République algérienne démocratique et populaire
Université Mohamed KHIDER - BISKRA
Faculté des sciences exactes, des sciences naturelles et de la vie
Département d'informatique

Numéro d'ordre : IA 10

Mémoire

Présenté pour obtenir le diplôme de master académique en

Informatique

Parcours : **Intelligence Artificielle**

Reconstruction des architectures logicielles à partir des systèmes basés sur des modules Java 9

Réalisé par :
RABIE Zineb

Soutenue le 27/06/2022, devant le jury composé de :

Dr. TIGANE Samir	MCB	Président
Dr. KERDOUDI Mohamed Lamine	MCA	Superviseur
Dr. HMIDI Zohra	MCB	Examinateur

Année universitaire : 2021 - 2022

Remerciements

On remercie dieu le tout puissant de nous avoir donné la santé et la volonté d'entamer et de terminer ce mémoire.

Tout d'abord, ce travail ne serait pas aussi riche et n'aurait pas pu avoir le jour sans l'aide et l'encadrement de **Dr. KERDOUDI Mohamed Lamine**, on le remercie pour le temps qu'il a consacré et la qualité de son encadrement exceptionnel, pour sa patience, sa rigueur et sa disponibilité durant notre préparation de ce mémoire.

Mes remerciements vont également aux membres du jury pour m'avoir honoré de leur relecture et de leur évaluation de mon mémoire.

Mes derniers remerciements vont à tous mes professeurs du département d'informatique de université de Biskra.

Dédicace

Avec l'expression de ma reconnaissance, je dédie ce modeste travail à ceux qui je n'arriverais jamais à leur exprimer mon amour sincère.

A la femme qui a souffert sans me laisser souffrir, et qui n'a jamais passé un jour sans me bénir de ses prières :
mon adorable mère **Souad**.

A l'homme, qui je doit ma vie, ma réussite et tout mon respect :
mon cher père **Aissa**.

A mes chère sœurs **Aya** et **Khadidja** et tout mes frères, spécialement mes petits **Sedik** et **Amdjed**.

A mes amies **Ines**, **Chafika** et **Rania** qui n'ont pas cessée de me conseiller, encourager et soutenir tout au long de mes études.

Que Dieu les protège et leurs offre la chance et le bonheur. Merci pour leurs amours et leurs encouragements.

Reconstruction Des Architectures Logicielles A Partir Des Systèmes Basés Sur Des Modules Java 9

Zineb Rabie

2022

Sommaire

Sommaire 3

Table des figures 4

Liste des tableaux 6

Introduction générale 7

1 Modularité et modules dans Java 9 10
- 1.1 Introduction . . . 10
- 1.2 Aspects généraux de modularité . . . 10
  - 1.2.1 But de modularité . . . 11
  - 1.2.2 Avantages modularité . . . 11
- 1.3 Projet JIGSAW . . . 12
  - 1.3.1 Objectifs du projet . . . 13
  - 1.3.2 Différences entre OSGi et Jigsaw . . . 13
- 1.4 JDK modulaire . . . 14
  - 1.4.1 Graphe de module JDK . . . 15
- 1.5 Définition de module . . . 16
  - 1.5.1 Exemple de modules . . . 19
- 1.6 Déclaration de Module . . . 22
  - 1.6.1 Types de dépendances entre les modules . . . 22
  - 1.6.2 Contenu de fichier module-info.java . . . 22
- 1.7 Différents types de modules . . . 24
- 1.8 Avantages d'utilisation des modules Java . . . 26
- 1.9 Conclusion . . . 27

2 Architectures logicielles et évolution logiciel 28
- 2.1 Introduction . . . 28
- 2.2 Architectures logicielles . . . 28
  - 2.2.1 Définition de l'architecture logicielle . . . 28

2.2.2 Activités d'un architecte 29
2.2.3 Concepts de base de l'architecture logiciel 31
2.2.4 Styles d'architecture logicielle 34
2.2.5 Construction de l'architecture 35
2.2.6 Récupération de l'architecture logicielle 35
2.2.7 Rétro-ingénierie 36
2.3 Évolution de logiciel 36
2.4 Conclusion 37

3 Travaux connexes 38
3.1 Introduction 38
3.2 Récupération de l'architecture basée sur les composants 38
3.3 Récupération de l'architecture orientée services 40
3.4 Évolution statique de l'architecture logicielle 41
3.5 Évolution dynamique de l'architecture logicielle 41
3.6 Conclusion 42

4 Reconstruction des architectures logicielles à partir des systèmes basés sur les modules 43
4.1 Introduction 43
4.2 Processus générale 43
4.2.1 Présentation de l'approche proposée 43
4.2.2 Méta-modèle pour les modules Java 9 45
4.3 Reconstruction de l'architecture logicielle 49
4.4 Architecture logicielle basée sur les modules 50
4.5 Compréhension de système logiciel 50
4.6 Évolution de système logiciel 50
4.7 Conclusion 52

