République Algérienne Démocratique et Populaire
الجمهورية الجزائرية الديمقراطية الشعبية
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
وزارة التعليم العالي و البحث العلمي

---

**Mémoire de fin d'études**

Pour l'obtention du diplôme d'Ingénieur d'Etat en Informatique

**Option : Systèmes Informatiques et Logiciels (SIL)**

---

# Analyse de flot de données en utilisant la technique du program slicing sur Moose

---


*Réalisé par :*
M. Ahmed Zaki Bennecer

*Promoteurs :*
Prof. Anne Etien (INRIA - RMod)
HDR. Nicolas Anquetil (INRIA - RMod)

*Encadré par :*
Mme. Lamia Yessad (ESI)


Promotion : 2021/2022

# Dédicaces

Je tiens sincèrement, et avec un grand plaisir, à dédier ce travail :

- À l'être le plus précieux de ma vie, ma mère,
- À celui qui a fait de moi un homme, mon père,
- À mes chers frère et soeur,
- À tous mes proches.

**- Ahmed Zaki -**

# Remerciements

C'est avec grand plaisir que je réserve cette page, en signe de gratitude et de reconnaissance à tout ceux qui m'ont aidé de prêt ou de loin à la réalisation de ce travail.

Tout d'abord, je remercie ALLAH Le Tout Puissant, Le Miséricordieux qui m'a donné la force, le courage et la volonté pour achever ce travail.

Je tiens à remercier particulièrement Anne ETIEN et Nicolas ANQUETIL mes promoteurs de stage pour leur aide et ainsi leur disponibilité incontestable. Ils ont su m'écouter, m'orienter et me conseiller. Je salue leur disponibilités, surtout leur compréhension et soutien dans les situation critiques que j'ai affronté et je leur serai très reconnaissant.

Je tiens à exprimer mes remerciements les plus distingués à Lamia YESSAD mon encadrante pour ses qualités scientifiques et humaines.

Plus généralement, je remercie tous les membres de l'équipe RMod de INRIA pour l'excellent atmosphère de travail et aussi toute l'équipe administrative pour leur efforts inestimables.

Pour finir, je tiens à remercier toute la famille de l'Ecole nationale Supérieure d'Informatique, enseignants et personnel, pour la richesse et la qualité de la formation.

**- Ahmed Zaki -**

# Résumé

Aujourd'hui, les industries dépendent fortement des technologies logicielles. Elles sont au cœur de la plupart des industries comme les banques, les assurances, les hôpitaux voire les grandes entreprises tels que Apple, Google, Facebook et Amazon. Et comme l'économie dépend de ces logiciels, il est incontournable de réfléchir aux moyens pour maintenir et analyser ces systèmes, en utilisant des plateformes de rétro ingénierie.

Lors de l'analyse de ces systèmes, il est important de faire un compromis entre le niveau de détail du code et la scalabilité. En effet, vu que ces programmes comptent souvent plusieurs centaines de classes, leur abstraction s'arrête souvent à l'appel de méthode, les accès aux variables sans donner le détail de la méthode elle-même. Parfois, ce niveau de détail ne suffit pas et il faut descendre, et effectuer les analyses, au niveau de l'AST (Abstract Syntax Tree). Par exemple, ce type d'analyse est important dans le cadre du RGPD (Règlement Général sur la Protection des Données), les entreprises doivent être capables, pour chaque donnée, de comprendre d'où elle vient, comment et où elle a été modifiée.

Ce projet vise à étudier la méta-plateforme Moose, à concevoir et implémenter un outil d'analyse visuel permettant la compréhension des flux de données dans les programmes en utilisant le program slicing.

---

**Mots clés :** Analyse de flux de données, Modèles, Méta-modèles, Ingénierie logicielle, Rétro-ingénierie, Programmation Orienté Objet.

---

# Abstract

Today, industries rely heavily on software technologies. They are at the heart of most industries such as banks, insurance, hospitals and even large companies such as Apple, Google, Facebook and Amazon. As the economy depends on softwares, it is inevitable to think about ways to maintain and analyze these systems, by using reverse engineering tools.

When analyzing these systems, it is important to make a trade-off between the level of code detail and scalability. Since these programs often have several hundred classes, their abstraction often stops at the method call, access to variables without giving the details of the method itself. Sometimes, this level of detail is not enough and you have to go down, and perform the analyses, to the AST level (Abstract Syntax Tree). For example, this type of analysis is important in the context of the GDPR (General Data Protection Regulation), companies must be able, for each piece of data, to understand where it comes from, how and where it was modified.

This project aims to study the Moose meta-platform, to design and implement a visual analysis tool allowing the understanding of data flows in programs using program slicing.

---

**Keywords :** Data flow analysis, Models, Meta-models, Software engineering, Reverse engineering, Object Oriented Programming.

---

# ملخص

اليوم ، تعتمد الصناعات بشكل كبير على تقنيات البرمجيات. هم في قلب أغلب الشركات مثل البنوك، التأمين، المستشفيات وحتى الكبرى منها مثل Apple و Google و Facebook و Amazon. وبما أن الاقتصاد يعتمد على هذه البرمجيات ، فلا مفر من التفكير في طرق لصيانة هذه الأنظمة وتحليلها ، من خلال استعمال أدوات الهندسة العكسية.

عند تحليل هذه الأنظمة ، يجب التوفيق بين مستوى التفصيل في الكود ومايعرف بـ Scalability . في الواقع ، نظرًا لأن برامج هذه الأنظمة غالبًا ما تحتوي على عدة مئات من الفئات Classes ، فإن تجريدها غالبًا ما يتوقف عند استدعاء الطريقة ،Method والوصول إلى المتغيرات دون إعطاء تفاصيل الـ Method نفسها. في بعض الأحيان، هذا المستوى من التفصيل لا يكون كافيًا وعلينا النزول، وإجراء التحليلات، على مستوى الـ AST (شجرة التركيب المجردة). على سبيل المثال، هذا النوع من التحليل مهم أيضًا في سياق GDPR (اللائحة العامة لحماية البيانات)، يجب أن تكون الشركات قادرة ، لكل جزء من البيانات، على فهم مصدرها وكيف وأين تم تعديلها.

يهدف هذا المشروع إلى دراسة منصة Moose الوصفية، تصميم وتنفيذ أداة تحليل مرئي تسمح بفهم تدفقات البيانات في البرامج التي تستخدم تقطيع البرامج.

---

**كلمات مفتاحية :** تحليل تدفق البيانات، النماذج، النماذج الوصفية، هندسة البرمجيات، الهندسة العكسية، البرمجة الكائنية.

---

# Table des matières

# Table des figures

# Liste des tableaux

# Liste des sigles et acronymes

| | |
|---|---|
| **SDLC** | *Software Development LifeCycle* |
| **AST** | *Abstract Syntax Tree* |
| **MDA** | *Model-Driven Architecture* |
| **OMG** | *Object Management Group* |
| **UML** | *Unified Modeling Language* |
| **CFA** | *Control Flow Analysis* |
| **DFA** | *Data Flow Analysis* |
| **ASG** | *Abstract Semantic Graph* |
| **JIT** | *Just In Time* |
| **DCU** | *Diagramme de Cas d'Utilisation* |
| **MM** | *Méta-Modèle* |

# Introduction générale

Aujourd'hui, les industries dépendent fortement des technologies logicielles, et la majorité des entreprises, indépendamment de leur taille et du secteur d'activité, font désormais face au problème bien connu de devoir gérer, maintenir, faire évoluer ou remplacer leurs systèmes logiciels existants (Lehman 1980). Et comme l'économie dépend de ces logiciels (Koskinen et al. 2005), les entreprises doivent régulièrement faire évoluer (au moins en partie) leurs systèmes logiciels déjà existants avant que ceux-ci ne deviennent véritablement obsolètes ou ne commencent à dysfonctionner.

Les projets de migration/modernisation logiciels, dans les entreprises, ne peuvent donc pas être pris à la légère, puisqu'ils ne viennent jamais sans d'importants challenges associés (El Emam et Koru 2008). Dans le cas parfait, les ingénieurs devront adopter les disciplines et les approches de développement adéquats pour leur projet et anticiper les problèmes à venir. Cependant, en pratique, ils ont une vue partielle et limitée sur la complexité et les potentielles conséquences de leurs produits logiciels. (Khadka et al. 2014).

Afin de mettre en oeuvre ces projets de migration, la modélisation joue un rôle inévitable. C'est un paradigme du génie logiciel basé sur la création, la manipulation et l'utilisation des modèles. Des études relativement récentes sur l'application concrète de ces techniques, dans l'industrie, ont montré que les techniques basées sur les modèles ont déjà engendré des bénéfices intéressants dans le cadre de différents types de projets ou d'activités à forte composante logicielle (Hutchinson et al. 2011 ; Whittle, Hutchinson et Rouncefield 2013). Les modèles permettent une abstraction subjective d'un objet, d'un concept ou d'un logiciel. Ils sont donc particulièrement utiles dans le contexte de la rétro-ingénierie.

L'existence de plusieurs modèles signifie l'existence de plusieurs niveaux d'abstraction. Souvent, l'abstraction qui s'arrête à l'accès aux variables et aux appels méthodes permet une analyse déjà pertinente sur les programmes. Dans d'autres cas, cette abstraction n'est pas suffisante et on doit descendre au niveau de l'AST (Abstract Syntax Tree). Par exemple, pour le debuggage, le niveau de l'AST peut être important pour analyser comment les variables sont utilisées et comment les données « circulent » dans le code. Ce type d'analyse est aussi important par exemple dans le cadre du RGPD (Réglement Général sur la Protection des Données), les entreprises doivent être capables, pour chaque donnée, de comprendre d'où elle vient, comment et où elle a été modifiée. Pour répondre à cette problématique, il est important de pouvoir représenter le flux des données dans les programmes informatiques en utilisant le slicing. Le slicing est une technique de rétro-

ingénierie qui accepte comme entrée une variable à une ligne donnée du code source, et nous donne comme sortie l'ensemble des instructions du programme où cette variable peut être changée.

Dans ce rapport, nous présenterons un état de l'art sur la modélisation, la rétro-ingénierie et l'analyse des flux de données. Après cela, nous décrivons le contexte organisationnel et technique du projet. Finalement, nous présentons la solution en passant par l'expression des spécification, la conception et puis la réalisation.

**Organisation du mémoire**

Ce mémoire est organisé en six chapitres :

Le premier chapitre "**État de l'art**" présente une synthèse bibliographique sur le domaine de la rétro-ingénierie et ainsi les concepts liés à l'analyse des flux de données. La première partie du chapitre définit *la maintenance et l'évolution* dans l'ingénierie logicielle. Ensuite nous abordons *la modélisation* et *la rétro-ingénierie*. Et nous finissons ce chapitre par présenter la technique *d'analyse de flux de données* et ainsi *le slicing* des programmes.

Le deuxième chapitre "**Étude de l'existant**" décrit le contexte du projet. Le laboratoire, ses missions, sa vision ainsi que le département d'accueil sont présentés dans la première partie du chapitre. La deuxième partie est consacrée à la présentation de l'environnement Moose et ainsi les méta-modèles existants au sein de ce dernier.

Le troisième chapitre "**Expression des besoins**" décrit les besoins et les objectifs à atteindre pour l'analyse des flux de données. Nous commençons par identifier les fonctionnalités à développer et les acteurs du système, ensuite nous modélisons et représentons par des diagrammes UML (Unified Modeling Language) les fonctionnalités de la solution.

Le quatrième chapitre "**Conception**" discute la solution proposée. Dans ce chapitre la conception de la solution est détaillée avec des explications et des justifications de chaque concept adopté.

Le cinquième chapitre "**Implémentation et Tests**" présente la mise en œuvre de la solution. Il décrit les différents choix technologiques utilisés, l'implémentation de la solution conçue et les résultats obtenus au cours de la réalisation.

Le sixième chapitre "**Résultats**" aborde les résultats obtenus au cours de la réalisation de la solution en effectuant un test end-to-end. Il présente aussi une analyse de ces résultat en détéctant une anomalie de flux de donnée.

Le mémoire est clôturé par une conclusion générale dont l'objectif principal est de résumer tout ce qui a été vu au sein de ce mémoire et de présenter les différentes perspectives et ainsi la poursuite de ce projet.

# Chapitre 1

# Etat de l'art

## 1.1 Introduction

La rétro-ingénierie est presque aussi ancienne que l'informatique elle-même. Ciblant initialement l'analyse du hardware (Rekoff 1985), elle a rapidement étendu son champ d'application pour se concentrer également sur les systèmes logiciels (Chikofsky et Cross 1990). Après, suite à l'expansion spectaculaire et à l'avènement des logiciels à partir de la fin des années 80, la rétro-ingénierie a généralement été considérée dans le contexte de la gestion des legacy[1] systèmes.

Pour gérer ces systèmes, une compréhension profonde est exigée afin de les représenter sous une forme ou un formalisme différent et à un niveau d'abstraction plus élevé, ou ce qu'on appelle la *modélisation* (Chikofsky et Cross 1990).

Nous définissons dans la première partie de cette synthèse bibliographique la maintenance et l'évolution logicielle. Ensuite nous abordons la modélisation et la rétro-ingénierie, où nous synthétisons quelques techniques de ce domaine. Et nous finissons ce chapitre en exposant l'analyse des flux de données et le programme slicing.

## 1.2 Maintenance et évolution

L'activité de la maintenance est la dernière phase dans le cycle de vie du développement logiciel (SDLC). Ce processus définit les différentes étapes que devraient suivre un produit logicielle pour garantir sa qualité.

Aujourd'hui dans les organisations, ces activités, la maintenance et l'évolution, représentent une grande partie des coûts des logiciels. Cela, nous poussent à élaborer ces notions, leur importance dans le domaine de l'ingénierie logicielle et ainsi leurs coûts et difficultés.

### 1.2.1 Définitions

La maintenance logicielle consiste principalement à corriger les anomalies d'un système logicielle tout en préservant ses fonctionnalités. La maintenance est effectuée après le lancement du produit pour plusieurs raisons, notamment pour améliorer l'ensemble du logiciel, augmenter les performances, répondre aux besoins des clients, ... etc. On a quatre type différents de maintenance (Swanson 1976) :

- **La maintenance logicielle corrective :** a pour but de corriger les défaillances : *défaillances de traitement et défaillances de performance*. Un programme produisant une mauvaise sortie est un exemple de défaillance de traitement. De même, un programme incapable de répondre aux exigences en temps réel est un exemple de défaillance de performance. Le processus de maintenance corrective comprend l'isolement et la correction des éléments défectueux dans le logiciel. En principe, la maintenance corrective est un *processus réactif* (Tripathy et Naik 2014), ce qui

[1] Le terme legacy est bien défini dans l'annexe. C'est un terme qui est utilisé souvent dans le domaine de la rétro-ingénierie signfiant les systèmes obsolètes.

signifie que la maintenance corrective est effectuée après avoir détecté des défauts dans le système.

- **La maintenance logicielle perfective :** Le but de la maintenance parfaite est d'apporter une variété d'améliorations, à savoir l'expérience utilisateur, l'efficacité du traitement et la maintenabilité. Par exemple, les sorties du programme peuvent être rendues plus lisibles pour une meilleure expérience utilisateur ; le programme peut être modifié pour le rendre plus rapide, augmentant ainsi l'efficacité du traitement ; et le programme peut être réfactoré pour améliorer sa lisibilité, augmentant ainsi sa maintenabilité. On trouve dans la littérature qu'on l'appelle aussi « maintenance for the sake of maintenance » (Eick et al. 2001).

- **La maintenance logicielle adaptative :** a pour but de permettre au système de s'adapter aux évolutions de son environnement de données ou de traitement. Ce processus modifie le logiciel pour communiquer correctement avec un environnement changeant ou modifié. La maintenance adaptative comprend les modifications, les ajouts, les suppressions, les modifications, les extensions et les améliorations du système pour répondre aux besoins évolutifs de l'environnement dans lequel le système doit fonctionner (Tripathy et Naik 2014).

- **La maintenance logicielle évolutive** [2]

Les quatre différents types de maintenance logicielle sont chacun réalisés pour des raisons et des objectifs divers. Un logiciel donné peut être soumis à un, deux ou tous les types de maintenance au cours de sa durée de vie.

L'évolution logicielle est *la maintenance évolutive* qui est, comme déjà mentionné, une branche de la maintenance. C'est un changement continu d'un état inférieur, plus simple à un état supérieur ou meilleur (Arthur 1988). Elle consiste à faire évoluer une application, par exemple à la suite de demandes d'utilisateurs, pour modifier son comportement ou pour proposer de nouvelles fonctions.

### 1.2.2 Difficultés et coûts

Modifier un logiciel, peu importe comment et quand, est difficile et coûteux. Lorsque le logiciel et le matériel sont étroitement intégrés, le logiciel est également souvent considéré comme la partie la plus facile à modifier (Sneed 2004).

Les facteurs clés qui distinguent le développement et la maintenance et qui entraînent des coûts de maintenance plus élevés sont divisés en deux sous-catégories (GeeksforGeeks 2021) :

[2] En fait, la maintenance logicielle évolutive est ce qu'on appelle l'évolution logicielle qui sera défini dans le paragraphe suivant 1.2.1.

**Coûts techniques :**

Certains facteurs de coûts techniques courants sont (GeeksforGeeks 2021) :

- **Complexité du code :** la structure complexe de contrôle et de logique est difficile à comprendre et donc difficile à modifier.
- **Changements dans les langages de programmation :** Si le code à changer est écrit dans un langage qui n'est plus couramment utilisé, il est difficile à changer. Encore plus difficile si le nouveau code doit être écrit dans un langage différent.
- **Changements dans l'infrastructure logicielle :** Si le logiciel, le middleware ou les bibliothèques sous-jacentes ont changé, les programmeurs doivent comprendre comment le logiciel à modifier interagit avec eux, ce qui est difficile également.
- **Qualité du formatage et du style du code :** Dans le SDLC, si le logiciel exécutant l'environnement de support (y compris le matériel, le réseau et les logiciels système) change, le logiciel doit changer en conséquence pour s'adapter au nouvel environnement de support, donc l'adaptation pour maintenir la charge de travail sera énorme (Martin 2009).

**Coûts non-techniques :**

D'autres facteurs de coûts humains courants sont (Ren et al. 2011) :

- **Stabilité de l'équipe :** Les coûts de maintenance sont plus élevés si les développeurs d'origine ne sont pas disponibles.
- **Environnement externe :** Cela réfère à l'environnement logiciel, à savoir les règles métier, le flux de travail, les rapports, etc. Si un logiciel dépend trop de l'environnement externe, lorsque l'environnement externe change, alors le logiciel doit effectuer les modifications appropriées. Donc, la charge de travail de maintenance adaptative sera énorme (Ren et al. 2011).

La question qu'on se pose maintenant est : comment soulever ces difficultés et diminuer le coût de maintenance et d'évolution dans les logiciels ? La rétro-ingénierie répond a cette question en proposant des modèles et des techniques qui sont aujourd'hui un paradigme du Génie Logiciel reposant sur la création, la manipulation et l'utilisation intensive de modèles de natures diverses et variées. Nous verrons plus de détails dans le chapitre suivant.

## 1.3 Modélisation, modèles et méta-modèles

Dans cette section, nous donnons la définition générale de la modélisation ainsi que le modèle MDA (Model-Driven Architecture) de L'OMG (Object Management Group). Ensuite, nous allons parcourir les différents concepts fondamentaux qui sont relatifs à

la modélisation. Ce qui servira à comprendre le rétro-ingénierie en général et le MDRE en particulier (Model-driven Reverse Engineering) que nous détaillerons dans la section suivante.

### 1.3.1 Définition générale

La modélisation est un paradigme du génie logiciel reposant sur la création, la manipulation et l'utilisation intensive de modèles de nature diverses et variées. Même s'ils peuvent différer en termes de portée, les sous catégories fréquemment rencontrées (Model-Driven Engineering (Kent 2002 ; Schmidt 2006 ; Bézivin 2005a), Model-based Engineering (Estefan et al. 2007), Model-driven Development (Völter et al. 2013), etc.) peuvent être toutes regroupées sous le nom générique de modélisation (Modeling (Brambilla, Cabot et Wimmer 2017)). Plusieurs auteurs affirment que de nombreux avantages peuvent être obtenus en passant d'approches traditionnelles centrées sur le code à des approches basées sur des modèles ou pilotées par des modèles, augmentant ainsi le niveau d'abstraction considéré.

**Objectifs :**

Les principaux objectifs de la modélisation peuvent être résumés comme suit (Bruneliere 2018) :

- Exploiter les connaissances génériques existantes (par exemple sur les processus, les technologies et les données) afin de favoriser l'extensibilité et la réutilisabilité.
- Augmenter la compréhension du problème et/ou du système afin de mieux gérer la complexité, et ainsi faciliter les phases d'intégration, de maintenance ou d'évolution.
- Améliorer l'efficacité globale des différentes activités d'ingénierie logicielle, notamment en automatisant certaines tâches.

### 1.3.2 Model-Driven Architecture

Model Driven Architecture® (MDA®) est une approche de conception logicielle pour le développement des systèmes logiciels. Elle fournit un ensemble de lignes directrices pour la structuration des spécifications, qui sont exprimées sous forme de modèles. MDA a été lancé par l'Object Management Group (OMG) en 2001 (OMG 2001).

Les trois principaux objectifs du MDA sont la portabilité, l'interopérabilité et la réutilisabilité grâce à la séparation architecturale des préoccupations. D'après cela et comme indiqué dans (Miller et Mukerji 2003), MDA fournit une approche et permet de fournir des outils pour :

- Spécifier un système indépendamment de la plate-forme qui le prend en charge,
- Spécifier des plates-formes,

- Choisir une plate-forme particulière pour le système,
- Transformer la spécification du système en une plate-forme particulière.

L'initiative MDA est fortement liée à UML. C'est la première initiative pilotée par un modèle, voir (Miller et Mukerji 2003 ; Miller et Mukerji 2001).

MDA, et toutes les initiatives OMG, suivent le principe "tout est un modèle" comme indiqué dans (Bézivin 2005b). La sous-section suivante 1.3.3 décrit la notion de modèle et ses concepts associés.

### 1.3.3 Concepts fondamentaux

**Modèle :**

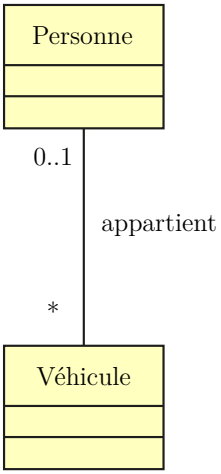
Plusieurs définitions de modèle (Kleppe et al. 2003 ; Mellor et al. 2004 ; Miller et Mukerji 2003) sont résumées dans (Favre 2005). Par exemple, dans le contexte de la norme UML, le terme modèle est défini comme suit :

"*Un modèle est une abstraction d'un système physique, avec un certain but.*"

D'autres auteurs définissent le terme en fonction d'une langue (Kleppe et al. 2003) :

"*Une description (d'une partie) d'un système écrite dans un langage bien défini.*"

L'OMG clarifie également dans (Miller et Mukerji 2003) qu'« un modèle est souvent présenté comme une combinaison de dessins et de textes. Le texte peut être dans un langage de modélisation ou dans un langage naturel.»




.

Fig. 1.1 : Exemple de modèle « L'entité Personne peut posséder des voitures » .

La figure 1.1 montre un exemple de modèle dans lequel deux entités différentes, Personne et Voiture, sont liées entre elles via la relation "appartient".

**Méta-modèle :**

La définition suivante est donnée dans (Favre 2005) :

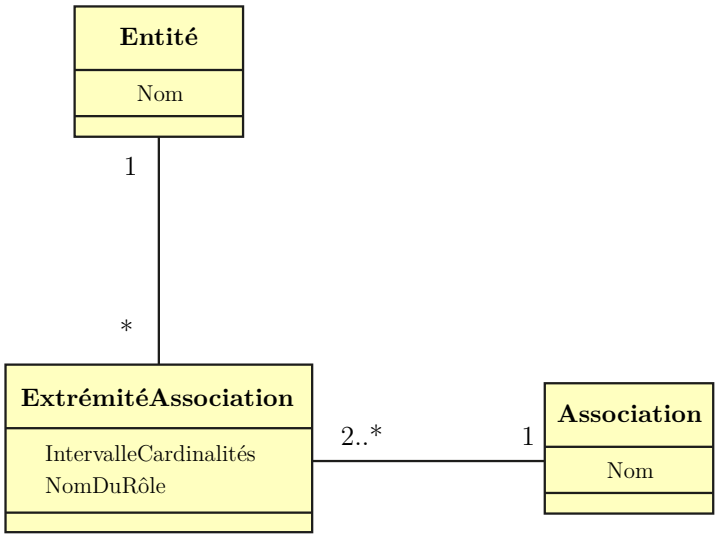
"*Un méta-modèle est un modèle d'un langage de modélisation.*"

Ainsi, les méta-modèles sont là pour décrire un langage de modélisation. Favre a déclaré dans (Favre 2005) que :

"*Un méta-modèle est un modèle d'un ensemble de modèles.*"

Similairement à une légende pour une carte, un méta-modèle définit les concepts et les relations possibles entre ces concepts qui permettent de définir un modèle.

