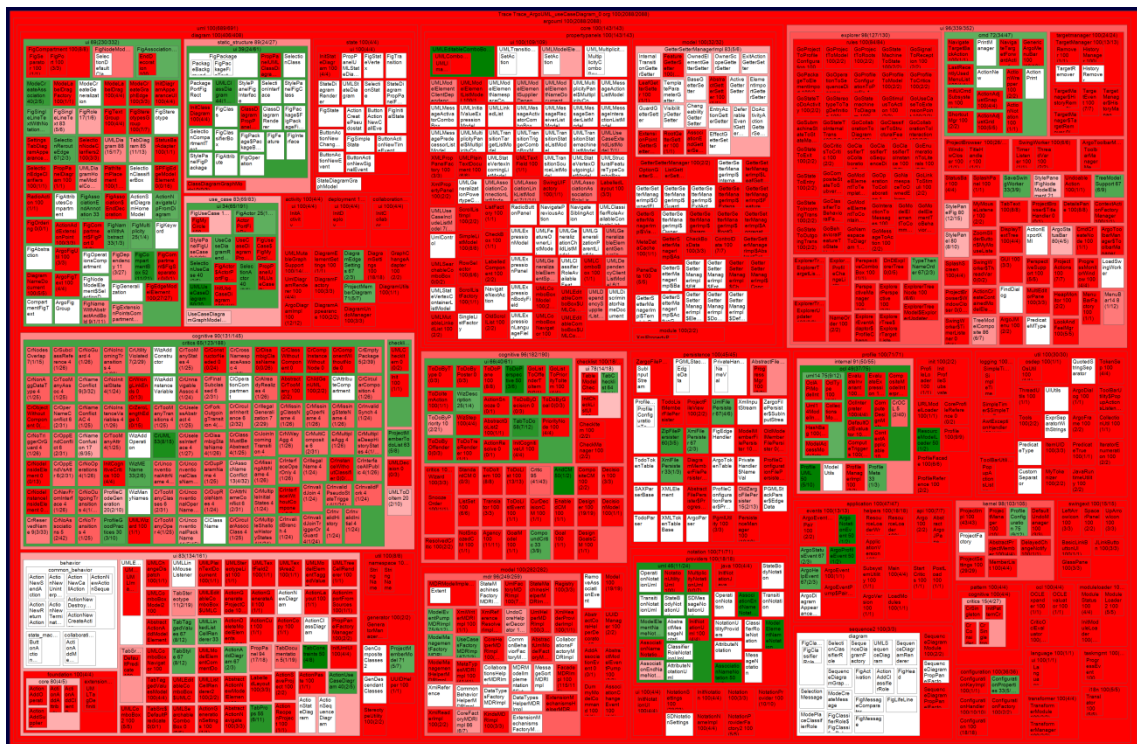
## Annexe 2.3 : Scénario – Création d'un diagramme d'états et sauvegarde