5 Outil ArchBaseDevModules et Editeur pour les applications basées module Java 9 53
5.1 Introduction 53
5.2 Environnement et outils de développement 53
5.3 Notre outil *ArchBaseDevModules* 54
5.3.1 Architecture de l'outil *ArchBaseDevModules* 54
5.3.2 Framework de modélisation de Eclipse (EMF) 55
5.3.3 Éditeur graphique pour l'architecture logicielle 55
5.4 Exécution de notre outil : 68
5.4.1 Configuration de l'exécution 68
5.4.2 Visualisation graphique et récupération de l'architecture 68
5.5 Conclusion 69

Bibliographie **72**

Table des figures

1.1 Une partie du graphe de modules du JDK 9 représentant les modules standards SE [1]. 16
1.2 Un module spécifie les classes non encapsulées (A,B,C) et encapsulées (D,E,F). 17
1.3 Structure générale d'un module logiciel [2]. 18
1.4 Création de projet "com.modules.example" et son fichier "module-info.java". 20
1.5 Création de projet "com.modules.client" et son fichier "module-info.java". 20
1.6 Résultat d'exécution. 21
1.7 Relation entre modules. 21
1.8 Classification des modules dans le JPMS [2]. 24

2.1 Interfaces de composant. 33

4.1 Approche proposée. 44
4.2 Méta-modèle 46

5.1 Architecture de l'outil *ArchBaseDevModules* 54
5.2 Création un nouveau projet EMF vide. 56
5.3 Création un nouveau projet EMF vide 57
5.4 Étapes de création du méta-modèle Ecore. 58
5.5 Étapes de création du méta-modèle Ecore. 58
5.6 Visualisation graphique du méta-modèle Ecore diagramme. . . . 59
5.7 les champs du package. 60
5.8 Modèle Ecore. 60
5.9 GMF Dashboard. 61
5.10 Génération des plug-ins de "ModulesModel.genmodel". 62
5.11 "ModulesModel.gmfgraph". 63
5.12 "ModuleModel.gmftool". 64
5.13 Éléments de modèle "map domain". 65

5.14 Propriétés des connecteurs de "map domain". 66
5.15 le fichier "ModuleModel.gmfmap". 67
5.16 "generator model" et "diagram code" 68

Liste des tableaux

1.1 Les modules du système d'exécution Java. 15
1.2 Table to test captions and labels. 23

Introduction générale

Les systèmes logiciels doivent évoluer avec le temps. Ils sont modifiés pour améliorer leurs performances ou modifier leurs fonctionnalités en réponse à de nouvelles exigences, à des bogues détectés, etc. Certains changements font partie de la maintenance du système ; d'autres font évoluer le système, généralement en ajoutant de nouvelles fonctionnalités, en modifiant son architecture, etc. Ainsi, il existe plusieurs phases d'évolution pour lesquelles différents processus peuvent être employés. Le processus d'évolution commence idéalement par la compréhension du système et se poursuit par la recherche d'un ensemble approprié de modifications du système. Il a été mesuré que dans les phases de maintenance et d'évolution, au moins la moitié du temps des ingénieurs est consacrée à la compréhension du système[3].

Ainsi, pour réussir à faire évoluer un système complexe, il est essentiel de le comprendre. La phase de compréhension nécessite un temps et un efforts considérable pour plusieurs raisons, parmi lesquelles : la taille du système (les grands systèmes sont constitués de millions de lignes de code et des centaines de composants), le manque de vues d'ensemble du système, ses évolutions antérieures ne sont pas documentées, etc. Le contexte de ce travail est les applications à base des composants (OSGI, modules java 9, ..) de très grande taille. Ses application constitués de centaines de composants et milliers de connexions (orienté services, interfaces ...), ce qui rend la tâche de compréhension très difficile . Les architectures servent ainsi de moyens utile pour faciliter la compréhension et améliorer l'évolution des logiciels tout en fournissant un modèle de haut niveau.

En pratique, les évolutions sont faites directement sur le code source et ne sont pas documentées. Par conséquent, les architectures logicielles existantes ne reflètent pas le système actuel. Par ailleurs, la reconstruction d'architecture logicielle est une approche d'ingénierie inverse qui vise à reconstruire des vues architecturales viables d'applications logicielles.
Dans la littérature plusieurs travaux de reconstruction d'architecture à partir des système existant ont été proposés parmi eux, [4], [5], [6],[7], et [8]. Dans

ce travail, on s'intéresse à la reconstruction des architecture logiciel à partir des systèmes logiciels basé sur les modules Java 9.

Nous nous concentrons sur l'extraction de vues architecturales de systèmes logiciels existants basés sur des modules. Il est largement admis que plusieurs vues architecturales sont utiles pour décrire l'architecture logicielle. Pour aider les développeurs à faciliter la phase de compréhension de l'ensemble du logiciel sans analyser le code source du système afin de faire évoluer leurs systèmes au moment de l'exécution.
Nos objectives dans ce travail sont de :

- proposer une approche de reconstruction d'architecture à base de modules Java 9.
- implémenter l'approche sous forme d'un plugin Eclipse.
- développer un éditeur graphique pour visualiser les architectures à base de module Java 9 reconstruites.