La figure 1.2 montre un exemple de méta-modèle. Le modèle présenté précédemment à la figure 1.1 est conforme à ce méta-modèle.




.

Fig. 1.2 : Exemple d'un méta-modèle.

De la même manière que les modèles sont définis en termes de méta-modèles, les méta-modèles sont définis en termes de méta-méta-modèles, qui sont introduits dans la section suivante 1.3.3.

**Méta-méta-modèle :**

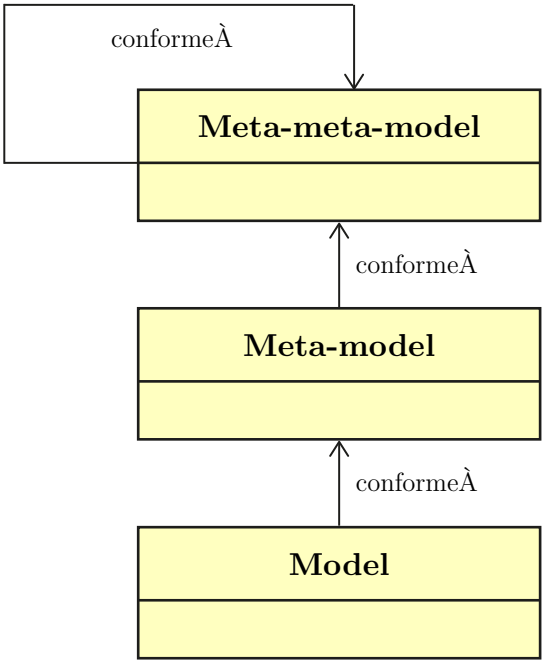
On peut reprendre la première définition de méta-modèle pour définir le concept de méta-méta-modèle comme « le modèle du langage de modélisation du méta-modèle », ou simplement :

"*Un méta-méta-modèle est un modèle d'un ensemble de méta-modèles.*"

Pour éviter une infinité de « méta-niveaux », les méta-méta-modèles sont dits auto-descriptifs ou réflexifs, c'est-à-dire qu'ils peuvent être définis récursivement par eux-

mêmes.

La figure 1.3 décrit graphiquement les différents niveaux d'abstraction et leurs relations telles que définies dans l'architecture pilotée par les modèles.




.

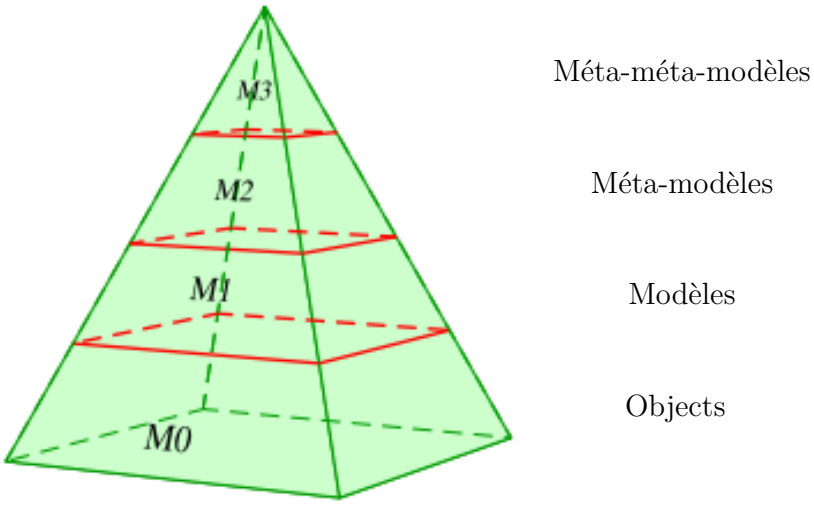
Fig. 1.3 : Model-Driven Architecture (GROUP et INRIA 2006).

**Modélisation à 4 niveaux :**

OMG désigne quatre niveaux différents nommés avec un M et le numéro du niveau. Ces niveaux sont résumés dans la figure 1.4. Ces quatre niveaux déterminent ce que l'on appelle la « pyramide de méta-modélisation à quatre couches » (Favre 2005). Chacun des niveaux comprend l'un des concepts précédents de modèle, métamodèle et méta-méta-modèle déjà présentés. De plus, la pyramide montre les instances des modèles ou des objets à la base (Jonkers, Stroucken et Vdovjak 2006).

La description des niveaux est la suivante :

- **M3 :** Couche contenant les méta-méta-modèles, décrits par eux-mêmes en raison de la propriété réfléchissante.
- **M2 :** Couche contenant les méta-modèles (par exemple, les éléments UML tels que les classificateurs, les attributs et les opérations, ou les définitions d'un langage de modélisation quelconque).
- **M1 :** Couche contenant les modèles (par exemple, une classe UML représentant des véhicules).
- **M0 :** Couche contenant les objets de l'application (par exemple, une instance de la classe Véhicule représentant la voiture immatriculée 12345).




.

FIG. 1.4 : Pyramide à quatre niveaux montrant chacun des niveaux OMG.

## 1.4 Rétro-ingénierie, techniques et solutions

Dans cette section, nous allons commencer par une vue globale en précisant la liaison entre la modélisation et la rétro-ingénierie. Ensuite, nous présenterons les différentes étapes et techniques de la rétro-ingénierie. Enfin, nous terminerons par présenter les solution spécifiques et génériques de ce domaine.

### 1.4.1 Model-driven Reverse Engineering

L'application de la modélisation à la rétro-ingénierie (i.e MDRE ou Model-Driven Reverse Engineering) est un domaine déjà assez ancien (RUGABER et STIREWALT 2004). Au début, les modèles étaient principalement utilisés pour la conception et spécifier les systèmes avant leur mise en œuvre (forward engineering). Au lieu de cela, la rétro-ingénierie vise à construire des modèles logiciels de plus haut niveau, plus abstrait, à partir du code source. La compréhension du programme est une activité qui tente de donner un sens aux informations produites par la rétro-ingénierie, en construisant des modèles d'architecture, de la structure et du comportement global du logiciel (MENS 2008). La compréhension du programme comprend également des activités telles que la modélisation des tâches, les problèmes d'interface utilisateur et bien d'autres.

La rétro-ingénierie est une étape d'un processus global qu'on appelle la réingénierie (TRIPATHY et NAIK 2014). L'objectif de la réingénierie est d'arriver à un nouveau système logiciel qui soit plus évolutif, et éventuellement doté de plus de fonctionnalités, que le système logiciel d'origine. Le processus de réingénierie est généralement (TRIPATHY et NAIK 2014) composé de trois activités :

1. Reverse Engineering
2. Re-design
3. Forward Engineering

Fortement fondé sur trois principes (respectivement) :

1. Abstraction
2. Altération
3. Raffinement

Une métaphore visuelle appelée "horseshoe" (Fer à cheval, en français), illustrée à la figure 1.5, a été développée par Kazman et al. (Jones 1984) pour décrire un processus de réingénierie architecturale en trois étapes. Trois segments distincts du "horseshoe" sont le côté gauche, la partie supérieure et le côté droit. Ces trois parties désignent les trois étapes du processus de réingénierie. La première étape, représentée à gauche, vise à extraire l'architecture du code source en utilisant le principe d'abstraction. La deuxième étape, représentée en haut, implique la transformation de l'architecture vers l'architecture cible en utilisant le principe d'altération. Enfin, la troisième étape, représentée sur le côté droit, implique la génération de la nouvelle architecture par raffinement. On peut regarder le fer à cheval de bas en haut pour remarquer comment la réingénierie progresse à différents niveaux d'abstraction : code source, modèle fonctionnel et conception architecturale.

.

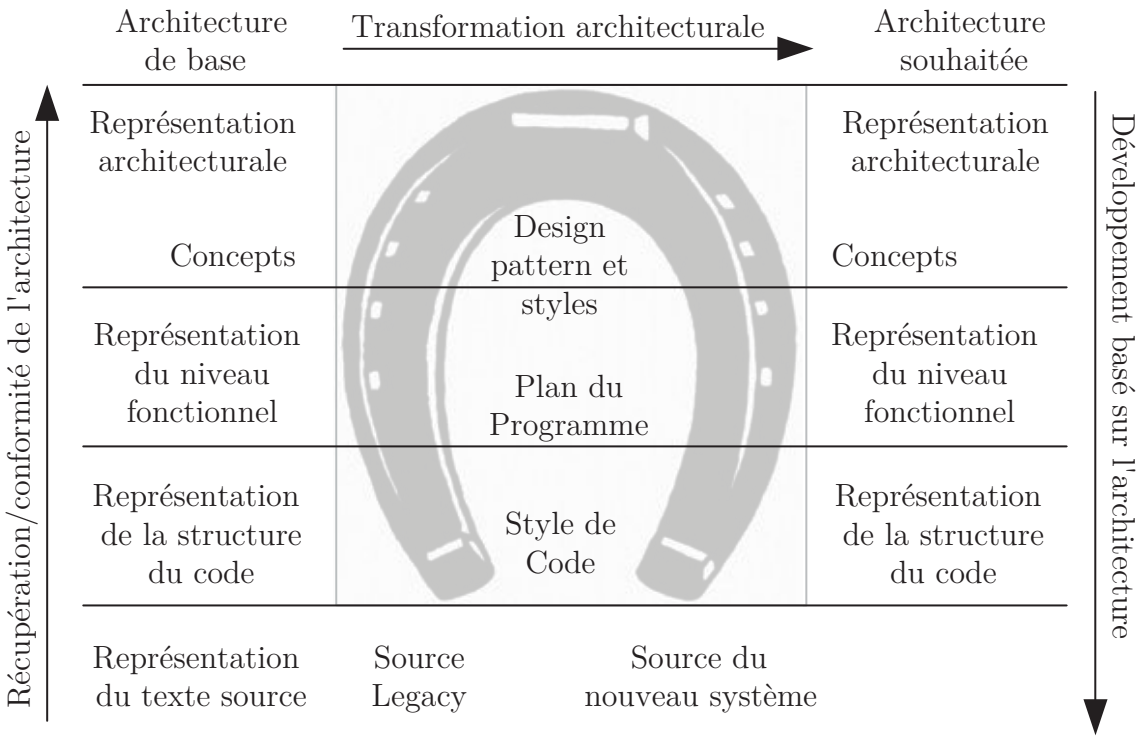



Fig. 1.5 : Le modèle de ré-ingénierie Horseshoe (Tripathy et Naik 2014) .

En résumé, la réingénierie implique : (i) la création d'une vision plus abstraite du système au moyen de certaines activités de la rétro-ingénierie ; (ii) la restructuration de la vision abstraite ; et (iii) la mise en œuvre du système sous une nouvelle forme au moyen d'activités d'ingénierie avancée.

### 1.4.2 Étapes clés de la rétro-ingénierie

Les six étapes clés de la rétro-ingénierie, telles que documentées dans la norme IEEE pour la maintenance logicielle (1219-1998 1998), sont :

- partitionner le code source en unités,
- décrire la signification de ces unités et identifier les unités fonctionnelles,
- créer les schémas d'entrée et de sortie des unités identifiées auparavant,
- décrire les unités connectées,
- décrire l'application du système,
- et créer une structure interne du système.

### 1.4.3 Techniques de rétro-ingénierie

La collecte d'informations et le fact-finding à partir du code source sont les clés de la modélisation (Tripathy et Naik 2014). Afin d'extraire des informations qui ne sont pas explicitement disponibles dans le code source, des techniques d'analyse automatisées sont utilisées.

Dans cette sous-section, nous allons présenter brièvement *l'analyse lexicale, l'analyse syntaxique, l'analyse de flux de contrôle et la visualisation.* Dans les sections suivantes, nous détaillerons les techniques relatives à notre projet *l'analyse de flux de données et le program slicing.*

**Analyse lexicale :**

L'analyse lexicale est la première phase d'un compilateur. C'est le processus de décomposition d'une séquence de caractères du code source en unités lexicales constitutives appelées tokens[3]. L'analyse lexicale nous fournit diverses représentations utiles sur les informations d'un programme (Tripathy et Naik 2014). L'information la plus largement utilisée, sur les programmes, est la liste de références croisées (Cross Reference Listing).

Un programme effectuant une analyse lexicale est appelé analyseur lexical et fait partie du compilateur d'un langage de programmation. En général, il utilise des règles décrivant les structures de programme lexical qui sont exprimées dans une notation mathématique appelée expressions régulières. Les analyseurs lexicaux modernes sont construits automatiquement à l'aide d'outils appelés générateurs d'analyseurs lexicaux, à savoir Lex et Flex (Fast Lexical analyzer) Battaglia, Savoia et Favaro 1998.

[3]Un token est une séquence de caractères qui peut être traitée comme une unité dans la grammaire des langages de programmation. Cela peut être un type (réel, entier, id ...), une ponctuation (if, void, ...) où un mot-clé (GfG 2021)

### Analyse syntaxique :

La deuxième forme, la plus complexe, d'analyse de programme automatisée est l'analyse syntaxique. Les compilateurs et autres outils tels que les interpréteurs déterminent les expressions, les instructions et les modules d'un programme. L'analyse syntaxique est effectuée par un parseur. A ce niveau, les propriétés linguistiques requises sont exprimées dans un formalisme mathématique appelé grammaires à contexte libre (Seki et al. 1991). Similairement aux analyseurs lexicales, les parseurs peuvent être construits automatiquement à partir d'une description des propriétés programmatiques d'un langage de programmation. YACC est l'un des outils d'analyse les plus couramment utilisés (Battaglia, Savoia et Favaro 1998). Dans cette analyse deux types de représentation sont généralement utilisés : *parse tree* [4] *et AST (abstract syntax tree).*

### Analyse de flux de contrôle (CFA) :

Après avoir déterminé la structure d'un programme, une analyse de flux de contrôle (CFA) peut être effectuée sur celui-ci. Toute analyse statique globale des expressions et la relations entre les données dans un programme nécessite une connaissance du flux de contrôle du programme Allen 1970. Étant donné que l'une des principales raisons de faire une telle analyse dans un compilateur est de produire des programmes optimisés, l'analyse de flux de contrôle a été intégrée dans de nombreux compilateurs et a été décrite dans plusieurs articles Cocke 1969 ; Prosser 1959 ; Lowry 1969.

Les deux types de CFA sont l'analyse intra-procédurale et l'analyse inter-procédurale Hecht 1977. Le premier montre l'ordre dans lequel les instructions sont exécutées dans un sous-programme, tandis que le second montre la relation d'appel entre les unités de programme.

### Visualisation :

La visualisation logicielle est une stratégie utile pour permettre à un utilisateur de mieux comprendre les systèmes logiciels Tripathy et Naik 2014. La visualisation consiste à représenter un système logiciel au moyen d'un objet visuel pour essayer de faire apparaître certaines particularités.

De nombreux outils de rétro-ingénierie présentent leurs résultats dans des notations graphiques. Ces schémas graphiques sont initialement développés comme conception de systèmes logiciels avant même de les programmer. Par exemple, dans l'orienté objet, il existe de nombreux outils qui, compte tenu du code source du programme, génèrent des diagrammes de classes UML et ainsi d'autres types de diagrammes UML, par exemple Borland Together, Rational Rose, ESS-Model, BlueJ et Fujaba Diehl 2005.

Généralement, dans la rétro-ingénierie, on opte pour faire des analyses sur les modèles obtenus à partir du code source pour finalement visualiser les résultats de ces analyses comme illustré dans le pipeline de la Figure 1.6.

[4] La définition du parse tree est donnée au niveau de l'annexe.

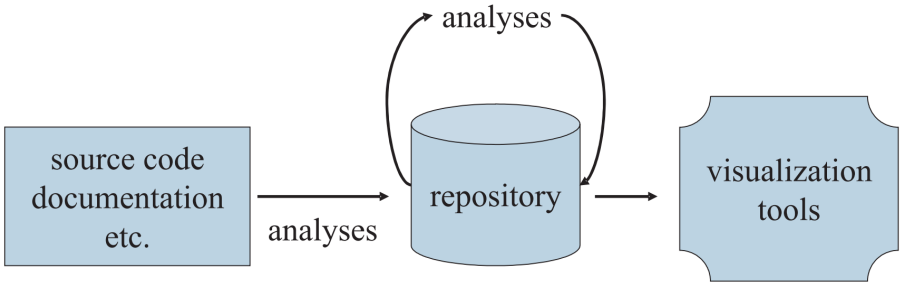



Fig. 1.6 : Pipeline typique des outils de rétro-ingénierie Diehl 2005.

## 1.5 Analyse de flux de données (DFA)

Bien que l'analyse de flux de contrôle (CFA) soit utile, de nombreuses questions ne peuvent pas être résolues au moyen de la CFA. Par exemple : après l'exécution d'une expression d'affectation, quelles sont les parties du code qui peuvent être impactées ? (Tripathy et Naik 2014). Afin d'avoir des réponses, une compréhension des définitions (def) des variables et des références (uses) des variables est nécessaire. En réalité, si une variable apparaît sur le côté gauche d'une instruction d'affectation, alors la variable est dite définie. Au contraire, si une variable apparaît sur le côté droit d'une instruction d'affectation, on dit alors qu'elle est référencée dans cette instruction.

L'analyse des flux de données (DFA) est principalement une analyse statique. Elle s'intéresse à la façon dont les valeurs des variables définies circulent et sont utilisées dans un programme (Osterweil et Fosdick 1976). CFA peut détecter la possibilité de boucles, tandis que DFA peut déterminer les anomalies de flux de données (Fosdick et Osterweil 1976). Un exemple d'anomalie de flux de données est qu'une variable indéfinie est référencée. Un autre exemple d'anomalie de flux de données est qu'une variable est définie dans deux endroits différents sans être référencée entre eux. DFA aussi permet d'identifier : le code qui ne peut jamais s'exécuter (code mort), les variables non-définies avant d'être utilisées et les instructions qui sont altérées lorsqu'un bug est corrigé.

L'analyse de flux de données calcule de manière statique des informations sur les données (c'est-à-dire les utilisations et les définitions des données) pour chaque point du code source en cours d'analyse. Ces informations doivent être une approximation sûre des propriétés souhaitées du comportement d'exécution du programme lors de chaque exécution possible de ce point de programme sur toutes les entrées possibles (Khedker, Sanyal et Karkare 2017). Par conséquent, DFA se caractérise par les éléments suivants :

- **Applications diverses :** la DFA peut être utilisée pour :
  - Déterminer la validité sémantique d'un programme (c'est-à-dire correction de type basée sur l'inférence, interdiction d'utiliser des variables non initialisées, etc.),
  - Comprendre le comportement d'un programme pour le débogage, la maintenance, la vérification ou les tests,
  - Transformer un programme. Il s'agit de l'application classique de l'analyse des flux de données et la DFA a été initialement conçue dans ce contexte.

- **Scope du programme :** L'analyse des flux de données peut être effectuée à presque tous les scope level[5]. Traditionnellement (Khedker, Sanyal et Karkare 2017), les termes suivants ont été associés à l'analyse des flux de données pour différentes scope :
  - À travers les instructions mais confiné à une séquence d'instructions (au sein d'un bloc local) : Analyse du flux de données local.
  - À travers les blocs de base mais confinés à une fonction/procédure : analyse globale (intra-procédurale) des flux de données.
  - À travers les fonctions/procédures : analyse de flux de données inter-procédurale.
- **Les représentations :** l'analyse de flux de donnée peut utiliser plusieurs représentations internes possibles comme : les arbres de syntaxe abstraite (AST), les graphes acycliques dirigés (DAG), les graphes de flux de contrôle (CFG), les graphes de flux de programme (PFG), les multigraphes d'appel (CG), les graphes de dépendance de programme (PDG), les affectations uniques statiques (SSA), etc. Les représentations les plus courantes pour l'analyse globale des flux de données sont les CFG, les PFG, les SSA et les PDG, tandis que les analyses de flux de données inté-procédurales utilisent une combinaison de CG (et de CFG ou de PFG).

## 1.6 Slicing d'un programme

Le program slicing est l'ensemble des instruction d'un programme qui donnent valeur et impactent une variable donnée à une ligne de code donnée (Weiser 1984).

### 1.6.1 La définition originale

La définition du program slicing varie légèrement d'un article à l'autre. Ces différences peuvent être expliquées généralement en termes de critère de slicing. Un critère de slicing indique les conditions du calcul du slice (le résultat du slicing), telles que à quel point du programme et pour quelle(s) variable(s) un slice est requis (Weiser 1984). Ce critère a été classé, (Xu et al. 2005), en 4 types :

- Le critère de slicing statique $C = (p, v)$ nécessite le calcul d'un slice statique par rapport à un point de programme $p$ et une variable $v$, qui n'est pas nécessairement utilisée ou définie en $p$.
- Le critère de slicing statique $C = (p, v)$ nécessite le calcul d'un slice statique par rapport à un point de programme $p$ et une variable $v$, qui est utilisée ou définie en $p$.
- Le critère de slicing statique orienté $C = (p, v, o)$ nécessite le calcul d'un slice statique par rapport à un point de programme $p$ et une variable $v$ dans une orientation

[5]Le scope level d'un programme est une frontière bien définie, qui englobe toutes les activités qui sont faites pour développer et livrer ce programme. d'un programme

$o$ (vers le haut ou bien vers le bas), qui n'est pas nécessairement utilisée ou définie en $p$.

- Le critère de slicing dynamique $C = (x, p, v)$ nécessite le calcul d'une slice dynamique d'un programme exécuté sur l'entrée x par rapport à un point d'exécution p et une variable $v$, qui n'est pas nécessairement utilisée ou définie en $p$.

.

**Algorithm :** Class to apply Program Slicing

```
1  class MyClassA
2     public method integer sumOfEvnNbrs()
3        var n, sum, i: integer
4        sum ← 0
5        i ← 0
6        read(n)
7        while (n > 0) do
8           sum ← sum + i
9           i ← i + 2
10          n ← n - 1
11       write(sum)
12    end method
13 end class
```

**Résultat:** Program Slicing

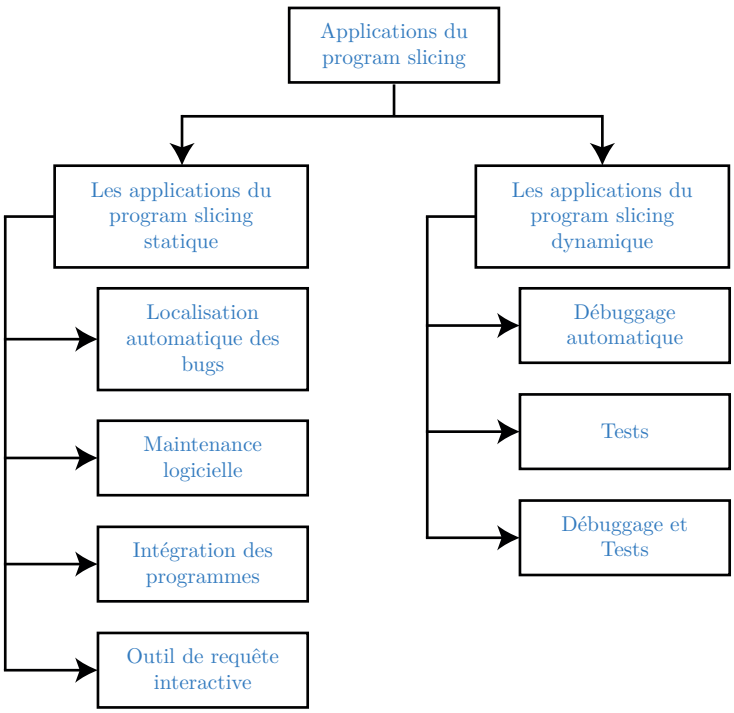
```
3  var n: integer
6  read(n)
7  while (n > 0) do
10    n ← n - 1
```

FIG. 1.7 : Exemple du program slicing avec le crièitre $C = (7, n, up)$.Le résultat du slicing est l'ensemble des instructions : $Res = \{3, 6, 7, 10\}$

### 1.6.2 Les applications du program slicing

Le type de slicing dépend du but de l'application de cette technique et l'information qu'on dispose. Le slice statique, bien qu'imprécis, contient des informations beaucoup plus larges que le slice dynamique. Le slice statique contient une quantité d'information énorme car c'est la collection de tous les calculs possibles de valeurs pour la même occurrence de variable, et ces calculs peuvent se chevaucher (KAMKAR 1995) et être successive. D'autre part, le slice dynamique isole le calcul unique d'une valeur d'une occurrence de variables pour une entrée donnée (KAMKAR 1995).