## Annexe 2.4 : Scénario – Création d'un diagramme use-case et sauvegarde



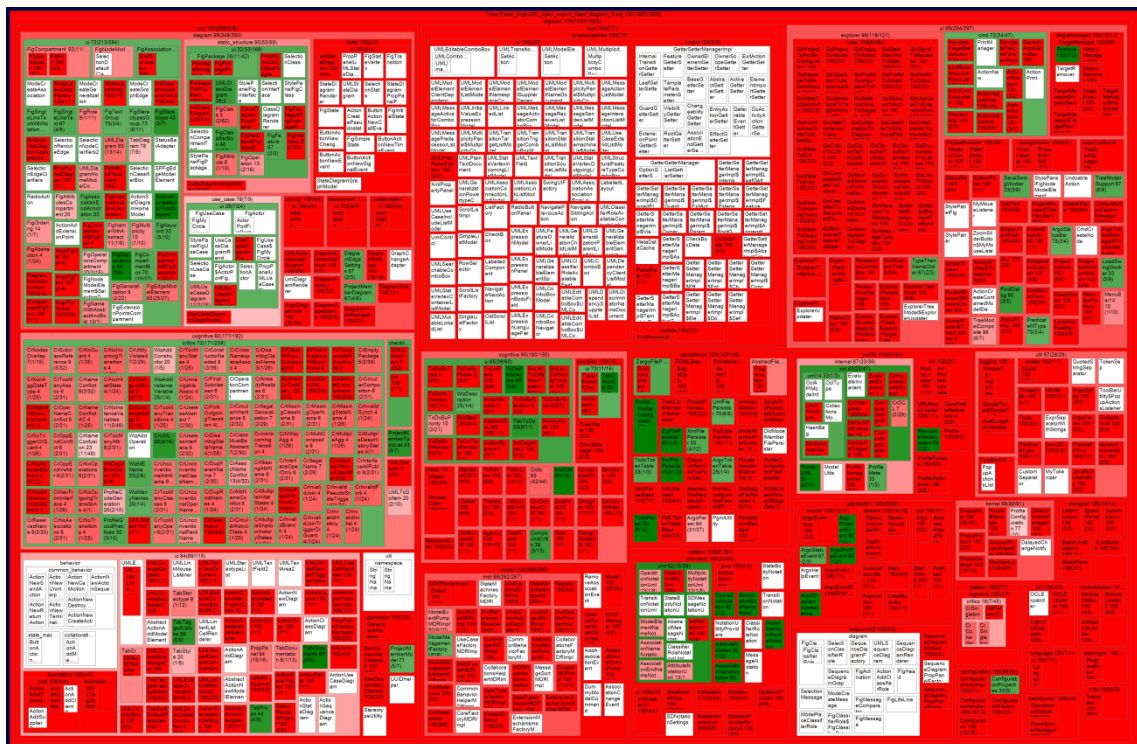
## Annexe 2.4 : Scénario – Création d'un diagramme use-case et sauvegarde