Ce mémoire est organisé comme suit :

- **Introduction générale :**
  Nous débuterons notre mémoire par une introduction au contexte de ce travail, au problème visé, et à la solution que nous proposons.

- **Chapitre 1 :** *Modularité et modules dans Java 9*
  Ce chapitre, nous allons introduire le concept de la modularité et la définition des modules et leur déclaration.Ensuite, nous avons parlé du projet JIGSAW et du déférence entre celui-ci et OSGI. Ensuite, nous avons parlé des types de modules.

- **Chapitre 2 :** *Architectures logicielles et évolution logiciel*
  Ce chapitre, présenter les concepts et la terminologie de les architectures logicielles, leurs différents styles de présentation des systèmes et leur importance et le concept d'évolution logicielle et aussi les techniques existantes et les travaux d'évolutions.

- **Chapitre 3 :** *État de l'art*
  Dans ce chapitre, nous présenterons l'état de l'art de la récupération d'architecture logicielle. Nous présenterons les techniques existantes de récupération d'architecture logicielle et aussi le concept de la reconstruction des systèmes logicielles.

- **Chapitre 4 :** *Reconstruction des architectures logicielles à partir*

des systèmes basés sur les modules
Dans ce chapitre, nous présenterons l'approche et le processus proposé et le méta-modèle du modèle de composants qui définit la syntaxe des architectures récupérées pour les systèmes à base de modules. Enfin, nous présenterons l'outil "ArchBaseDevModules" qui implémente le processus proposé ainsi que son architecture et ses fonctionnalités.

• **Chapitre 5 :** *Outil ArchBaseDevModules et Éditeur pour les application à base module Java 9*
Dans ce chapitre, nous allons introduire l'architecture de notre outil "ArchBaseDevModules" et la mise en œuvre de notre approche proposée, nous allons présenter aussi l'environnement de développement et aussi les outils inclus dans la réalisation de notre outil.

• **Conclusion générale :**
Nous terminerons notre mémoire par une conclusion générale et quelques perspectives et orientations futures.

Chapitre 1

Modularité et modules dans Java 9

1.1 Introduction

Le langage de programmation Java a ajouté de nouvelles fonctionnalités dans chaque nouvelle version de Java , petites, moyennes et grandes. La sortie de la version Java 9 est prévue en septembre 2017, plus de trois ans après la sortie de Java 8.
Dans cette version Java a introduit une évolution conceptuelle importante qui est : les modules. Le système de module de plate-forme Java 9 "Java 9 Platform Module System (JPMS)", est la nouvelle technologie d'ingénierie logicielle la plus importante de Java depuis sa création.
Dans ce chapitre nous allons présenter le projet Jigsaw qui qui est le point de départ du concept modulaire ,et nous allons également introduire les modules et leur concepts, leur structure générale et leur déclaration,nous donnons un exemple, et on finissons par les types et les avantages des modules.

1.2 Aspects généraux de modularité

La modularité spécifie l'interrelation et l'intercommunication entre les parties qui composent un système logiciel.La programmation modulaire définit un concept appelé le module.Les modules sont des composants logiciels qui contiennent des données et des fonctions.Intégrés à d'autres modules, ils forment ensemble un système logiciel unitaire.La programmation modulaire fournit une technique pour décomposer un système entier en modules logiciels indépendants.La modularité joue un rôle crucial dans l'architecture logicielle moderne.Il divise un grand système logiciel en entités distinctes

et aide à réduire la complexité des applications logicielles tout en réduisant simultanément l'effort de développement [2].

1.2.1 But de modularité

Le but de la modularité est de définir de nouvelles entités faciles à comprendre et à utiliser. La programmation modulaire est un style de développement d'applications logicielles en divisant la fonctionnalité en différents modules - des unités logicielles qui contiennent une logique métier et ont pour rôle de mettre en œuvre une fonctionnalité spécifique. La modularité permet une séparation claire des préoccupations et assure la spécialisation. Il masque également les détails d'implémentation du module. La modularité est un élément important du développement logiciel agile car elle nous permet de modifier ou de refactoriser des modules sans casser d'autres modules. Deux des aspects les plus importants de la modularité sont la maintenabilité et la réutilisabilité, qui apportent toutes deux de grands avantages [2].

1.2.2 Avantages modularité

Si le processus de développement d'applications suit une conception modulaire appropriée, la modularisation en Java peut offrir plusieurs avantages à la fois à long et à court terme, en plus d'être simplement facile à gérer[2]. Voici quelques-uns des avantages les plus importants de la modularité en Java :

- Maintenabilité : Une grande application logicielle monolithique est difficile à maintenir, surtout si elle a de nombreuses dépendances à l'intérieur du code. L'architecture du système et les modèles de conception utilisés nous aident à créer un code maintenable.

- Augmentation de la lisibilité : Un code modulaire est relativement plus lisible et maniable qu'un code monolithique.

- Facile à personnaliser : Il est plus facile de maintenir et de mettre à jour le code car les composants individuels ont des problèmes distincts et en changer un n'affecterait pas les autres composants. Grâce à la modularité de Java, il est désormais simple d'apporter les modifications nécessaires à un module, en minimisant autant que possible les modifications apportées aux autres modules.