Dans la littérature, on retrouve principalement deux applications du program slicing. La figure 1.8, illustre les différentes applications selon le type du slice (KAMKAR 1995) :




.

FIG. 1.8 : Les application du program slicing (KAMKAR 1995).

### 1.6.3 DFA en utilisant program slicing

Après avoir défini les deux concepts, on peut dire que le slicing et la DFA sont étroitement liés dans le contexte de la maintenance logicielle, plus spécifiquement dans la définition et l'utilisation des données. En effet, le résultat du slicing, qui est l'ensemble des instructions nécessaires pour un endroit précis dans le programme, est intéressant pour représenter le flux des données et savoir comment et où les valeurs des variables circulent dans notre programme.

Lors de notre piste de recherche, et après avoir effectuer cet état d'art, nous avons prévu d'utiliser la technique du slicing orienté dans le but de comprendre et analyser le flux de données dans un programme.

## 1.7 Conclusion

Dans le présent chapitre nous avons établi un état de l'art des concepts liés à notre projet. Dans la première partie, nous avons défini la maintenance et l'évolution. Dans la deuxième partie du chapitre, nous avons présenté la modélisation et ses concepts fondamentaux. Après, nous avons évoqué la liaison entre le domaine de la rétro-ingénierie et la modélisation (MDRE). Dans la fin du chapitre, nous avons élaboré les techniques de la rétro-ingénierie, en détaillant le program slicing et l'analyse des flux de données et la liaison entre eux dans le but de notre projet.

# Chapitre 2

# Etat de l'existant

## 2.1 Introduction

La première étape dans la résolution de n'importe quel problème est la compréhension du contexte. Ce chapitre a pour but de positionner le projet dans son environnement organisationnel et contextuel. Au début du chapitre, nous présentons l'organisme d'accueil, après nous décrivons l'équipe où ce projet a été mis en place et ainsi l'environnement technique du travail.

## 2.2 Présentation de l'organisme d'accueil

### 2.2.1 L'institut Inria

#### Présentation

Inria, ou l'Institut national de recherche en informatique et en automatique, est un établissement public à caractère scientifique et technologique français spécialisé en mathématiques appliquées et informatique, placé sous la double tutelle du ministère de l'Enseignement supérieur, de la Recherche et de l'Innovation et du ministère de l'Économie et des Finances (Foundation 1970). L'institut fournit un transfert de technologie fort et porte une attention particulière pour la formation par la recherche, la diffusion du développement scientifique et technique, l'expertise et la participation à des programmes internationaux.

#### Composition

Inria accueille 4 200 personnes dans ses dix centres de recherche, figure 2.1 situés à Rocquencourt, Rennes, Sophia Antipolis, Grenoble, Nancy, Bordeaux, Lille, Saclay, Paris et Chile. 2 600 membres du personnel sont des scientifiques de l'Inria et l'autre partie sont des partenaires d'organisations (CNRS, universités, … etc). Au total, il y a plus de 210 équipes-projets qui travaillent sur des projets divers de recherche.


Fig. 2.1 : Les centres de recherches Inria.

### 2.2.2 Inria Lille - Nord Europe (LNE)

Le contre Inria Lille - Nord Europe, dirigé par Mireille Régnier, est basé sur les axes scientifiques suivants :

- Science des données ;
- Génie logiciel ;
- Systèmes cyberphysiques.

Le centre compte 15 équipes-projets chacune son domaine et thème de recherche comme illustré dans la figure suivante 2.2.

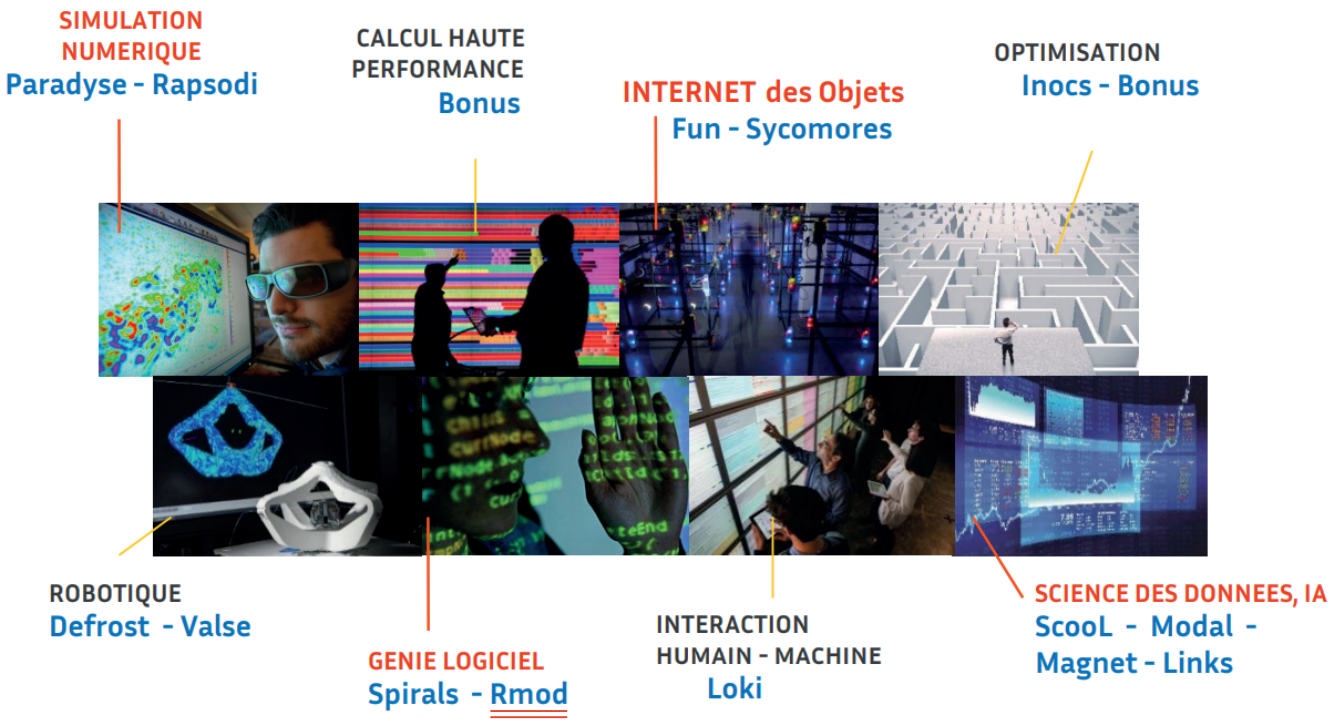



FIG. 2.2 : Les équipe projet Inria LNE et le positionnement de l'équipe RMod.

### 2.2.3 L'équipe RMoD

#### Présentation

Le but de l'équipe RMoD est de contribuer à re-modulariser des systèmes logiciels existants. Cet objectif fait suite à deux lignes complémentaires, la réingénierie et la définition de nouvelles constructions pour les langages de programmation.

Pour aider à la réingénierie, de nouvelles méthodes d'analyses sont proposées afin de comprendre les larges systèmes (re-modulariser les logiciels, expression de métriques, visualisations adaptées, ... etc). Dans le contexte de la réingénierie, des méta-modèles et des outils ont été développés pour atteindre l'objectif de l'équipe en utilisant Moose. Moose est une méta-plateforme utilisée pour l'analyse et la maintenance logicielle, nous verrons plus de détails dans la section 2.3

L'équipe travaille également sur un noyau efficace et sécurisé pour Pharo, un environnement de développement intégré pour Smalltalk, utilisé et maintenu par RMoD.

## 2.3 Moose

Dans cette section, nous allons présenter Moose (ANQUETIL et al. 2020), qui est une plateforme générique pour l'analyse et la maintenance des logiciels. Nous commençons par définir l'environnement Moose et ses objectifs. Ensuite, nous verrons l'architecture extensible utilisée par Moose. Nous terminons par voir les différents outils réutilisables par cette plateforme.

### 2.3.1 Définition

Moose est un environnement indépendant du langage pour la rétro-ingénierie et la réingénierie des systèmes logiciels complexes. Moose fournit un ensemble de services comprenant un méta-modèle commun, une évaluation et une visualisation des métriques, un référentiel de modèles et une prise en charge générique de l'interface graphique pour l'interrogation, la navigation et le regroupement (Nierstrasz, Ducasse et Gundefinedrba 2005).

Moose est né dans le cadre de Famoos, un projet européen[1] dont le but était d'accompagner l'évolution des logiciels orientés objet de première génération vers des frameworks orientés objet (Nierstrasz, Ducasse et Gundefinedrba 2005).

### 2.3.2 Objectifs

Il y a différentes façons d'appréhender la rétro-ingénierie :

1. Soit on se focalise sur un langage en particulier et on devient expert de ce langage en développant des outils ;
2. Soit on ne se focalise pas sur un seul langage et on a une équipe suffisamment grande pour être expert de ces langages en développant des outils dédiés ;
3. Soit on ne se focalise pas sur un seul langage mais on essaie de réutiliser des outils génériques facilement spécialisables pour être expert de tous ces langages sans avoir une très grande équipe.

C'est la dernière stratégie qui régit la méta-plateforme Moose. Pour cela, il a fallu développer des MM spécifiques pour chacun des langages mais communs pour adapter facilement des outils existants.

### 2.3.3 Architecture et Outils

Dans cette section, nous allons présenter l'architecture de Moose qui permet aux ingénieurs de concevoir des outils spécifiques de rétro-ingénierie en utilisant son infrastructure. Ensuite, on va citer quelques outils génériques et configurables de Moose.

#### Architecture

Moose est architecturé sur les principes suivants (Anquetil et al. 2020) afin de garantir la collaboration et l'ajout de nouveaux outils d'une manière flexible :

- Les outils font partie de l'environnement ModMoose [2] qui fait office de maître et centralise les données,

[1] ESPRIT Project 21975: "Framework-based Approach for Mastering Object-Oriented Software Evolution". Sept. 1996-Sept. 1999.

[2] ModMoose fait référence à une nouvelle version de moose 2020 (Anquetil et al. 2020)

- Les outils communiquent via des bus, ils « lisent » des entités modèles sur leur bus et « réécrivent » des entités sur leur bus (voir figure 2.3),

- Les outils se concentrent sur une seule tâche : par exemple, le navigateur de requêtes fonctionne sur un ensemble d'entités de modèle et produit un autre ensemble d'entités.

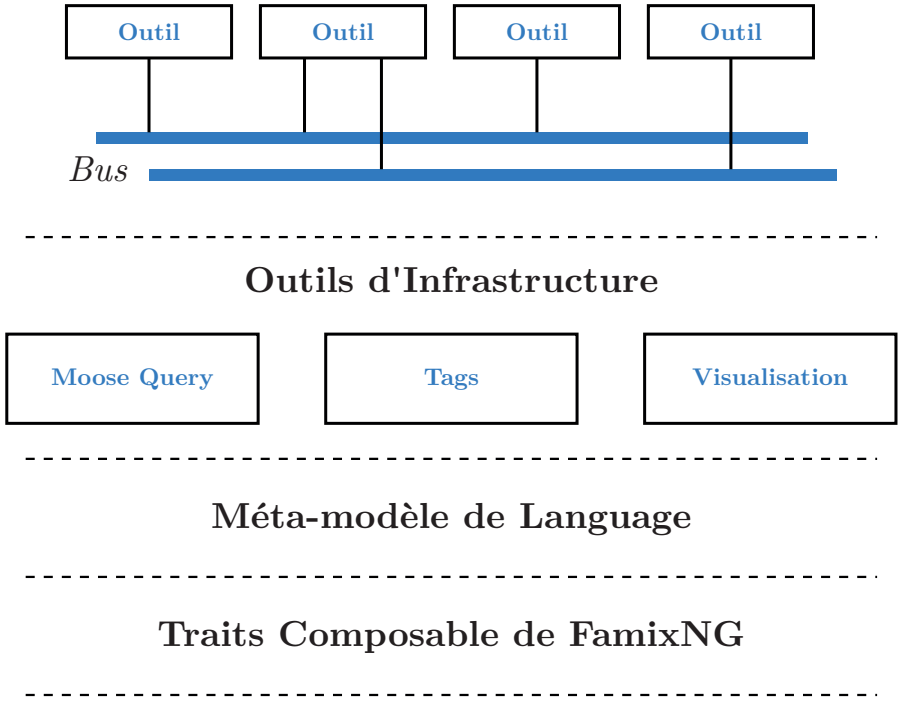



Fig. 2.3 : Nouvelle architecture de Moose (Anquetil et al. 2020).

**Outils d'infrastructure :** Les outils d'infrastructure de Moose sont :

- **Moose Query :** est une API pour faire des requêtes et naviguer sur n'importe quel Famix méta-modèle.

- **Tags :** sont des étiquettes attachées par l'utilisateur à toute entité, soit de manière interactive, soit à la suite de requêtes (Govin et al. 2017). Les balises enrichissent les modèles avec des informations supplémentaires.

- **Engine de Visualisation :** utilise Roassal (Bergel 2016), un moteur de visualisation générique, pour scénariser des graphiques ou des diagrammes interactifs. Roassal (Bergel 2016) est principalement utilisé pour afficher des entités logicielles sous différentes formes ou couleurs et leurs relations. Des exemples possibles consistent à afficher les classes d'un package sous la forme d'un diagramme de classes UML ou d'une matrice de structure de dépendance13 (Anquetil et al. 2020).

### 2.3.4 Outils réutilisables de Moose

Comme l'architecture l'indique 2.3.3, il existe de nombreux outils à réutiliser qui sont déja implémentés, dont on cite :

- **Model Browser :** un outil qui permet d'importer, créer de nouveau modèls pour un système donné. Il est considéré comme le point d'entrée pour commencer à travailler sur l'environnement (Anquetil et al. 2020).
- **Entity Inspector :** répertorie toutes les propriétés d'une entité sélectionnée et leurs valeurs (telles que les mesures, les tags et autres). Il permet de naviguer dans le modèle lorsque la valeur d'une propriété est une autre entité (par exemple, la propriété methods d'une classe) (Anquetil et al. 2020).
- **Query Browser :** c'est l'interface graphique de Moose Query (section 2.3.3).
- **Dependency Graph Browser :** affiche sous forme de graphique toutes les dépendances entrantes et sortantes directes d'un groupe d'entités (Anquetil et al. 2020).

### 2.3.5 Smalltalk / Pharo

Smalltalk est un langage et un environnement orienté objet entièrement réflexif. Il a inspiré de nombreux langages orientés objet au fil des ans et a évolué vers de nombreuses variantes. Pharo est l'une de ces variantes issues d'une autre implémentation de Smalltalk appelée Squeak.


Fig. 2.4 : Logo de Pharo.

**Aperçu :**

Pharo est un langage de programmation orienté objet et un environnement qui possède les propriétés suivantes :

- Réflexive[3] : car c'est un langage qui a la capacité d'un programme à examiner, et éventuellement à modifier, ses propres structures internes de haut niveau lors de son exécution.

[3]Dans le contexte des langages de programmation, la réflexion est basé sur la réification qui est le processus par lequel un programme utilisateur ou tout aspect d'un langage de programmation implicite dans le programme traduit et l'environnement d'exécution sont exprimés dans le langage lui-même (Foundation 2022b).

- Pureté : c'est est un langage orienté objet pur : tout est représenté comme un objet, même les types et classes primitifs.

- Basé sur une image : contrairement à la plupart des langages de programmation, il ne s'appuie pas sur une structure de fichiers sous-jacente pour stocker le code. Au lieu de cela, il utilise une image, c'est-à-dire un snapshot complet de tous les objets qui existent dans le système. Une image est donc un système vivant que l'on peut modifier progressivement afin de développer un nouveau logiciel.

- Late binding : comme la plupart des langages dynamiques orientés objet, il utilise le passage de messages et la liaison tardive (parfois appelée duck-typing).

La syntaxe de transmission de messages diffère de la plus connue de Java. Smalltalk utilise le passage de messages, pas l'invocation de méthode. L'exemple suivant montre les différences.

Etant donné la ligne de code suivante :

```
foo bar: baz.
```

#bar : est un sélecteur, foo est le récepteur d'un message appelé #bar :

baz est un argument.

Lorsque la ligne est exécutée, foo reçoit le message #bar : avec l'argument baz.

La syntaxe correspondante de

```
foo bar: baz.
```

dans Java est

```
foo.bar(baz);
```

En Java, le système d'exécution déterminerait le type réel de foo, trouve la méthode la plus appropriée et l'exécuterait.

Les choses se ressemblent presque dans Smalltalk. Lorsque nous envoyons un message à un objet, il recherche dans son dictionnaire de méthodes une méthode dont le nom correspond à celui du sélecteur du message. S'il n'en trouve pas, il recherche dans le dictionnaire de méthodes de sa superclasse, et ainsi de suite.

## 2.4 Famix & FAST

Pour aider à l'analyse de code, il existe deux métamodèles dans Moose, Famix et FAST qui dépendent du niveau de détail choisi. Dans cette section, nous allons présenter ces deux méta-modèles.

### 2.4.1 Famix

Famix est un métamodèle qui permet de décrire des programmes écrits dans différents langages. Il s'arrête au niveau de l'appel de méthode et ne descend pas au niveau de l'instruction. Ainsi, il permet d'avoir des informations structurelles.

Pour un programme objet, Famix nous permet de connaître les packages, les classes avec leurs attributs et leurs méthodes. Pour chaque méthode, on peut connaître quels attributs elle accède, quelles classes elle réfère et quelles méthodes elle appelle. Par ailleurs, il nous donne toute la hiérarchie des classes.

Pour un programme impératif, ce méta-modèle nous permet de connaître par exemple les fonctions, les variables, globales ou locales auxquelles elles accèdent ou les autres fonctions qu'elles appellent.

**Aperçu :**

Dans la plupart des cas, nous obtenons suffisamment d'informations si nous avons les entités de base qui modélisent un système orienté objet. Il s'agit de namespace, package, classe, méthode, attribut et des relations entre eux, à savoir l'héritage, les accès et l'invocation. La figure 2.5 ci-dessous donne un aperçu de ces classes. *Ce dernier n'est qu'une représentation générale qui ne tient pas compte de l'implémentation technique.*

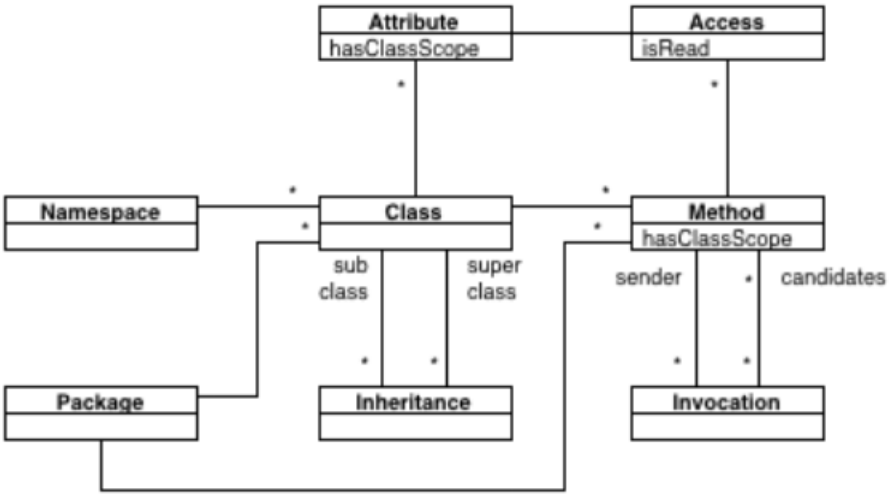



Fig. 2.5 : Un aperçu général de Famix (Stéphane Ducasse 2016).

D'après l'aperçu du méta-modèle, la figure 2.5, nous avons :

- le modèle de l'héritage est une relation d'association entre : la sous-classe et la super-classe,
- L'accès désigne une relation entre une entité de comportement (une méthode dans un système orienté objet) et une entité structurelle (souvent un attribut),
- L'invocation est une relation entre une source et une ou plusieurs entités comportementales cibles (souvent des méthodes),

- les classes ont des méthodes et les méthodes appartiennent aux classes.

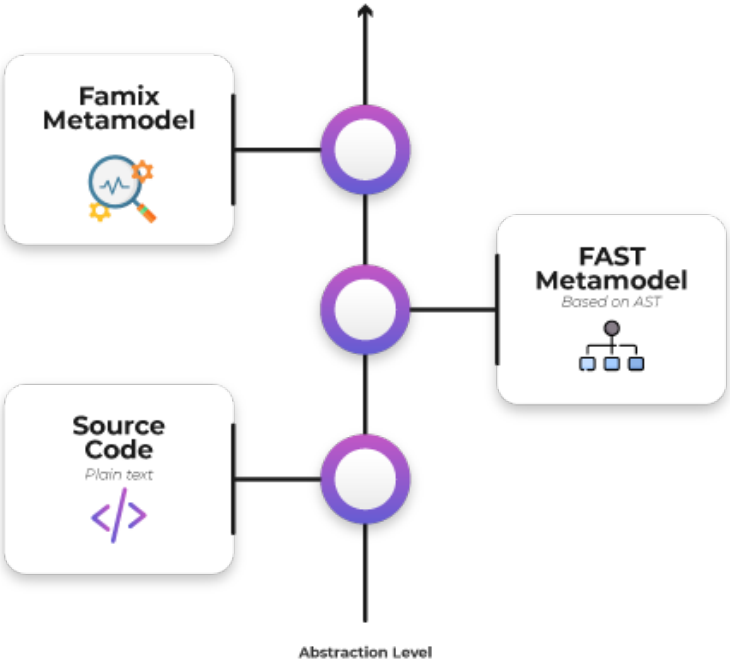
**Caractéristiques :**

Le méta-modèle Famix fournit une API que nous pouvons l'utiliser dans les requêtes et la navigation dans les modèles.

La nouvelle conception de Famix supporte la réutilisation des concepts de langages de programmation, et cette conception est basée sur les stateful traits (Tesone et al. 2020).

### 2.4.2 FAST

FAST signifie Famix AST. Contrairement à Famix qui représente un système logiciel dans un niveau d'abstraction élevé, FAST utilise une représentation de bas niveau : l'AST. Ces niveaux d'abstractions sont résumés dans la figure 2.6




.

Fig. 2.6 : Le niveau d'abstraction des différents méta-modèles.

**Conception :**

FAST définit un ensemble de traits pouvant être utilisés pour créer de nouveaux méta-modèles compatibles avec les outils de Moose. La conception du méta-modèle FAST, illustré dans la figure 3.2, est similaire aux traits prédéfinis de Famix.

