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

المدرسة الوطنية العليا للإعلام الآلي
(المعهد الوطني للتكوين في الإعلام الآلي سابقا)
École nationale Supérieure d'Informatique
ex. INI (Institut National de formation en Informatique)

Mémoire de Master

Pour l'obtention du diplôme de Master

Option : Systèmes Informatiques et Logiciels (SIL)

Plateformes et techniques de rétro-ingénierie orientée objet

Réalisé par :
M. Ahmed Zaki Bennecer

Promoteurs :
Mme. Anne Etien (INRIA - RMod)
M. Nicolas Anquetil (INRIA - RMod)

Encadré par :
Mme. Lamia Yessad (ESI)

Promotion : 2021/2022

Dédicaces

Je tiens, 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.

- 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 mon encadrante Lamia YESSAD pour ses qualités scientifiques et humaines. Elle a su m'écouter, m'orienter et me conseiller. Je salue sa disponibilité et je lui serai très reconnaissant.

Je tiens à exprimer mes remerciements les plus distingués à Anne ETIEN et Nicolas ANQUETIL mes promoteurs de stage pour leur encouragement et ainsi leur disponibilité incontestable.

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.

A la fin, 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 hopitaux 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 créant des outils de rétro ingénierie. Par exemple, la représentation des données pour permettre d'avoir une vue globale sur les modèles des logiciels.

Par conséquent, la contribution au développement des outils de la rétro-ingénierie est indispensable, pour faire face à la complexité de ces systèmes ainsi qu'aux nombreuses tâches de la réingénierie (migration, évolution, réarchitecture) quelque soit l'infrastructure utilisée. Donc, ces outils doivent s'adapter à un large éventail de situations et pouvoir réutiliser ces outils pour de nombreuses technologies, paradigmes et langages de programmation.

Dans ce mémoire, nous nous intéressons d'abord aux généralités liées à l'ingénierie logicielle. Nous présentons, ensuite, les briques principales composant la rétroingénierie, les approches, les techniques et les solutions utilisées dans ce domaine. Au final, nous mettrons l'accent sur la métaplateforme Moose et l'architecture la constituant. Le but de ce mémoire est de comprendre le contexte afin de proposer ultérieurement des solutions pour analyser le flux de données des programmes dans cette plateforme.

Mots clés : Ingénierie logicielle, Rétro-ingénierie, Architecture des logicielles, Modèles, Méta-modèles, 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 creating reverse engineering tools. For example, summarize data to allow a human to abstract a big picture understanding on software models.

Consequently, the contribution to the development of these tools is essential, to face the complexity of these systems as well as the many tasks of reengineering (migration, evolution, re-architecture) whatever the infrastructure used. Therefore, reverse engineering tools must adapt to a wide range of situations and be able to reuse these tools for many technologies, paradigms and programming languages.

In this essay, we will first focus on generalities related to software engineering. We will then see the main bricks making up reverse engineering, the approaches, the techniques and solutions used in this field. At the end, we will focus on the Moose platform and the architecture behind this platform. The goal is to propose solutions later to analyze the data flow of programs in this platform.

Keywords : Software engineering, Reverse-engineering, Software architecture, Models, Meta-models, Oriented Object 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 إنشاء أدوات الهندسة العكسية. على سبيل المثال ، تمثيل البيانات للسماح للإنسان بفهم الصورة الكبيرة لنماذج البرمجيات.

وبالتالي ، فإن المساهمة في تطوير هذه الأدوات ضرورية لمواجهة تعقيد هذه الأنظمة بالإضافة إلى المهام العديدة لإعادة الهندسة (الترحيل والتطور وإعادة الهندسة) مهما كانت البنية التحتية المستخدمة. لذلك ، يجب أن تتكيف أدوات الهندسة العكسية مع مجموعة واسعة من المواقف وأن تكون قادرة على إعادة استخدام هذه الأدوات للعديد من التقنيات والنماذج ولغات البرمجة.

في هذه المقال، سوف نركز أولاً على العموميات المتعلقة بهندسة البرمجيات. سنرى بعد ذلك كل الأساسيات التي تؤلف التي تشكل الهندسة العكسية ، والأساليب والتقنيات المستخدمة في هذا المجال. في النهاية ، سنركز على منصة Moose والمعمارية التي تقوم عليها هذه المنصة. الغرض من هذه الأطروحة هو اقتراح حلول لتحليل تدفق البيانات في هذه المنصة لاحقا.

كلمات مفتاحية : هندسة البرمجيات ، الهندسة العكسية ، معمارية البرمجيات ، النماذج ، النماذج الوصفية ، البرمجة الكائنية

Table des matières

Résumé . . . . . **I**

Abstract . . . . . **II**

III . . . . . **ملخص**

Introduction générale **1**

1 Généralités **3**
1.1 Introduction 4
1.2 Cycle de vie d'un logiciel 4
1.2.1 Définition 4
1.2.2 Phases impliquées dans le SDLC 4
1.2.3 Méthodologies standards du SDLC 5
1.3 Maintenance et évolution 10
1.3.1 Définitions 10
1.3.2 Importance 11
1.3.3 Difficultés et coûts 11
1.4 Conclusion 12

2 Rétro-ingénierie, techniques et solutions **13**
2.1 Introduction 14
2.2 Modélisation, modèles et méta-modèles 14
2.2.1 Définition générale 14
2.2.2 Model-Driven Architecture 15
2.2.3 Concepts fondamentaux 15
2.2.4 Norme et spécifications standard 20
2.2.5 Exemples de méta-modèles 21
2.3 Rétro-ingénierie, techniques et solutions 23
2.3.1 Model-driven Reverse Engineering 23
2.3.2 Étapes clés de la rétro-ingénierie 24
2.3.3 Techniques utilisées pour la rétro-ingénierie 25
2.4 Solutions existantes pour la rétro-ingénierie 31
2.4.1 Solutions spécifiques 31
2.4.2 Solutions génériques 33
2.5 Conclusion 34

3 Moose **36**

3.1 Introduction 37
3.2 Moose : Difficultés & Objectifs 37
3.2.1 Moose 37
3.2.2 Difficultés 38
3.2.3 Objectifs 38
3.3 FAMIX & FAST 39
3.3.1 FAMIX 39
3.3.2 FAST 39
3.4 Architecture et Outils 40
3.4.1 Architecture 40
3.4.2 Outils réutilisables de Moose 42
3.5 Conclusion 42

Conclusion générale et perspectives 43

Annexes 53

A Définitions et Terminologie 54

Table des figures

1.1 Les phases impliquées dans un modèle SDLC [MD13]. 5
1.2 Le modèle en cascade. 6
1.3 Le modèle spiral de Boehm [BH00] 7
1.4 La méthode RAD - Approche de prototypage - [Rup10] 8
1.5 Le modèle incrémental [MD13] 8

2.1 Exemple de modèle "L'entité Personne peut posséder des voitures" 16
2.2 Exemple d'un méta-model. 17
2.3 Model-Driven Architecture [GI06]. 18
2.4 Pyramide à quatre niveaux montrant chacun des niveaux OMG. 19
2.5 La transformation modèle à modèle. 19
2.6 La transformation modèle à texte - Génération de code. 20
2.7 Couches, packages et séparation du méta-modèle KDM [OMG16a]. 21
2.8 Le framework de modélisation ASTM [OMG11] 22
2.9 Le modèle de ré-ingénierie Horseshoe [TN14] 24
2.10 L'architecture de l'analyseur lexical. 26
2.11 Classification des analyseurs Bottom-Up. 27
2.12 Pipeline typique des outils de rétro-ingénierie [Die05]. 30

3.1 Le projet Moose Open source - Image capturé de Github. 37
3.2 FAST meta-model. 40
3.3 Nouvelle architecture de Moose [Anq+20]. 41
3.4 Call Graph Browser, exemple d'un outils spécialisé de Moose [Anq+20]. . . 42

Liste des tableaux

1.1 Comparaison de différents modèles SDLC à plusieurs caractéristiques [RR13] 9

2.1 Solutions Spécifiques MDRE. 32
2.2 Solutions génériques MDRE. 34

Liste des sigles et acronymes

GL	*Génie Logiciel*
SDLC	*Software Development LifeCycle*
ISO	*International Organization for Standardization*
IEC	*International Electrotechnical Commission*
AST	*Abstract Syntax Tree*
RAD	*Rapid Application Development*
MDA	*Model-Driven Architecture*
OMG	*Object Management Group*
UML	*Unified Modeling Language*
MOF	*Meta Object Facility*
OCL	*Object Constraint Language*
XML	*eXtensible Markup Language*
SysML	*Systems Modeling Language*
QVT	*Query/View/Transformation*
CFA	*Control Flow Analysis*
DFA	*Data Flow Analysis*
ASG	*Abstract Semantic Graph*
JIT	*Just In Time*

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 [Leh80]. Et comme l'économie dépend de ces logiciels [Kos+05], 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 [EK08]. Dans le cas parfait, il faudra avoir une vision plus claire sur le projet et anticiper les problèmes à venir. Mais, en pratique, les ingénieurs ont une vue partielle et limitée sur la complexité et les potentielles conséquences de leurs produits logiciels [Kha+14].

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 [Hut+11 ; WHR13]. 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.

Après avoir l'abstraction des logiciels à travers la modélisation, la compréhension du système sera possible en utilisant les techniques de rétro-ingénierie. Ces techniques consistent à examiner, comprendre et restructurer les différentes parties du système analysé. Généralement, ces techniques sont groupées dans une plateforme ou une solution de rétro-ingénierie afin de pouvoir les utiliser ensemble pour aboutir aux résultats désirés.

L'objectif de ce mémoire de master est de fournir une vision complète de la rétro-ingénierie logicielle allant ainsi des définitions les plus simples aux techniques et solutions les plus récentes.

Organisation du mémoire

Ce mémoire est organisé en trois chapitres :

Le premier chapitre "**Généralités**" a pour objectif de mettre en avant quelques généralités liées au génie logiciel. Des notions importantes et nécessaires à la bonne compré-

hension du contexte de la rétro-ingénierie. En effet, le logiciel et son cycle de vie seront présentés. Par la suite, nous allons étudier *la maintenance et l'évolution* ainsi que leur rôle dans le domaine de l'ingénierie logicielle.

Le second chapitre "**Rétro-ingénierie, techniques et solutions**" a pour but de présenter la rétro-ingénierie. Nous verrons le principe de cette ingénierie, les modèles et les méta-modèles. Nous terminerons ce chapitre par les techniques et les solutions utilisées pour la rétro-ingénierie.

Le dernier chapitre "**Moose**" a pour but d'illustrer une des solution de rétro-ingénierie et élaborer son architecture, c'est la plateforme d'analyse et de maintenance *Moose*. En effet, nous présenterons l'architecture globale, ses outils et leurs caractéristiques et ainsi ses méta-modèles existants.

Le mémoire est clôturé par une conclusion générale dont l'objectif principal est de résumer et rappeler tout ce qui a été vu au sein de ce mémoire.

Chapitre 1

Généralités

1.1 Introduction

Dans ce premier chapitre, nous allons présenter les notions de bases liées au génie logiciel (GL), nous verrons le cycle de vie du développement logiciel (SDLC) et ses méthodologies standards. Ensuite, nous expliquerons la maintenance et l'évolution des systèmes logiciels. Ce qui servira à comprendre la motivation derrière les techniques et les plateformes de rétro-ingénierie.

1.2 Cycle de vie d'un logiciel

1.2.1 Définition

Le cycle de vie du développement logiciel (SDLC) est un processus suivi pour un projet logiciel, au sein d'une organisation. Il s'agit d'un plan détaillé décrivant comment recueillir les exigences, analyser les activités, concevoir des systèmes, maintenir ou améliorer des logiciels spécifiques [KGK07]. Le cycle de vie définit les différentes étapes que devraient suivre un logiciel pour garantir sa qualité.

Le SDLC vise à produire un logiciel de haute qualité qui répond aux attentes des clients et atteint l'achèvement dans les délais et les estimations de coûts. Il est également appelé processus de développement logiciel. Les tâches à effectuer à chaque étape du processus du développement logiciel sont précisément définies dans la norme ISO/IEC 12207 [Pre01].

1.2.2 Phases impliquées dans le SDLC

La figure 1.1 ci-après est une représentation graphique des phases qui sont généralement[1] présentes dans chaque modèle de cycle de vie de développement logiciel. Ces phases sont :

- Comprendre le problème (A partir de la collecte des besoins),
- Décider d'un plan pour une solution (Conception),
- Réaliser la solution envisagée,
- Tester le programme réalisé,
- Implémentation & maintenance du produit.

[1] Dans la littérature on peut trouver plusieurs phases, mais la raison est expliquée dans le paragraphe qui suit 1.2.2

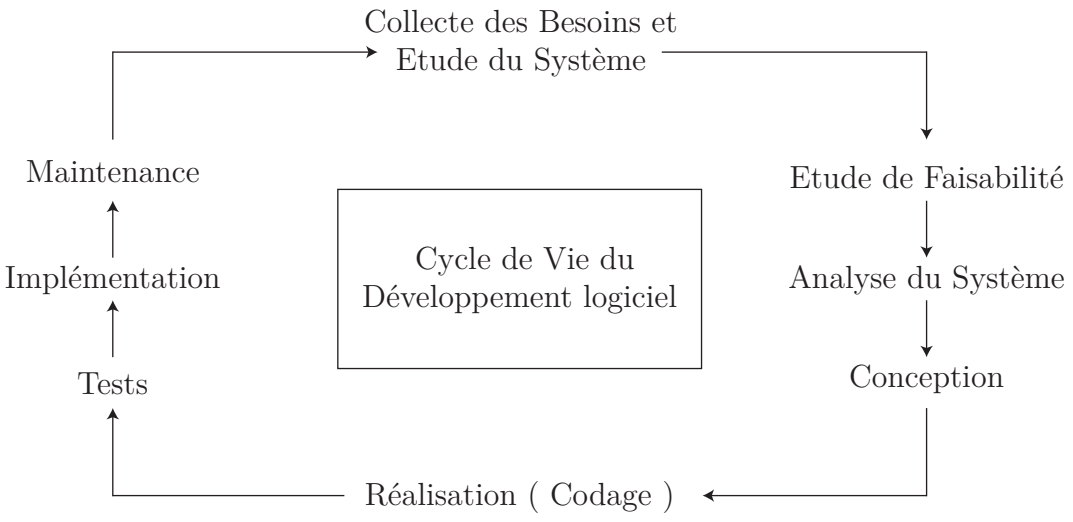


Fig. 1.1 : Les phases impliquées dans un modèle SDLC [MD13].