- Test et débogage : Les programmes modulaires sont également relativement faciles à déboguer et à tester en raison de leur nature

découplée. Tous les composants individuels peuvent être facilement sélectionnés pour des tests individuels. Il sera également plus facile de garder une erreur localisée.

1.3 Projet JIGSAW

Nous pouvons considérer le projet Jigsaw comme un projet parapluie avec les nouvelles fonctionnalités visant deux aspects : l'introduction du système de modules dans le langage Java. et son implémentation dans la source JDK et l'environnement d'exécution Java. Il est inclus plusieurs JEP *"JDK Enhancement Proposal"* (un processus rédigé par Oracle Corporation pour recueillir des propositions d'améliorations du kit de développement Java et d'OpenJDK), et JSR *"Java Specification Requests"* (essentiellement les demandes de modification du langage Java, des bibliothèques et d'autres composants) [2].

• Le projet JIGSAW a d'abord commencé en 2005, JSR 277 a été publié, puis en 2008, le travail proprement dit sur le projet a commencé. Il n'est sorti qu'en 2017. Il a donc fallu près de 10 ans pour terminer correctement les modules Java. Ce qui, en fait, met l'accent sur toute l'ampleur du travail et des changements qui ont été apportés lors de la mise en œuvre des modules.

• Project Jigsaw représente l'implémentation du nouveau système de modules évolutifs introduit dans Java 9. Il a été développé sous "Open JDK", qui est l'implémentation gratuite et "open source de Java Platform Standard Edition". L'objectif du système de module nouvellement conçu pour la plate-forme Java SE est de modulariser le JDK et d'appliquer le système de module au JDK lui-même. Jigsaw modularise la plate-forme Java SE.

• Le processus de modularisation de la plate-forme Java a été un effort compliqué et énorme. Un grand nombre de décisions de conception difficiles ont dû être prises. La modularisation de la plate-forme est un énorme changement avec un impact majeur sur l'ensemble de l'écosystème. Il introduit le nouveau concept de modules et modifie considérablement la façon dont nous développons des applications logicielles à l'aide du langage de programmation Java. Les modules sont placés au premier plan et constituent le concept clé sur lequel repose Project Jigsaw. Des techniques de programmation entières doivent être ajustées pour correspondre au concept nouvellement introduit.

1.3.1 Objectifs du projet

Les objectifs de Project Jigsaw, tels qu'ils sont répertoriés sur le site Web Open JDK [9].

• Faciliter le développement de grandes applications et bibliothèques.

• Améliorer les performances des applications.

• Permettre à Java SE et JDK de se réduire pour une utilisation dans de petits appareils afin de ne pas consommer trop de mémoire.

• Rendre la plate-forme Java SE et le JDK plus facilement évolutifs jusqu'aux petits appareils informatiques.

• Améliorer la sécurité et la maintenabilité des implémentations de la plate-forme Java SE en général et du JDK en particulier.

• faciliter la construction et la maintenance des bibliothèques et des applications volumineuses pour les développeurs, à la fois pour les plates-formes Java SE.

1.3.2 Différences entre OSGi et Jigsaw

Dans cette section nous allons présenter les différences entre OSGi et Jigsaw [2].

• OSGi (Open Service Gateway Initiative) est un framework bien connu qui permet de développer des applications modulaires dans le langage de programmation Java. La spécification pour l'implémentation de systèmes de modules dans Java à l'aide d'OSGi se trouve dans le document "JSR 291 - Dynamic Component Support for Java SE", qui a été publié en août 2007.

• Une différence majeure entre OSGi et Jigsaw est le fait qu'OSGi prend en charge la gestion des versions, mais que Jigsaw ne le fait qu'à un faible degré. Jigsaw vous permet de définir une version en tant que méta-attribut ou d'utiliser plusieurs versions pour un module dans une couche. Mais le système de gestion des versions proposé par Jigsaw est beaucoup moins puissant que celui proposé par OSGi. OSGi possède également certaines fonctionnalités liées au cycle de vie dynamique que Jigsaw n'a pas. En plus de cela, OSGi

fournit un registre de services dynamique et un modèle de sécurité amélioré.

• Jigsaw est plus sécurisé qu'OSGi car son mécanisme de sécurité ne peut pas être contourné. Le mécanisme de sécurité d'OSGi peut être contourné. Les bundles OSGi n'offrent pas le même niveau de sécurité que les modules Jigsaw.

• Jigsaw n'est pas destiné à remplacer OSGi. OSGi peut très bien fonctionner sur JDK 9. JCP vise à rendre les deux systèmes capables de fonctionner en parallèle et de coopérer. Il devrait même être possible pour OSGi de traiter un module Jigsaw comme un bundle OSGi.

• Project Jigsaw offre également des fonctionnalités importantes qui n'existent pas dans OSGi, telles que la modularité au moment de la compilation et la prise en charge intégrée des bibliothèques natives. Jigsaw, contrairement à OSGi, modularise la plate-forme Java et introduit le nouveau concept de modules en tant que un élément central du programme.