## Annexe 2.5 : Scénario – Ouvrir et exporter le diagramme de classes



## Annexe 2.5 : Scénario – Ouvrir et exporter le diagramme de classes

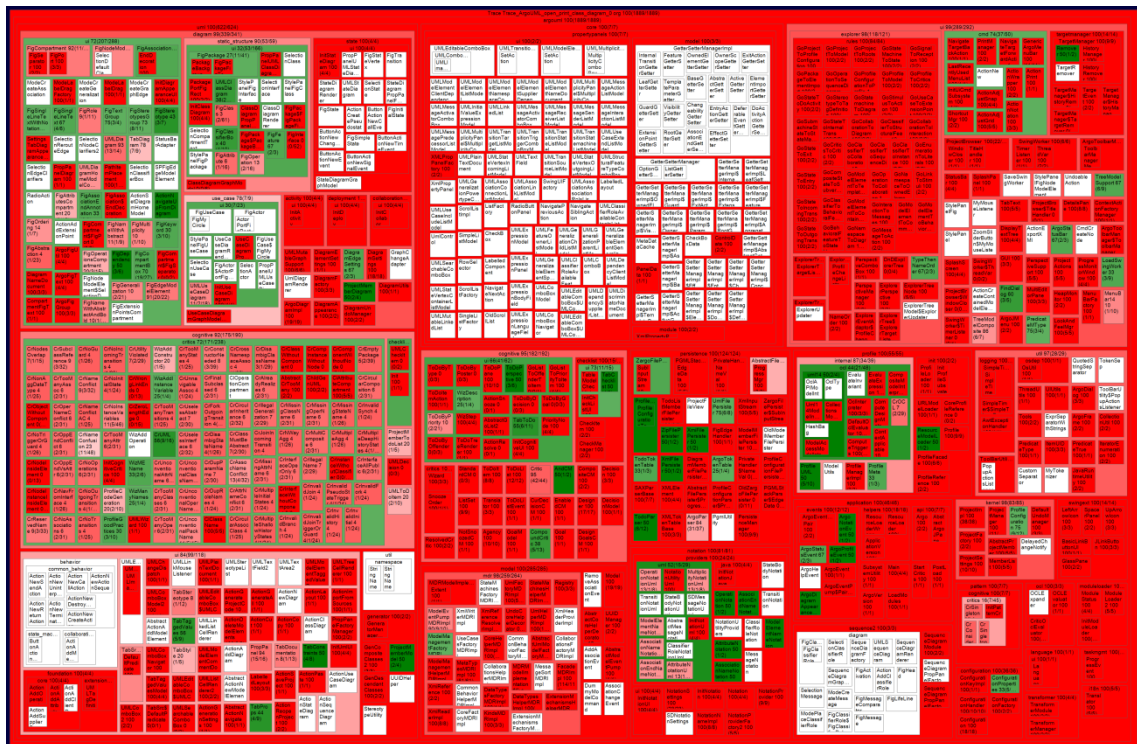


## Annexe 2.6 : Scénario – Ouvrir et imprimer le diagramme de classes





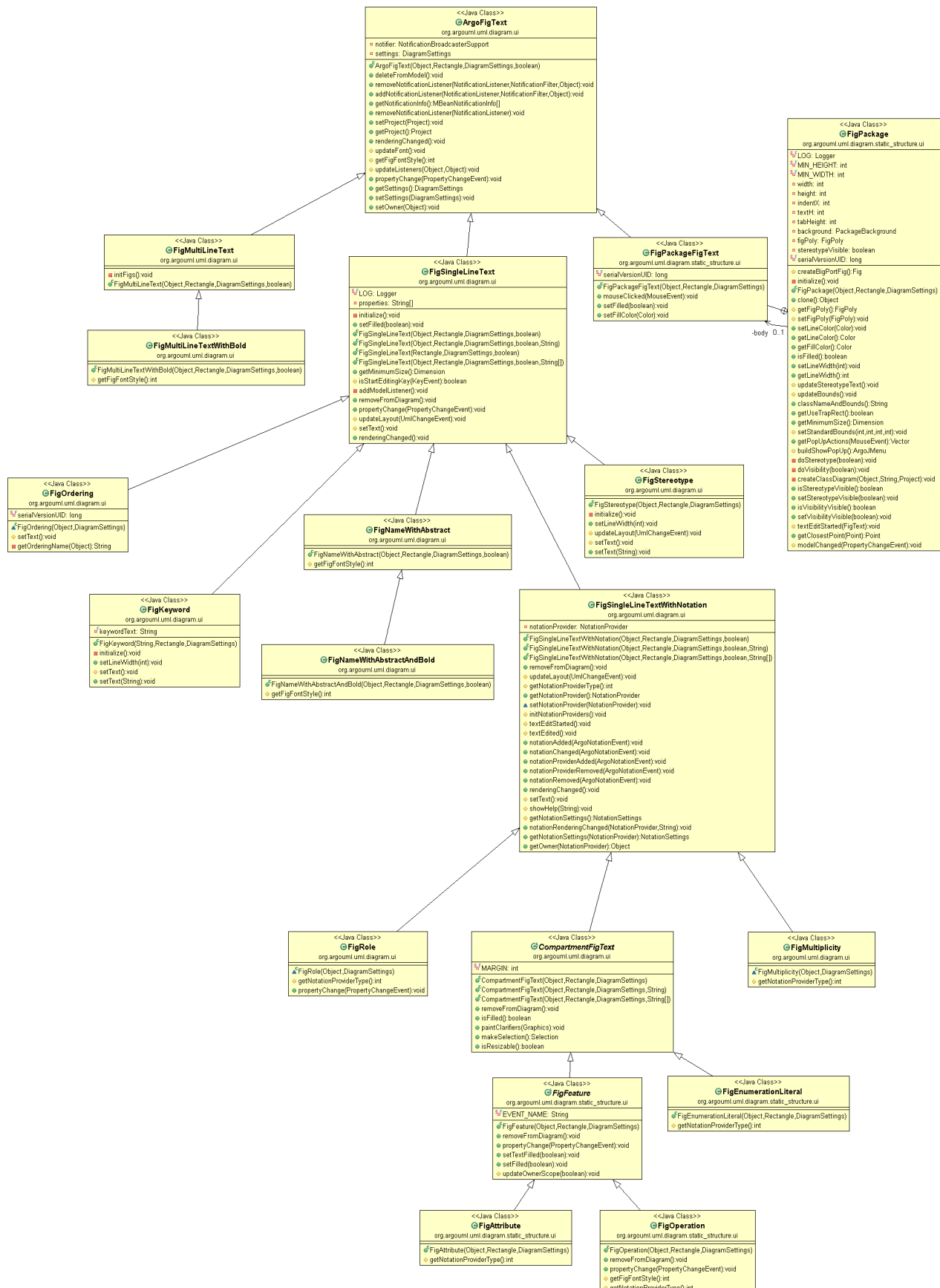
## Annexe 2.6 : Scénario – Ouvrir et imprimer le diagramme de classes



### Annexe 3.1 : Hiérarchie complète 1



### Annexe 3.2 : Hiérarchie complète 2



## **Annexe 4 : Installation des outils nécessaire à l'analyse du THD**

Voici les instructions pour installer tous les outils nécessaires au calcul et à la visualisation du THD sur une application java dont vous possédez le code source. Les outils sont fournis sur le DVD annexé à ce document.