Pour les systèmes larges, chaque activité peut être extrêmement complexe, c'est pourquoi des méthodologies et des procédures sont nécessaires pour exécuter cette activité efficacement et correctement. De plus, chacune des activités de base peut être si importante qu'elle ne peut être traitée en une seule étape et doit être divisée en étapes plus petites. Par exemple, la conception d'un grand système logiciel est toujours divisée en plusieurs phases de conception distinctes, allant d'une conception de très haut niveau spécifiant uniquement les composants du système à une conception détaillée où la logique des composants est élaborée. En plus des activités réalisées pendant le développement du logiciel, certaines activités sont réalisées après que le développement du logiciel soit terminé. Par exemple, la maintenance est une activité qui commence après le développement. Le logiciel doit être maintenu non pas parce que certains de ses composants sont obsolètes et doivent être remplacés, mais parce qu'il reste souvent des erreurs répercutées dans le système [MD13] qui doivent être supprimées plus tard au fur et à mesure qu'elles sont découvertes. Par conséquent, la maintenance est inévitable pour les systèmes logiciels.

1.2.3 Méthodologies standards du SDLC

Méthode Waterfall :

L'approche Waterfall (en cascade) a été proposée par Royce en 1970 [Roy+70], qui est un modèle de cycle de vie de développement logiciel séquentiel linéaire (SDLC). Les différentes phases suivies sont la collecte des besoins, l'analyse des exigences, la conception, le codage, les tests, maintenance. La mise en œuvre de cette méthodologie est faite de manière à ce que la phase ne soit pas répétée une fois de plus et que le développement ne passe pas à la phase suivante tant que la phase précédente n'est pas complètement terminée. Par conséquent, ce modèle n'est pas très utile lorsque les exigences du projet sont de nature dynamique [MD13].

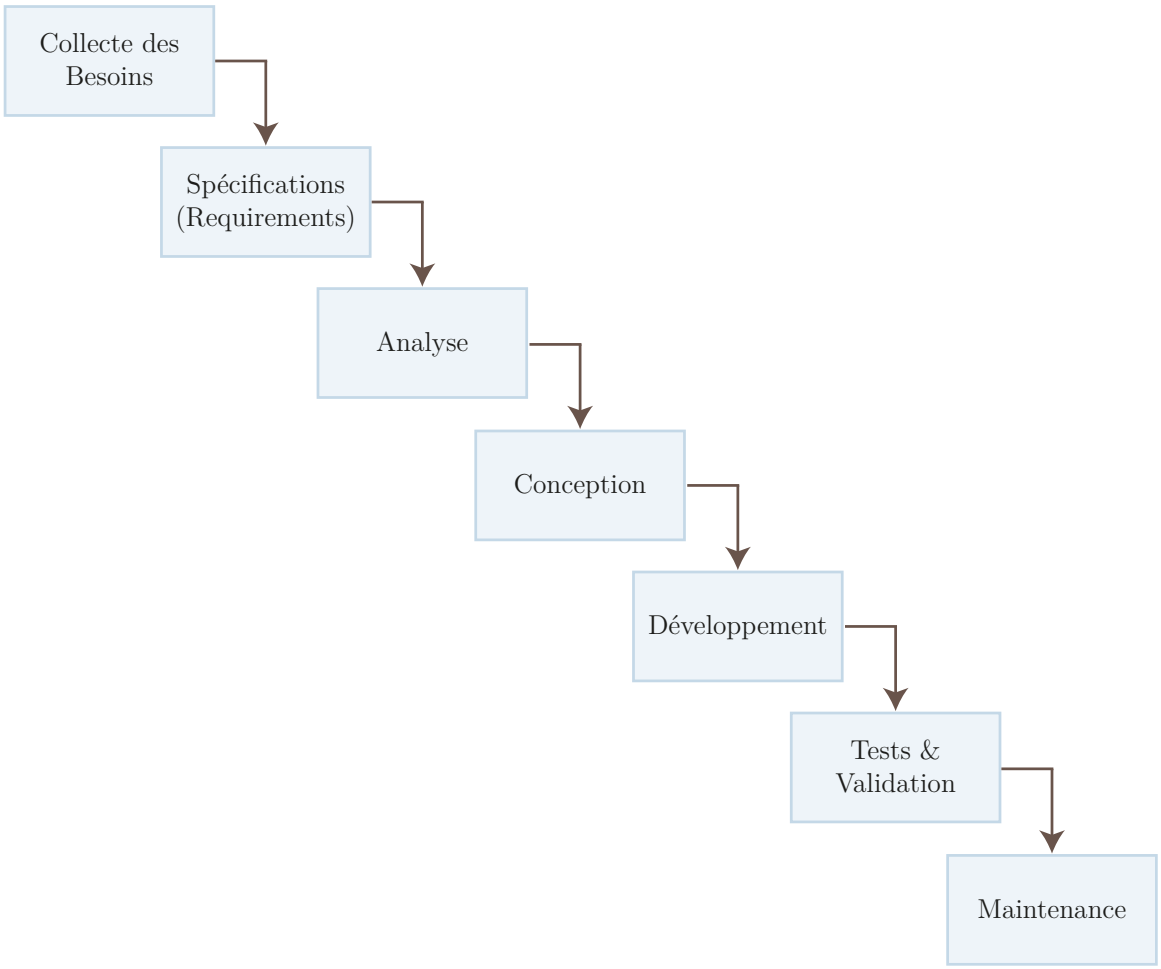


FIG. 1.2 : Le modèle en cascade.

Processus en Spirale :

En réponse aux faiblesses et aux échecs des méthodes Waterfall, de nombreux autres modèles ont été développés qui ajoutent une certaine forme d'itération au processus de développement logiciel. Dans l'approche en Spirale du SDLC, comme dans la figure 1.3, l'équipe de développement commence avec un petit ensemble d'exigences et passe par chaque phase de développement (sauf l'installation et la maintenance) pour cet ensemble d'exigences [MS12]. Sur la base des leçons tirées de l'itération initiale, l'équipe de développement ajoute des fonctionnalités pour des exigences supplémentaires dans des « spirales » sans cesse croissantes jusqu'à ce que l'application soit prête pour la phase d'installation et de maintenance.

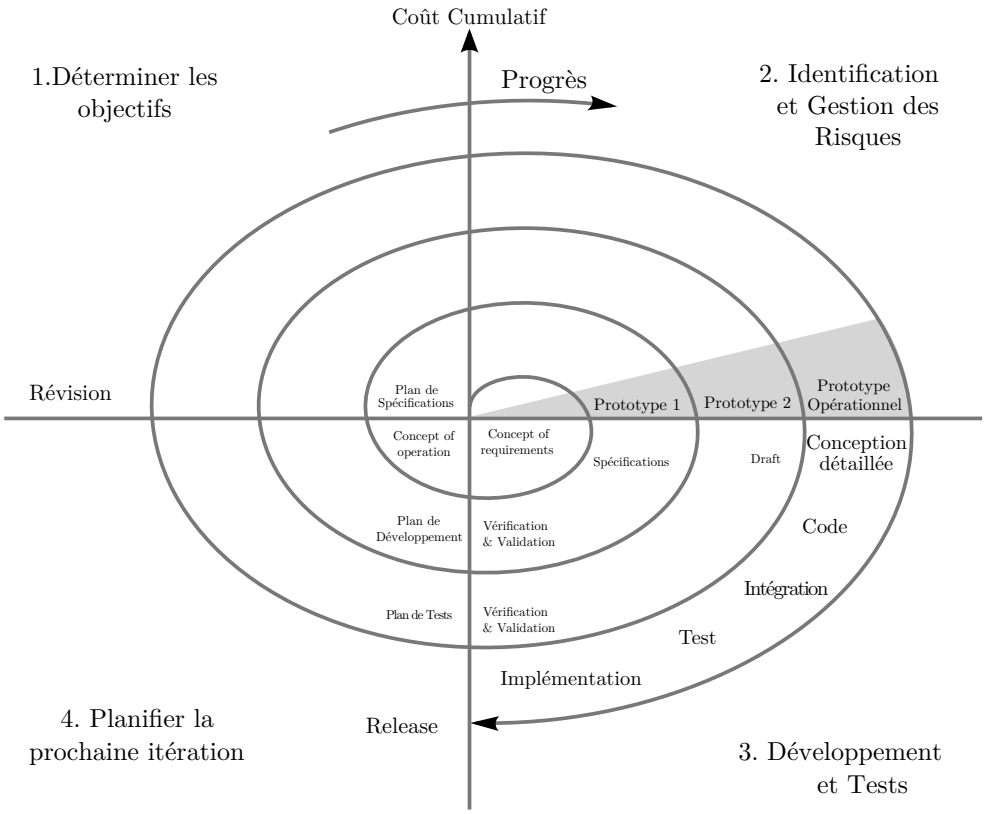


Fig. 1.3 : Le modèle spiral de Boehm [BH00] .

Méthode RAD :

Si les exigences sont bien comprises et que la portée du projet est limitée, le processus de développement rapide d'applications (RAD) permet à une équipe de développement de créer un « système entièrement fonctionnel » dans des délais très courts (par exemple, 60 à 90 jours). Développé principalement par James Martin en 1991, c'est une méthodologie qui utilise le prototypage comme mécanisme, comme illustré à la figure 1.4, pour le développement itératif.

Récemment, RAD est devenu utilisé dans un sens générique plus large qui englobe une variété de techniques visant à accélérer le développement de logiciels. Parmi celles-ci : *Extreme Programming (XP), Lean Development (LD), Joint Application Development (JAD) et Scrum.*

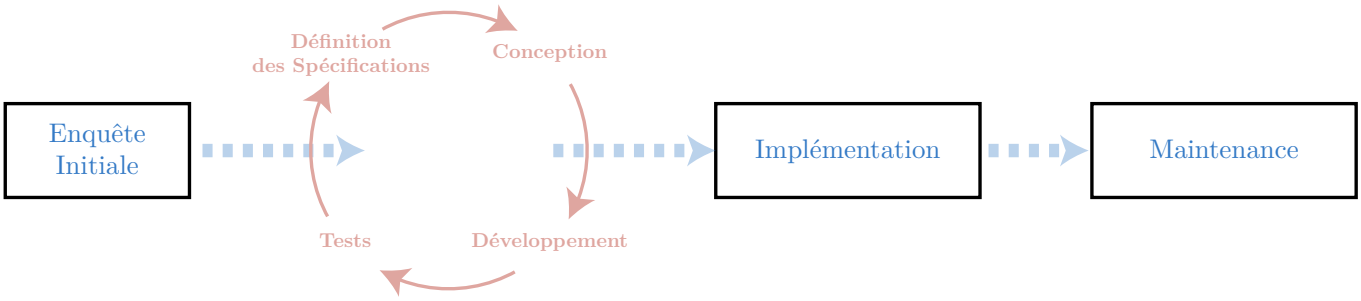


FIG. 1.4 : La méthode RAD - Approche de prototypage - [Rup10] .

Modèle incrémental :

L'approche incrémentale, figure 1.5, est une évolution de la méthode en cascade qui comporte un certain nombre d'itérations et après chaque itération, nous obtenons un produit fonctionnel (artefact). Au départ, les exigences sont analysées et les phases de conception, de codage et de test sont établis pour obtenir le premier artefact. Le produit de travail de la première itération est remis au client et après avoir obtenu les commentaires du client, le produit est modifié pour avoir le produit de la deuxième itération et ainsi de suite. L'approche incrémentale réduit le coût de la construction d'un prototype car au lieu de construire un prototype, on s'adapte aux modifications apportées au produit et çela dépend du feedback de client. Cependant, en raison des itérations, le calendrier peut être dépassé, ce qui à son tour affecte le coût et l'effort.

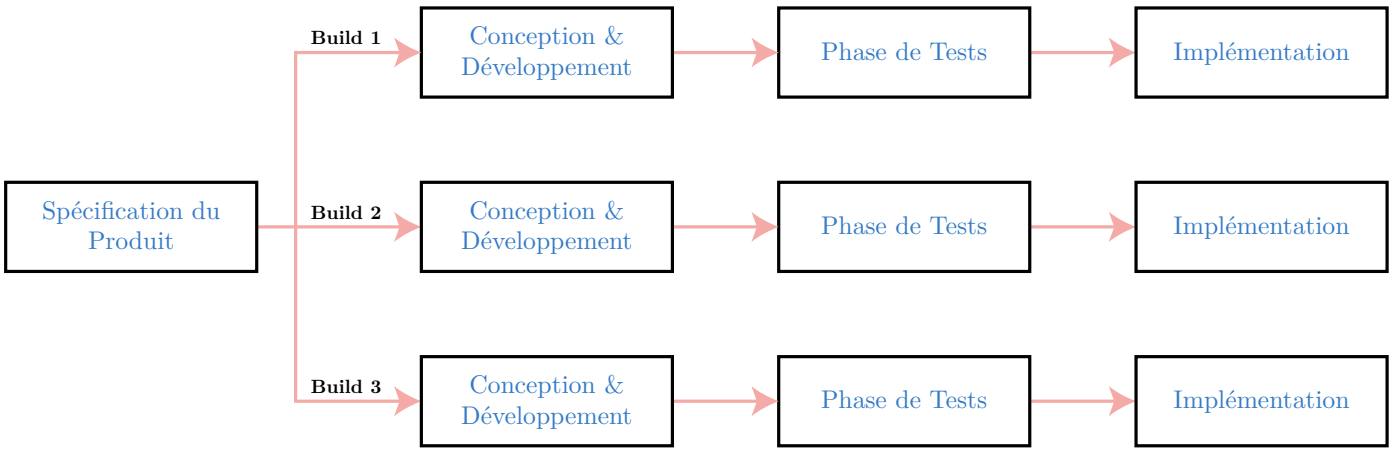


FIG. 1.5 : Le modèle incrémental [MD13] .