1.4 JDK modulaire

Le récapitulatif du module JDK comprend des informations complètes sur les modules qui existent actuellement dans la plate-forme Java [10]. Pour chaque module, il spécifie les éléments suivants :

- Le nombre de classes et de ressources qu'il contient.
- La taille totale du module ainsi que la taille totale de ses dépendances.
- Les modules dont il a besoin.
- Les types qu'il exporte.
- Les services qu'il utilise et les services qu'il fournit.

Dans Java 9, le JDK est modularisé. Afin de répertorier tous les modules qui existent dans le système d'exécution, le lanceur Java peut être utilisé avec l'option de ligne de commande **- -list-modules**. En exécutant la commande suivante, nous obtenons une liste complète des modules existants dans notre environnement d'exécution : **$ java - -list-modules**
Le Tableau 1.1 affiche les résultats.

java.activation	java.xml.crypto	jdk.jfr
java.base	java.xml.ws	jdk.jsobject
java.compiler	java.xml.ws.annotation	jdk.localedata
java.corba	javafx.base	jdk.management
java.datatransfer	javafx.controls	jdk.management.agent
java.desktop	javafx.deploy	jdk.naming.dns
java.instrument	javafx.fxml	jdk.naming.rm
java.jnlp	javafx.graphics	jdk.net
java.logging	javafx.medi	a jdk.pack
java.management	javafx.swing	jdk.plugin
java.management.rmi	javafx.web	jdk.plugin.dom
java.naming	jdk.accessibility	jdk.plugin.server
java.prefs	jdk.charsets	jdk.scripting.nashorn
java.rmi	jdk.crypto.cryptoki	jdk.scripting.shell
java.scripting	jdk.crypto.ec	jdk.sctp
java.se	jdk.crypto.mscapi	jdk.security.auth
java.se.ee	jdk.deploy	jdk.security.jgss
java.security.jgss	jdk.deploy.controlpanel	jdk.snmp
java.security.sasl	jdk.dynalink	jdk.unsupported
java.smartcardio	jdk.httpserver	jdk.xml.dom
java.sql	jdk.incubator.httpclient	jdk.zipfs
java.sql.rowset	jdk.internal.le	oracle.desktop
java.transaction	jdk.internal.vm.ci	oracle.net
java.xml	jdk.javaws	
java.xml.bind	jdk.jdwp.agent	

TABLE 1.1 – Les modules du système d'exécution Java.

1.4.1 Graphe de module JDK

La modularisation de la plate-forme Java 9 peut être bien représentée sous la forme d'un graphe de modules. La Figure 1.1 montre un extrait du nouveau graphe de modules du JDK contenant uniquement les modules SE standard. Il est obtenu après avoir divisé le JDK en modules [1].