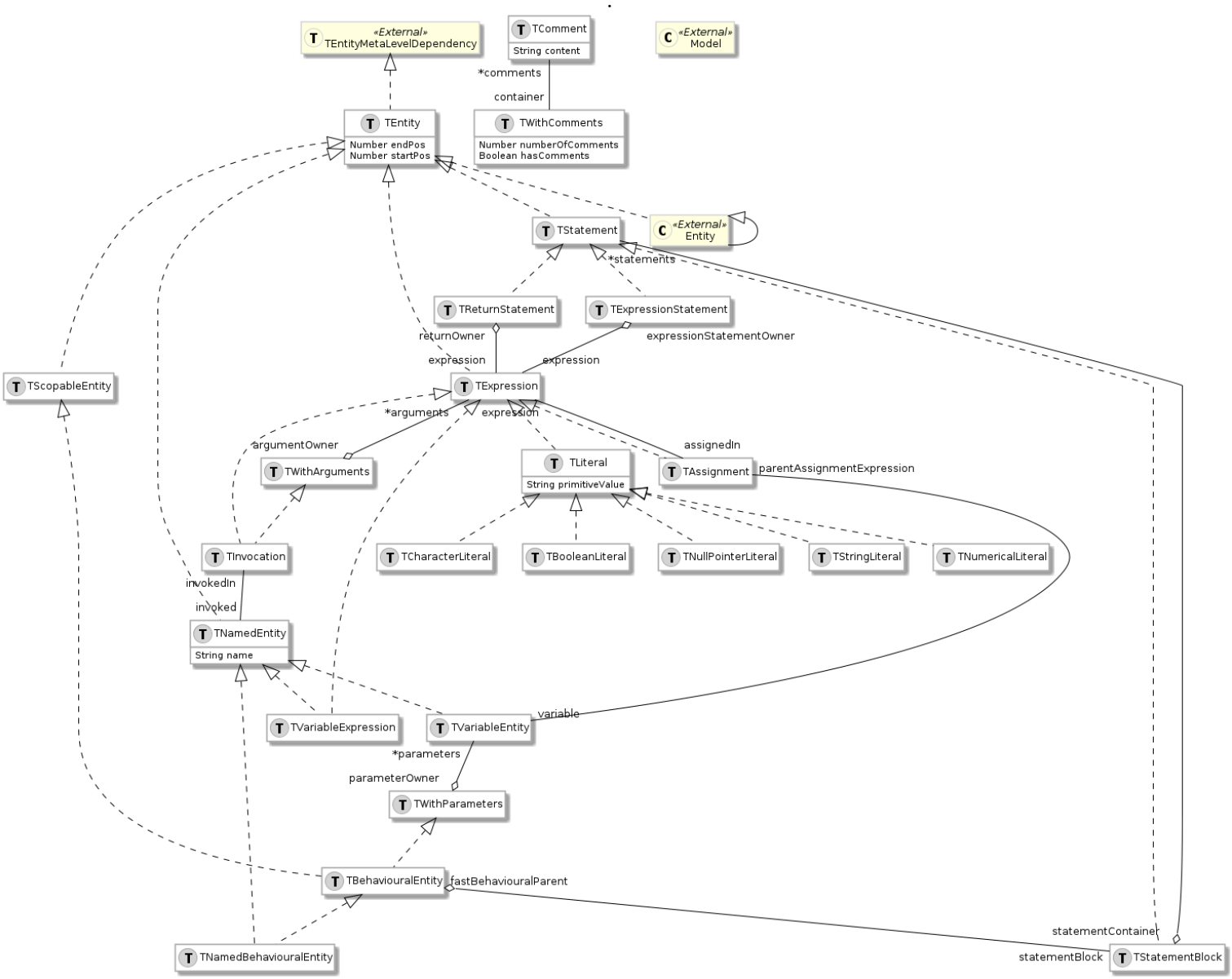



Fig. 2.7 : FAST meta-model.

D'après le diagramme de classe ci-dessus, les entités de l'AST ont comme attributs startPos et endPos car il utilisent #TEntity. Ces deux attributs définissent la position de l'entité dans le code source. A titre d'exemple, toutes les expressions sont des entités qui ont un pointeur de début et de fin.

Les expressions #TExpression peuvent être une affectation #TAssignement, un littéral (valeur fixe) #TLiteral, une déclaration de variable #TVariableExpression ou une invocation #TInvocation.

### 2.4.3 Carrefour

Carrefour représente un lien à double sens entre Famix et FAST, il permet de naviguer sur l'AST et en même temps de revenir aux éléments de Famix en cas de besoin.

**Etude de cas :**

Pour avoir une vision complète des méta-modèles décrits ci-dessus, nous étudions le cas de d'un simple exemple (code java), figure 2.8, en mettant l'accent sur chaque méta-modèle.

.

```
class MyClass {

    public int methodAB(int a, int b) {
        if (a>b) {
            a = 2;
        } else {
            b = 1;
        }
        return b;
    }
}
```

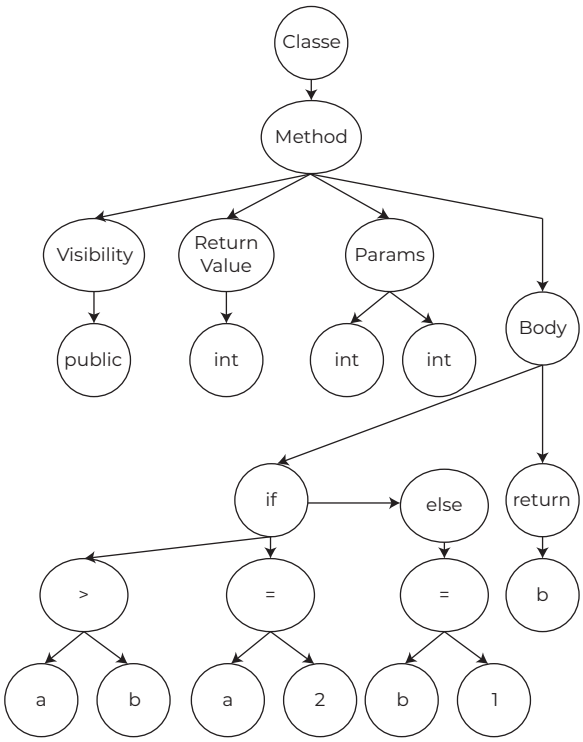



Fig. 2.8 : Un exemple d'un code Java et l'AST correspendant à ce code.

Le méta-modèle carrefour consiste à créer des liens entre les différents entités de Famix et FAST, comme illustré dans 2.9 :

- Le modèle Famix est représenté par les entitè à gauche
- Le modèle FAST est représenté par les entitè à droite
- Pour transiter du modèle Famix à FAST d'une entité variable, nous y envoyer le message self fastAccesses
- Pour transiter du modèle FAST à Famix d'une entité variable, nous y envoyer le message self famixVariable

*NB : La transition entre les deux modèles Famix et FAST sur les entités de type variables se fait par fastAccesses et famixVariable. Pour les entités de type méthode nous utilisons les messages famixMethod et fast*

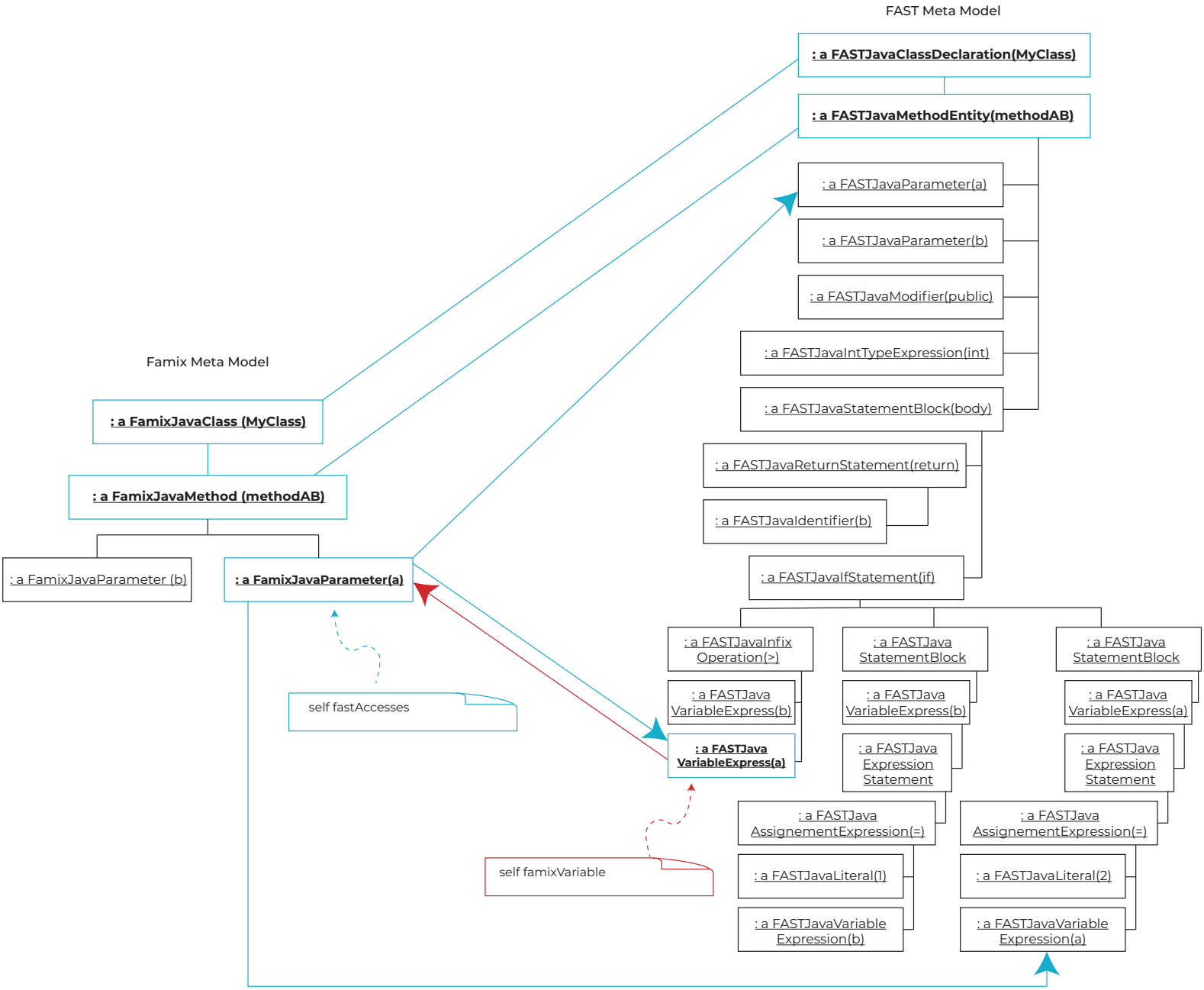



FIG. 2.9 : Un exemple de liaisons entre Famix et FAST en utilisant Carrefour.

## 2.5 Conclusion

La compréhension du contexte organisationnel et de l'environnement de travail est une étape primordiale dans n'importe quel projet. Nous avons présenté dans ce chapitre l'organisme d'accueil, le centre et ainsi l'équipe où nous avons effectué notre stage. Nous avons également détaillé la méta-plateforme Moose et ainsi ses méta-modèles.

À la fin de ce chapitre, nous pouvons dire que Moose dispose d'une architecture qui ne cesse d'évoluer en termes de volume de données, de langage de programmation et de niveau d'abstraction. D'où le besoin de développer et maintenir ses outils est primordial.

# Chapitre 3

# Expression des Besoins

## 3.1 Introduction

Dans n'importe quel domaine, un projet dont les objectifs ne sont pas clairement définis a de fortes chances d'échouer. Et les projets en informatique n'échappent pas à la règle. Il est vital au bon déroulement du projet de bien recenser et formaliser les besoins et les tâches à mettre en place. Plusieurs réunions de synchronisation ont été effectuées et m'ont permis de comprendre la finalité du projet et dégager les spécifications et les fonctionnalités de la solution.

Ce chapitre relate les spécifications modélisées et représentées par des diagrammes UML (Unified Modeling Language).

## 3.2 Modèle de Spécifications

Les fonctionnalités sont exprimées par les spécifications fonctionnelles. Elles sont listées dans le tableau suivant 3.1 où la priorité varie entre 1 et 4 avec 1 la priorité la plus grande. Une spécification critique est celle qui doit être implémentée sinon le système ne pourra pas fonctionner. Tandis qu'une spécification importante peut être omise mais ceci impacte négativement l'utilisabilité du système. En revanche, une spécification utile peut être omise sans impacter le système. Quant à la stabilité de la spécification, ce n'est autre que la probabilité de son changement avec le temps.

| ID | Spécification | Criticisme | Priorité | Flexibilité |
|---|---|---|---|---|
| En tant qu'acteur utilisateur, le système doit me permettre de : | | | | |
| 1 | Importer le modèle | Utile | 4 | Stable |
| 2 | Visualiser l'AST du code Java/Pharo | Important | 3 | Stable |
| 3 | Faire le slicing d'un programme | Critique | 1 | Instable |
| 4 | Représenter le flot de donnée d'un programme | Important | 1 | Stable |

Tab. 3.1 : Les spécifications fonctionnelles.

## 3.3 Modèle des cas d'utilisation

Nous modélisons les spécifications recensées à l'aide du diagramme de cas d'utilisation dans la sous-section 3.3.1. Ensuite, nous documentons ces cas d'utilisation dans la sous-section 3.3.2 selon le tableau 3.2.

### 3.3.1 Diagramme de cas d'utilisation - DCU

Dans cette sous-section, nous allons décrire le DCU, figure 3.1. Les différents cas d'utilisations mentionnés peuvent être effectué par l'acteur du système. Les cas d'utilisations les plus important sont documentés dans ce qui suit selon le tableau 3.2.

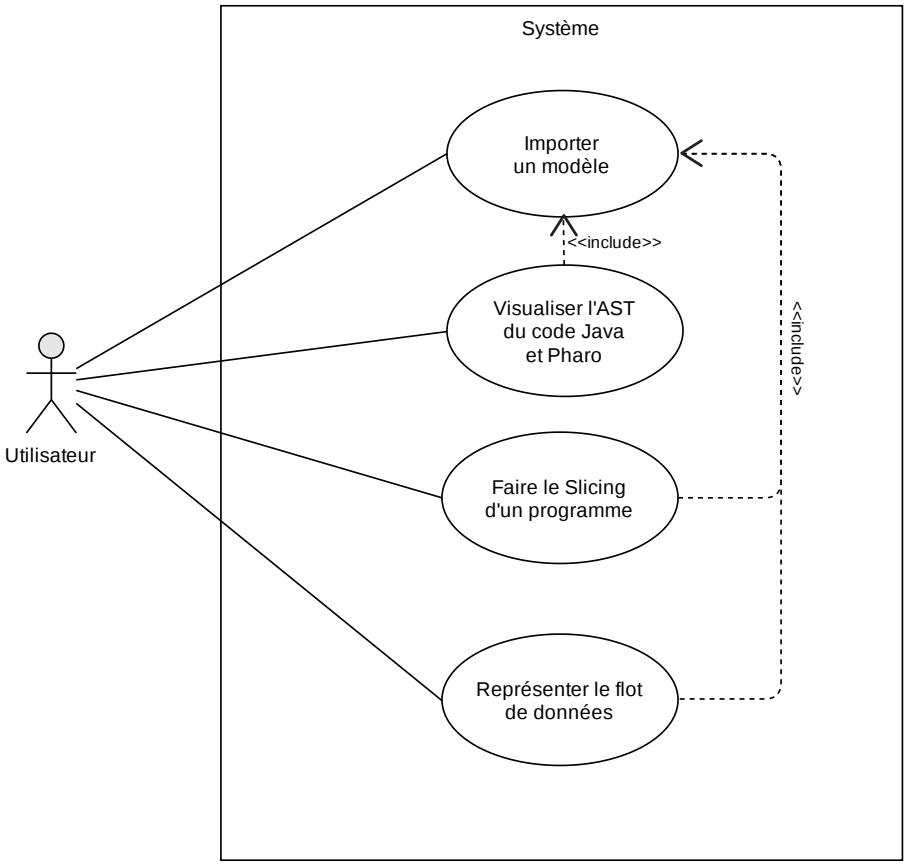



Fig. 3.1 : Diagramme de cas d'utilisations.

### 3.3.2 Documentation des cas d'utilisation

| ID | CU | Documentation | Diagramme d'activité |
|---|---|---|---|
| 1 | Importer le modèle | | ✓ |
| 2 | Visualiser l'AST du code Java/Pharo | | ✓ |
| 3 | Faire le slicing d'un programme | ✓ | ✓ |
| 4 | Représenter le flot de donnée | ✓ | ✓ |

Tab. 3.2 : Liste de la documentation des cas d'utilisation.

### Documentation du CU #3

| |
|---|
| **CU** : Faire le slicing d'un programme |
| **ID** : 3 |
| **Description brève :** Avoir le slice d'un programme pour une variable donnée dans une ligne donnée et dans une direction spécifique. |
| **Acteurs primaires** : Utilisateur |
| **Acteurs secondaires** : / |
| **Pré condition :** Le modèle doit être importé et chargé. |
| **Enchaînement principal** :<br>Le cas d'utilisation démarre lorsque l'utilisateur souhaite faire le slicing:<br>1. Il navigue sur le modèle pour arriver à la ligne et la donnée correspondante,<br>2. il sélectionne l'entité souhaite faire le slice,<br>3. il envoie le message "allSliceUp" pour faire le slicing dans la direction de haut,<br>4. il envoie le message "allSliceDown" pour faire le slicing dans la direction du bas. |
| **Post condition :** Le résultat de l'algorithme de slicing, qui est dans ce cas l'ensemble des instructions, est retourné |
| **Enchaînement alternatif** : Si l'entité sélectionnée n'est pas adéquate, par exemple un mot clé du langage "for" ou "class", une exception est soulevée. |

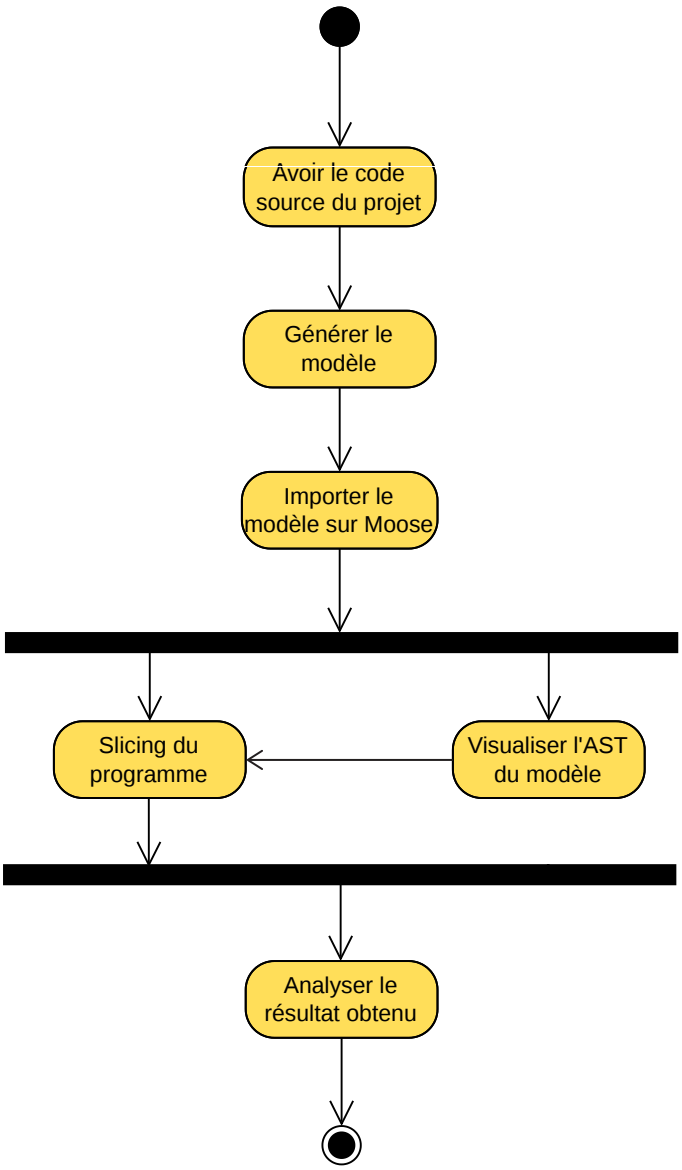
Tab. 3.3 : Documentation du CU : Faire le slicing d'un programme.

### Documentation du CU #4

| |
|---|
| **CU** : Représenter le flot de donnée |
| **ID** : 4 |
| **Description brève :** Avoir une visualisation de la donnée selon le slicing choisi |
| **Acteurs primaires** : Utilisateur |
| **Acteurs secondaires** : / |
| **Pré condition :** Le modèle doit être importé et chargé. |
| **Enchaînement principal** :<br>Le cas d'utilisation démarre lorsque l'utilisateur souhaite représenter le flot de données dans l'AST:<br>1. Il applique le slicing sur l'entité et l'endroit voulu,<br>2. Il sélectionne la rubrique graph de cette entité,<br>3. Le système représente le flux de données à analyser. |
| **Post condition :** le graphe AST sera affiché, le flux de la donnée dans le programme est mis en évidence au niveau de la représentation affichée. |
| **Enchaînement alternatif** : Si l'entité sélectionnée n'est pas adéquate, par exemple un mot clé du langage "for" ou "class", une exception est soulevée. |

Tab. 3.4 : Documentation du CU : Représenter le flot de donnée.

### 3.3.3 Diagramme d'activités




.

FIG. 3.2 : Diagramme d'activité des CUs.

## 3.4 Conclusion

Dans ce chapitre nous avons commencé par identifier les besoins du projet, ensuite nous avons modélisé et représenté par des diagrammes UML (Unified modeling Langage) les fonctionnalités de la solution.

Pour résumer, le besoin principalement est d'analyser le flot de données dans les programme et pour atteindre cet objectif nous allons développer une solution extensible à la base des méta-modèles existants (Famix, FAST & Carrefour) avec une visualisation qui nous permet d'avoir une représentation de la solution.

# Chapitre 4

# Conception

## 4.1 Introduction

Après avoir réalisé une étude bibliographique sur les techniques de rétro-ingénierie, plus spécifiquement l'analyse de flux de données, et après avoir établi une description sur l'environnement existant et défini les besoins, nous allons nous concentrer dans ce chapitre sur la partie la plus importante dans le projet, en l'occurrence la conception de la solution. Nous commençons dans ce qui suit par des concepts avancés de l'orienté objet que nous allons utiliser, ensuite on passe au diagramme de classes de notre projet et finalement au diagramme de séquences.

## 4.2 Concepts OOP & Méthodologie

### 4.2.1 Les Traits

Un trait est un concept utilisé dans la programmation orientée objet, qui représente un ensemble de méthodes pouvant être utilisées pour étendre les fonctionnalités d'une classe (*Wiki Trait* 2021). Autrement dit (Anquetil et al. 2020), « Un trait est un ensemble de méthodes, séparé de toute hiérarchie de classes. Les traits peuvent être composés dans un ordre arbitraire. L'entité composée (classe ou trait) a un contrôle total sur la composition et peut résoudre les conflits explicitement ». Dans l'exemple suivant, figure 4.1, on a le Trait #TNameOfMyTrait qui est utilisé par la classe #MyClass.

```
Trait named: #TNameOfMyTrait
        uses: {}
        package: 'MyPackage'
```

```
MySuperClass subclass: #MyClass
        uses: TNameOfMyTrait
        slots: {  }
        classVariables: {  }
        package: 'MyPackage'
```

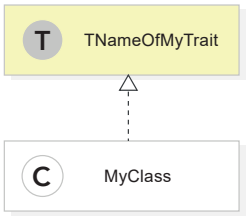



Fig. 4.1 : Simple exemple de l'utilisation de Trait (sans état).

Les traits sont des unités de comportement pures. C'est à dire, c'est des objets qui peuvent être composés pour enrichir d'autres traits.

Le mécanisme de composition des traits est une alternative à l'héritage multiple ou mixin dans lequel le compositeur a un contrôle total sur la composition des traits. Il permet plus de réutilisation que l'héritage unique sans introduire les inconvénients de l'héritage multiple ou mixin.

Dans leur forme originale, les traits étaient sans état, c'est-à-dire des groupes de méthodes pures sans aucun attribut. Les traits avec état étendent les traits sans état en introduisant un seul opérateur d'accès aux attributs pour donner aux clients du trait le contrôle de la visibilité des attributs (Tesone et al. 2020).

### 4.2.2 Extension de classe

Smalltalk fait un usage intensif d'une fonctionnalité appelée extension de classe (Teruel, Ducasse et Denker 2012). Une extension de classe consiste à ajouter des méthodes à une classe qui existe déjà dans le système. Des fonctionnalités similaires dans d'autres langages dynamiques sont appelées Monkey Patching ou Open Classes. Dans la figure 4.2, lorsque le package réseau est chargé, il ajoute une nouvelle méthode asUrl à la classe String. Lorsque le package Network est déchargé, la méthode asUrl est alors supprimée de la classe String.