### **Eclipse**

Il est nécessaire de posséder l'environnement Eclipse. La mise en œuvre de ce travail a été effectuée avec le SDK Eclipse version : 4.2.2 Build id : M20130204-1200. Le logiciel que vous voulez analyser doit être installé avec ses sources et configuré comme un Java Eclipse Project. Il est conseillé de faire un backup du code source du logiciel analysé car celui-ci va être affecté lors de l'instrumentation. Toutefois, il est très facile de revenir à la forme d'origine. Ceci n'est qu'une mesure de sécurité.

### **Instrumentor**

Copiez le plugin servant à l'instrumentation du code nommé « ch.hesge.csim2.instrumentor\_1.0.0.jar » du répertoire Tools\Instrumentor à votre répertoire de plugin d'installation Eclipse. En règle générale, ce répertoire se trouve « C:\Program Files\eclipse\plugins ». Copiez l'instrumentorRuntime.jar à l'endroit de votre choix. Il est conseillé de le copier dans le répertoire du projet java (l'application analysée) car cette librairie va être utilisée par ce projet. Finalement, rendez-vous à la fenêtre de paramètres du projet java, et ajoutez le fichier jar externe « instrumentorRuntime.jar ». Pour ce faire, suivez l'emplacement suivant : « Project settings windows (du projet java analysé) \ Library pane \ Add External Jars » et sélectionnez votre fichier « instrumentorRuntime.jar ».

### **Trace Loader**

Lancez le Trace Loader en exécutant le fichier « TraceLoaderOracle.jar » si vous utilisez une base de données Oracle. Pour toute autre base de données relationnelle qui utilise SQL, utilisez le fichier « TraceLoaderODBC.jar » pour tout autre base de données relationnelle qui utilise SQL. Ces fichiers se trouvent dans le répertoire Tools\TraceLoader du DVD. Une fois lancé, appuyez sur le bouton « Database Parameter » pour paramétrer la connexion à votre base de données. J'ai personnellement utilisé Oracle en version 10g.

## **PL/SQL Developer**

Vous avez besoin d'un outil permettant de gérer des scripts SQL qui vont s'effectuer sur votre base de données. La mise en œuvre de ce travail a été effectuée avec la version 8.04.1514. Vous trouverez dans le répertoire Tools\PLsqlDev l'exécutable permettant son installation. Cet outil est nécessaire pour exécuter les scripts permettant de traiter les traces analysées mais également afin d'exporter les résultats en format CSV. Ce fichier permettra par la suite d'être traité par le « TreeMapGenerator ». Pour traiter les scripts, on peut utiliser des outils équivalents. Cependant, dans notre travail, il est nécessaire d'utiliser PL/SQL car le format du fichier CSV est spécifique à PL/SQL et l'application java « TreeMapGenerator » prend en compte ce format lors de son exécution. Si pour une raison quelconque vous devez utiliser un autre outil que PL/SQL, il faudra alors modifier le code de l'application « TreeMapGenerator » pour adapter le format du fichier CSV généré par votre outil SQL.