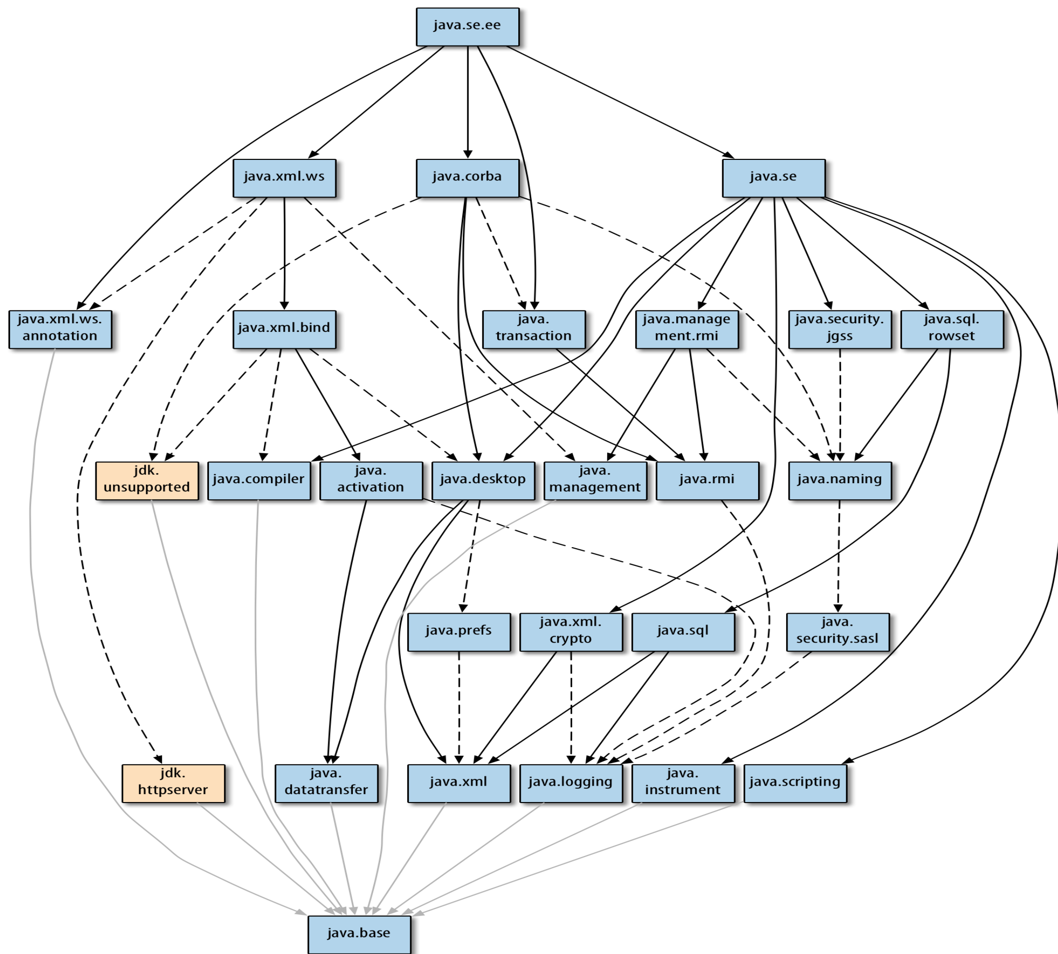


FIGURE 1.1 – Une partie du graphe de modules du JDK 9 représentant les modules standards SE [1].

1.5 Définition de module

Le module est nouvel élément clé du langage, et un nouveau niveau d'agrégation de packages et de ressources.

Un module logiciel est un composant logiciel indépendant et déployable d'un système plus vaste qui interagit avec d'autres modules et cache son implémentation interne. Il dispose d'une interface qui permet une communication inter modulaire. L'interface définit les composants qu'elle fournit pour un usage externe et les composants dont elle a besoin pour un usage interne. Un module détermine une limite en spécifiant quelle partie du code source se trouve à l'intérieur du module. Il offre également de la flexibilité et augmente

la réutilisabilité du système logiciel

Les modules peuvent être découverts à partir de la compilation. Un module peut exposer certaines de ses classes à l'extérieur ou les encapsuler afin d'empêcher tout accès externe. La Figure 1.2 illustre ce concept avec un exemple de module contenant des classes exposées à l'extérieur (A,B,C) et des classes non exposées à l'extérieur (E,F,G)[2].

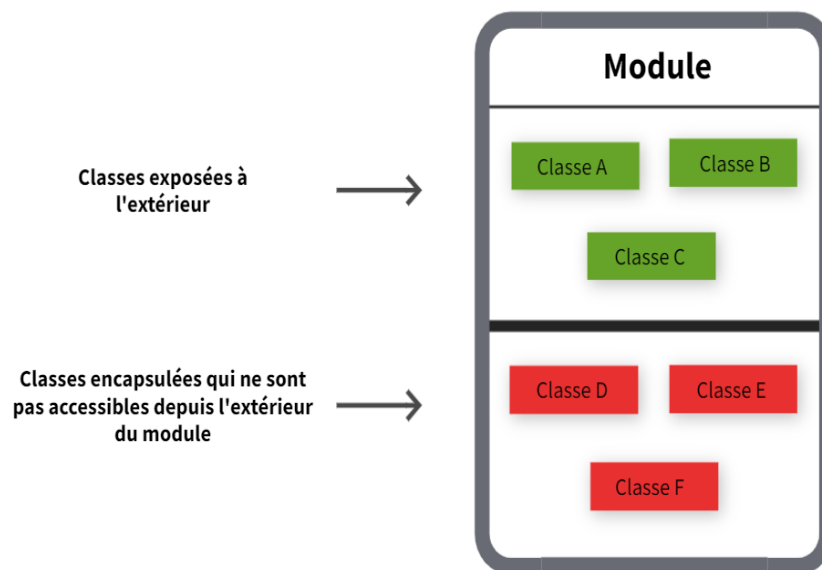


FIGURE 1.2 – Un module spécifie les classes non encapsulées (A,B,C) et encapsulées (D,E,F).

Un module logiciel est réutilisable, testable, gérable et déployable. Plusieurs modules peuvent être combinés pour former un nouveau module. La programmation modulaire est la clé pour réduire au minimum le nombre de bogues dans les systèmes logiciels complexes. En divisant l'application en très petits modules, chaque module aura moins de défauts car sa fonctionnalité n'est pas complexe. L'assemblage de ces modules moins sujets aux erreurs se traduit par une application avec moins d'erreurs.

L'une des facettes clés de la modularité consiste à décomposer l'application en petits modules minces faciles à mettre en œuvre car ils ne possèdent pas un niveau de complexité élevé. Les modules peuvent être interconnectés plus tôt lors de la compilation ou plus tard lors de l'exécution. Chaque module doit pouvoir être lié à l'application principale, La Figure 1.3 [2] montre la structure générale d'un module.