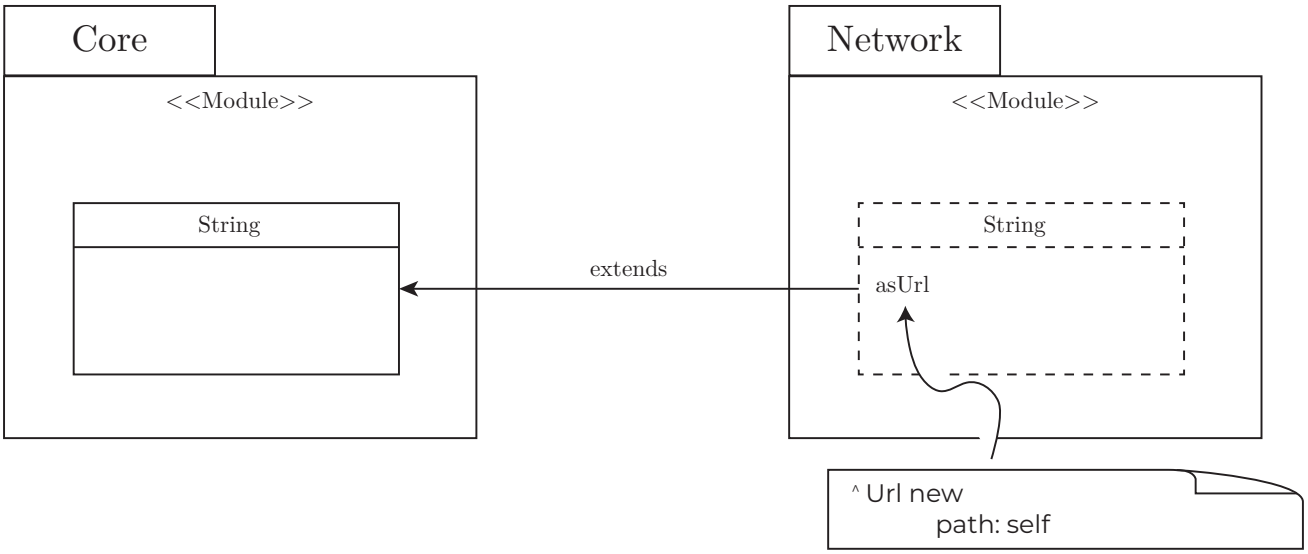
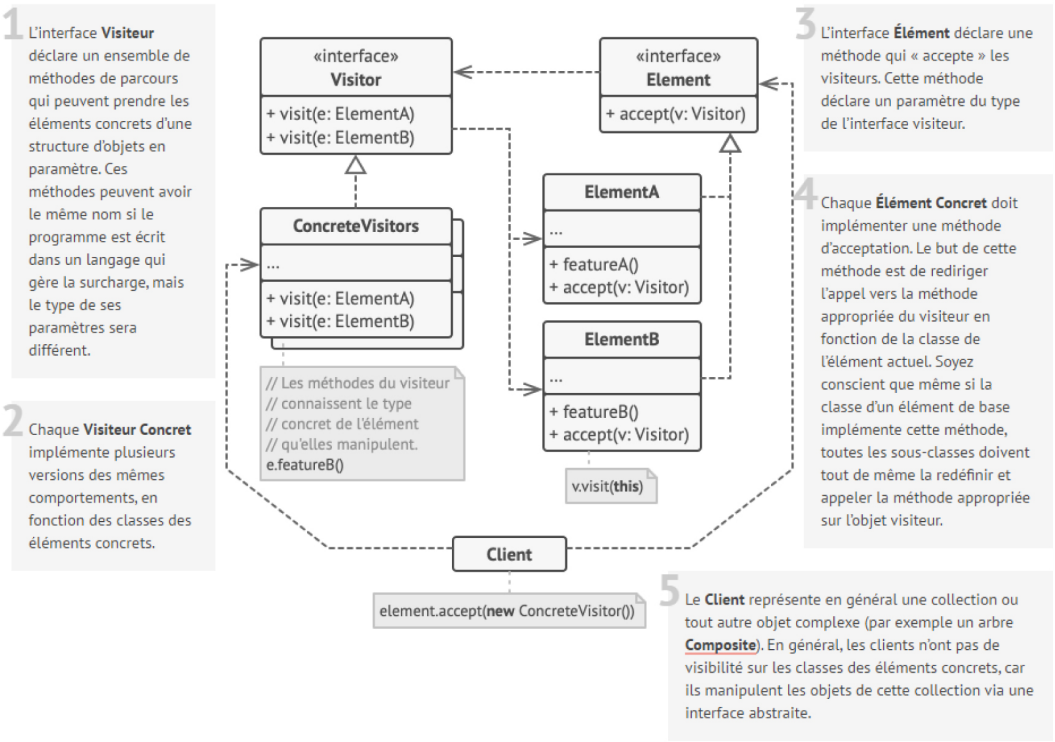



Fig. 4.2 : Exemple d'extension de classe dans SmallTalk.

### 4.2.3 Le patron Visitor

Le patron de conception Visitor, est comportemental, permet de faire une séparation entre les algorithmes et les objets sur lesquels ils opèrent (Gamma et al. 1994).

Ce patron de conception nous propose de placer un nouveau comportement dans une classe séparée que l'on appelle visitor, plutôt que de l'intégrer dans des classes existantes. L'objet qui devait lancer ce traitement à l'origine est effectivement passé en paramètre des méthodes du visitor, ce qui permet à la méthode d'avoir accès à toutes les données nécessaires qui se trouvent à l'intérieur de l'objet (Gamma et al. 1994). La figure suivante 4.3 explique le mécanisme de ce patron de conception.




.

FIG. 4.3 : Le patron de conception Visitor (GURU 2022).

### 4.2.4 Le Test-Driven Developement - TDD

Le Test-Driven Development (ou développement piloté par les tests) est une approche de développement de logiciel qui consiste à concevoir un logiciel par petites étapes, de façon progressive, en écrivant avant chaque partie du code source propre au logiciel les tests correspondants et en remaniant le code continuellement (WIKIPEDIA 2022).

Le processus préconisé par le TDD est basé sur 3 lois, Robert C. Martin les reformule (BECK 2002) comme suit :

- Il faut écrire un test qui échoue avant d'écrire le code de production correspondant.
- Il faut écrire une seule assertion à la fois, qui fait échouer le test ou qui échoue à la compilation.
- Il faut écrire le minimum de code de production pour que l'assertion du test actuellement en échec soit satisfaite

Lors de la conception de notre solution, nous allons suivre cette approche de développement afin de concevoir nos fonctionnalités progressivement et en se basant sur des tests case (tests unitaires).

## 4.3 Modèles statique et dynamique

Nous avons opté pour un découpage modulaire, où chaque module devrait avoir la responsabilité d'une fonctionnalité de la solution, et il devrait encapsuler cette fonctionnalité. Tous les services de ce module doivent être étroitement alignés sur cette responsabilité. Dans les sections suivantes, nous allons modéliser un modèle statique (diagramme de classes) et un modèle dynamique (diagramme de séquences) pour chaque module.

La conception des module est guidée par les tests i.e TDD. En effet, on écrit notre cas de test en amont des détails d'implémentation et cela nous permet d'avoir une couverture globale des cas, meilleure qualité du code (moins de bugs) mais aussi le processus de refactoring (Red -> Green -> Refactor). Dans les modèles de conceptions qui suivent, nous avons montré uniquement les tests case important dans le but de minimiser la taille des figures.

### 4.3.1 Défintions

#### Modèle Statique - Diagramme de classes

Le diagramme de classes est une modélisation orientée objet qui permet de définir les classes du système et ses attributs ainsi que les différentes règles d'associations entre ces classes. Il permet de visualiser les différents objets manipulés dans notre système (Beck 2002).

#### Modèle Dynamique - Diagramme de séquences

Le diagramme de séquences est une représentation UML, dont l'objectif est de représenter graphiquement les différentes interactions entre les acteurs et les éléments du système selon un ordre chronologique en plus des méthodes utilisées lors de la manipulation de ces objets (Beck 2002).

Dans les sous-sections suivantes, nous allons présenter les diagramme de classes et de séquences pour chaque module de la solution :

### 4.3.2 Importeur de modèle - UI

L'importeur de modèles est un sous-module de l'importeur existant déjà au niveau de Moose. Il s'agit d'une interface utilisateur qui utilise le patron de conception command déja existant comme Controller. Ce module est mis en place en utilisant l'architecture MVC (Model-Vue-Controller). En effet, notre sous-classe MiImportFromFileCommand est le controller tandis que MiImportModelFromFileDialog est le modèle, figure 4.4. L'objectif de ce module est d'importer un modèle pour les deux formats JSON et MSE.

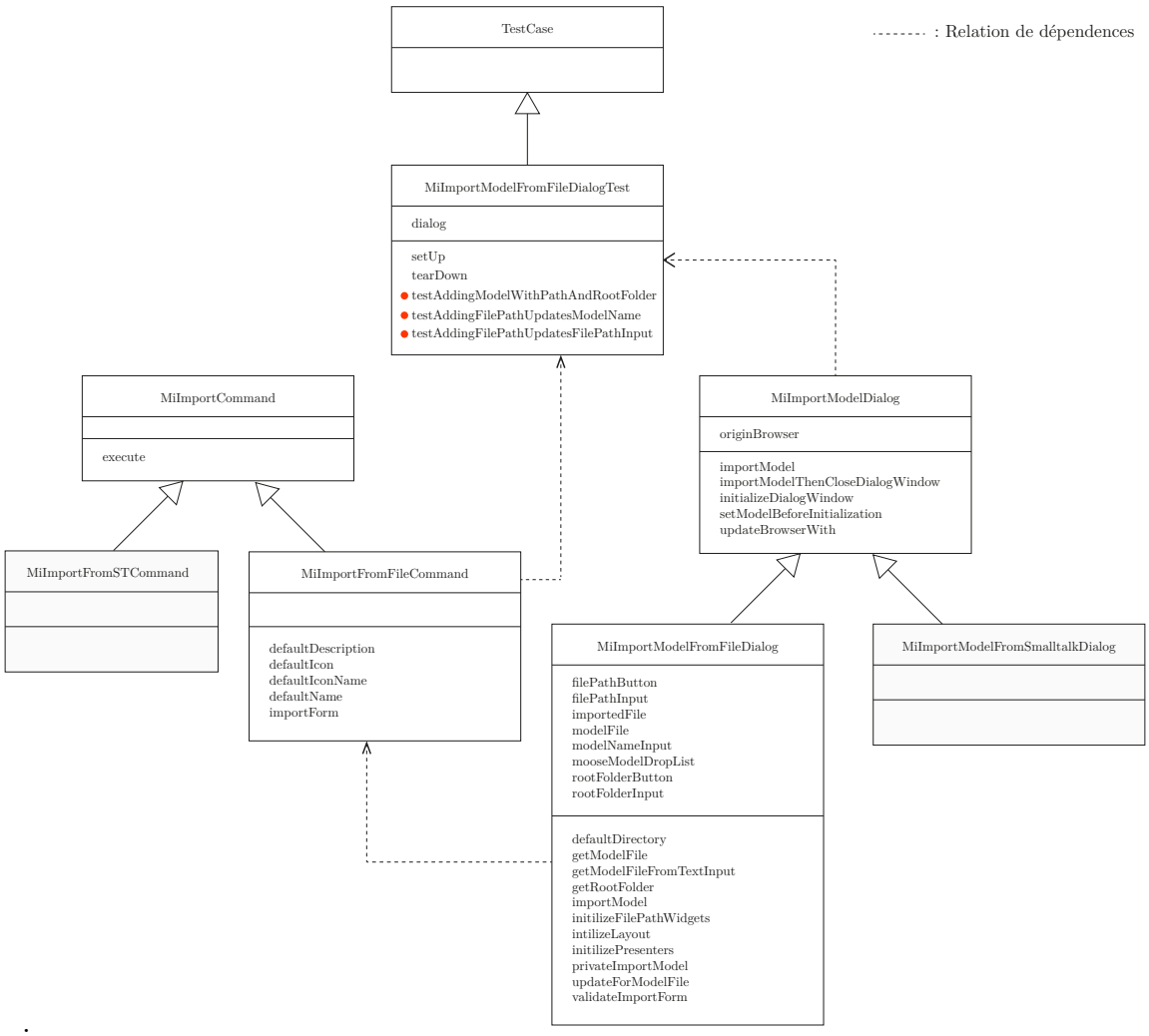



Fig. 4.4 : Diagramme de classes du module UI - Importeur modèle.

Dans le but d'importer un modèle et l'utiliser, une interface utilisateur sera développée. Cette fenêtre contient plusieurs champs à remplir : le type du modèle, la location du modèle et ainsi le code source du projet associé à ce modèle. En effet, l'utilisateur ouvre cette interface et remplit tous les champs nécessaires. Une fois ces champs sont corrects, le modèle est validé et bien importé. Les interactions entre les objets de ce processus sont illustrées dans la figure 4.5.

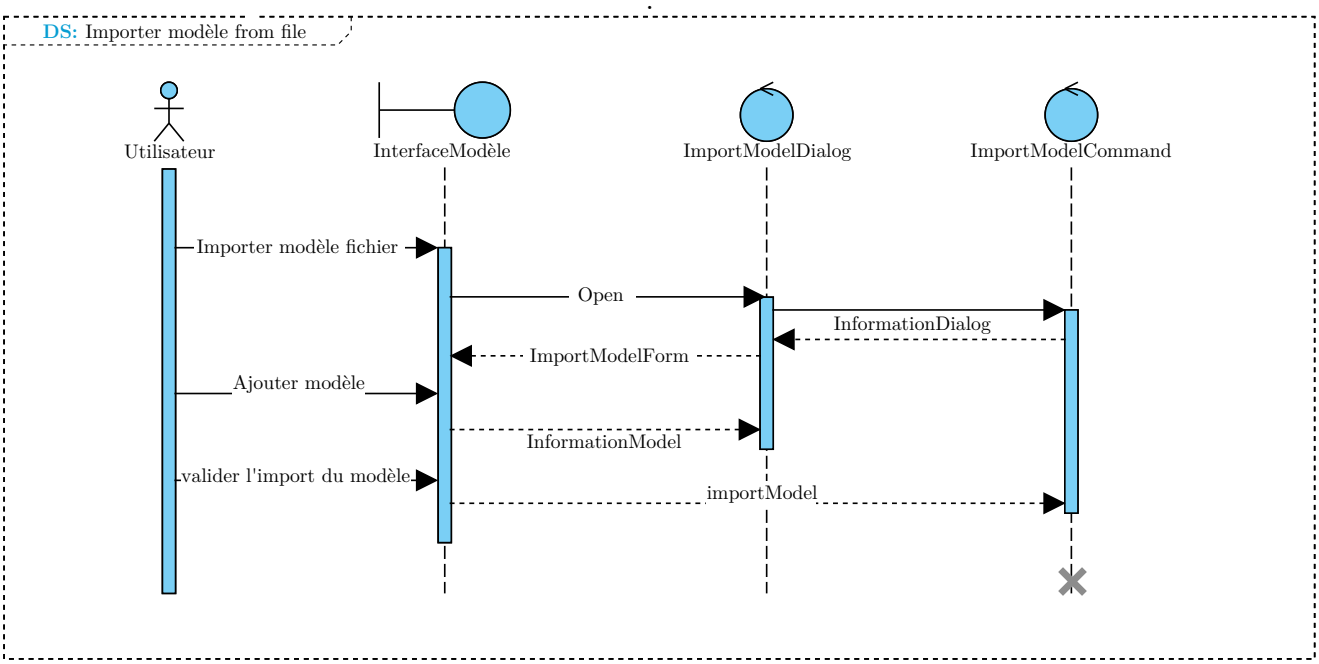



FIG. 4.5 : Diagramme de séquences du module .

### 4.3.3 Visualisation de l'AST

Le module de visualisation de l'AST, illustré dans la figure 5.1, est un remplaçant d'une représentation du méta-modèle FAST qui était textuelle et non significative dans certains cas. L'extension de certaines entités FAST était dans le but de donner un nom représentatif au nœud de l'AST par exemple dans le cas d'une entité FASTJavaLiteral on met la valeur du littéral au lieu du nom de la classe.

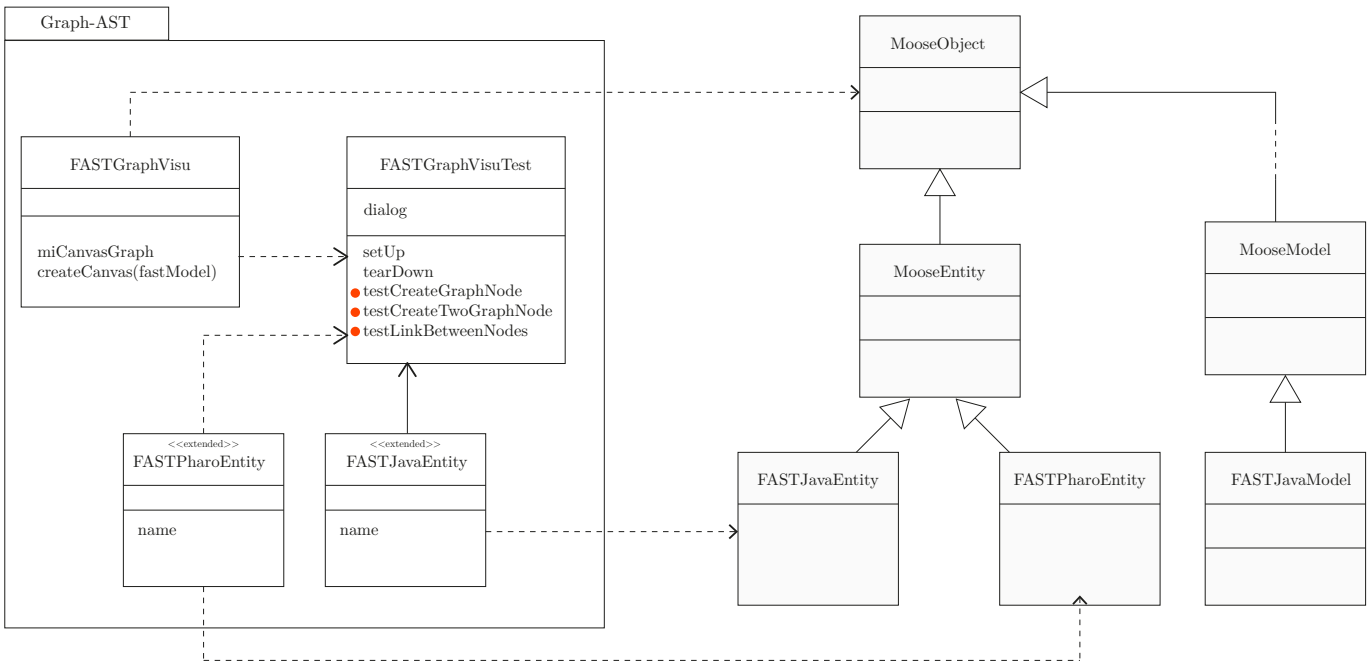



FIG. 4.6 : Diagramme de classe du module visualisation AST.

Le diagramme 4.7 décrit les étapes de visualiser l'AST d'un modèle, l'utilisateur génère l'AST à travers le méta-modèle FAST, ceci est fait en envoyant le message generateFast-Java au modèle importé. Une fois que l'AST est généré avec succès, l'utilisateur peut voir la représentation graphique de l'arbre. Cette représentation est la réponse du message createGraph envoyé à l'objet FASTGraphVisu qui est responsable du rendu graphique de l'AST.

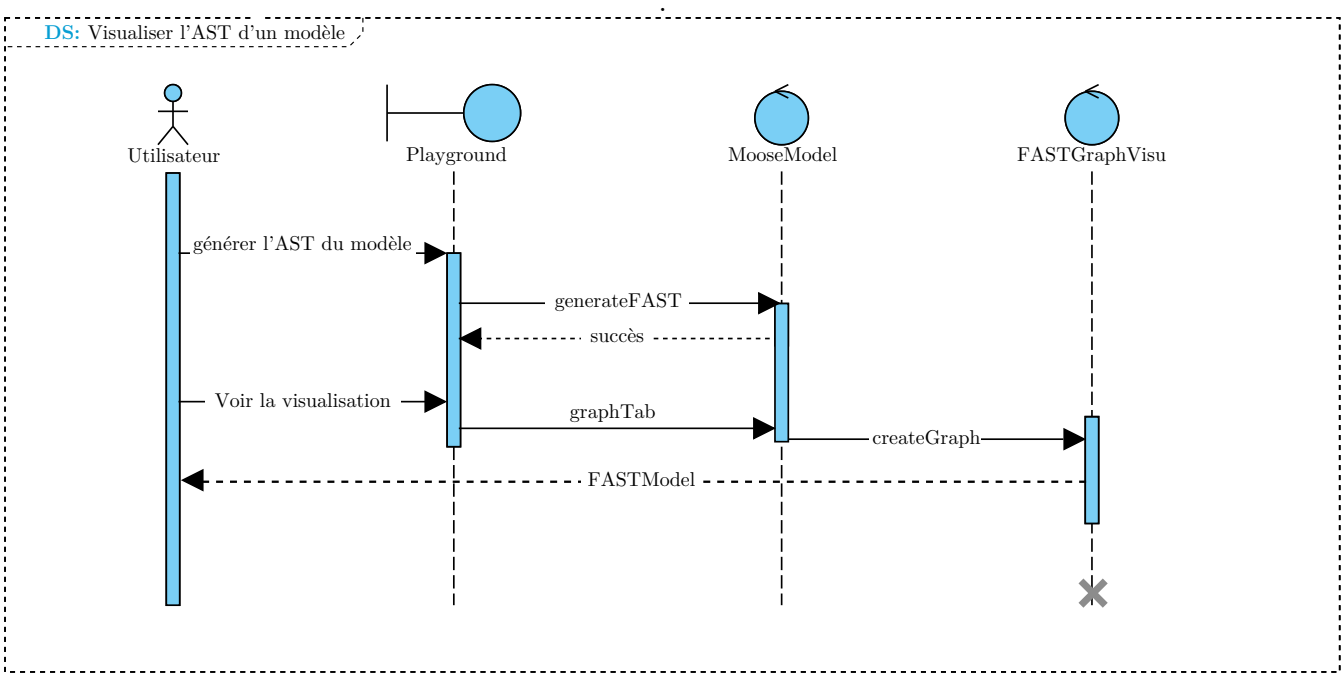



Fig. 4.7 : Diagramme de séquences du module.

### 4.3.4 Program Slicing

Le module de la technique de slicing, illustré dans la figure 5.1, est basé sur 3 classes principales :

- Slicer : c'est une classe contenant l'ensemble des instructions qui est le résultat du slicing comme attribut. Les méthodes de cette classes les types de slicing que nous allons utilisé (sliceUp et SliceDown).
- SliceVisitor : cette classe représente le visiteur du slicing, il permet de visiter chaque nœud de l'AST et appliquer le slicing selon le nœud.
- JavaModelGraph : c'est une classe UI de visualisation de flux de données contenant les différents intéractions avec le graph de l'AST.

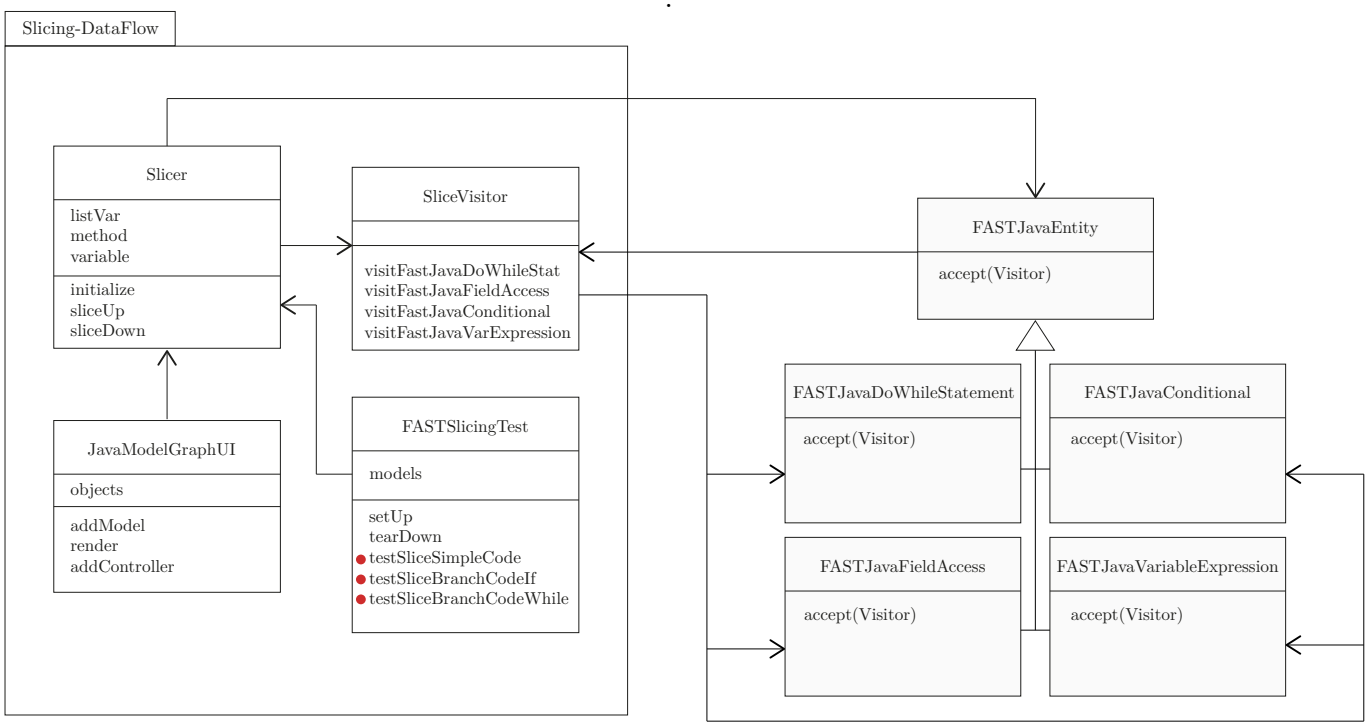



FIG. 4.8 : Diagramme de classes du module program slicing.