Comparaison entre les méthodologies

Comme il existe différentes méthodologies de cycle de vie de développement logiciel, chacune a ses propres avantages et inconvénients. Une comparaison des différents modèles est représentée dans le tableau suivant 1.1 sur la base de certaines caractéristiques.

Caractéristiques\Modèles	Waterfall	Incrémental	RAD	Spiral
Spécifications claires	Niveau initial	Niveau initial	Niveau initial	Niveau initial
Retour de l'utilisateur	Non	Oui	Oui	Non
Rapidité du changement	Faible	Élevé	Non [2]	Élevé
Prévisibilité	Faible	Faible	Faible	Moyen
Implémentation pratique	Non	Faible	Non	Moyen
Existence des variations	Oui V-Model	Non	Non	Oui Win Win Spiral [Sac+11]
Compréhension	Simple	Intermédaire	Intermédiaire	Difficile
Préconditions	Définition claire des spécifications	Le core du produit doit être clair	Idée claire	Non
Utilisabilité	Simple	Moyen	Moyen	Moyen
Priorité du client	/	/	/	Intermédiaire
Approche industrielle	Simple	Simple	Moyen	Moyen
Coût	Faible	Faible	Elevé	Très élevé
Organisation des ressources	Oui	Oui	Oui	Non
Élasticité	Non	Non	Oui	Non

Tab. 1.1 : Comparaison de différents modèles SDLC à plusieurs caractéristiques [RR13]

1.3 Maintenance et évolution

Aujourd'hui dans les organisations, les activités de maintenance et d'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.

1.3.1 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 [Swa76] :

- **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* [TN14], ce qui 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 » [Eic+01].

- **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 [TN14].

- **La maintenance logicielle évolutive** [3]

[3] En fait, la maintenance logicielle évolutive est ce qu'on appelle l'évolution logicielle qui sera définit dans le paragraphe suivant 1.3.1.

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 [Art88]. 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.

1.3.2 Importance

Dans le monde d'aujourd'hui, la majeure partie du budget logiciel est consacrée à la modification de logiciels existants (maintenance évolutive) plutôt qu'au développement de nouveaux logiciels.

Les produits logiciels réussis sont généralement utilisés beaucoup plus longtemps qu'en cours de développement, ce qui rend la maintenance encore plus cruciale. Si le logiciel ne continue pas à s'adapter aux changements des besoins et de l'environnement, il devient progressivement moins utile (1ère Loi de Lehman, évolution logicielle) [Leh96]. Les changements sont inévitables et se produisent pour plusieurs raisons [Som15] :

- De nouvelles exigences émergent lors de l'utilisation du logiciel ;
- L'environnement industriel change ;
- Les défauts doivent être réparés ;
- De nouveaux ordinateurs ou équipements sont ajoutés au système ;
- La croissance de la base d'utilisateurs rend les performances ou la fiabilité insuffisantes.

1.3.3 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 [Sne04].

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 [Gee21] :

Coûts techniques :

Certains facteurs de coûts techniques courants sont [Gee21] :

- **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 différente.
- **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, l'adaptation pour maintenir la charge de travail sera énorme [Mar09].

Coûts non-techniques :

D'autres facteurs de coûts humains courants sont [Ren+11] :

- **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+11].

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.

1.4 Conclusion

Pour conclure, nous avons présenté dans ce chapitre le cycle de vie du développement de logiciel. Nous sommes passés par les différentes étapes du SDLC ainsi que ses méthodologies standards, et finalement, nous avons présenté les notions de maintenance et d'évolution des systèmes logiciels. L'objectif de ce chapitre est d'introduire des concepts fondamentaux pour pouvoir présenter la rétro-ingénierie d'une manière détaillée dans le chapitre suivant.

Chapitre 2

Rétro-ingénierie, techniques et solutions

2.1 Introduction

La rétro-ingénierie est presque aussi ancienne que l'informatique elle-même. Ciblant initialement l'analyse du hardware [Rek85], elle a rapidement étendu son champ d'application pour se concentrer également sur les systèmes logiciels [CC90]. 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* [CC90].

Dans ce chapitre, nous introduisons tout d'abord les connaissances de base qui sont pertinentes afin de bien comprendre la modélisation. Ensuite, nous présenterons la rétro-ingénierie avant de passer sur ses techniques, notamment ses solutions spécifiques et génériques.

2.2 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. On terminera par présenter les normes et les techniques de ce domaine.

2.2.1 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 [Ken02 ; Sch06 ; Béz05a], Model-based Engineering [Est+07], Model-driven Development [Völ+13], etc.) peuvent être toutes regroupées sous le nom générique de modélisation (Modeling [BCW17]). 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 [Bru18] :

- Exploiter les connaissances génériques existantes (par exemple sur les processus, les technologies et les données) afin de favoriser l'extensibilité et la réutilisabilité.

[1] Le terme legacy est bien définit dans l'annexe. C'est un terme qui est utilisé souvant dans le domaine de la rétro-ingénierie

- 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.

2.2.2 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 [OMG01].

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 [MM03], 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,
- et transformer la spécification du système en une pour une plate-forme particulière.

L'initiative MDA est fortement liée à UML, et c'est la première initiative pilotée par un modèle (voir [MM03 ; MM01]).

MDA, ainsi que toutes les initiatives OMG, suivent le principe "tout est un modèle" comme indiqué dans [Béz05b]. La sous-section suivante 2.2.3 décrit la notion de modèle et ses concepts associés.

2.2.3 Concepts fondamentaux

Modèle :

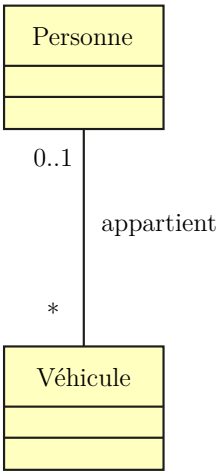
Plusieurs définitions de modèle [Kle+03 ; Mel+04 ; MM03] sont résumées dans [Fav05]. Par exemple, dans le contexte de la norme UML, le terme modèle est défini comme suit :

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

D'autres auteurs définissent le terme en fonction d'une langue [Kle+03] :

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

L'OMG clarifie également dans [MM03] qu'« un modèle est souvent présenté comme une combinaison de dessins et de texte. Le texte peut être dans un langage de modélisation ou dans un langage naturel.»



.

FIG. 2.1 : Exemple de modèle "L'entité Personne peut posséder des voitures" .

La figure 2.1 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 [Fav05] :

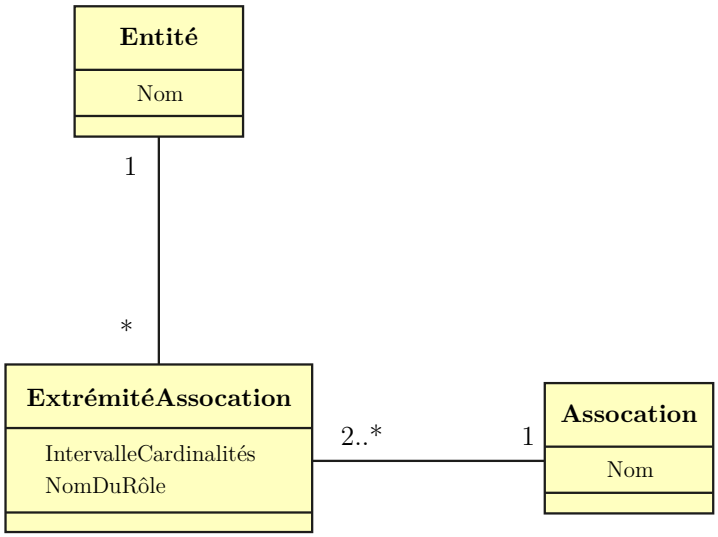
"*Un méta-modèle est un modèle d'un langage de modélisation.*"

Ainsi, les méta-modèles sont là pour décrire un langage de modélisation. Favre a déclaré dans [Fav05] 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 2.2 montre un exemple de méta-modèle. Le modèle présenté précédemment à la figure 2.1 est conforme à ce méta-modèle.



.

Fig. 2.2 : Exemple d'un méta-model.

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 2.2.3.

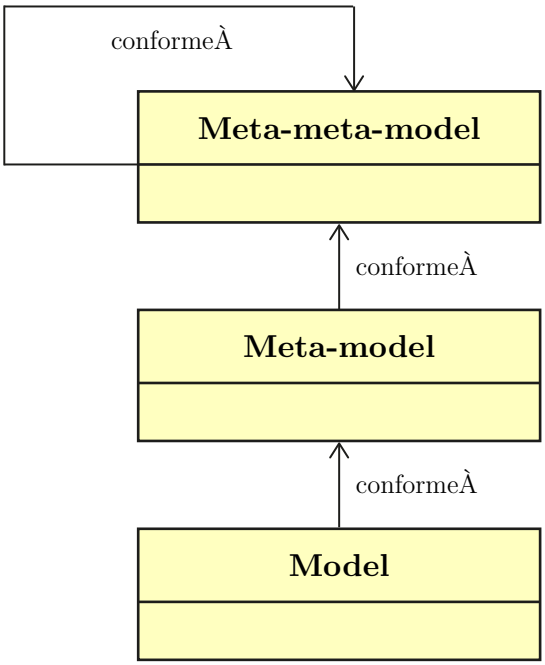
Méta-méta-modèle :

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



.

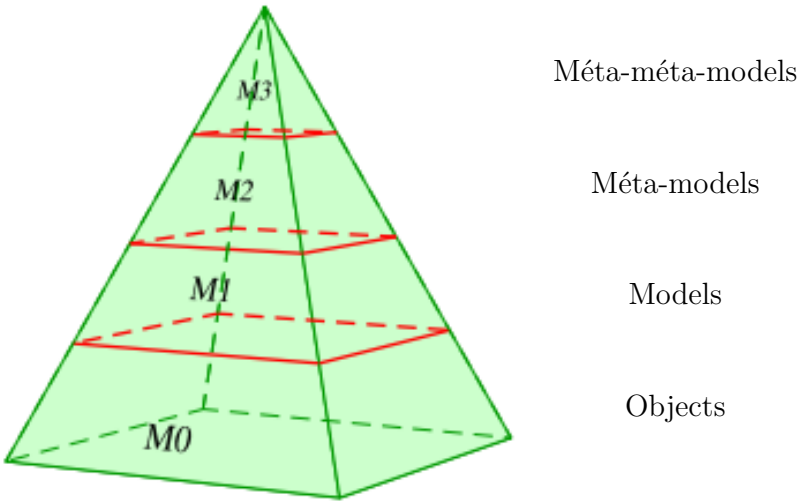
Fig. 2.3 : Model-Driven Architecture [GI06].

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 2.4. Ces quatre niveaux déterminent ce que l'on appelle la « pyramide de méta-modélisation à quatre couches » [Fav05]. 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 [JSV06].

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).

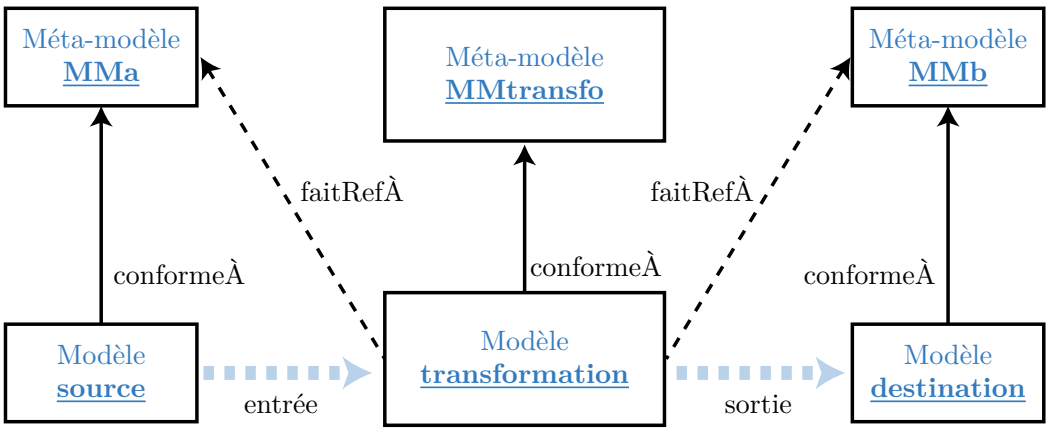


.

FIG. 2.4 : Pyramide à quatre niveaux montrant chacun des niveaux OMG.

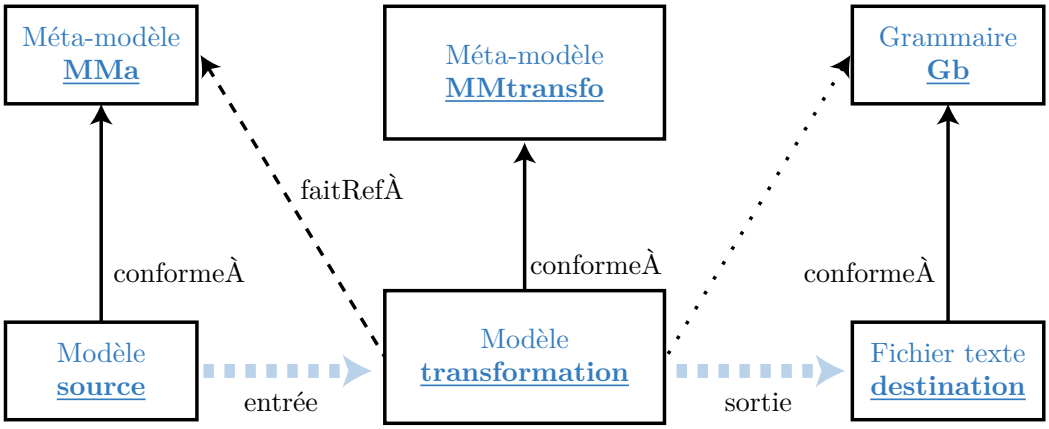
Transformation des modèles :

La transformation de modèles est un autre concept fondamental en modélisation [SK03 ; MV06]. Selon l'état de l'art dans ce domaine, il existe deux principaux types de transformation de modèles : *les transformations modèle à modèle*, représenté à la figure 2.5, et *les transformations modèle à texte*, représenté à la figure 2.6 [Bru18]. Alors que le premier type est communément appelé transformation de modèle, le second est très souvent appelé génération de code [Her03 ; KT08].