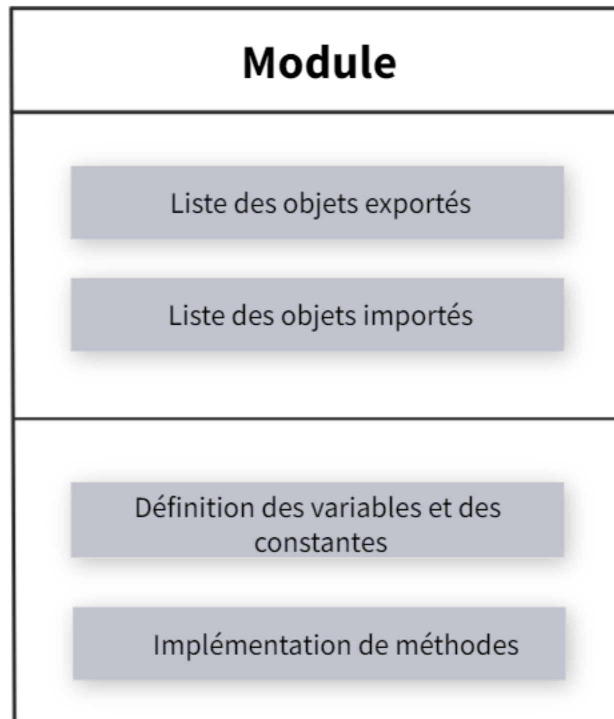


FIGURE 1.3 – Structure générale d'un module logiciel [2].

Un module se compose généralement de deux parties : l'interface du module et l'implémentation du module. L'interface du module définit les objets qu'il exporte et les objets qu'il importe. Les objets exportés sont les objets aptes à être disponibles en dehors du module. Les objets importés sont les objets dont le module a besoin de l'extérieur pour une utilisation interne. L'implémentation du module définit les variables, les constantes et l'implémentation des méthodes.

L'utilisation d'un module comme variable, variable d'instance, constante ou fonction n'est pas autorisée. Un module peut être composé d'objets qui ne peuvent être utilisés qu'en interne à l'intérieur du module et d'objets qui peuvent être exportés vers les autres modules pour une utilisation externe. L'abstraction des données, un concept de base de la modularité, est obtenue en masquant des informations afin qu'elles ne soient pas accessibles de l'extérieur à moins qu'elles ne soient explicitement spécifiées par une exportation. Par défaut, la structure interne et l'implémentation interne d'un module sont masquées des autres modules.

Une modification effectuée sur un module spécifique ne doit pas avoir

d'impact sur les autres modules. De plus, il devrait être possible d'ajouter un nouveau module au système central sans casser le système. Étant donné que seule l'interface d'un module est visible de l'extérieur du module, il devrait être possible pour les développeurs de modifier l'implémentation interne du module sans casser le code dans l'application. La structure d'une application logicielle modulaire est essentiellement définie par les connexions et les corrélations entre les modules.

Certaines des caractéristiques d'un module sont les suivantes :

- Un module doit définir des interfaces pour communiquer avec d'autres modules.
- Un module définit une séparation entre l'interface du module et le module implémentation.
- Un module doit présenter un ensemble de propriétés contenant des informations.
- Deux ou plusieurs modules peuvent être imbriqués ensemble.
- Un module doit avoir une responsabilité claire et définie. Chaque fonction doit être mis en œuvre par un seul module.
- Un module doit pouvoir être testé indépendamment des autres modules.
- Une erreur dans un module ne doit pas se propager aux autres modules.

1.5.1 Exemple de modules

Pour faire une vision plus claire de la manière de l'utilisation des modules, on va créer un simple exemple.

- Nous créons un projet Java "com.modules.example" en sélectionnant l'option "jre-9.0.0". Il peut s'agir de n'importe quelle version supérieure à la version 9. Clique sur Terminer, Eclipse vous invite à créer module-info.java. Clique sur Créer.
- Après cela, nous devons choisir le package qu'il va rendre accessible au autre module, et nous l'écrivons après le mot clé "exports", La Figure 1.4.

FIGURE 1.4 – Création de projet "com.modules.example" et son fichier "module-info.java".

• Nous créons un autre projet Java "com.modules.client", avec son fichier module-info.java aussi.

• Nous devons ajouter le package qui le premier module est fourni après le mot clé "requires" dans le fichier "module-info.java" associé au deuxième projet comme le montre La Figure 1.5.

FIGURE 1.5 – Création de projet "com.modules.client" et son fichier "module-info.java".

• Ensuite ,on exécute la classe "ClientExample" qui va appeler la méthode "sayHello" déclarée dans le premier module et nous donnons le résultat montré dans La Figure 1.6

```
Problems  Javadoc  Declaration  Console  Debug  GMF Dashboard  Properties
<terminated> ClientExample (1) [Java Application] C:\Program Files\Java\jdk-16.0.1\bin\javaw.exe
Hello je suis example
```

Figure 1.6 – Résultat d'exécution.

• Cela signifie que les deux modules que nous avons créés fonctionnent ensemble comme le montre le schéma ci-dessous, La Figure 1.7.