Le diagramme 4.9 décrit les étapes d'effectuer un program slicing. Elle se fait suite à la demande de l'utilisateur qu'après avoir entré le modèle dans le Playground. D'abord, le MooseModel reçoit le message Slicing avec l'entité correspondante. Le MooseModel vérifie ensuite si le slicing est applicable sur cette entité, si c'est le cas il envoie le message du slicing (up/down) au Slicer pour récupérer le résultat de l'algorithme du slicing associé (up/down) qui est un ensemble d'instructions représentant les endroits où cette donnée peut être modifiée. Une fois le résultat du slicing obtenu, nous pouvons visualiser le flux de l'entité choisie.

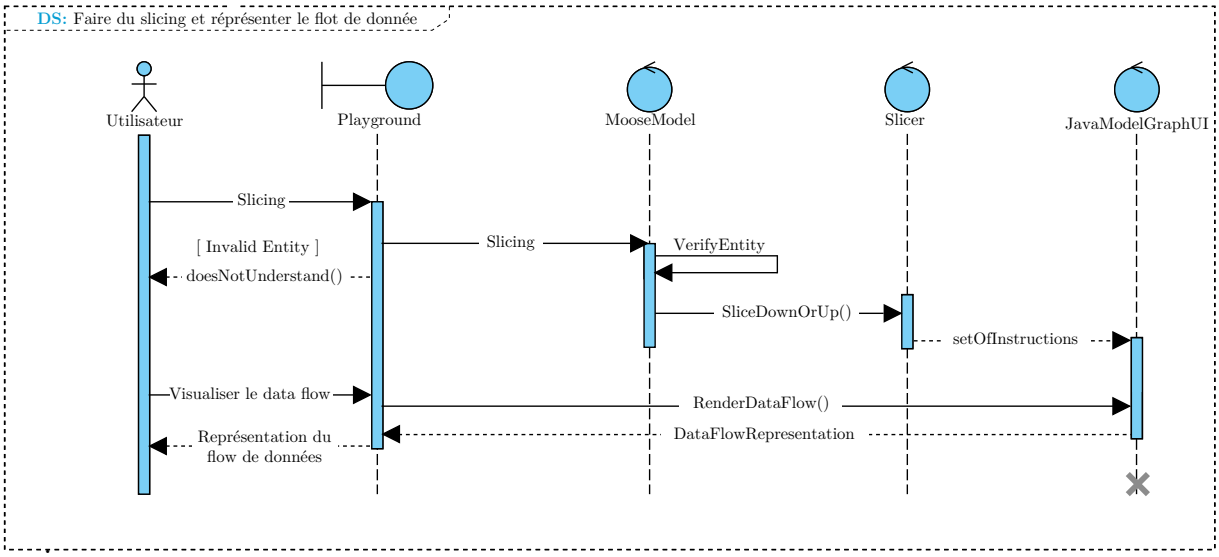



FIG. 4.9 : Diagramme de séquences du module.

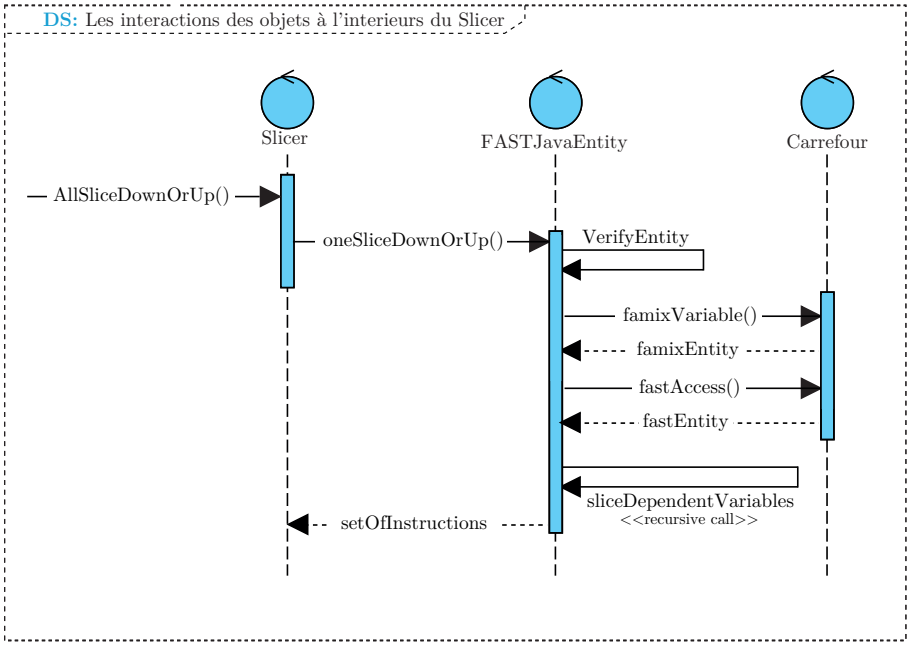



Fig. 4.10 : Diagramme de séquences interne du Slicer.

## 4.4 Conclusion

Pour conclure, dans ce chapitre nous avons cerné tous les concepts et les points relatifs à la partie la plus importante de notre projet, en l'occurrence la conception. Nous avons commencé par décrire les concepts utilisés lors de notre solution. Ensuite, nous avons modélisé le modèle statique du projet par le diagramme des classes ainsi que le développement dynamique via le diagramme de séquences qui illustre les différentes interactions entre l'utilisateur et les objets du système.

Cette étape est cruciale pour lever l'abstraction de la phase d'analyse et ajouter les concepts techniques de développement et diriger la phase de réalisation. Dans les prochains chapitres nous exposons la phase de réalisation ainsi que les résultats obtenus.

# Chapitre 5

# Implémentation et Tests

## 5.1 Introduction

Après la définition des aspects conceptuels de notre système, l'étape suivante est l'implémentation. Cette étape comprend la mise en œuvre de chaque partie du système ainsi que l'établissement des liens entre ces parties. Dans ce chapitre nous présentons le processus global de la solution. Par la suite, nous détaillons les technologies utilisées dans chaque module. Et nous terminons par présenter la solution développée.

## 5.2 Processus global de la solution

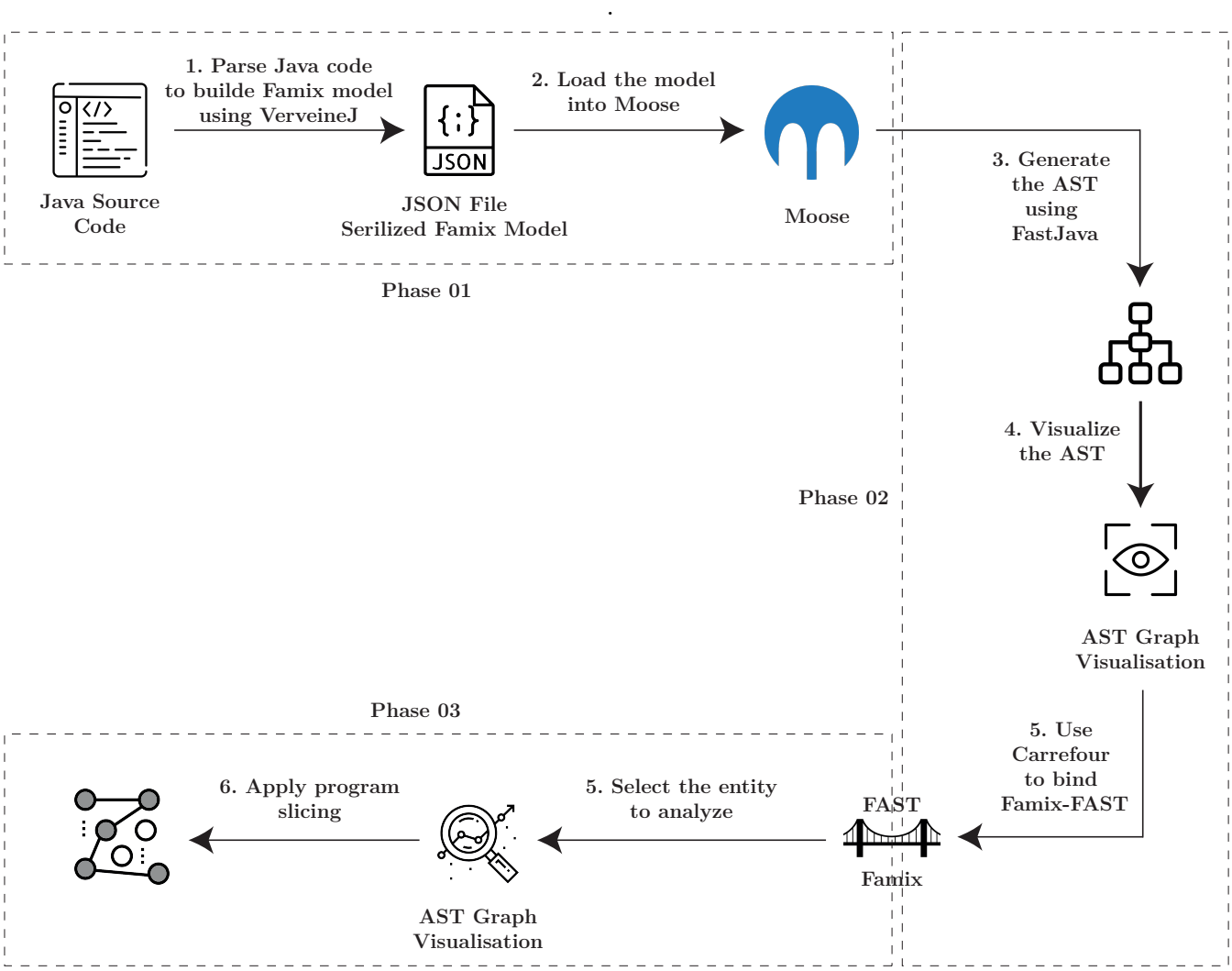



Fig. 5.1 : Workflow global de notre solution.

La figure illustre notre solution end-to-end. Nous distinguons 3 phases principales : Préparation du modèle, Génération du FAST avec Carrefour et L'analyse de flux de données. La préparation et la construction du modèle est faite par le Parser VerveinJ, le résultat obtenu peut-être importé sur Moose. Le modèle importé est un modèle Famix pour lequel doit générer l'AST et avoir plus de détails sur le code source en utilisant FAST. L'AST peut être représenté grâce à notre module de visualisation Graph-AST. En

naviguant sur l'AST, on choisit la donnée pour laquelle nous avons besoin de connaître son flux dans le programme et on applique le slicing dans l'ordre voulu descendant ou ascendant.

## 5.3 Technologies et outils utilisés

Nous citons dans cette section les outils et technologies utilisés dans la solution et ainsi les outils d'infrastructure et gestion de projet :

### Java

Java[1] est un langage de programmation orienté objet créé par James Gosling et Patrick Naughton, employés de Sun Microsystems, avec le soutien de Bill Joy (cofondateur de Sun Microsystems en 1982), présenté officiellement le 23 mai 1995 au SunWorld. La société Sun a été rachetée en 2009 par la société Oracle qui détient et maintient désormais Java. Une particularité de Java est que les logiciels écrits dans ce langage sont compilés vers une représentation binaire intermédiaire qui peut être exécutée dans une machine virtuelle Java (JVM) en faisant abstraction du système d'exploitation.


Fig. 5.2 : Logo de Java.

### VerveineJ

VerveineJ[2] est un, projet open source, qui permet de parser du code java vers un modèle Famix au format MSE ou JSON. Nous allons l'utiliser pour exporter le modèle java et importer ce dernier sur la plateforme Moose.


Fig. 5.3 : Le projet VerveinJ.

[1] https://www.java.com.
[2] https://github.com/NicolasAnquetil/VerveineJ.

## Roassal3

Roassal3[3] est un moteur de visualisation agile pour Pharo 8 et Pharo 9. Roassal a été créé pour permettre la visualisation interactive des données. Roassal3 est gratuit et open source, mais soutenu par Object Profile. Des offres d'assistance commerciale sont disponibles, y compris des services de formation et de conseil personnalisé.

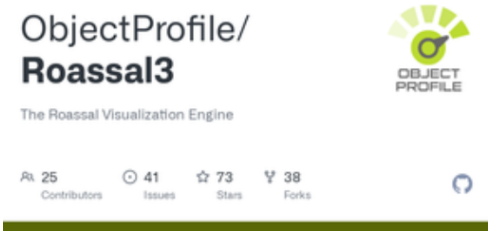



Fig. 5.4 : Le projet Roassal3.

## SmalltalkCI

SmallTalkCI [4] est un framework pris en charge par la communauté pour tester les projets Smalltalk sur Linux, OS X et Windows avec prise en charge intégrée des actions GitHub, Travis CI, AppVeyor et GitLab CI/CD.Il est inspiré de builderCI et vise à fournir un moyen uniforme et facile de charger et de tester des projets Smalltalk.


Fig. 5.5 : Le projet SmalltalkCI.

## Git

Git [5] est un logiciel de gestion des versions décentralisé pour gérer et traquer les changements durant le développement d'un logiciel. Il a été conçu pour faciliter la collaboration entre les développeurs sur un même projet. Il supporte les flux de travail non-linéaire distribués et il garde l'intégrité des données. Similaire en cela à BitKeeper, Git ne repose pas sur un serveur centralisé, mais il utilise un système de connexion pair à pair. Le code informatique développé est stocké non seulement sur l'ordinateur de chaque contributeur du projet, mais il peut également l'être sur un serveur dédié. C'est un outil de bas niveau, qui se veut simple et performant.

Fig. 5.6 : Logo de git.

[3] https://github.com/ObjectProfile/Roassal3.
[4] https://github.com/hpi-swa/smalltalkCI.
[5] https://git-scm.com/.

### Github

Github [6] est un service web d'hébergement et de gestion de développement de logiciels, utilisant le logiciel de gestion de versions Git. Ce site est développé en Ruby on Rails et Erlang par Chris Wanstrath, PJ Hyett et Tom Preston-Werner. GitHub propose des comptes professionnels payants, ainsi que des comptes gratuits pour les projets de logiciels libres. Le site assure également un contrôle d'accès et des fonctionnalités destinées à la collaboration comme le suivi des bugs, les demandes de fonctionnalités, la gestion de tâches et un wiki pour chaque projet.

FIG. 5.7 : Logo de github.

## 5.4 Implémentation

Dans le but d'avoir une meilleure séparation des responsabilités, la solution est découpée en module comme décrit dans le chapitre conception **??**. Nous allons décrire dans cette section la mise en place de chaque module.

### 5.4.1 Imprteur de modèle

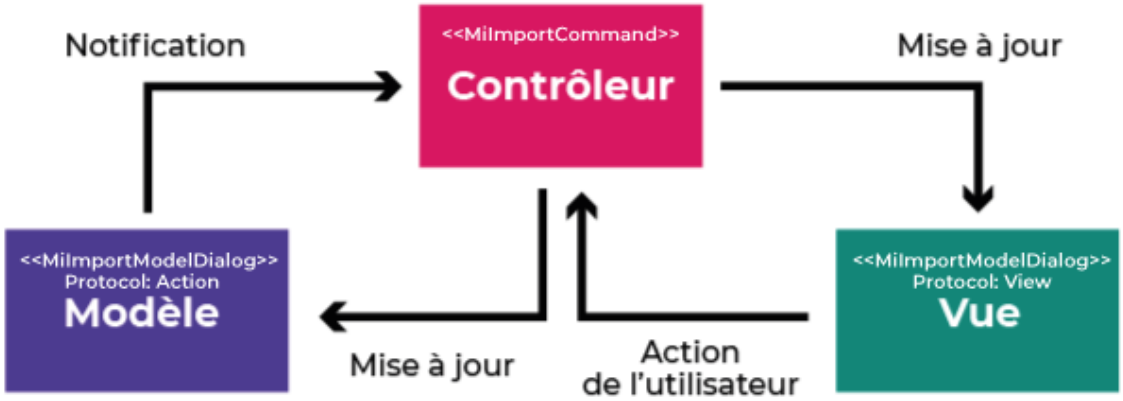



FIG. 5.8 : L'architecture MVC dans le module Importeur de modèle.

Ce module est mis en place en utilisant l'architecture MVC. La figure 5.8 représente cette architecture au niveau de ce package. La classe MiImportCommand est une classe abstraite qui prend une action à effectuer (par exemple l'ouverture de l'imprteur) et la transforme en un objet à utiliser par le modèle qui est dans ce cas MiImportModelDialog. Le modèle cette architecture effectue plusieurs actions comme nous le voyons dans la figure 5.9 au niveau de la 4ème onglet (l'onglet droite).

[6] https://github.com/.

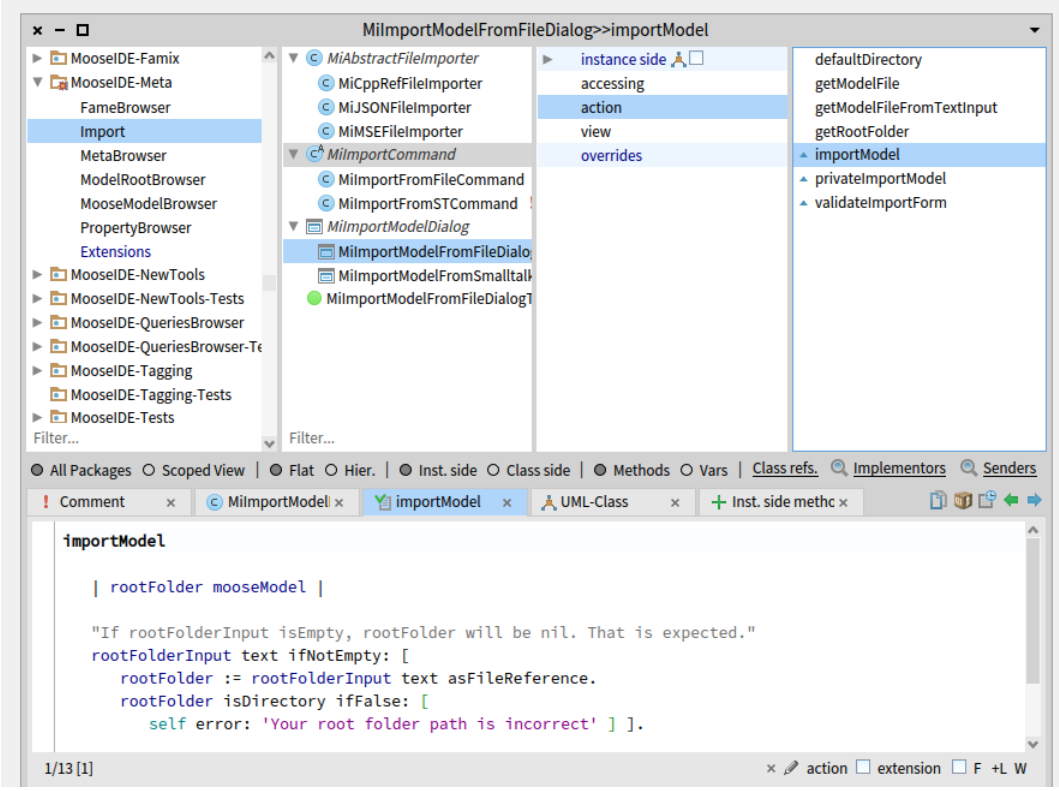



FIG. 5.9 : Structure du module Importeur de modèle dans Moose.

### 5.4.2 Visualisation d'AST

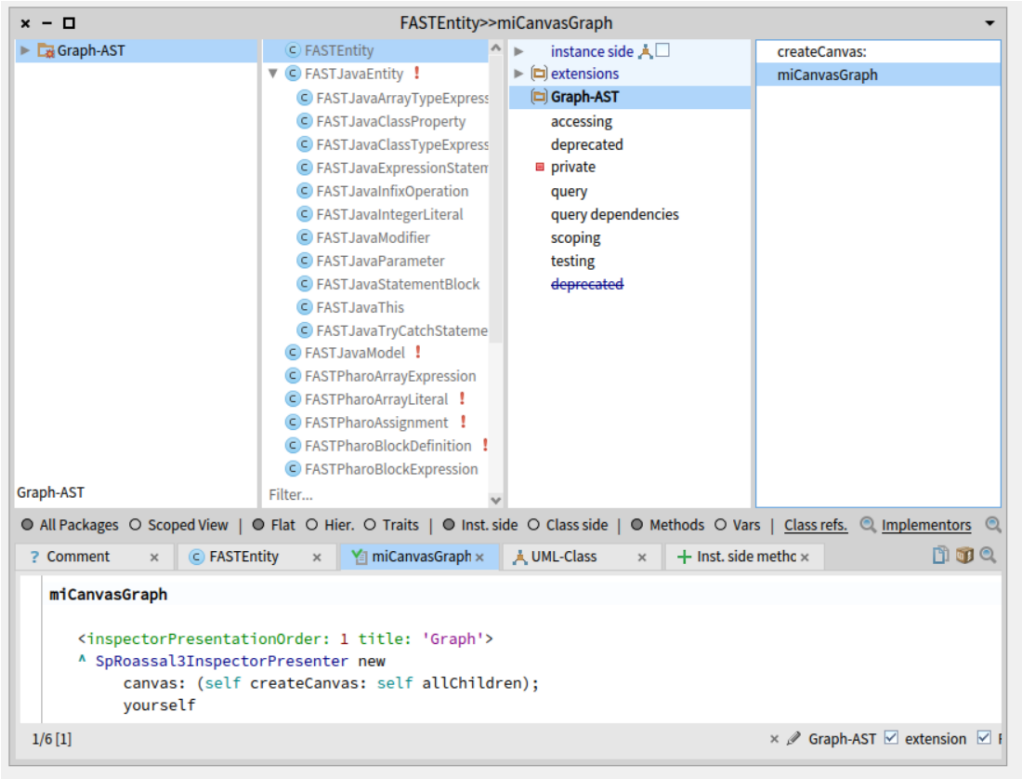



FIG. 5.10 : Structure du module Graph-AST pour la visualisation de l'AST dans Moose.

Ce module est intégré dans le core du modèle FAST sous forme d'une extension de la classe dans FASTEntity, figure 5.10. L'implémentation de ce module est basée sur le moteur de visualisation Roassal3, figure 5.11 où nous parcourons l'AST et pour chaque nœud, nous créons une boite avec un label adéquat. Le choix de label peut être une valeur si le nœud est une feuille de l'arbre sinon il prendra le nom de sa classe. Le nom du label est une redéfinition de la méthode #name du nœud au niveau de la classe de l'entité.

FASTEntity>>createCanvas:

Comment | FASTEntity | createCanvas: | UML-Class | Inst. side methc

```
1  createCanvas: data
2
3        | canvas shapes |
4        canvas := RSCanvas new.
5        shapes := data collect: [ :entity |
6                        | box label |
7                        box := RSBox new
8                                    border: (RSBorder new color: Color black);
9                                    noPaint.
10                       label := RSLabel new
11                                     text: entity name;
12                                     color: 'black';
13                                     yourself.
14                       box extent: label extent + 10.
15                       {
16                             box.
17                             label } asShape
18                             model: entity;
19                             draggable;
20                             popup;
21                             yourself ].
22       canvas addAll: shapes.
23       RSLineBuilder orthoVertical
24             withVerticalAttachPoint;
25             canvas: canvas;
26             shapes: canvas nodes;
27             connectToAll: #children.
28
29       RSTreeLayout on: canvas nodes.
30       canvas addInteraction: RSCanvasController.
31
32       ^ canvas
```

FIG. 5.11 : Création du graph en utilisant Roassal3.