.

FIG. 2.5 : La transformation modèle à modèle.



.

FIG. 2.6 : La transformation modèle à texte - Génération de code.

2.2.4 Norme et spécifications standard

En complétant ce qui a été introduit dans la section 2.2.2, MDA s'appuie sur un ensemble de langages de modélisation standards à utiliser ensemble afin de matérialiser sa vision [Bru18]. Parmi toutes les spécifications standards disponibles fournies par l'OMG, on peut notamment lister les suivantes[2] :

- MOF [OMG16b], un méta-méta-modèle de base qui définit les types de référence pour spécifier divers méta-modèles (y compris ceux listés juste après),
- Object Constraint Language (OCL) [OMG14a], un langage textuel déclaratif qui permet de définir des contraintes et des requêtes (et donc de naviguer) sur n'importe quel méta-modèle ou modèle basé sur MOF,
- XML Metadata Interchange (XMI) [OMG14b], un format commun d'échange de modèles utilisés pour sérialiser puis partager de manière uniforme tous les métamodèles et modèles basés sur MOF,
- Langages de modélisation à usage général comprenant plus particulièrement
  - UML [OMGb], un langage de modélisation dédié au génie logiciel et à la conception de systèmes logiciels (couvrant à la fois les aspects structurels et comportementaux de tels systèmes),
  - Systems Modeling Language (SysML) [OMGa], un langage de modélisation (défini comme une extension d'UML via son mécanisme de profil) dédié à l'ingénierie système et à la conception de systèmes complexes,

[2] **Celles-ci correspondent à des normes bien connues et largement utilisées.**

- Langages de transformations génériques (au sens d'indépendants du métamodèle) et complémentaires
  - Query/View/Transformation (QVT) [OMG16c], trois langages de transformation model-to-model nommés :
    * QVT-Operational (impératif, pour la transformation unidirectionnelle uniquement),
    * QVT-Relations (déclaratif, pour la transformation unidirectionnelle et bidirectionnelle),
    * QVT- Core (déclaratif, un langage exécutable simple cible pour QVT-Relations).
  - MOF Model to Text Transformation Language (MOF2Text) [OMG08], un langage de transformation de modèle en texte pour générer du code source ou de la documentation à partir de modèles basés sur MOF.

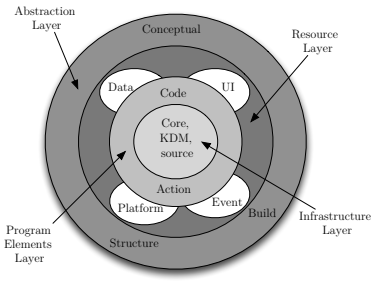
2.2.5 Exemples de méta-modèles

Au niveau maintenance et évolution, on donne deux modèles MDA qui sont plus utilisés : *le KDM (Knowledge Discovery Meta-model, voir la sous-section 2.7)* et *ASTM (Abstract Syntax Tree Meta-model, voir la sous-section 2.8)).*

KDM :

La définition officielle actuelle du KDM selon L'OMG [OMG16a] est la suivante :

"*Le KDM fournit un format d'échange commun qui permet l'interopérabilité entre les outils d'analyse et de modernisation de logiciels existants, les services et leurs modèles respectifs. Plus précisément, (KDM) définit une ontologie commune et un format d'échange qui facilite l'échange de données actuellement contenues dans des modèles d'outils individuels qui représentent des logiciels existants. Le méta-modèle représente les éléments physiques et logiques du logiciel ainsi que leurs relations à différents niveaux d'abstraction.*"



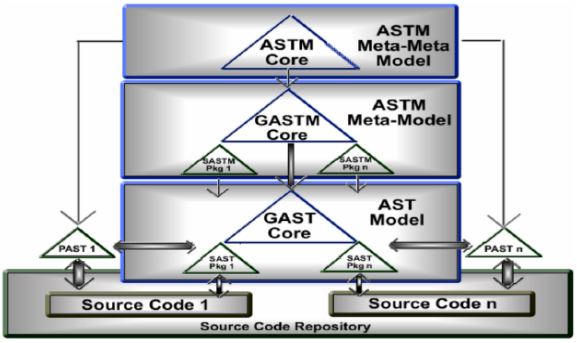
.

Fig. 2.7 : Couches, packages et séparation du méta-modèle KDM [OMG16a].

ASTM :

Selon le document standard de l'OMG [OMG11], la spécification principale de l'ASTM, figure 2.8, est :

"*de faciliter l'échange de métadonnées logicielles détaillées entre les outils de développement et de modernisation de logiciels, les plate-formes et les référentiels de métadonnées dans des environnements hétérogènes distribués. Le métamodèle d'arbre de syntaxe abstraite définit une spécification pour modéliser des éléments afin d'exprimer des arbres de syntaxe abstraite (AST) dans une représentation qui peut être partagée entre plusieurs outils de différents fournisseurs.*"



.

Fig. 2.8 : Le framework de modélisation ASTM [OMG11] .

La métamodélisation de l'arborescence de syntaxe abstraite (ASTM [OMG11]) et la métamodélisation de la découverte des connaissances (KDM [OMG16a]) sont deux spécifications de modélisation complémentaires développées par l'OMG Architecture Driven Modernization Task Force. Leur relation peut être clairement comprise en reconnaissant que le KDM établit une spécification pour les modèles de graphes sémantiques abstraits ASG, tandis que l'ASTM établit une spécification pour les modèles d'arbres de syntaxe abstraite AST. Ainsi, contrairement à d'autres normes de représentation logicielle, telles que Knowledge Discovery Metamodel ou le Unified Modeling Language, l'ASTM prend en charge un mapping direct 1 à 1 de toutes les instructions de langage logiciel au niveau du code dans des modèles logiciels de bas niveau [OMG11].

2.3 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.

2.3.1 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 [RS04]. 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 [Men08]. 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 [TN14]. 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 [TN14] 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 2.9, a été développée par Kazman et al. [Jon84] 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.

.

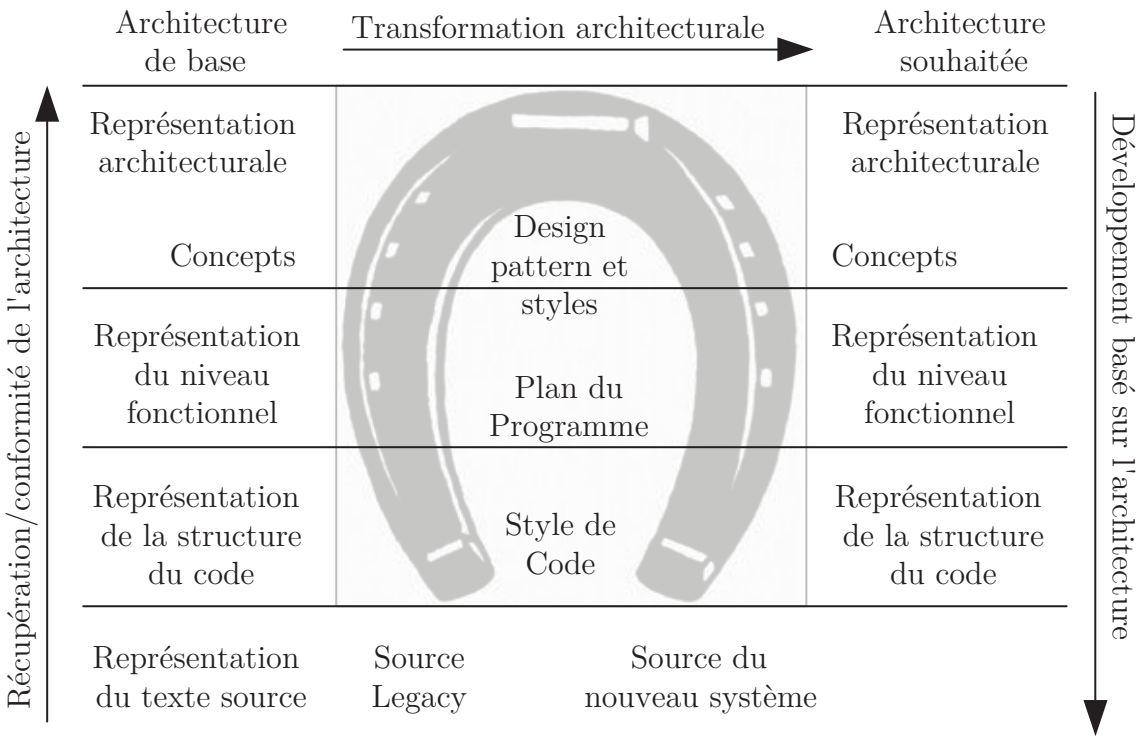


FIG. 2.9 : Le modèle de ré-ingénierie Horseshoe [TN14] .

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.

2.3.2 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 [12198], 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.

2.3.3 Techniques utilisées pour la rétro-ingénierie

La collecte d'informations et le fact-finding à partir du code source sont les clés du paradigme Objectif/Modèles/Outils [TN14]. Afin d'extraire des informations qui ne sont pas explicitement disponibles dans le code source, des techniques d'analyse automatisées sont utilisées. Les techniques d'analyse bien connues qui facilitent la rétro-ingénierie sont *l'analyse lexicale, l'analyse syntaxique, l'analyse de flow de contrôle, l'analyse de flow de données, le slicing d'un programme et la visualisation.*

Analyse lexicale :

L'analyse lexicale 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 [TN14]. 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)[BSF98].

L'analyse lexicale est la première phase d'un compilateur. Il prend comme entrée le code source modifié des préprocesseurs du langage qui sont écrits sous forme de phrases. L'analyseur lexical divise ces syntaxes en une série de tokens, en supprimant tout espace ou commentaire dans le code source. On a aussi, les scanners, qui sont à leur tour, généralement implémentés pour produire uniquement des tokens à la demande d'un analyseur. La figure 2.12 décrit le processus de reconnaissance des tokens dans l'architecture d'un compilateur :

1. "Avoir le token suivant" est une commande qui est envoyée de l'analyseur à l'analyseur lexical.
2. A la réception de cette commande, l'analyseur lexical analyse l'entrée jusqu'à ce qu'il trouve le token suivant.
3. Il renvoie le token au Parser.

[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 [GfG21]

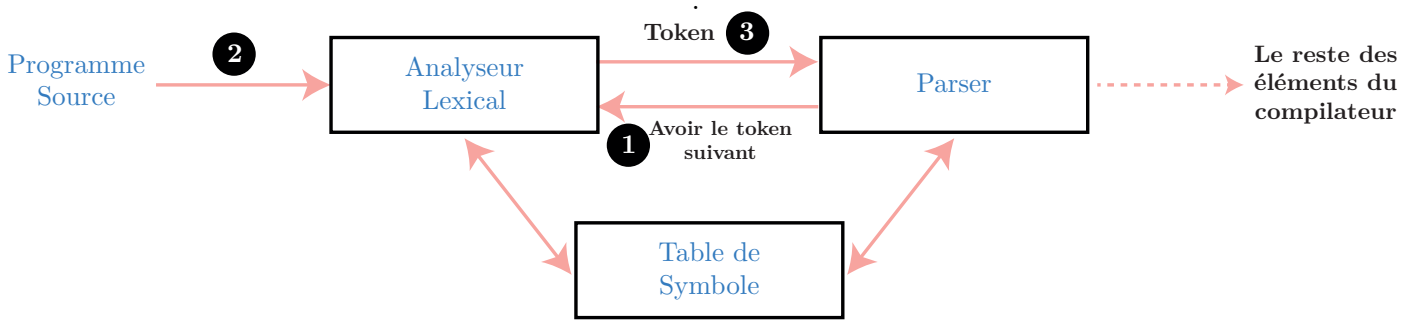


FIG. 2.10 : L'architecture de l'analyseur lexical.

On résume le rôle d'un analyseur lexical dans le processus de compilation comme suit :

- Aide à identifier le token dans la table des symboles,
- Supprime les espaces blancs et les commentaires du programme source,
- Met en corrélation les messages d'erreur avec le programme source,
- Aide à développer des macros si elles se trouvent dans le programme source
- Lire les caractères d'entrée du programme source

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 [Sek+91]. 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 [BSF98].

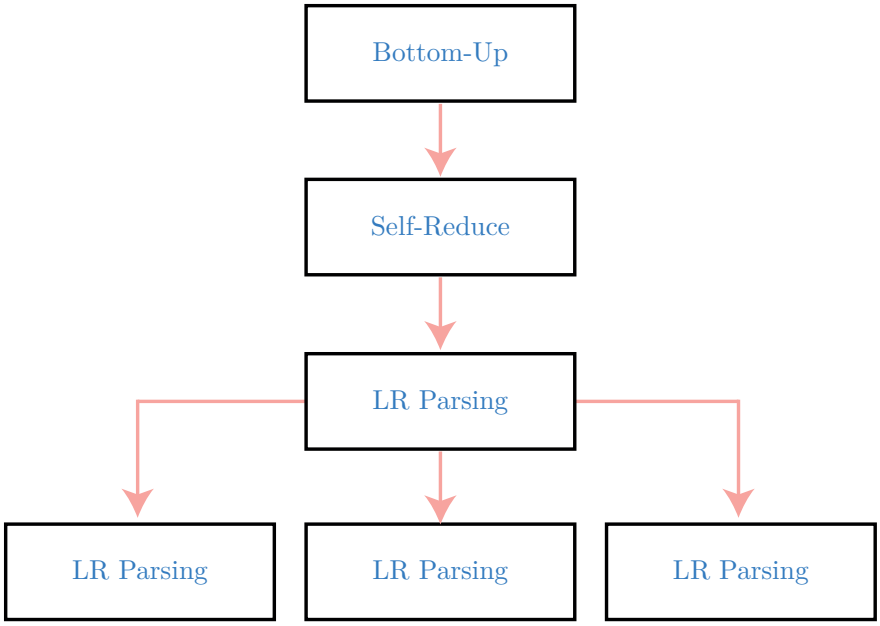
Dans cette analyse deux types de représentation sont généralement utilisés : *parse tree et AST (abstract syntax tree)*. Un parse tree contient des détails sans rapport avec la signification réelle du programme, comme la ponctuation, dont le rôle est de diriger le processus de parse. Par exemple, les 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). Un AST contient uniquement les détails qui se rapportent à la signification réelle d'un programme. Étant donné qu'un AST est un arbre, les nœuds de l'arbre peuvent être visités d'une manière prédéfinie, telle que depth-first search, et pour chaque nœud visité les informations contenues dans ce nœud sont fournies à l'analyseur.