## **Scripts SQL**

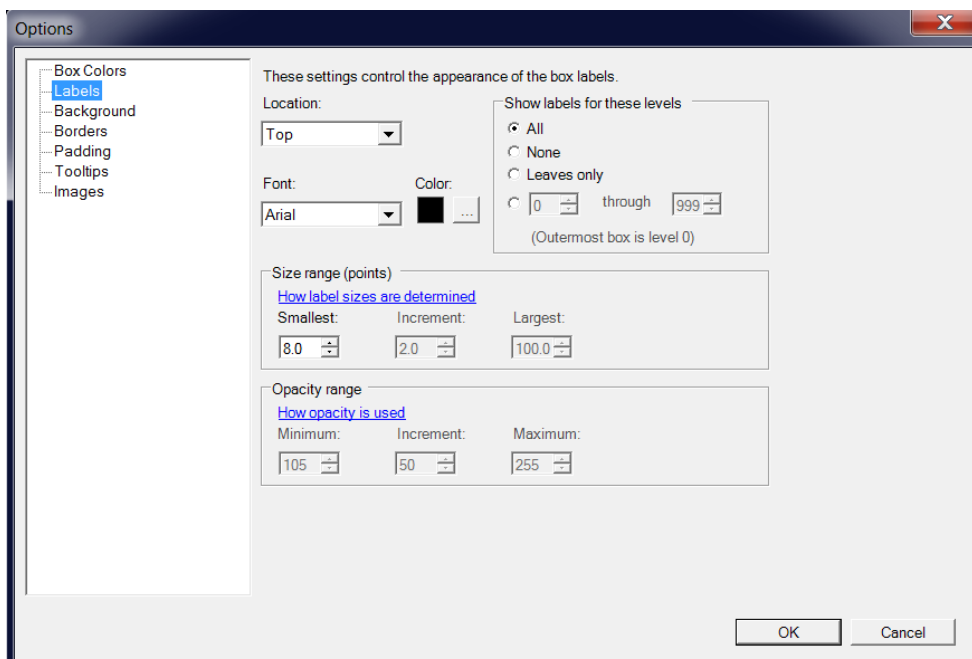
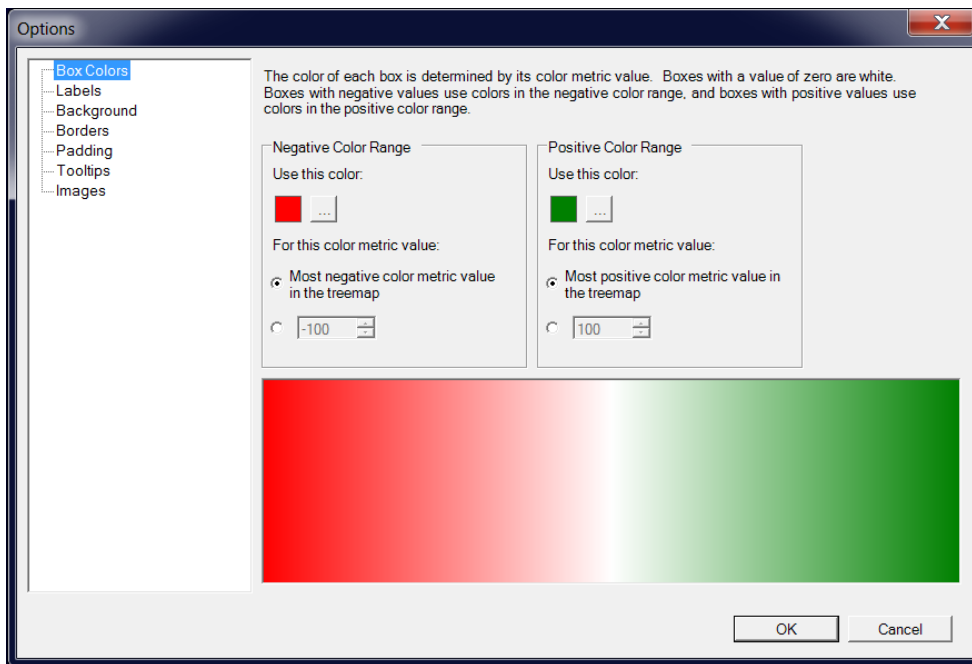
Lancez PL/SQL après avoir démarré votre base de données Oracle. Exécutez le script « Trace\_package\_list\_GRANT\_NEEDED.sql » se trouvant dans le répertoire SQL\_Scripts\1\_Grant\_Privileges. Exécutez ensuite le script « Create\_Table\_Trace\_taux\_heritage\_dynamique » se trouvant dans le répertoire SQL\_Scripts \2\_Create\_Table. Maintenant, ouvrez et exécutez tous les scripts présents dans le répertoire SQL\_Scripts\3\_Oracle\_Procedures&Functions. Ceci va permettre de créer les fonctions et procédures stockées dans la base de données.