### 5.4.3 Slicing et Analyse de flux de données

Le package de Slicing et représentation de flux de données est structuré de la manière suivante :

- Slicer : c'est une classe contenant l'ensemble des instructions qui est le résultat du slicing comme attribut. Cette classe, figure 5.13, est considérée comme le client et l'interface pour ce module.

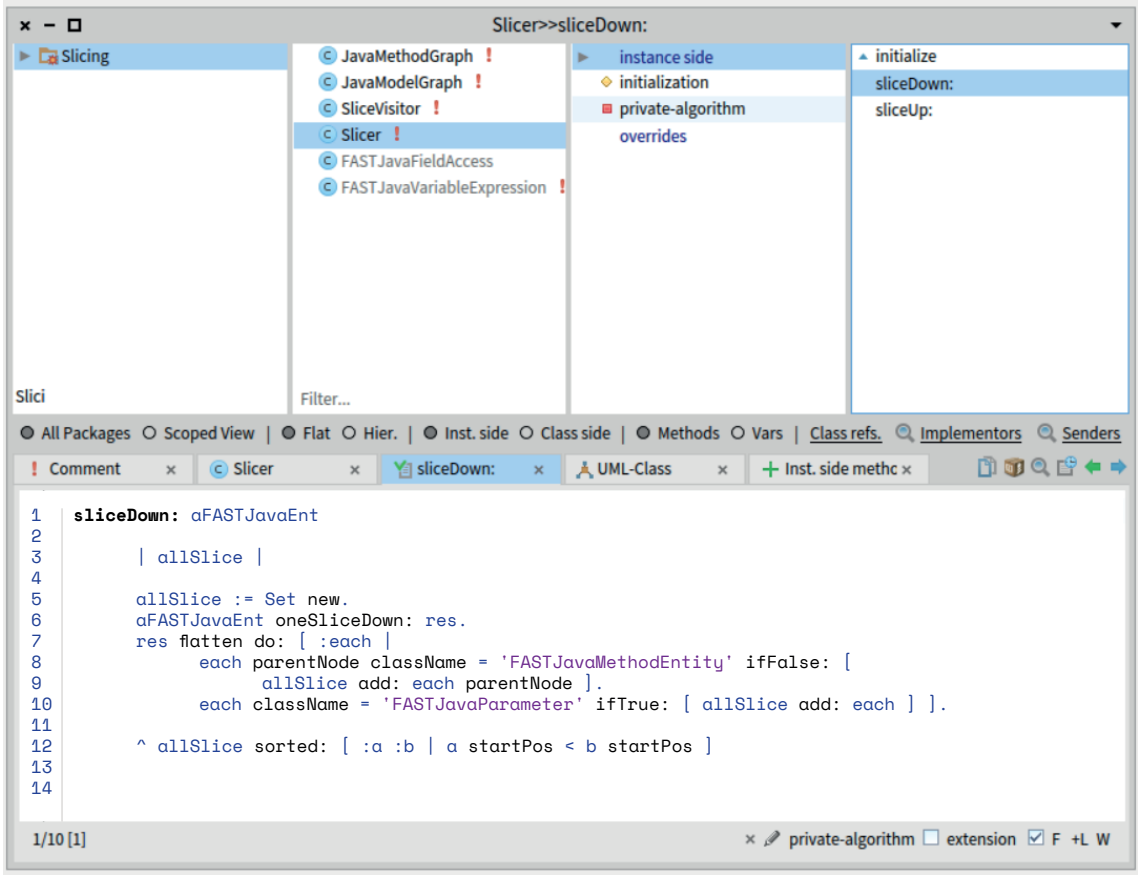



FIG. 5.12 : Structure du module Slicing dans Moose.

- SliceVisitor : cette classe représente le visiteur du slicing, il permet de visiter chaque nœud de l'AST et appliquer le slicing selon le nœud. A titre d'exemple, l'algorithme dans la figure 5.14 : pour un nœud de condition : on doit rajouter toutes les variables qui sont dans la condition comme variable de slicing. Pour illustrer l'exemple précédent, on prend au début une variable de slicing n (ligne 9) et une fois arriver à une condition (ligne 7) de type n>i (ou n<i ou n=!i .. etc) les variable de slicing deviennent n et i à partir de cet endroit là (ligne 7). Donc, la donnée n est impacté par i au moment où nous sommes arrivés à la condition.

- JavaGraph : c'est des classes de visualisation de flux de données pour intéragir avec l'AST en appliquant le slicing dans l'ordre voulu (Slicing Up / Slicing Down).

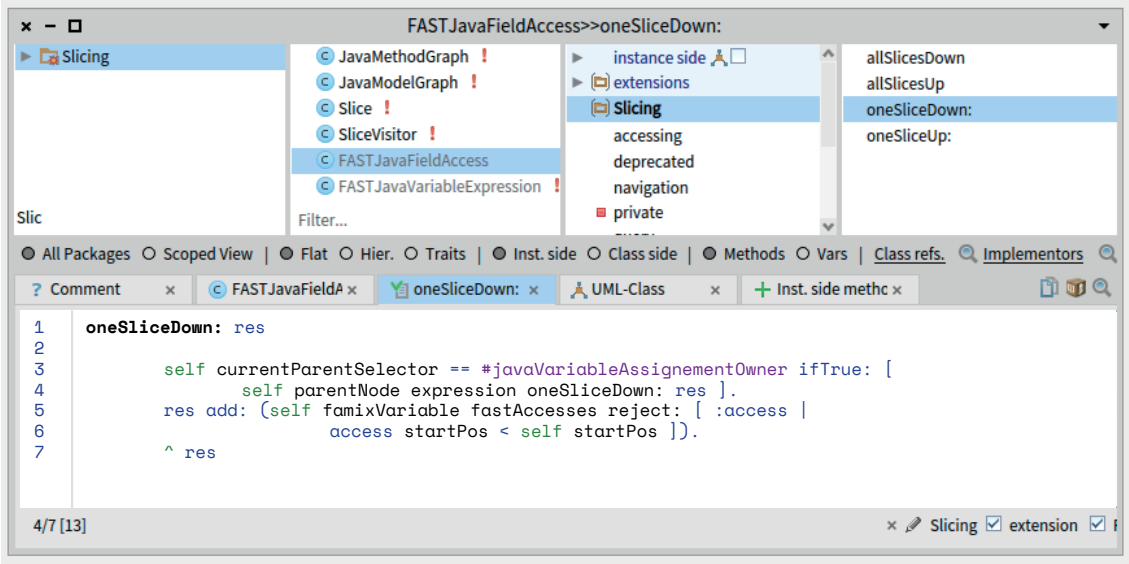



FIG. 5.13 : Le corp de la méthode oneSliceDown.

**Algorithm :** Class to apply Program Slicing

```
1  class MyClassA
2      public method integer exmpleSLicing()
3          var n, sum, i: integer
4          sum ← 0
5          i ←— 0
6          read(n)
7          while (n > i) do
8              sum ← sum + i
9              n ←— n - 1
10         write(sum)
11     end method
12 end class
```

**Résultat:** Program Slicing

```
3  var n: integer
5  i ←— 0
6  read(n)
7  while (n > i) do
9      n ←— n - 1
```

FIG. 5.14 : Slicing sur un noeud de condition.

## 5.5 Conclusion

Dans ce chapitre nous avons présenté la partie réalisation de notre projet. Nous avons commencé par illustrer le processus global de notre solution en expliquant les différents modules utilisés dans chaque phase. Nous avons ensuite décrit les technologies et les projets que nous avons utilisés pour la solution. Finalement, nous avons discuter l'implémentation de chaque module.

# Chapitre 6

# Résultats

## 6.1 Introduction

Dans ce chapitre, nous présentons les résultats obtenus au cours du développement de la solution. Ces résultats nous ont permis de faire des analyses de code et de déterminer les anomalies de flux de données. Les résultats et l'analyse présentés dans ce chapitre correspondent à un étude de cas du début jusqu'à la fin de la solution.

## 6.2 Etude de cas

Nous allons présenter dans cette section, les IHMs « Interface Homme Machine » correspondant aux fonctionnalités de la solution en déroulant un exemple concret, voir la figure 6.1, et passant sur tout le processus.

**Algorithm :** Exemple pour la partie résultat

```
1  class MyClassA
2      private attribute  b: integer
3      public attribute  i: integer
4      public method  integer exmple(a: integer)
5          var n: integer
6          this.i ← 0
7          a ←—— 0
8          read(n)
9          if (a > i) then
10             this.b ← 5
11             this.b ← a
12             a ←—— 6
13             return a
14         return b
15     end method
16 end class
```

.

Fig. 6.1 : Exemple de déroulement.

### Préparation et génération du modèle

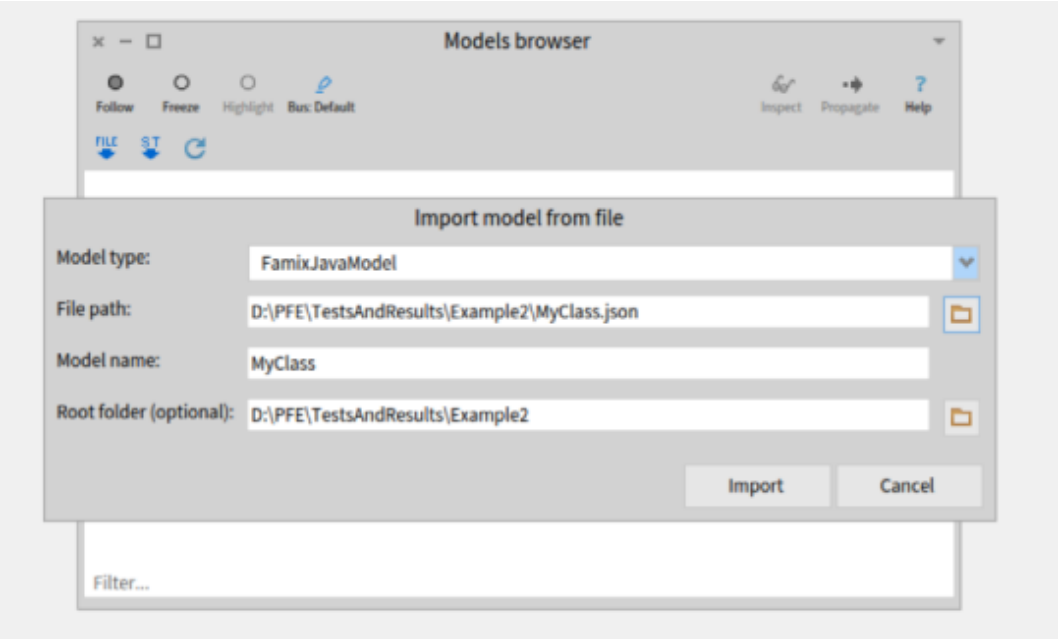
Tout d'abord, nous allons créer le fichier et générer le modèle comme suit, figure 6.2 :

```
MyClass.java ×
MyClass.java
class MyClass {
  private int b;
  public int i;
  public int methodAB(int a){
    int n = 0;
    this.i = 0;
    Scanner myObj = new Scanner(System.in);
    a = 8;
    System.out.println("Enter a number between 5 and 10");
    n = myObj.nextLine();
    if(a>i){
      this.b = 5;
      this.b = a;
      a = 6;
      return a;
    }
    return b;
  }
}

PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

PS D:\PFE\TestsAndResults\Example2> 'D:\Path\To\VerveineJ\verveinej.bat'
-format json -o testing.json -anchor assoc -autocp './src' './src'
```

.

FIG. 6.2 : Slicing sur un noeud de condition.

Ensuite, importons le modèle dans la plateforme Moose, figure 6.3.




.

FIG. 6.3 : Importeur de modèle sur Moose.

Exécutons le script suivant pour générer l'AST en utilisant FAST et faire le binding entre Famix et FAST, 6.4.

```
1 myClass := MooseModel root at: 1.
2 method := myClass allModelClasses first generateFastAndBind .
```

Fig. 6.4 : Script pour générer l'AST.

### Visualisation de l'AST

La figure ci-dessous 6.5 montre la représentation graphique de l'AST d'un modèle FAST.

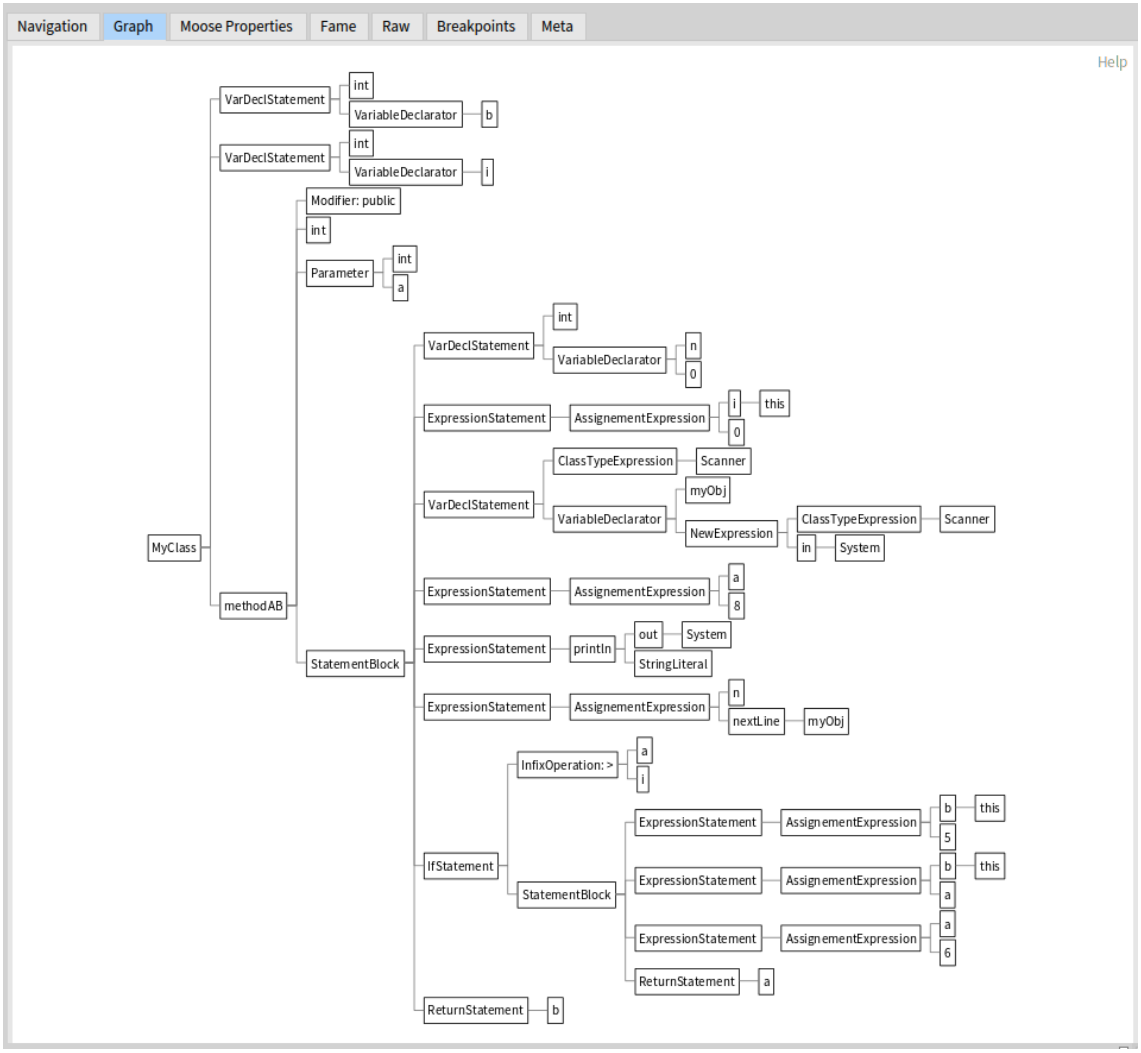



Fig. 6.5 : La représentation graphique de l'AST - IHM.

### Program Slicing

Nous appliquons notre slicing, orienté vers le haut, sur la variable b de la ligne 7 en envoyant le message allSlicingUp, figure 6.6. Comme résultat, la partie droite de la figure

6.6, on aura l'ensemble des instructions représentant le slice.

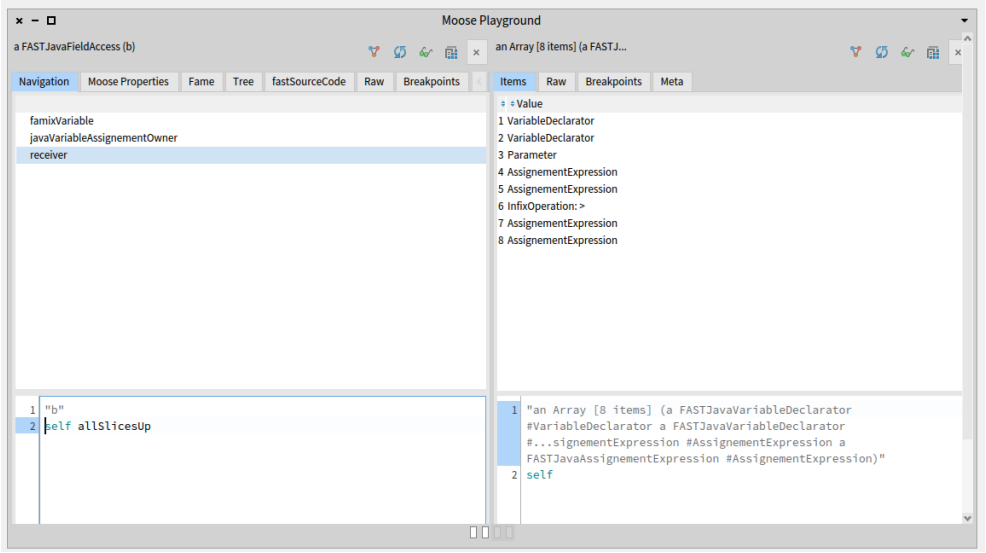



FIG. 6.6 : L'envoie du message slicingUp à l'entité b - IHM.

### 6.2.1 Visualisation de flux de données

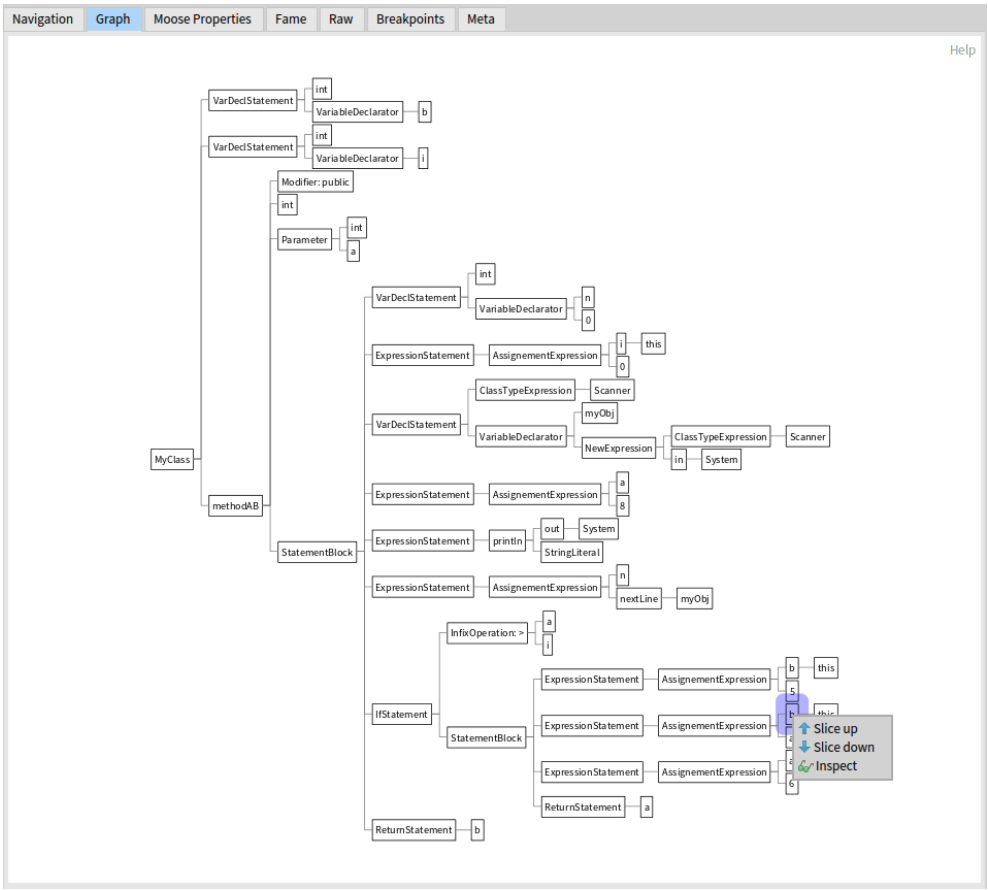



FIG. 6.7 : Opérations de flux de données - IHM.

Dans l'AST l'utilisateur peut faire les opérations suivantes, figure 6.7 :

- Slice up : voir le flux de la donnée dans le sens du haut, figure 6.8
- Slice down : voir le flux de la donnée dans le sens du bas, figure 6.9
- Inspect : permet d'inspecter et de voir la structure interne de l'entité séléctionnée.

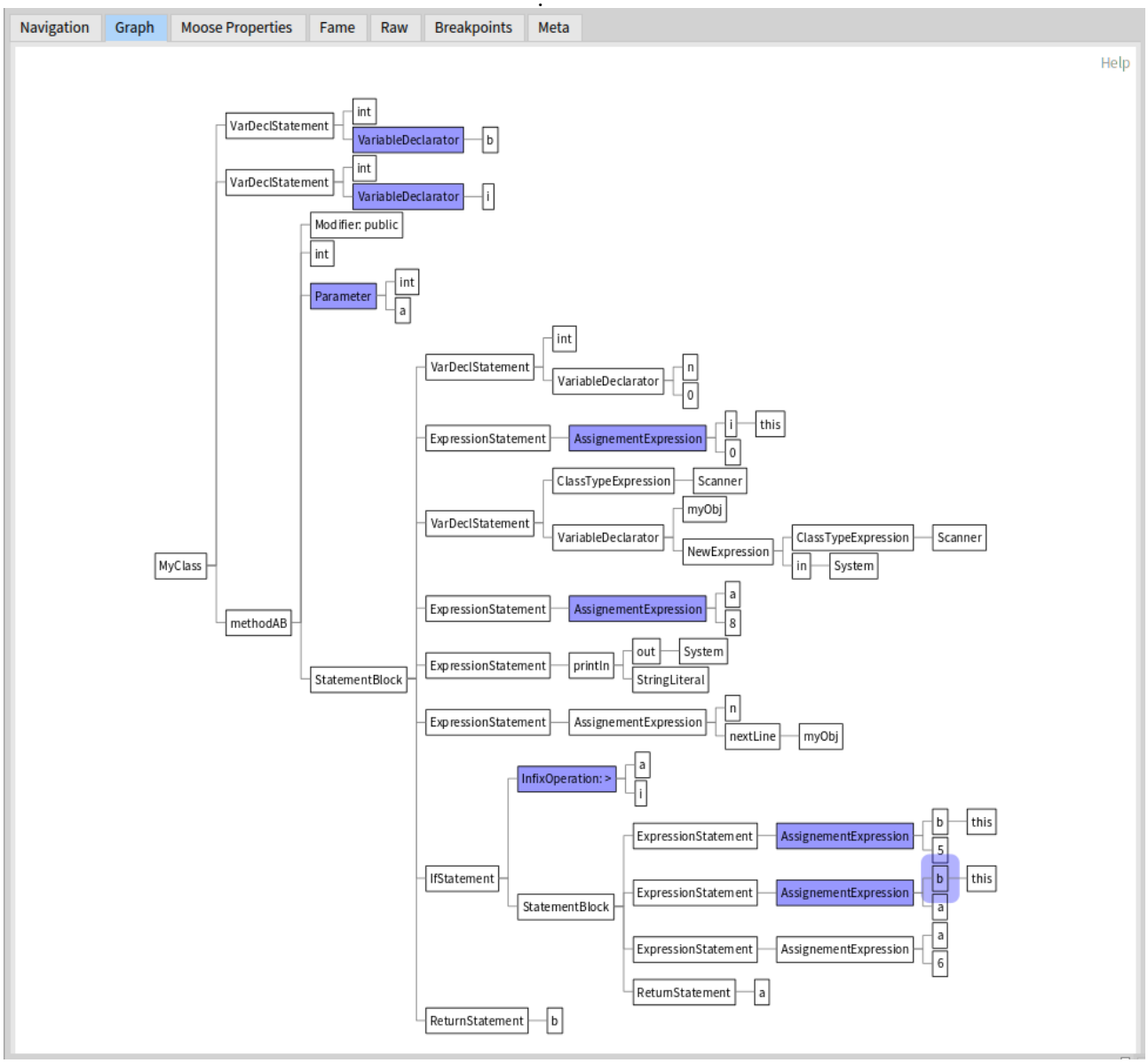



Fig. 6.8 : Opérations de flux de données - IHM.