Les techniques d'analyse syntaxique sont divisés en deux groupes :

- **Top-Down Parsing :** dans le Top-Down Parsing, la construction de l'arbre d'analyse commence à partir de la racine, puis se poursuit vers les feuilles. Deux type dans cette catégorie sont :

  1. Predictive Parsing : C'est une analyse prédictive qui peut prédire quelle production doit être utilisée pour remplacer la chaîne d'entrée spécifique. L'analyseur prédictif utilise un point d'anticipation, qui pointe vers les symboles d'entrée suivants. Le retour en arrière n'est pas un problème avec cette technique d'analyse. Il est connu sous le nom d'analyseur LL(k).
  2. Recursive Descent Parsing : cette technique récursive analyse l'entrée pour produire une parse Tree. Elle se compose de plusieurs petites fonctions, une pour chaque non terminal de la grammaire.

- **Bottom-Up Parsing :** dans le Bottom-Up Parsing, la construction de l'arbre d'analyse commence à partir des feuilles, puis se poursuit vers la racine. C'est-à-dire, on part d'une phrase puis on applique les règles de production en sens inverse pour arriver au symbole de départ. La figure 2.11 décrit la classification des analyseurs Bottom-Up disponibles.



.

Fig. 2.11 : Classification des analyseurs Bottom-Up.

Analyse de flux de contrôle :

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 [All70]. 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 [Coc69 ; Pro59 ; Low69].

Les deux types de CFA sont l'analyse intra-procédurale et l'analyse inter-procédurale [Hec77]. 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.

Analyse de flux de données :

Bien que la 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 impacté ? [TN14]). Afin de résoudre ces situations, 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) s'intéresse à la façon dont les valeurs des variables définies circulent et sont utilisées dans un programme [OF76]. CFA peut détecter la possibilité de boucles, tandis que DFA peut déterminer les anomalies de flux de données [FO76]. 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 de programme dans le programme 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 [KSK17]. 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, le slicing 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.
- **JIT :** L'analyse des flux de données est principalement une analyse statique. La

compilation Just-In-Time (JIT) et le slicing dynamic, etc. impliquent une analyse dynamique des flux de données.

- **Portée du programme :** L'analyse des flux de données peut être effectuée à presque tous les niveaux de portée d'un programme. Traditionnellement [KSK17], les termes suivants ont été associés à l'analyse des flux de données pour différentes portées dans le domaine des langages impératifs :

  – À travers les instructions mais confiné à une séquence maximale d'instructions sans transfert de contrôle autre que le transfert (c'est-à-dire au sein d'un bloc de base) : 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.

  Dans le domaine de la rétro-ingénierie, il est courant d'utiliser le terme analyse de flux de données locales pour l'analyse d'une seule instruction et analyse de flux de données globale pour l'analyse de plusieurs instructions dans une fonction/procédure. Effectivement, les blocs de base pour de telles analyses consistent en une seule expression.

- **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).

Slicing d'un programme :

On parle du slicing d'un programme pour une variable donnée à une ligne de code donnée. On le définit par la partie du programme qui donne valeur à cette variable (définition) à ce niveau là [Wei84]. Par conséquent, si l'on détermine lors du débogage que la valeur d'une variable à une ligne spécifique est incorrect, on peut regarder le slicing du programme correspondant pour trouver le code défectueux. C'est une technique utilisée également dans les tests de logiciels qui prend un slice ou un groupe d'instructions du programme pour tester des conditions ou des cas de test particuliers et qui peut affecter une valeur à un point d'intérêt particulier [Gee19]. On trouve dans la littérature plusieurs types de slicing. Cependant, on se restreint aux standards : *slicing statique et slicing dynamique.*

Le slicing statique est définie par le fait que le comportement du programme d'origine doit être préservé sur n'importe quelle entrée [Wei84]. Weiser a proposé une définition pour ce type d'algorithme de la manière suivante : un slice est constitué de l'ensemble des instructions de programme et des prédicats qui affectent directement ou indirectement le

calcul des variables dans V avant l'exécution de p, telle que p est un point d'un programme et V est un sous-ensemble de variables de ce programme [De 01].

Le deuxième type est le slicing dynamique. Il a été proposé par Korel and Lasky [KL88 ; KL90] qui utilise l'analyse dynamique pour identifier toutes et uniquement les instructions qui affectent les variables d'intérêt sur la trace d'exécution. De cette façon, la taille du slice peut être considérablement réduite, permettant ainsi une localisation plus aisée des bugs [KL88 ; KL90].

Visualisation :

La visualisation logicielle est une stratégie utile pour permettre à un utilisateur de mieux comprendre les systèmes logiciels [TN14]. 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 [Die05].

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 2.

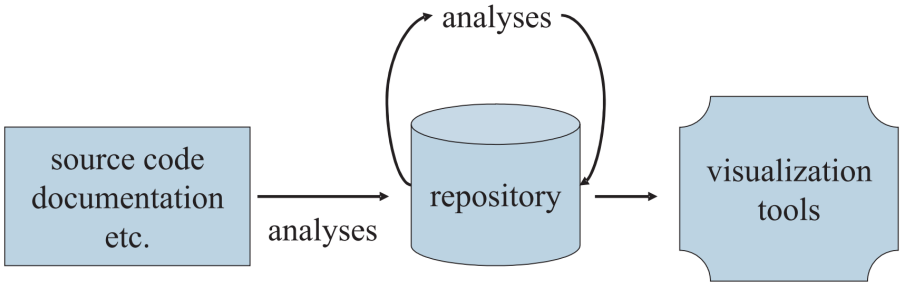


Fig. 2.12 : Pipeline typique des outils de rétro-ingénierie [Die05].

L'un des exemples les plus intéressant dans la POO est Fujaba [TOO18], qui détecte les design patterns dans un code source et étend les diagrammes UML en indiquant les rôles joués par les différentes classes dans le design pattern. On a aussi les blueprints [LD01] qui représentent les appels graphs dans les classes et entre les classes.

2.4 Solutions existantes pour la rétro-ingénierie

En ce qui concerne les solutions existantes supportant la rétro-ingénierie, on distingue deux grandes familles : les solutions spécifiques et les solutions à usage général. Celles-ci sont déterminées selon qu'elles visent à rétro-concevoir le système à partir d'une technologie unique et/ou avec un scénario prédéfini (par exemple un type d'analyse concret), ou à être la base de tout autre type de manipulation dans les étapes ultérieures du processus de la rétro-ingénierie [Bru18].

Chacune de ces solutions possède une ou plusieurs fonctionnalités distinctes par rapport aux autres. On va mettre l'accent sur l'objectif de chacune et sur les fonctionnalités qu'elle prend en charge. Il est important de noter que la rétro-ingénierie logicielle est un processus complexe que les outils ne peuvent que supporter, pas complètement automatiser [TN14]. Il y a donc, toujours, un besoin d'intervention humaine dans tout projet de rétro-ingénierie.

Dans les tableaux 2.1 et 2.2, [Bru18] a fournit des exemples représentatifs pour chacune de ces deux familles de solutions. [Bru18] a également classé ces différentes solutions selon des catégories du simple à haut niveau en fonction de leur famille [4].

2.4.1 Solutions spécifiques

Les catégories :

- Découverte : la solution fournit des métamodèles et des model discovery features (KDM) correspondants à partir du code source,
- Migration : la solution est à la base un projet de migration impliquant la restructuration du code et la traduction automatique de la langue,
- Intégration : La solution est intégrée dans un autre environnement (Model-based Technology Integration) [Béz05a],
- Analyse et Compréhension : la solution est dédiée à l'analyse et la compréhension du code source mais aussi des modèles associés, des spécifications textuelles ... etc.

Il existe plusieurs solutions spécifiques qui nous permettent d'extraire des modèles à partir d'un legacy system :

- Columbus [Fer+02 ; Tiw20] offre des capacités d'analyse à partir du code source C/C++ et permet de sérialiser les informations obtenues en utilisant différents formats (par exemple XML, UML XMI). Columbus est un framework qui fournit la manipulation du projet, l'extraction de données, la représentation de données, le stockage de données, la modification et la visualisation. Le code source analysé est fourni via le plug-in d'extraction C/C++ de Columbus, qui appelle un programme distinct appelé CAN (C++ Analyzer). CAN est un logiciel en ligne de commande

[4] Quelques approches ont été mises à jour, par conséquent les tableaux 2.1 et 2.2 sont également mis à jour par rapport à [Bru18].

	Découverte	Migration	Intégration	Anaylse et Compréhension
JaMoPP [Hei+09]	✓			
Spoon [Paw+15]	✓			
Briand et al. [BLL06]	✓			
Sneed [Sun+09] [Sne92] [Sne10]	✓	✓		
RE Analyzer [OT94]	✓	✓		
Barbier et al. [Eve+09]	✓	✓		
Fleurey et al. [Fle+07]	✓	✓		
Clavreul et al. [CBJ10]	✓		✓	
Alnusair et al. [AZ09]	✓		✓	
Ramon et al. [SSM13]	✓		✓	
ConQAT [Dei+10]	✓			✓
GUPRO [Ebe+02]	✓			✓
SWAG Kit [RG08]	✓			✓
CodeCrawler [Lan+05],CodeCity [WLR11]	✓			✓
Columbus [Fer+02 ; Tiw20]	✓			✓
SolidFx [Tiw20]	✓			✓
Pacione et al.[PRW04]	✓			✓
Olsson et al. [OG12]	✓			✓
Fradet et al.[FLP99b]				✓

Tab. 2.1 : Solutions Spécifiques MDRE.

pour analyser le code C/C++. Cela permet son intégration dans les fichiers de création du client et d'autres fichiers de configuration, accélérant ainsi l'exécution automatisée en parallèle avec le processus de construction du logiciel.

- Spoon [Paw+15] et JaMoPP [Hei+09] fournissent des méta modèle et qui correspondent à un KDM à partir d'un code source Java. Le méta modèle Java publié par Spoon est basé sur un langage de modélisation standard Ecore, mais il n'était pas complet car il ne supporte pas toutes les spécification du langage Java [Hei+09]. Par conséquent, le métamodèle JaMoPP couvre tous les éléments du langage Java (e.g., classifiers, imports, types, modifiers, members, statements, variables, expressions etc..).

- ConQAT [Dei+10], au niveau software, est dédié à l'analyse du code source mais également des modèles associés, des spécifications textuelles, etc. Son objectif est d'effectuer des activités d'évaluation de la qualité telles que la détection de clones, la conformité de l'architecture ou la génération de tableaux de bord.

- GUPRO [Ebe+02], au niveau code, est une solution basée sur un référentiel central de graphes ainsi que des requêtes et algorithmes associés. Pour le langage C, la navigation GUPRO inclut un traitement complet des installations du préprocesseur graçe au référentiel central de graphes.

- RE-Analyzer [OT94] est un système de rétro-ingénierie automatisé offrant un haut niveau d'intégration avec un outil d'ingénierie logicielle assistée par ordinateur (CASE) développé chez IBM. L'analyseur RE inverse automatiquement le code source en représentations graphiques et textuelles dans un outil CASE prenant en charge la méthodologie d'analyse structurée [You89]. Autrement dit, il transforme le code source en un ensemble de diagrammes de flux de données, de diagrammes de transition d'état et de modèles de données entité-relation dans la base de données de conception d'un outil CASE. Étant donné que les représentations résultantes peuvent être parcourues et modifiées dans l'environnement de l'outil CASE, un large éventail d'activités d'ingénierie logicielle est pris en charge, y compris la compréhension du programme, la réingénierie et la redocumentation.

- SWAG Kit [RG08] aide à générer une architecture de bas niveau qui constitue la base de l'analyse des systèmes C/C++ grâce à des visualisations graphiques.

- CodeCrawler [Lan+05] est un outil de rétro-ingénierie indépendant du langage qui combine métriques et visualisation logicielle. CodeCrawler est écrit en Visualworks Smalltalk et fonctionne sur toutes les principales plates-formes. Il utilise des écrans bidimensionnels pour visualiser les logiciels. Les nœuds représentent des entités logicielles ou une abstraction de celles-ci, tandis que les bords représentent les relations entre ces entités [Lan+05]. Cet outil combine la capacité d'afficher les entités logicielles et leurs relations avec la capacité de visualiser les métriques logicielles à l'aide de vues polymétriques. CodeCrawler a été utilisé avec succès pour rétro-concevoir plusieurs grands systèmes logiciels industriels.

- SolidFx présente un environnement intégré de rétro-ingénierie (IRE) pour C et C++. SolidFX a été spécialement conçu pour prendre en charge l'analyse de code, l'extraction de faits, le calcul de métriques et l'analyse visuelle interactive de la sortie de la même manière que les IDE et les outils de conception le prévoient pour le pipeline d'ingénierie avancée.

2.4.2 Solutions génériques

- Approche single-core méta-modèle : la solution est basée sur des métamodèles spécifiques à une seule technologie (les technologies orienté objet par exemple) dans leur processus de modélisation,

- Approche à méta-modèle multiple : la solution est basée sur des métamodèles spécifiques à plusieurs technologies (les technologies orientés objet ou autre) dans leur processus de modélisation,

Contrairement aux solutions/outils spécifiques de rétro-ingénierie orienté à une activité ou un scénario particulier, il y a eu moins d'initiatives de recherche fournissant des environnements intégrés plus génériques qui pourraient être étendus et adaptés à différents scénarios d'application.

	Approche single-core méta-modèle	Approche à méta-modèle multiple	Logiciel de modélisation général
CORUM-II [FLP99a]	✓		
Moose [KWC98 ; Anq+20]		✓	
MoDisco [Bru+14 ; Bru18]		✓	
Garces et al. [Gar+18 ; Gar+16]		✓	
Rational Software Architect [IBM21]			✓
MagicDraw [Das20]			✓
Enterprise Architect [Sys21]			✓
Modelio [Sou20]			✓
Visual Paradigm [21b]			✓

Tab. 2.2 : Solutions génériques MDRE.