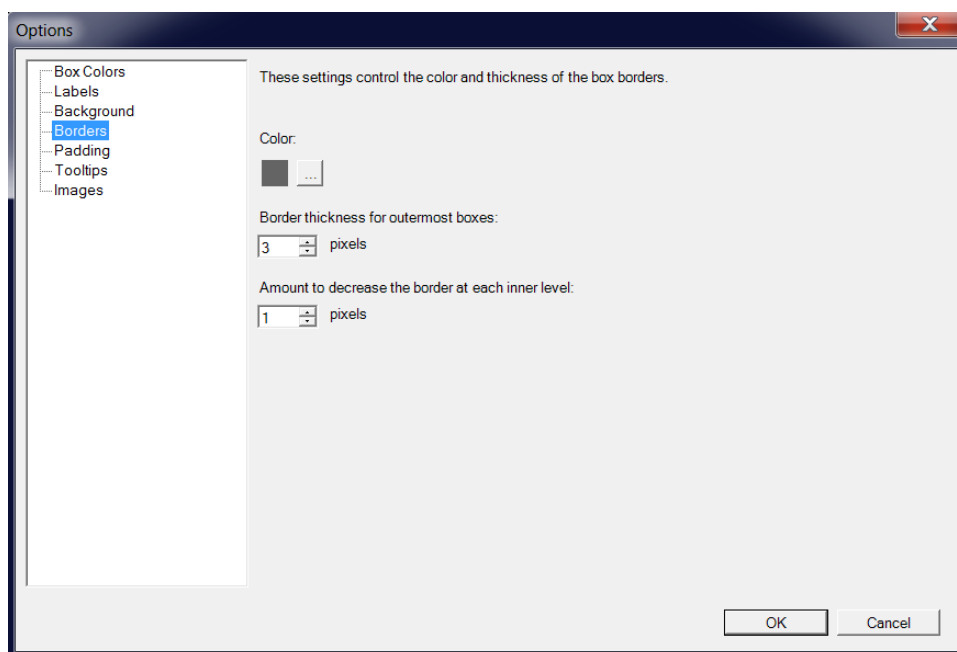
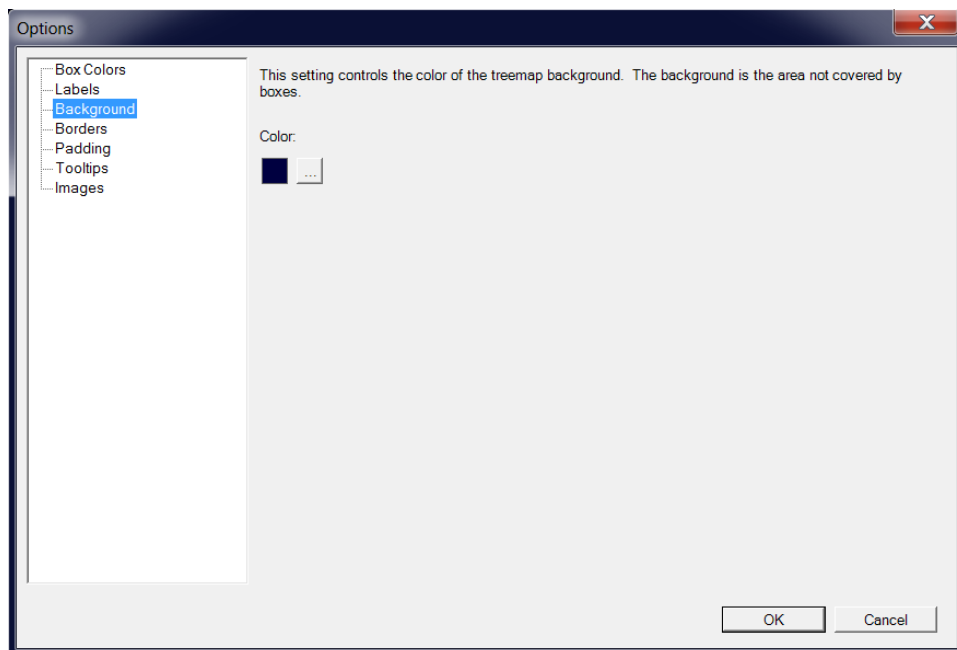
## **TreeMapGenerator**