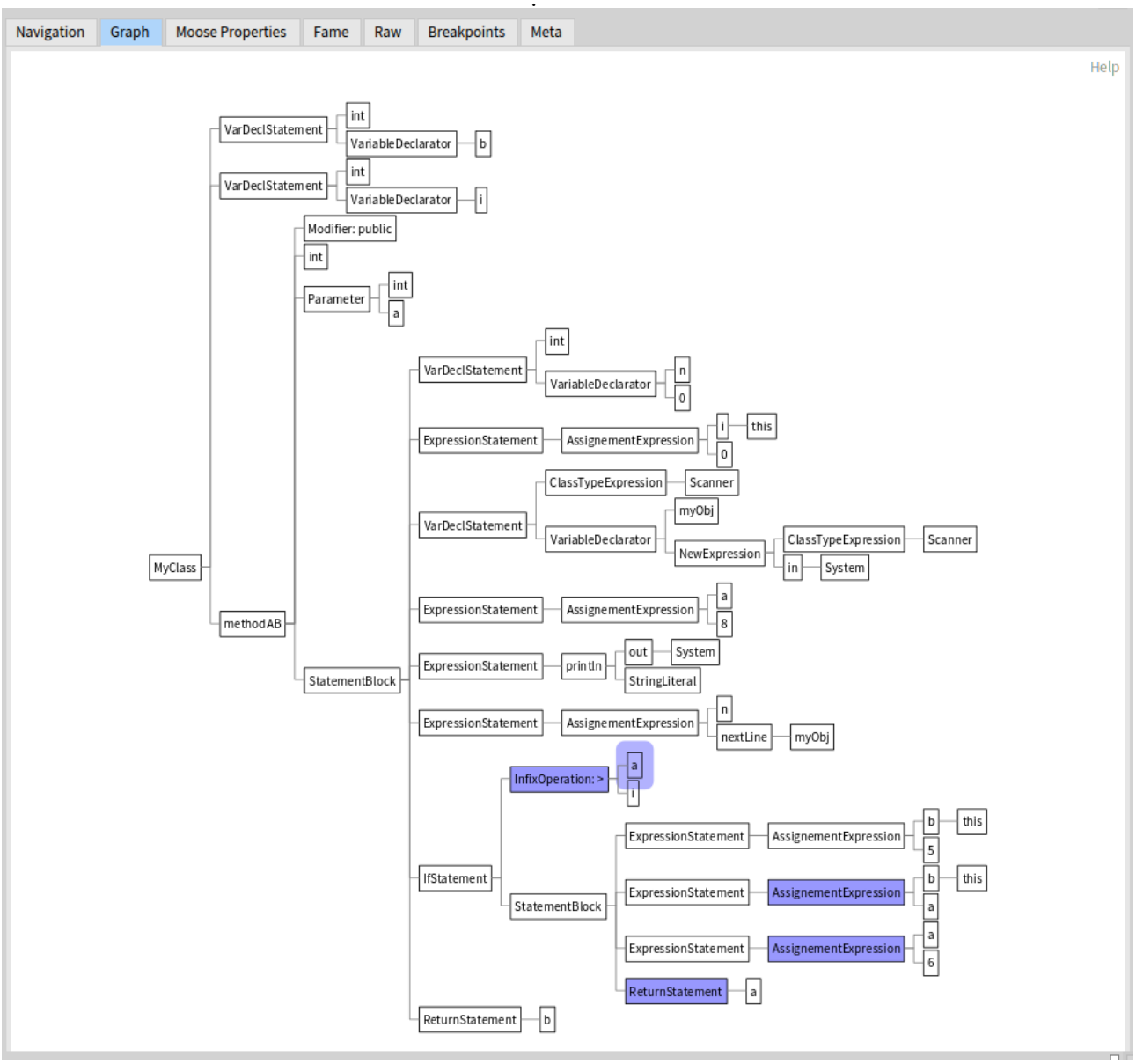



FIG. 6.9 : Opérations de flux de données - IHM.

### 6.2.2 Analyse des résultats

Dans cette rubrique, nous allons analyser les résultats obtenus lors de l'opération slicing up de l'exemple précédent. Nous avons l'ensemble des instructions obtenus est comme suit, figure 6.10.

**Algorithm :** Exemple pour la partie résultat

```
2   private attribute  b: integer
3   public attribute  i: integer
4   public method  integer exmple(a: integer)
5       var n: integer
6       this.i ← 0
7       a ←—— 0

9       if (a > i) then
10          this.b ← 5
11          this.b ← a
```

.

FIG. 6.10 : Résultat du slicing pour l'exemple précédent.

Nous remarquons dans figure ci-dessus que la variable b est définie dans deux endroits successifs (la ligne 10 et la ligne 11). Ceci est une anomalie de flux de données car il n'y a pas un référencement entre les définitions, nous pouvons le constater également dans la figure de l'AST 6.11 que nous avons deux noeud d'AssignementExpression pour la même donnée b l'une au dessus de l'autre.

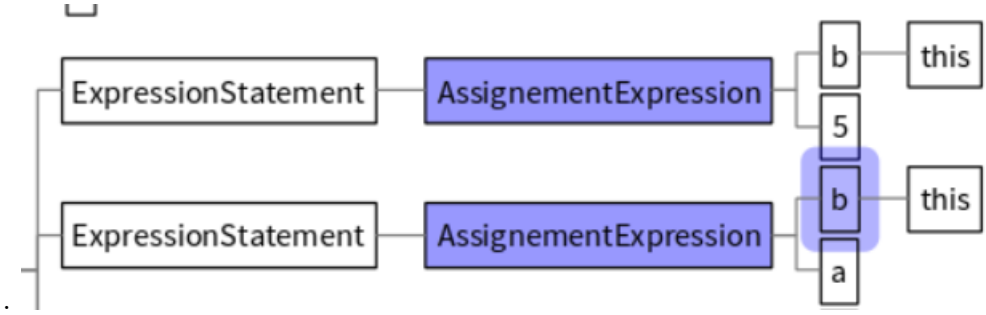



FIG. 6.11 : Définition successives de la variable "b" AST.

## 6.3 Conclusion

Dans de ce dernier chapitre nous avons présenté un exemple end-to-end d'un code Java afin d'analyser le flux de donnée d'une de ces variable. Nous clôturons l'exemple par une analyse de résultats où nous avons mis en évidence l'anomalie existante au niveau de ce code.

# Conclusion générale et perspectives

## Conclusion générale

Le projet qui nous a été confié par l'organisme d'accueil « Inria Lille - Nord Europe », au sein de l'équipe RMoD, consistait à concevoir et à réaliser une solution pour analyser le flux de données dans la méta-plateforme Moose. L'abstraction dans les méta-modèles nous permet de faire des analyses intéressantes, cependant dans certains cas on est obligé d'avoir plus de détails et de se rapprocher du code source. Pour analyser le flux de données, nous étions près du code source grâce au méta-modèle FAST et en utilisant la technique du program slicing nous avons pu représenter ce flux dans le programme.

Nous avons entamé notre rapport par une étude bibliographique sur les différents concepts de la maintenance, l'évolution et la rétro-ingénierie. Cette première étape nous a permis de bien comprendre le contexte de notre projet, afin d'acquérir plus de connaissances sur les méthodes d'analyse de flux de données et du slicing.

Ensuite, pour comprendre les spécificités et le contexte de notre organisme d'accueil, nous avons effectué une étude de l'existant. Une présentation de l'environnement de travail nous a permis de guider notre solution de manière à utiliser les méta-modèles existants.

Dans l'étape qui a suivi, et afin de mieux comprendre les spécifications, nous avons documenté et analysé le besoin. Suivi de l'étape de la conception où nous avons rajouté des concepts techniques de développement et modélisé notre solution.

Au cours de la dernière partie de notre projet, à savoir la réalisation et les tests de la solution, nous avons exploité les résultats de la phase de conception ainsi que les différents diagrammes que nous avons élaborés et nous avons utilisé certaines technologies qui nous ont permis d'obtenir un outil d'analyse visuel pour le flux de données.

Les contributions que notre projet a pu apporter peuvent se résumer dans les points suivants :

- Visualisation de l'AST (Abstract Syntaxe Tree) pour le méta-modèle FAST.
- Un outil permettant de représenter et analyser le flux de données dans les programmes.
- Implémenter le UI de l'importeur de modèle et le déployé dans la dernière version de Moose(v10)

## Perspectives

Cette solution se considère comme un début et une première version qui peut s'améliorer au fur et à mesure. Ainsi, plusieurs pistes d'amélioration sont envisageables dans le futur, nous pouvons citer les perspectives suivantes :

- **Améliorer la scalabilité :** notre solution propose de suivre le flot de données d'un modèle FAST, cependant elle admet seulement les modèles avec des attributs, des méthodes et une seule classe. Mais les modèles de Moose peuvent aller plus loin à travers les larges projets, pour cela nous pouvons étendre la solution afin de gérer les appels entres les différents classes et packages en utilisant d'autres types de program slicing.
- **Multi-Langage :** La solution utilisée actuellement est supportée par seulement le langage Java et cela à cause de la génération des modèles qui ne contiennent pas la position des identifiants, d'où l'ordre des instructions de l'AST n'est pas pris en compte. Notre solution peut être étendue en préservant la position de chaque identifiant dans le code source au niveau de la génération des modèles, comme c'est le cas pour le langage Java en utilisant VerveinJ.
- **Antécédent pour ma thèse de doctorat :** Ce PFE sera poursuivi par une thèse de doctorat intitulée "Traçage des règles métier dans le code source" avec mes promoteurs du stage où l'état de l'art, la solution et la connaissance achevé lors de ce projet sont considérés comme antécédents pour ce sujet de thèse.

## Appréciation personnelle

Ce stage a été une expérience très intéressante, tant sur le plan personnel que professionnel. C'était ma première expérience dans le domaine de la recherche, et il m'a donné la motivation pour poursuivre ma carrière dans le domaine de la recherche en général et dans l'ingénierie logicielle en particulier. Il m'a donné la chance de travailler avec des chercheurs hautement qualifiés et d'approfondir et de compléter mes connaissances dans le domaine du génie logiciel. De plus, ce stage était une étape complémentaire et importante pour ma formation d'ingénieur en Systèmes Informatiques et Logiciel où il m'a permis d'appliquer les connaissances acquises lors de ma formation à l'ESI et de faire face à des problèmes concrets du monde réel.

# Bibliographie


1219-1998, IEEE Standard (1998). “IEEE standard for software maintenance”. In : *IEEE Computer Society Press*. DOI : 10.1109/ieeestd.1993.115570.

Allen, Frances E. (juill. 1970). “Control Flow Analysis”. In : *SIGPLAN Not.* 5.7, p. 1-19. ISSN : 0362-1340. DOI : 10.1145/390013.808479.

Anquetil, Nicolas et al. (2020). “Modular Moose : A new generation software reverse engineering environment”. In : *arXiv preprint arXiv :2011.10975*.

Arthur, Lowell Jay (1988). *Software evolution : the software maintenance challenge*. Wiley-Interscience.

Battaglia, Marco, Giancarlo Savoia et John Favaro (1998). “Renaissance : A method to migrate from legacy to immortal software systems”. In : *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*. IEEE, p. 197-200.

Beck (2002). *Test Driven Development : By Example*. USA : Addison-Wesley Longman Publishing Co., Inc. ISBN : 0321146530.

Bennett, Keith (1995). “Legacy systems : Coping with success”. In : *IEEE software* 12.1, p. 19-23.

Bergel, A. (2016). *Agile Visualization*. Lulu.com. ISBN : 9781365314094.

Bézivin, Jean (2005a). “Model driven engineering : An emerging technical space”. In : *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, p. 36-64.

— (2005b). “On the unification power of models”. In : *Software & Systems Modeling* 4.2, p. 171-188.

Brambilla, Marco, Jordi Cabot et Manuel Wimmer (2017). “Model-driven software engineering in practice”. In : *Synthesis lectures on software engineering* 3.1, p. 1-207.

Bruneliere, Hugo (2018). “Generic model-based approaches for software reverse engineering and comprehension”. Thèse de doct. Nantes.

Chikofsky, Elliot J. et James H Cross (1990). “Reverse engineering and design recovery : A taxonomy”. In : *IEEE software* 7.1, p. 13-17.

Cocke, John (1969). *Programming languages and their compilers : Preliminary notes*. New York University.

Diehl, Stephan (jan. 2005). “Software visualization”. In : p. 718-719. DOI : 10.1145/1062455.1062634.

Eick, Stephen G et al. (2001). “Does code decay ? assessing the evidence from change management data”. In : *IEEE Transactions on Software Engineering* 27.1, p. 1-12.

El Emam, Khaled et A Günes Koru (2008). “A replicated survey of IT software project failures”. In : *IEEE software* 25.5, p. 84-90.

ESTEFAN, Jeff A et al. (2007). "Survey of model-based systems engineering (MBSE) methodologies". In : *Incose MBSE Focus Group* 25.8, p. 1-12.

FAVRE, Jean-Marie (2005). "Foundations of meta-pyramids : Languages vs. metamodels–episode ii : Story of thotus the baboon1". In : *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

FOSDICK, Lloyd D et Leon J OSTERWEIL (1976). "Data flow analysis in software reliability". In : *ACM Computing Surveys (CSUR)* 8.3, p. 305-330.

GAMMA, Erich et al. (1994). *Design Patterns : Elements of Reusable Object-Oriented Software*. 1re éd. Addison-Wesley Professional. ISBN : 0201633612.

GOVIN, Brice et al. (2017). "Managing an Industrial Software Rearchitecting Project With Source Code Labelling". In : *CSD&M 2017-Complex Systems Design & Management conference*.

HECHT, Matthew S (1977). *Flow analysis of computer programs*. Elsevier Science Inc.

HUTCHINSON, John et al. (2011). "Empirical assessment of MDE in industry". In : *Proceedings of the 33rd international conference on software engineering*, p. 471-480.

JONES, T. Capers (1984). "Reusability in Programming : A Survey of the State of the Art". In : *IEEE Transactions on Software Engineering* SE-10.5, p. 488-494. DOI : 10.1109/TSE.1984.5010271.

JONKERS, Hans, Marc STROUCKEN et Richard VDOVJAK (jan. 2006). "Bootstrapping Domain-Specific Model-Driven Software Development within Philips". In.

KAMKAR, Mariam (1995). "An overview and comparative classification of program slicing techniques". In : *Journal of Systems and Software* 31.3, p. 197-214.

KENT, Stuart (2002). "Model driven engineering". In : *International conference on integrated formal methods*. Springer, p. 286-298.

KHADKA, Ravi et al. (2014). "How do professionals perceive legacy systems and software modernization ?" In : *Proceedings of the 36th International Conference on Software Engineering*, p. 36-47.

KHEDKER, Uday P, Amitabha SANYAL et Bageshri KARKARE (2017). *Data flow analysis : theory and practice*. CRC Press.

KLEPPE, Anneke G et al. (2003). *MDA explained : the model driven architecture : practice and promise*. Addison-Wesley Professional.

KOSKINEN, Jussi et al. (2005). "Software modernization decision criteria : An empirical study". In : *Ninth European Conference on Software Maintenance and Reengineering*. IEEE, p. 324-331.

LEHMAN, M.M. (1980). "Programs, life cycles, and laws of software evolution". In : *Proceedings of the IEEE* 68.9, p. 1060-1076. DOI : 10.1109/PROC.1980.11805.

LOWRY, ES (1969). "CW MEPLOCK,"". In : *Object code optimisation" CACM Janvier*.

MARTIN, Robert C (2009). *Clean code : a handbook of agile software craftsmanship*. Pearson Education.

MELLOR, Stephen J et al. (2004). *MDA distilled : principles of model-driven architecture*. Addison-Wesley Professional.

MENS, Tom (jan. 2008). "Introduction and Roadmap : History and Challenges of Software Evolution". In : DOI : 10.1007/978-3-540-76440-3_1.

NIERSTRASZ, Oscar, Stéphane DUCASSE et Tudor GUNDEFINEDRBA (2005). "The Story of Moose : An Agile Reengineering Environment". In : *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT Inter-*

*national Symposium on Foundations of Software Engineering*. ESEC/FSE-13. Lisbon, Portugal : Association for Computing Machinery, p. 1-10. ISBN : 1595930140. DOI : 10.1145/1081706.1081707.

OSTERWEIL, Leon J et Lloyd D FOSDICK (1976). "DAVE—a validation error detection and documentation system for Fortran programs". In : *Software : Practice and Experience* 6.4, p. 473-486.

PROSSER, Reese T (1959). "Applications of boolean matrices to the analysis of flow diagrams". In : *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, p. 133-138.

REKOFF, Michael G (1985). "On reverse engineering". In : *IEEE Transactions on systems, man, and cybernetics* 2, p. 244-252.

REN, Yongchang et al. (2011). "Research on Software Maintenance Cost of Influence Factor Analysis and Estimation Method". In : *2011 3rd International Workshop on Intelligent Systems and Applications*, p. 1-4. DOI : 10.1109/ISA.2011.5873461.

RUGABER, Spencer et Kurt STIREWALT (2004). "Model-driven reverse engineering". In : *IEEE software* 21.4, p. 45-53.

SCHMIDT, Douglas C (2006). "Model-driven engineering". In : *Computer-IEEE Computer Society-* 39.2, p. 25.

SEKI, Hiroyuki et al. (1991). "On multiple context-free grammars". In : *Theoretical Computer Science* 88.2, p. 191-229. ISSN : 0304-3975.

SNEED, Harry (oct. 2004). "A cost model for software maintenance and evolution". In : p. 264-273. ISBN : 0-7695-2213-0. DOI : 10.1109/ICSM.2004.1357810.

SWANSON, E. Burton (1976). "The Dimensions of Maintenance". In : ICSE '76. San Francisco, California, USA : IEEE Computer Society Press, p. 492-497.

TERUEL, Camille, Stéphane DUCASSE et Marcus DENKER (2012). "Toward a modularization of Pharo : Analysis of the design space for a new module system." In : *9ème édition de la conférence MAnifestation des JEunes Chercheurs en Sciences et Technologies de l'Information et de la Communication-MajecSTIC 2012 (2012)*.

TESONE, Pablo et al. (avr. 2020). "A new modular implementation for Stateful Traits". In : *Science of Computer Programming* 195. DOI : 10.1016/j.scico.2020.102470.

TRIPATHY, Priyadarshi et Kshirasagar NAIK (2014). *Software evolution and maintenance : a practitioner's approach*. John Wiley & Sons.

VÖLTER, Markus et al. (2013). *Model-driven software development : technology, engineering, management*. John Wiley & Sons.

WEISER, Mark (1984). "Program slicing". In : *IEEE Transactions on software engineering* 4, p. 352-357.

WHITTLE, Jon, John HUTCHINSON et Mark ROUNCEFIELD (2013). "The state of practice in model-driven engineering". In : *IEEE software* 31.3, p. 79-85.

XU, Baowen et al. (2005). "A brief survey of program slicing". In : *ACM SIGSOFT Software Engineering Notes* 30.2, p. 1-36.

# Webographie


FOUNDATION, Wikipedia (avr. 1970). *Institut national de recherche en informatique et en automatique*. URL : https://fr.wikipedia.org/wiki/Institut_national_de_recherche_en_informatique_et_en_automatique#:~:text=L'Institut%20national%20de%20recherche,l'Innovation%20et%20du%20minist%C3%A8re.

— (mai 2022a). *Mixin*. URL : https://en.wikipedia.org/wiki/Mixin.

— (mars 2022b). *Reification (computer science)*. URL : https://en.wikipedia.org/wiki/Reification_(computer_science).

GEEKSFORGEEKS (nov. 2021). *Cost and efforts of software maintenance*. URL : https://www.geeksforgeeks.org/cost-and-efforts-of-software-maintenance/.

GFG, GeeksforGeeks (juin 2021). *Introduction of lexical analysis*. URL : https://www.geeksforgeeks.org/introduction-of-lexical-analysis/.

GROUP, ATLAS et LINA INRIA (fév. 2006). *ATL : Atlas Transformation Language ATL user manual*. URL : https://www.eclipse.org/atl/documentation/old/ATL_User_Manual[v0.7].pdf.

GURU, Refactoring (2022). *Visitor*. URL : https://refactoring.guru/design-patterns/visitor.

MILLER, Joaquin et Jishnu MUKERJI (juill. 2001). *Model Driven Architecture (MDA)*. URL : https://www.omg.org/cgi-bin/doc?ormsc%2F2001-07-01.

— (juin 2003). URL : http://www.omg.org/cgi-bin/doc?omg%2F+03-06-01.pdf.

OMG (2001). URL : http://xml.coverpages.org/OMG-MDA2001-03-08a.htm.

STÉPHANE DUCASSE, Tudor Girba (2016). *The Moose Book*. URL : https://github.com/girba/themoosebook.

*Wiki Trait* (déc. 2021). URL : https://en.wikipedia.org/wiki/Trait_(computer_programming).

WIKIPEDIA, Foundation (mars 2022). *Test driven development*. URL : https://fr.wikipedia.org/wiki/Test_driven_development.

# Annexes

# Annexe A

# Définitions et Terminologie

## A.1 Legacy

Les systèmes legacy sont « de grands systèmes logiciels auxquels on ne sait pas faire face mais qui sont vitaux pour notre organisation » Bennett 1995. Ils ont été écrits il y a des années en utilisant des techniques obsolètes, mais ils continuent de faire un travail utile. La migration et la mise à jour de ce bagage présentent un défi majeur dans le domaine de la rétro-ingénierie.

## A.2 Mixin

Dans les langages de programmation orientés objet, un mixin (ou mix-in) est une classe qui contient des méthodes à utiliser par d'autres classes sans avoir à être la classe parente de ces autres classes. La manière dont ces autres classes accèdent aux méthodes du mixin dépend du langage. Les mixins sont parfois décrits comme étant "inclus" plutôt qu'"hérités" (Foundation 2022a).

Les mixins encouragent la réutilisation du code et peuvent être utilisés pour éviter l'ambiguïté d'héritage que l'héritage multiple peut causer (le "problème du diamant"), ou pour contourner le manque de prise en charge de l'héritage multiple dans un langage. Un mixin peut également être considéré comme une interface avec des méthodes implémentées (Foundation 2022a).

## A.3 Parse Tree

Un parse tree est un arbre qui contient des détails sans rapport avec la signification réelle du programme, comme la ponctuation, dont le rôle est de diriger le processus de parse. Par exemple, les parenthèses de regroupement sont implicites dans la structure arborescente, qui peut être supprimée du parse tree. La suppression de ces détails superflus produit une structure appelée arbre de syntaxe abstraite (AST).

## A.4 Abstract syntax tree (AST)

Un arbre de syntaxe abstraite est une représentation arborescente de la syntaxe d'un code source qui a été écrit dans un langage de programmation donné. Chaque nœud de l'arbre désigne un élément apparaissant dans le code source. L'arbre est abstrait dans le sens où il peut ne pas représenter certains éléments qui apparaissent dans la code source. Un AST est souvent construit par un analyseur au niveau de la compilation.

## A.5 Open source software

Logiciel dont le code source est disponible pour que les utilisateurs et les tiers puissent l'inspecter et l'utiliser. Il est mis à la disposition du grand public avec des restrictions de propriété intellectuelle. Il est généralement utilisé comme synonyme de logiciel libre même si les deux termes ont des connotations différentes. "Open" met l'accent sur l'accessibilité au code source, tandis que "free" met l'accent sur la liberté de modifier et de redistribuer selon les termes de la licence d'origine.

## A.6 Migration

Dans la migration, un programme est transformé dans un autre langage au même niveau d'abstraction. Cela peut être une traduction entre dialectes, ou une traduction d'un langage à un autre.

## A.7 Paradigme

En science et en philosophie, un paradigme (/ pærəda m/) est un ensemble distinct de concepts ou de schémas de pensée, y compris des théories, des méthodes de recherche, des postulats et des normes pour ce qui constitue des contributions légitimes à un domaine. En informatique, les paradigmes sont un moyen de classer les langages de programmation en fonction de leurs fonctionnalités. Les langages peuvent être également classées en plusieurs paradigmes.