L'une des premières solutions est CORUM-II [FLP99a], qui tentait d'intégrer plusieurs outils de rétro-ingénierie et d'architecture. Elle est basée sur un schéma commun d'interopérations appelé CORUM pour être considéré comme une référence.

Moose [Anq+20] est une plateforme reconnue fournissant une variété d'outils dédiés à la construction de diverses solutions logicielles et d'analyse de données. Ainsi la Suite Moose propose un ensemble d'outils construits autour de cette même plate-forme, reposant tous sur un métamodèle composable nommé FAMIX. Par ailleurs, Moose sera l'objet du chapitre 2.5 afin d'illustrer une des solution générique et comprendre l'architecture derrière ces outils de rétro-ingénierie.

Il existe plusieurs solutions commerciales offrant des fonctionnalités de rétro-ingénierie couvrant les langages de programmation les plus courants (par exemple Java, C/C++, C#, VB.NET, etc.) et les langages de modélisation standard (par exemple UML, SysML, BPMN). Des exemples bien connus de tels outils sont Rational Software Architect - RSA (par IBM) [IBM21], MagicDraw (par NoMagic/3DS) [Das20] ou Enterprise Architect (par Sparx Systems) [Sys21]. Certains outils sont également livrés avec une version de base open source ou gratuite, proposant ensuite des extensions spécifiques à acheter pour compléter la version de base, comme Modelio (de Softeam) [Sou20] ou Visual Paradigm (de Visual Paradigm International) [21b].

2.5 Conclusion

Le nombre de systèmes logiciels à maintenir, à étendre ou à faire évoluer de manière générale a considérablement augmenté au cours des dernières décennies et continuera à le faire. Pour faire face à tous ces logiciels hérités, tant sur le plan économique que technologique, des solutions de rétro-ingénierie (semi-)automatisées fiables doivent être fournies. Pour atteindre cet objectif, les techniques de modélisation dans les solutions de rétro-ingénierie montre des potentiels prometteurs et innovants.

Dans ce chapitre, nous avons vu les connaissances de base autour de la modélisation en donnant quelques exemples. Ensuite, nous présentions la rétro-ingénierie et ainsi ses étapes et techniques. Enfin, nous avons vu différentes solutions spécifiques et génériques. Parmi les solutions génériques, il existe la plateforme Moose qui sera décrite dans le prochain chapitre.

Chapitre 3

Moose

3.1 Introduction

Dans ce dernier chapitre, nous allons illustrer une des solutions de rétro-ingénierie Moose [Anq+20], qui est une plateforme générique pour l'analyse et la maintenance des logiciels. Nous commençons par définir l'environnement Moose, ses objectifs et les solutions apportées. Ensuite, nous verrons l'architecture extensible utilisée par Moose. Nous terminons par voir les différents outils réutilisables par cette plateforme.

3.2 Moose : Difficultés & Objectifs

Comme déja mentionné dans la section 2.2.1, la modélisation repose sur la création, la manipulation et l'utilisation intensive de modèles afin de les exploiter pour extraire et analyser l'information. Dans cette section, nous allons présenter Moose, les difficultés de la mise en oeuvre des modèles et les objectifs visés par Moose afin de résoudre ces difficultés.

3.2.1 Moose

La première description de Moose dans [NDG05] est : *Moose est un environnement indépendant du langage pour la rétro-ingénierie et la réingénierie de 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.*

Moose, figure 3.1 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 [NDG05].

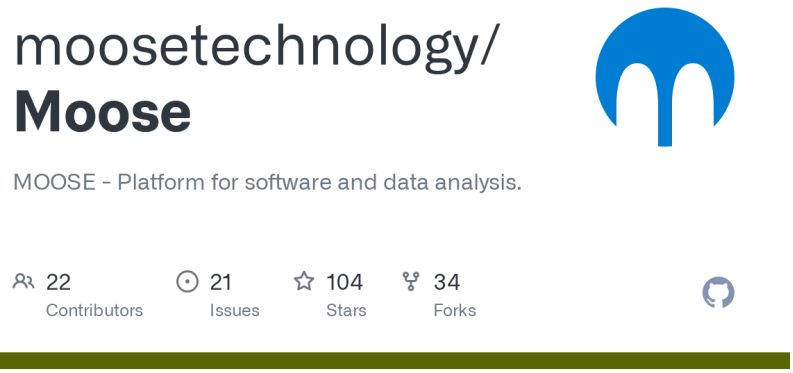


Fig. 3.1 : Le projet Moose Open source - Image capturé de Github.

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

3.2.2 Difficultés

Le défi pour la rétro-ingénierie est de savoir comment supporter le développement de plusieurs méta-modèles dédiés et spécifiques pour différentes langages de programmation. Donc ces méta-modèles doivent être génériques, multi-langages et détaillés [Anq+20].

Problème : Manipuler plusieurs langages

Les méta-modèles étaient censés résoudre ce problème. Normalement, on pourrait espérer qu'un seul "méta-modèle orienté objet" serait capable de représenter les programmes dans tous les langages de programmation OO et qu'un autre "méta-modèle procédural" représenterait tous les langages de programmation procédurale. Mais en pratique, [LTP04] se rend compte que « divers langages de programmation présentent des différences sémantiques mineures quant à la manière dont ils implémentent ces concepts ». Par exemple la visibilité est un exemple concret de la complexité de représenter des détails variés pour des concepts apparemment universels : l'abstraction de l'API d'une classe nécessite de comprendre les règles de visibilité des classes, des méthodes et des attributs [Anq+20]. La visibilité n'est pas la même pour Java et C++ .

3.2.3 Objectifs

L'environnement modulaire Moose a fixé des objectifs à deux niveaux : meta-modeling et tooling.

Objectif au niveau meta-modeling :

Pour avoir un modèle générique, il est nécessaire d'avoir une représentation unifiée [Anq+20] qui :

- Permet de représenter de nombreuses caractéristiques de langage de programmation,
- Soit compatible avec plusieurs outils d'analyses,
- Se méfie de la consommation de mémoire,
- Soit extensible.

Objectif au niveau tooling :

Les outils ont besoin de [BG98 ; KM10] :

- Automatiser les tâches fastidieuses afin de minimiser l'effort, temps et erreurs
- Gérer une grande quantité de données et dans ce cas tous les détails de plusieurs millions de lignes de code
- Résumer et représenter les données pour permettre à un être humain d'avoir une vue globale.

3.3 FAMIX & FAST

Moose définit un méta-modèle composable de langage de programmation qui est FAMIX. Il est conçu de manière à réutiliser tous les élements d'un language donné.

3.3.1 FAMIX

FAMIX est un méta-modèle pour représenter des modèles liés à diverses facettes de systèmes logiciels. Ces modèles sont généralement orientés pour l'analyse et fournissent une API riche qui peut être utilisée pour les requêtes et la navigation.

Les méta-modèles sont implémentés dans Pharo, et comme Pharo peut être utilisé comme langage de script, les API définies par les méta-modèles fournissent également un langage de requête.

La nouvelle conception de FAMIX supporte la réutilisation des concepts de languages, et cette conception est basée sur les stateful traits [Tes+20].

3.3.2 FAST

FAST signifie Famix AST. Contrairement à Famix qui représente l'application à un niveau d'abstraction élevé, FAST utilise une représentation de bas niveau : l'AST. FAST définit un ensemble de traits pouvant être utilisés pour créer de nouveaux méta-modèles compatibles avec les outils Moose. Cette stratégie est similaire aux traits prédéfinis de Famix. La conception du méta-modèle FAST est illustrée dans la figure 3.2.

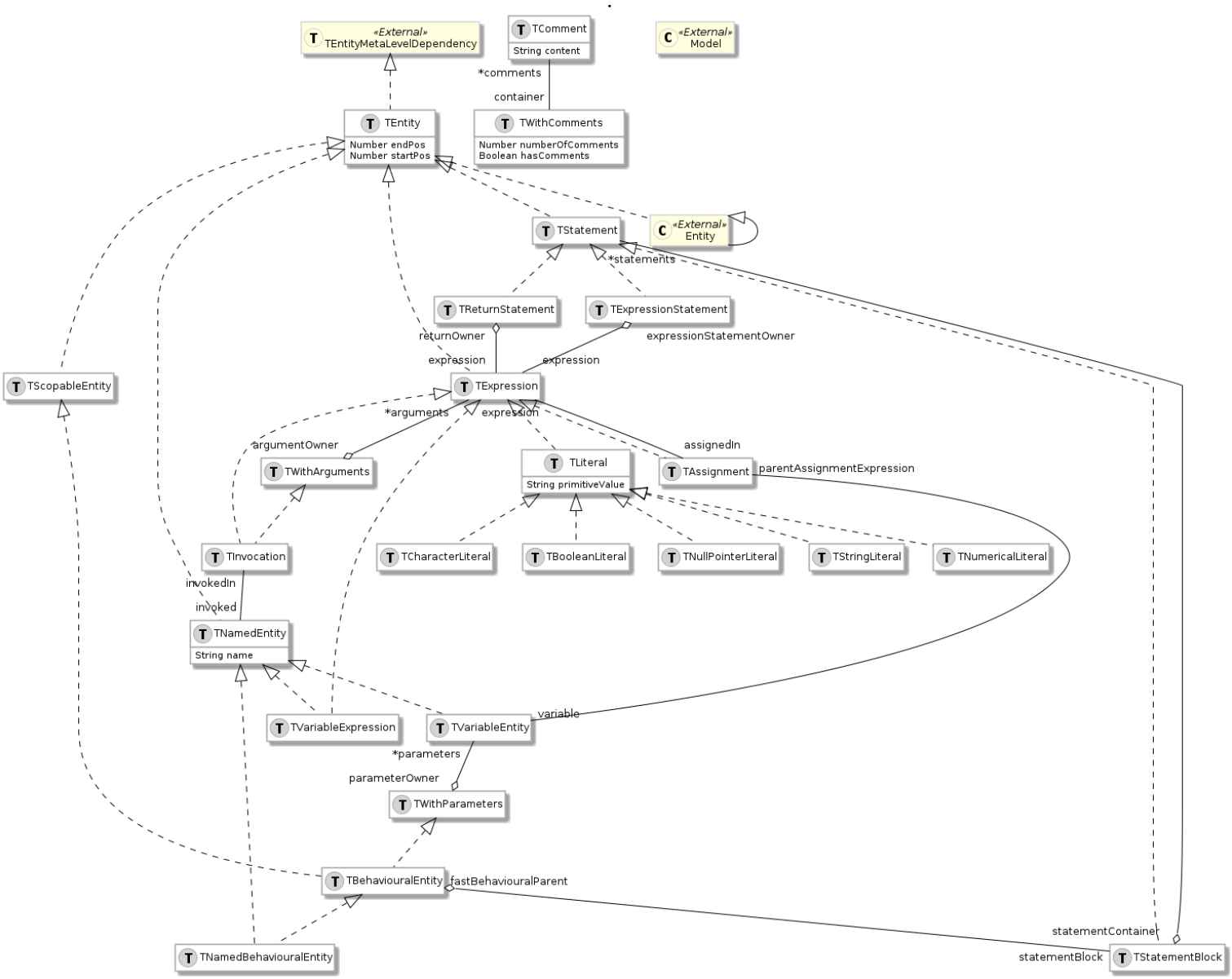


Fig. 3.2 : FAST meta-model.

3.4 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.

3.4.1 Architecture

Moose est architecturé sur les principes suivants [Anq+20] 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

[2]ModMoose fait référence à une nouvelle version de moose 2020 [Anq+20]

centralise les données

- Les outils communiquent via des bus, ils « lisent » des entités modèles sur leur bus et « réécrivent » des entités sur leur bus (voir figure 3.3)

- 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.

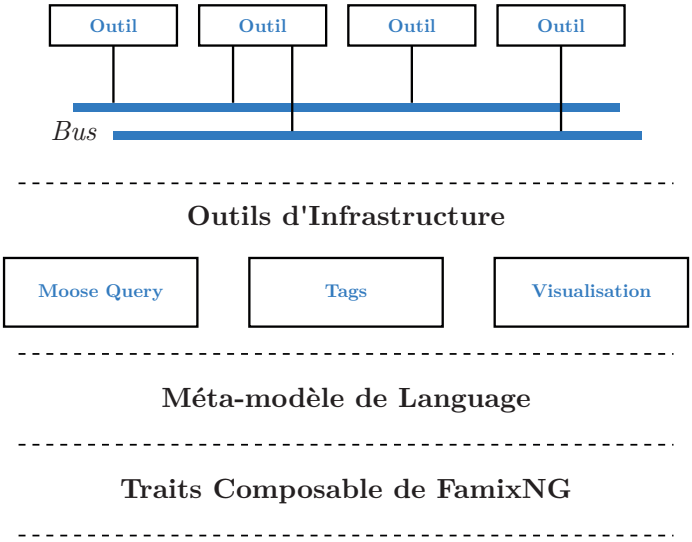


FIG. 3.3 : Nouvelle architecture de Moose [Anq+20].

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 [Gov+17]. Les balises enrichissent les modèles avec des informations supplémentaires.

- **Engine de Visualisation :** utilise Roassal [Ber16], un moteur de visualisation générique, pour scénariser des graphiques ou des diagrammes interactifs. Roassal [Ber16] 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 [Anq+20].

3.4.2 Outils réutilisables de Moose

Comme l'architecture l'indique 3.4.1, 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 [Anq+20].
- **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) [Anq+20].
- **Query Browser :** c'est l'interface graphique de Moose Query (section 3.4.1).
- **Dependency Graph Browser :** affiche sous forme de graphique toutes les dépendances entrantes et sortantes directes d'un groupe d'entités (voir Figure 3.4) [Anq+20].