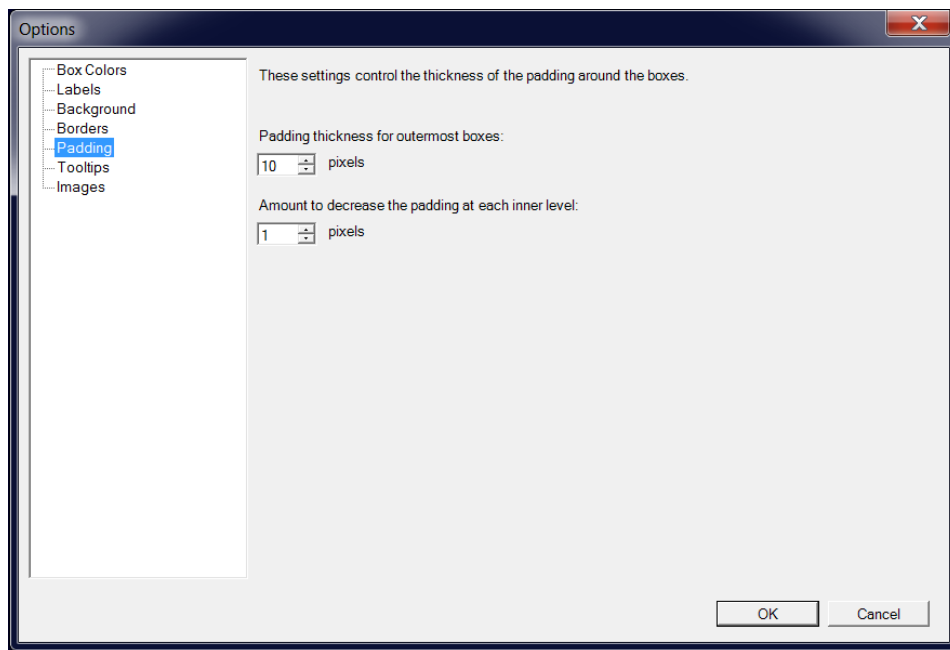
Ce programme java va générer une carte THD à partir du fichier CSV généré par PL/SQL. Le programme est fourni comme un projet Eclipse avec son code source. Vous n'avez pas besoin de le modifier à moins d'utiliser un autre outil que PL/SQL. Copier le répertoire TreeMapGenerator à l'emplacement de votre choix sur votre machine puis importez-le en Java Eclipse Project depuis l'emplacement choisi.

## **Microsoft Treemapper**

Il s'agit d'un logiciel gratuit qui permet de visualiser les cartes THD. Il faut l'installer sur votre machine. Dans la mise en œuvre de ce travail, j'ai utilisé la version 1.0.134. Afin de rendre les cartes exactement comme décrites dans ce document, allez dans « View \ Options » et paramétrez comme suit :









## **Annexe 5 : Guide pour utiliser les outils installés**

Ce guide décrit les étapes à suivre pour pouvoir générer des cartes THD pour un logiciel que vous souhaitez analyser. Ces étapes ont été effectuées pour l'analyse d'ArgoUML. Vous pouvez vous référer à la figure 34 qui résume visuellement ces étapes.

### **1. Sélectionner les scénarios**

Choisissez les différents scénarios que vous souhaitez analyser. La sélection devrait se faire selon les use-cases qui ont de la valeur pour l'entreprise concernée. Ils devraient également couvrir toutes les fonctionnalités que vous souhaitez analyser.

### **2. Instrumenter**

Instrumenter le code source de l'application à analyser. Pour les détails vous pouvez regarder le manuel utilisateur [20] contenu dans le DVD annexé.

### **3. Lancer l'application**

Exécutez l'application par le biais d'Eclipse. Réalisez un des scénarios que vous souhaitez analyser. A la fin de chaque scénario, fermez votre application et prenez la trace générée (« Trace.txt ») se trouvant dans le répertoire du projet Eclipse. Renommez-la comme bon vous semble. Ce fichier contient les traces des méthodes appelées lors de l'exécution de votre scénario. Il est conseillé de renommer le fichier avec un nom en rapport avec votre scénario. Pour plus de détail, consultez le manuel utilisateur [20].