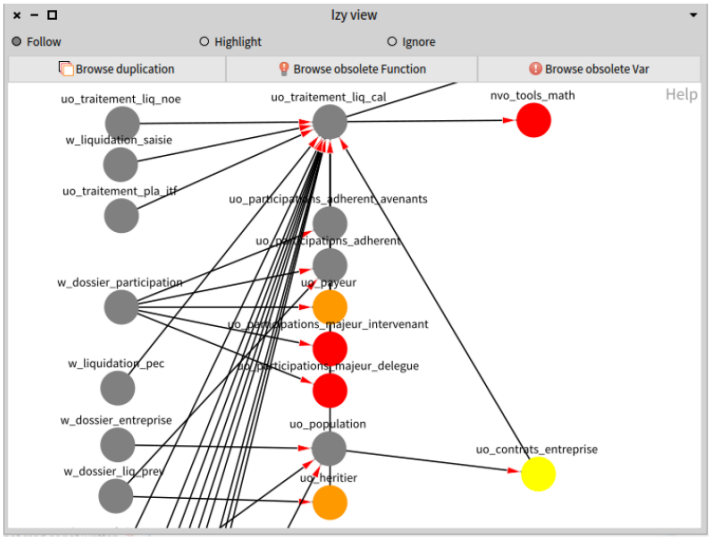


Fig. 3.4 : Call Graph Browser, exemple d'un outils spécialisé de Moose [Anq+20].

3.5 Conclusion

Pour conclure, dans ce chapitre, nous avons présenté l'environnement Moose, la solution pour avoir un méta-modèle extensible et adaptable pour représenter un langage de programmation, les méta-modèles FAMIX et FAST, l'architecture déployée pour exploiter les modèles ainsi que les différents outils spécialisés pour l'utilisateur afin d'intéragir avec les données.

Conclusion générale et perspectives

Dans ce travail de master, nous avons étudié les différentes techniques et solutions de rétro-ingénierie existantes en illustrant à la fin une des plateformes génériques Moose qui nous permet de comprendre la composition et l'architecture derrière ces outils. Notre choix est motivé par le fait que nous allons implémenter une technique de rétro-ingénierie analyse des flux de données, au niveau de cette plateforme Moose, dans le projet de fin d'étude. Le domaine de rétro-ingénierie est très vague où chaque technique représente un axe de recherche à part entière, d'où la nécessité de comprendre le contexte et d'avoir une vision générale sur les différentes techniques et solutions existantes.

Nous avons, dans le premier chapitre, commencé par aborder des généralités sur quelques notions du génie logicielle (GL). Nous avons défini le cycle de vie du développement logiciel et ses méthodologies standards. Nous avons conclu ce chapitre par les notions de maintenance et l'évolution des systèmes logiciels.

Le deuxième chapitre est le cœur du mémoire, la modélisation et ses concepts fondamentaux ont été détaillés, ainsi que l'application de la modélisation dans la rétro-ingénierie à travers le MDRE (Model-Driven Reverse Engineering). Aussi, le chapitre présente les différentes techniques de la rétro-ingénierie ainsi que les solutions génériques et spécifiques pour les appliquer.

Dans le troisième chapitre, nous avons mis en valeur un exemple d'une solution générique de rétro-ingénierie Moose qui est une plateforme d'analyse et de maintenance des logiciels. Cette illustration a pour but de comprendre l'infrastructure déployée derrière ces plateformes de rétro-ingénierie et de pouvoir implémenter une solution au-dessus de cette architecture pour le projet de fin d'étude.

Pour clôturer ce mémoire, nous souhaitons mentionner que ce travail a été l'occasion de mettre en pratique nos connaissances sur la recherche et synthèse bibliographiques et n'est que la première étape de notre projet, qui consiste à proposer une solution pour l'analyse des flux de données dans les programmes. Nos perspectives pour les étapes qui suivent se résument dans les points suivants :

- La conception et l'implémentation d'une infrastructure au sein de la méta-plateforme Moose pour l'analyse et la compréhension des flux de données.
- L'adaptation du langage MooseQuery pour permettre l'analyse et la compréhension des flux de données.
- L'intégration de la solution conçue sur Moose.

Bibliographie

[12198] IEEE Standard 1219-1998. "IEEE standard for software maintenance". In : *IEEE Computer Society Press* (1998). DOI : 10.1109/ieeestd.1993.115570.

[All70] Frances E. ALLEN. "Control Flow Analysis". In : *SIGPLAN Not.* 5.7 (juil. 1970), p. 1-19. ISSN : 0362-1340. DOI : 10.1145/390013.808479.

[Anq+20] Nicolas ANQUETIL et al. "Modular Moose : A new generation software reverse engineering environment". In : *arXiv preprint arXiv :2011.10975* (2020).

[Art88] Lowell Jay ARTHUR. *Software evolution : the software maintenance challenge.* Wiley-Interscience, 1988.

[AZ09] Awny ALNUSAIR et Tian ZHAO. "Towards a model-driven approach for reverse engineering design patterns". In : *CEUR Workshop Proceedings* 531 (jan. 2009).

[BCW17] Marco BRAMBILLA, Jordi CABOT et Manuel WIMMER. "Model-driven software engineering in practice". In : *Synthesis lectures on software engineering* 3.1 (2017), p. 1-207.

[Ben95] Keith BENNETT. "Legacy systems : Coping with success". In : *IEEE software* 12.1 (1995), p. 19-23.

[Ber16] A. BERGEL. *Agile Visualization.* Lulu.com, 2016. ISBN : 9781365314094.

[Béz05a] Jean BÉZIVIN. "Model driven engineering : An emerging technical space". In : *International Summer School on Generative and Transformational Techniques in Software Engineering.* Springer. 2005, p. 36-64.

[Béz05b] Jean BÉZIVIN. "On the unification power of models". In : *Software & Systems Modeling* 4.2 (2005), p. 171-188.

[BG98] Berndt BELLAY et Harald GALL. "An evaluation of reverse engineering tool capabilities". In : *Journal of Software Maintenance : Research and Practice* 10.5 (1998), p. 305-331.

[BH00] Barry BOEHM et Wilfred J HANSEN. "SPECIAL REPORT CMU/SEI-2000-SR-008". In : (2000).

[BLL06] L.C. BRIAND, Y. LABICHE et J. LEDUC. "Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software". In : *IEEE Transactions on Software Engineering* 32.9 (2006), p. 642-663. DOI : 10.1109/TSE.2006.96.

[Bru+14] Hugo Bruneliere et al. “MoDisco : a Model Driven Reverse Engineering Framework”. In : *Information and Software Technology* 56.8 (août 2014), p. 1012-1032. doi : 10.1016/j.infsof.2014.04.007.

[Bru18] Hugo Bruneliere. “Generic model-based approaches for software reverse engineering and comprehension”. Thèse de doct. Nantes, 2018.

[BSF98] Marco Battaglia, Giancarlo Savoia et John Favaro. “Renaissance : A method to migrate from legacy to immortal software systems”. In : *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*. IEEE. 1998, p. 197-200.

[CBJ10] Mickael Clavreul, Olivier Barais et Jean-Marc Jézéquel. “Integrating Legacy Systems with MDE”. In : *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. ICSE ’10. Cape Town, South Africa : Association for Computing Machinery, 2010, p. 69-78. isbn : 9781605587196. doi : 10.1145/1810295.1810306.

[CC90] Elliot J. Chikofsky et James H Cross. “Reverse engineering and design recovery : A taxonomy”. In : *IEEE software* 7.1 (1990), p. 13-17.

[Coc69] John Cocke. *Programming languages and their compilers : Preliminary notes*. New York University, 1969.

[De 01] Andrea De Lucia. “Program slicing : Methods and applications”. In : *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE. 2001, p. 142-149.

[Dei+10] Florian Deissenboeck et al. “Flexible Architecture Conformance Assessment with ConQAT”. In : *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. ICSE ’10. Cape Town, South Africa : Association for Computing Machinery, 2010, p. 247-250. isbn : 9781605587196. doi : 10.1145/1810295.1810343.

[Die05] Stephan Diehl. “Software visualization”. In : jan. 2005, p. 718-719. doi : 10.1145/1062455.1062634.

[Ebe+02] Jürgen Ebert et al. “GUPRO - Generic understanding of programs : An overview”. In : *Electronic Notes in Theoretical Computer Science* 72 (nov. 2002), p. 47-56. doi : 10.1016/S1571-0661(05)80528-6.

[Eic+01] Stephen G Eick et al. “Does code decay ? assessing the evidence from change management data”. In : *IEEE Transactions on Software Engineering* 27.1 (2001), p. 1-12.

[EK08] Khaled El Emam et A Günes Koru. “A replicated survey of IT software project failures”. In : *IEEE software* 25.5 (2008), p. 84-90.

[Est+07] Jeff A Estefan et al. “Survey of model-based systems engineering (MBSE) methodologies”. In : *Incose MBSE Focus Group* 25.8 (2007), p. 1-12.

[Eve+09] Sylvain Eveillard et al. “Model-Driven Reverse Engineering of COBOL-Based Applications”. In : (juin 2009). doi : 10.1016/B978-0-12-374913-0.00011-1.

[Fav05] Jean-Marie Favre. "Foundations of meta-pyramids : Languages vs. metamodels–episode ii : Story of thotus the baboon1". In : *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2005.

[Fer+02] Rudolf Ferenc et al. "Columbus - reverse engineering tool and schema for C++". In : *International Conference on Software Maintenance, 2002. Proceedings*. (2002), p. 172-181.

[Fle+07] Franck Fleurey et al. "Model-Driven Engineering for Software Migration in a Large Industrial Context". In : *Proc. of the Int. Conf. on Model Driven Engineering Languages and Systems (MODELS)*. 2007, p. 482-497.

[FLP99a] Pascal Fradet, Daniel Le Métayer et Michaël Périn. "Consistency Checking for Multiple View Software Architectures". In : ESEC/FSE-7. Toulouse, France : Springer-Verlag, 1999, p. 410-428. isbn : 3540665382.

[FLP99b] Pascal Fradet, Daniel Le Métayer et Michaël Périn. "Consistency Checking for Multiple View Software Architectures". In : *SIGSOFT Softw. Eng. Notes* 24.6 (oct. 1999), p. 410-428. issn : 0163-5948. doi : 10.1145/318774.319258.

[FO76] Lloyd D Fosdick et Leon J Osterweil. "Data flow analysis in software reliability". In : *ACM Computing Surveys (CSUR)* 8.3 (1976), p. 305-330.

[Gar+16] Kelly Garcés et al. "White-Box Modernization of Legacy Applications". In : t. 9893. Sept. 2016, p. 274-287. isbn : 978-3-319-45546-4. doi : 10.1007/978-3-319-45547-1_22.

[Gar+18] Kelly Garcés et al. "White-box modernization of legacy applications : The oracle forms case study". In : *Computer Standards & Interfaces* 57 (2018), p. 110-122.

[Gov+17] Brice Govin et al. "Managing an Industrial Software Rearchitecting Project With Source Code Labelling". In : *CSD&M 2017-Complex Systems Design & Management conference*. 2017.

[Hec77] Matthew S Hecht. *Flow analysis of computer programs*. Elsevier Science Inc., 1977.

[Hei+09] Florian Heidenreich et al. "Closing the Gap between Modelling and Java". In : *Proceedings of the Second International Conference on Software Language Engineering*. SLE'09. Denver, CO : Springer-Verlag, 2009, p. 374-383. isbn : 3642121063. doi : 10.1007/978-3-642-12107-4_25.

[Her03] Jack Herrington. *Code generation in action*. Manning Publications Co., 2003.

[Hut+11] John Hutchinson et al. "Empirical assessment of MDE in industry". In : *Proceedings of the 33rd international conference on software engineering*. 2011, p. 471-480.

[Jon84] T. Capers Jones. "Reusability in Programming : A Survey of the State of the Art". In : *IEEE Transactions on Software Engineering* SE-10.5 (1984), p. 488-494. doi : 10.1109/TSE.1984.5010271.

[JSV06] Hans Jonkers, Marc Stroucken et Richard Vdovjak. “Bootstrapping Domain-Specific Model-Driven Software Development within Philips”. In : (jan. 2006).

[Ken02] Stuart Kent. “Model driven engineering”. In : *International conference on integrated formal methods*. Springer. 2002, p. 286-298.

[KGK07] Riaan Klopper, Stefan Gruner et Derrick G Kourie. “Assessment of a framework to compare software development methodologies”. In : *Proceedings of the 2007 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*. 2007, p. 56-65.

[Kha+14] Ravi Khadka et al. “How do professionals perceive legacy systems and software modernization ?” In : *Proceedings of the 36th International Conference on Software Engineering*. 2014, p. 36-47.

[KL88] Bogdan Korel et Janusz Laski. “Dynamic program slicing”. In : *Information processing letters* 29.3 (1988), p. 155-163.

[KL90] Bogdan Korel et Janusz Laski. “Dynamic slicing of computer programs”. In : *Journal of Systems and Software* 13.3 (1990), p. 187-195.

[Kle+03] Anneke G Kleppe et al. *MDA explained : the model driven architecture : practice and promise*. Addison-Wesley Professional, 2003.

[KM10] Holger M Kienle et Hausi A Müller. “The tools perspective on software reverse engineering : requirements, construction, and evaluation”. In : *Advances in Computers*. T. 79. Elsevier, 2010, p. 189-290.

[Kos+05] Jussi Koskinen et al. “Software modernization decision criteria : An empirical study”. In : *Ninth European Conference on Software Maintenance and Reengineering*. IEEE. 2005, p. 324-331.

[KSK17] Uday P Khedker, Amitabha Sanyal et Bageshri Karkare. *Data flow analysis : theory and practice*. CRC Press, 2017.

[KT08] Steven Kelly et Juha-Pekka Tolvanen. *Domain-specific modeling : enabling full code generation*. John Wiley & Sons, 2008.

[KWC98] Rick Kazman, Steven Woods et S. Carrière. “Requirements for Integrating Software Architecture and Reengineering Models : CORUM II”. In : jan. 1998, p. 154-163. doi : 10.1109/WCRE.1998.723185.

[Lan+05] Michele Lanza et al. “CodeCrawler : An Information Visualization Tool for Program Comprehension”. In : *Proceedings of the 27th International Conference on Software Engineering*. ICSE ’05. St. Louis, MO, USA : Association for Computing Machinery, 2005, p. 672-673. isbn : 1581139632. doi : 10.1145/1062455.1062602.