### **4. Charger les traces dans la base de données**

Ouvrez le Trace Loader. Celui-ci va vous permettre de charger votre fichier de traces dans votre base de données. Appuyez sur le bouton « Add » et sélectionnez le fichier trace que vous voulez insérer dans votre base. Vous pouvez choisir plusieurs fichiers à la fois, mais il faut savoir que ces fichiers peuvent être très grands (plusieurs GBs). Ceci peut devenir un problème si la taille de votre base de données est limitée (avec Oracle Express Edition, la limite est de 4 Go). C'est pourquoi il est conseillé de ne charger que deux ou trois traces à la fois. (Il faut environ 1 heure pour 1Go de trace). Une fois la trace chargée, vous pouvez directement fermer Trace Loader. Si vous voulez cependant utiliser Trace Loader pour gérer ou rechercher des traces spécifiques dans votre fichier je vous conseille de consulter le manuel utilisateur du Trace Loader [18].

## 5. Exécuter les scripts SQL

Rendez-vous sur le répertoire contenant les procédures et fonctions (SQL\_Scripts\3\_Oracle\_Procedures&Functions). Effectuez un clic droit sur le fichier « Process\_THD\_ALLTRACE » et sélectionnez Test, (dans le cas où la procédure n'est pas compilée il suffit de le faire avec un clic droit, compilation). Dans la fenêtre de test, rendez-vous sur « SGBD sortie » et modifiez la taille de la mémoire tampon à 100000 (afin de permettre aux messages de débogage de s'afficher sur la console : si la mémoire tampon n'est pas assez grande, il y aura une erreur et le script s'arrêtera). Exécutez le script en appuyant sur la touche F8. Attendez que le script se termine. La durée peut varier en fonction des réglages de votre installation de base de données Oracle et le nombre de vos scénarios chargés dans la base. Cela va créer pour chaque scénario (fichier trace) une `tracenodeX_package_list`, qui va s'ajouter à la table `Trace_Taux_Heritage_Dynamique`. Cette table contient les résultats du THD, pour toutes les traces (à noter que la table doit exister, la création se fait avec le script « `Create_Table_Taux_Heritage_Dynamique.sql` »). Vous pouvez utiliser une alternative en ouvrant un fichier SQL et en exécutant la requête suivante :

```
begin
Process_THD_ALLTrace ;
end ;
select * from trace_taux_heritage_dynamique ;
```

Une fois les résultats visibles, il vous suffit d'effectuer un clic droit sur les résultats et de choisir Export en fichier CSV. Ce fichier sera utile au Trace Map Generator.

## 6. Exécuter le TreeMapGenerator

Importez l'application TreeMapGenerator sous Eclipse. Le projet se trouve dans le DVD dans le répertoire Tools. Effectuez un clic droit sur le projet TreeMapGenerator et sélectionnez « Run as\ Run Configurations ». Sélectionnez l'onglet « Arguments » et saisissez sous le champ « Program arguments » le nom de votre fichier CSV généré par PL/SQL. Vous devez l'écrire en rajoutant à la fin le format, à savoir « .csv ». Sélectionnez ensuite le radio bouton « Other » dans la rubrique « Working directory » et saisissez le chemin d'accès à votre fichier CSV contenant les résultats. Une fois les modifications effectuées, appuyez sur le bouton « Run ». L'application va s'exécuter et créer des fichiers CSV formatés afin de pouvoir être lus par le Microsoft Treemapper. Il y aura un fichier par scénario distinct. Chaque fichier se nommera « `X_map.csv` », X

étant le nom du scénario (le nom du fichier trace.txt) en question. Ces fichiers seront localisés au même endroit que le fichier résultats.

## **7. Ouvrir la carte avec Microsoft Treemapper**

Lancez Microsoft Treemapper en exécutant « Treemapper.exe » qui se trouve dans le répertoire Tools\Treemapper. Ouvrez un des fichiers générés par le TreeMapGenerator en sélectionnant dans le menu « File\Open ». Pour pouvoir sélectionner des fichiers CSV et pour que le format généré par le TraceMapGenerator soit correctement lu par le Treemapper, il faut choisir comme type de fichier « Comma-Delimited Non-Cumulative Files ». Une fois le fichier sélectionné, vous pourrez enfin visualiser une carte THD concernant le scénario mentionné. Si vous avez bien renommé vos fichiers texte contenant les traces comme mentionné plus haut, vous n'aurez aucun mal à distinguer la carte THD que vous consultez, étant donné qu'elle porte le nom du scénario concerné.