[LD01] Michele Lanza et Stéphane Ducasse. “A categorization of classes based on the visualization of their internal structure : the class blueprint”. In : *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2001, p. 300-311.

[Leh80] M.M. Lehman. "Programs, life cycles, and laws of software evolution". In : *Proceedings of the IEEE* 68.9 (1980), p. 1060-1076. doi : 10.1109/PROC.1980.11805.

[Leh96] Manny M Lehman. "Laws of software evolution revisited". In : *European Workshop on Software Process Technology*. Springer. 1996, p. 108-124.

[Low69] ES Lowry. "CW MEPLOCK,"". In : *Object code optimisation" CACM Janvier* (1969).

[LTP04] Timothy C Lethbridge, Sander Tichelaar et Erhard Plödereder. "The dagstuhl middle metamodel : A schema for reverse engineering". In : *Electronic Notes in Theoretical Computer Science* 94 (2004), p. 7-18.

[Mar09] Robert C Martin. *Clean code : a handbook of agile software craftsmanship*. Pearson Education, 2009.

[MD13] Apoorva Mishra et Deepty Dubey. "A comparative study of different software development life cycle models in different scenarios". In : *International Journal of Advance research in computer science and management studies* 1.5 (2013).

[Mel+04] Stephen J Mellor et al. *MDA distilled : principles of model-driven architecture*. Addison-Wesley Professional, 2004.

[Men08] Tom Mens. "Introduction and Roadmap : History and Challenges of Software Evolution". In : (jan. 2008). doi : 10.1007/978-3-540-76440-3_1.

[MS12] Vishwas Massey et KJ Satao. "Comparing various SDLC models and the new proposed model on the basis of available methodology". In : *International Journal of Advanced Research in Computer Science and Software Engineering* 2 (2012).

[MV06] Tom Mens et Pieter Van Gorp. "A taxonomy of model transformation". In : *Electronic notes in theoretical computer science* 152 (2006), p. 125-142.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse et Tudor Gundefinedrba. "The Story of Moose : An Agile Reengineering Environment". In : *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-13. Lisbon, Portugal : Association for Computing Machinery, 2005, p. 1-10. isbn : 1595930140. doi : 10.1145/1081706.1081707.

[OF76] Leon J Osterweil et Lloyd D Fosdick. "DAVE—a validation error detection and documentation system for Fortran programs". In : *Software : Practice and Experience* 6.4 (1976), p. 473-486.

[OG12] Thomas Olsson et John Grundy. "Supporting traceability and inconsistency management between software artifacts". English. In : *Proceedings of the 6th IASTED International Conference on Software Engineering and Applications, SEA 2002*. IASTED International Conference on Software Engineering and Applications 2002, SEA 2002 ; Conference date : 04-11-2002 Through 06-11-2002. ACTA Press, 2012, p. 484-489. isbn : 0889863237.

[OT94] Anthony B. O'Hare et Erik W Troan. "RE-Analyzer : From source code to structured analysis". In : *IBM Systems Journal* 33.1 (1994), p. 110-130.

[Paw+15] Renaud Pawlak et al. "Spoon : A Library for Implementing Analyses and Transformations of Java Source Code". In : *Software : Practice and Experience* 46 (2015). update for oadoi on Nov 02 2018, p. 1155-1179. doi : 10.1002/spe.2346.

[Pro59] Reese T Prosser. "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*. 1959, p. 133-138.

[PRW04] M.J. Pacione, M. Roper et M. Wood. "A novel software visualisation model to support software comprehension". In : *11th Working Conference on Reverse Engineering*. 2004, p. 70-79. doi : 10.1109/WCRE.2004.7.

[Rek85] Michael G Rekoff. "On reverse engineering". In : *IEEE Transactions on systems, man, and cybernetics* 2 (1985), p. 244-252.

[Ren+11] Yongchang Ren et al. "Research on Software Maintenance Cost of Influence Factor Analysis and Estimation Method". In : *2011 3rd International Workshop on Intelligent Systems and Applications*. 2011, p. 1-4. doi : 10.1109/ISA.2011.5873461.

[RG08] Banani Roy et T.C. Graham. "An Iterative Framework for Software Architecture Recovery : An Experience Report". In : t. 5292. Sept. 2008, p. 210-224. isbn : 978-3-540-88029-5. doi : 10.1007/978-3-540-88030-1_16.

[Roy+70] Winston W Royce et al. "Managing the development of large software systems. proceedings of IEEE WESCON". In : *Los Angeles* (1970).

[RR13] Ratnmala Raval et Haresh Rathod. "Comparative Study of Various Process Model in Software Development". In : *International Journal of Computer Applications* 82 (nov. 2013), p. 16-19. doi : 10.5120/14263-2363.

[RS04] Spencer Rugaber et Kurt Stirewalt. "Model-driven reverse engineering". In : *IEEE software* 21.4 (2004), p. 45-53.

[Rup10] Nayan B. Ruparelia. "Software Development Lifecycle Models". In : *SIGSOFT Softw. Eng. Notes* 35.3 (mai 2010), p. 8-13. issn : 0163-5948. doi : 10.1145/1764810.1764814.

[Sac+11] Anupriya Sachin et al. "Efficiency of Spiral Model by applying Genetic Algorithm". In : (2011). issn : 2229-4333, 0976-8491.

[Sch06] Douglas C Schmidt. "Model-driven engineering". In : *Computer-IEEE Computer Society-* 39.2 (2006), p. 25.

[Sek+91] Hiroyuki Seki et al. "On multiple context-free grammars". In : *Theoretical Computer Science* 88.2 (1991), p. 191-229. issn : 0304-3975.

[SK03] Shane Sendall et Wojtek Kozaczynski. "Model transformation : The heart and soul of model-driven software development". In : *IEEE software* 20.5 (2003), p. 42-45.

[Sne04] Harry SNEED. “A cost model for software maintenance and evolution”. In : oct. 2004, p. 264-273. ISBN : 0-7695-2213-0. DOI : 10.1109/ICSM.2004.1357810.

[Sne10] Harry M. SNEED. “Migrating from COBOL to Java”. In : *2010 IEEE International Conference on Software Maintenance* (2010), p. 1-7.

[Sne92] Harry SNEED. “Migration of procedurally oriented COBOL programs in an object-oriented achitecture”. In : déc. 1992, p. 105-116. ISBN : 0-8186-2980-0. DOI : 10.1109/ICSM.1992.242552.

[Som15] Ian SOMMERVILLE. *Software Engineering 10th Edition*. Pearson, 2015.

[SSM13] Óscar SÁNCHEZ RAMÓN, Jesús SÁNCHEZ CUADRADO et Jesús MOLINA. “Model-driven reverse engineering of legacy graphical user interfaces”. In : *Automated Software Engineering* 21 (juil. 2013), p. 147-186. DOI : 10.1007/s10515-013-0130-2.

[Sun+09] Weijun SUN et al. “A Model-Driven Reverse Engineering Approach for Semantic Web Services Composition”. In : *2009 WRI World Congress on Software Engineering* 3 (2009), p. 101-105.

[Swa76] E. Burton SWANSON. “The Dimensions of Maintenance”. In : ICSE ’76. San Francisco, California, USA : IEEE Computer Society Press, 1976, p. 492-497.

[Tes+20] Pablo TESONE et al. “A new modular implementation for Stateful Traits”. In : *Science of Computer Programming* 195 (avr. 2020). DOI : 10.1016/j.scico.2020.102470.

[Tiw20] Ketki TIWARI. “Study and Assessment of Reverse Engineering Tool”. In : (2020).

[TN14] Priyadarshi TRIPATHY et Kshirasagar NAIK. *Software evolution and maintenance : a practitioner’s approach*. John Wiley & Sons, 2014.

[Völ+13] Markus VÖLTER et al. *Model-driven software development : technology, engineering, management*. John Wiley & Sons, 2013.

[Wei84] Mark WEISER. “Program slicing”. In : *IEEE Transactions on software engineering* 4 (1984), p. 352-357.

[WHR13] Jon WHITTLE, John HUTCHINSON et Mark ROUNCEFIELD. “The state of practice in model-driven engineering”. In : *IEEE software* 31.3 (2013), p. 79-85.

[WLR11] Richard WETTEL, Michele LANZA et Romain ROBBES. “Software Systems as Cities : A Controlled Experiment”. In : *Proceedings of the 33rd International Conference on Software Engineering*. ICSE ’11. Waikiki, Honolulu, HI, USA : Association for Computing Machinery, 2011, p. 551-560. ISBN : 9781450304450. DOI : 10.1145/1985793.1985868.

[You89] Edward YOURDON. *Modern structured analysis*. Yourdon press, 1989.

Webographie

[21a] *Pharo*. Nov. 2021. URL : https://en.wikipedia.org/wiki/Pharo.

[21b] *The 1 Development Tool Suite*. 2021. URL : https://www.visual-paradigm.com/.

[21c] *Trait (computer programming)*. Déc. 2021. URL : https://en.wikipedia.org/wiki/Trait_(computer_programming).

[Das20] Inc. DASSAULT SYSTÈMES No Magic. *Software engineering - magicdraw software modeling tool - CATIA - dassault systèmes®*. Juin 2020. URL : https://www.3ds.com/products-services/catia/products/no-magic/solutions/software-engineering/.

[Gee19] GEEKSFORGEEKS. *Software engineering : Slicing*. Déc. 2019. URL : https://www.geeksforgeeks.org/software-engineering-slicing.

[Gee21] GEEKSFORGEEKS. *Cost and efforts of software maintenance*. Nov. 2021. URL : https://www.geeksforgeeks.org/cost-and-efforts-of-software-maintenance/.

[GfG21] GeeksforGeeks GFG. *Introduction of lexical analysis*. Juin 2021. URL : https://www.geeksforgeeks.org/introduction-of-lexical-analysis/.

[GI06] ATLAS GROUP et LINA INRIA. *ATL : Atlas Transformation Language ATL user manual*. Fév. 2006. URL : https://www.eclipse.org/atl/documentation/old/ATL_User_Manual[v0.7].pdf.

[IBM21] IBM. Août 2021. URL : https://www.ibm.com/support/pages/rational-software-architect-designer-971.

[MM01] Joaquin MILLER et Jishnu MUKERJI. *Model Driven Architecture (MDA)*. Juil. 2001. URL : https://www.omg.org/cgi-bin/doc?ormsc%2F2001-07-01.

[MM03] Joaquin MILLER et Jishnu MUKERJI. Juin 2003. URL : http://www.omg.org/cgi-bin/doc?omg%2F+03-06-01.pdf.

[OMGa] Object Management Group - (OMG). *Systems Modeling Language (SysML)*. URL : https://www.omgsysml.org/.

[OMGb] Object Management Group - (OMG). *Unified Modeling Language (UML)*. URL : https://www.uml.org/.

[OMG01] OMG. 2001. URL : http://xml.coverpages.org/OMG-MDA2001-03-08a.htm.

[OMG08] Object Management Group - (OMG). *MOF Model To Text Transformation Language (MOFM2T)*. Jan. 2008. URL : https://www.omg.org/spec/MOFM2T.

[OMG11] Object Management Group - (OMG). *Architecture-driven modernization : Abstract syntax ... - OMG*. Jan. 2011. URL : https://www.omg.org/spec/ASTM/1.0/PDF.

[OMG14a] Object Management Group - (OMG). *Object Constraint Language*. Fév. 2014. URL : https://www.omg.org/spec/OCL.

[OMG14b] OMG. *Object Management Group Model Driven Architecture (MDA) MDA Guide rev. 2.0*. Juin 2014. URL : https://www.omg.org/cgi-bin/doc?ormsc%2F14-06-01.

[OMG16a] Object Management Group - (OMG). Sept. 2016. URL : https://www.omg.org/spec/KDM/1.4/PDF.

[OMG16b] Object Management Group - (OMG). *OMG'S METAOBJECT FACILITY™*. 2016. URL : https://www.omg.org/mof/.

[OMG16c] Object Management Group - (OMG). *Query/View/Transformation (QVT)*. Juin 2016. URL : https://www.omg.org/spec/QVT.

[Pre01] R Pressman. *Software Engineering : a Practitioner's Approach. 5/e. Fifth*. 2001.

[Sou20] Open Source. *UML and BPMN Free Modeling Tool*. 2020. URL : https://www.modelio.org/.

[Sys21] Sparx Systems. *Enterprise architect*. 2021. URL : https://sparxsystems.com/products/ea/index.html.

[TOO18] MDE TOOLS. *Fujaba (From UML to Java and back again)*. 2018. URL : http://www.mdetools.com/detail.php?toolId=50.

Annexes

Annexe

Définitions et Terminologie

1 Legacy

Les systèmes legacy sont « de grands systèmes logiciels auxquels on ne sait pas faire face mais qui sont vitaux pour notre organisation » [Ben95]. 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.

2 Pharo

Pharo est une implémentation open source et multiplateforme du langage de programmation et de l'environnement d'exécution classiques Smalltalk-80. Il est basé sur la machine virtuelle OpenSmalltalk appelée Cog (VM), qui évalue un langage de programmation dynamique, réflexif et orienté objet avec une syntaxe ressemblant étroitement à Smalltalk-80 [21a].

Pharo est livré avec un code source compilé dans une image système qui contient tous les logiciels nécessaires pour exécuter le système Pharo, et en compilation juste à temps (JIT) [21a].

3 CASE

Les technologies de génie logiciel assisté par ordinateur (Computer-Aided Software Engineering : CASE) sont des outils qui fournissent une assistance automatisée pour le développement de logiciels. Le but des outils CASE est la réduction du temps et du coût de développement logiciel et l'amélioration de la qualité des systèmes développés.

4 Trait

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 [21c].Autrement dit [Anq+20], « Un trait est un ensemble de méthodes, séparé de toute hiérarchie de classe. 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 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 [Tes+20].

5 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.

6 Artifact

Les artefacts qui correspondent à un produit logiciel peuvent être des types suivants : *document pouvant être subdivisé en documents textuels et graphiques, composants de produits prêts à l'emploi ...etc.* Les documents textuels incluent les listes de code source, les plans, la conception et les spécifications des exigences.

7 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.

8 Open source software

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.

9 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.