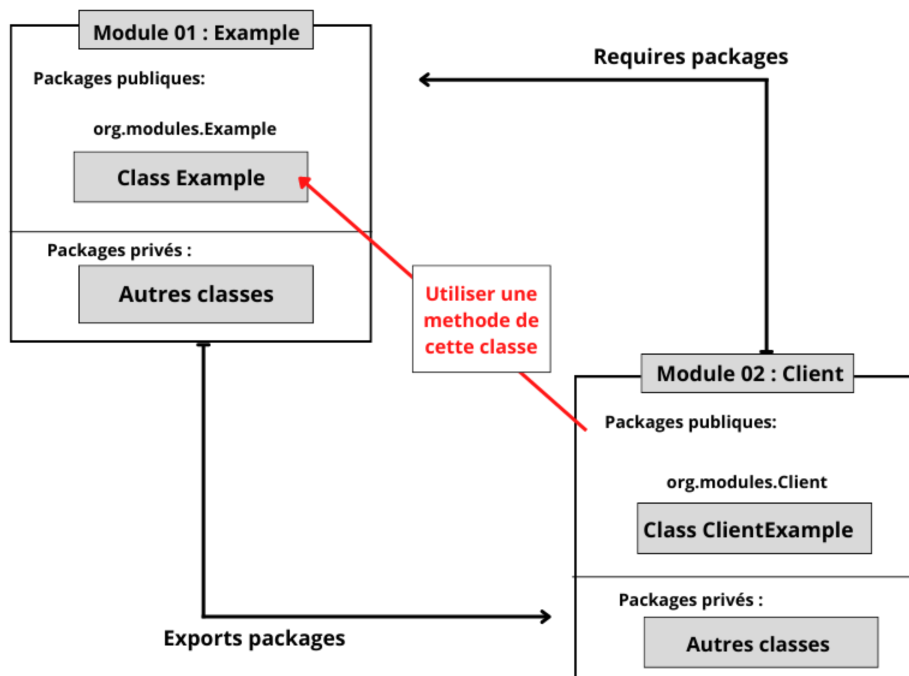


Figure 1.7 – Relation entre modules.

1.6 Déclaration de Module

Chaque module a une déclaration de module située dans un nouveau fichier spécial appelé module-info.java, situé au niveau supérieur du répertoire. Pour définir un module en Java 9, nous créons le fichier **module-info.java** et y mettons le nouveau mot clé module suivi du nom du module et de la déclaration du module entre accolades. Déclarer un module est simple et direct.
Ce fichier contient : le nom, les dépendances et les packages publics, les services consommés et offerts, les permissions de réflexion. Un module est un ensemble de paquets conçus pour réutiliser, et c'est un bloc de construction attendu depuis longtemps dans le langage java. En effet, les modules sont rappelés dans la structure de votre programme afin que les parties que vous souhaitez réutiliser puissent être réutilisées tandis que les parties que vous ne souhaitez pas réutiliser ne peuvent pas être réutilisées.
À l'intérieur d'une déclaration de module, nous pouvons avoir des différentes types de clauses, discutées ensuite :

1.6.1 Types de dépendances entre les modules

Une déclaration de module peut contenir les types de clauses suivants :

- **requires** les clauses spécifient le module requis par le module actuel.
- **exports** les clauses spécifient les packages qui sont exportés par le module actuel.
- **provides** les clauses spécifient les implémentations de service fournies par le module actuel.
- **uses** les clauses spécifient les services que le module actuel consomme.
- **opens** les clauses spécifient les packages que le module actuel ouvre pour une réflexion approfondie.

1.6.2 Contenu de fichier module-info.java

Pour déclarer un fichier jar en tant que module Java nommé, il faut fournir un fichier "module-info.class", qui est, naturellement, compilé à partir d'un fichier "module-info.java". Il déclare les dépendances (dependencies) au sein du système de modules et permet au compilateur et au Runtime de contrôler les limites/violations d'accès entre les modules de votre application. Regardons la syntaxe du fichier et les mots clés que vous pouvez utiliser. Le Tableau 1.2.

Déclaration	**Action**
module module.name	Déclare un module appelé "module.name".
requires module.name	Spécifie la dépendance du module, permet au module d'accéder aux types publics exportés dans le module cible.
requires transitive module.name	Tout module qui lit implicitement votre module lit également le module transitif ou le module spécifiquement référencé.
exports pkg.name	Indique que notre module exporte les membres publics dans le package "pkg.name" pour chaque module nécessitant celui-ci.
exports pkg.name to module.name	Comme ci-dessus, mais limite les modules qui peuvent utiliser les membres publics du package pkg.name.
uses class.name	Fait du module courant un consommateur pour le service "class.name".
provides class.name with class.name.impl	Enregistre la classe "class.name.impl" un service qui fournit une implémentation du service "class.name".
opens pkg.name	permet à d'autres modules d'utiliser la réflexion pour accéder aux membres privés du package "pkg.name".
opens pkg.name to module.name	Comme ci-dessus, mais limite les modules qui peuvent avoir un accès de réflexion aux membres privés dans le "pkg.name".

Table 1.2 – Table to test captions and labels.

Une grande chose à propos de la syntaxe module-info.java est que les IDE modernes soutiendraient pleinement vos efforts pour les écrire.

1.7 Différents types de modules

Jigsaw définit deux principaux types de modules :
les modules nommés (*Named modules*) et le module sans nom (*Unnamed module*). Les modules nommés sont divisés en modules normaux (*Normal modules*) et en modules automatiques (*Automatic modules*). Les modules normaux sont également séparés en modules de base (*Basic modules*) et en modules ouverts (*Open modules*). La Figure 1.8 [2] illustre la classification des modules.

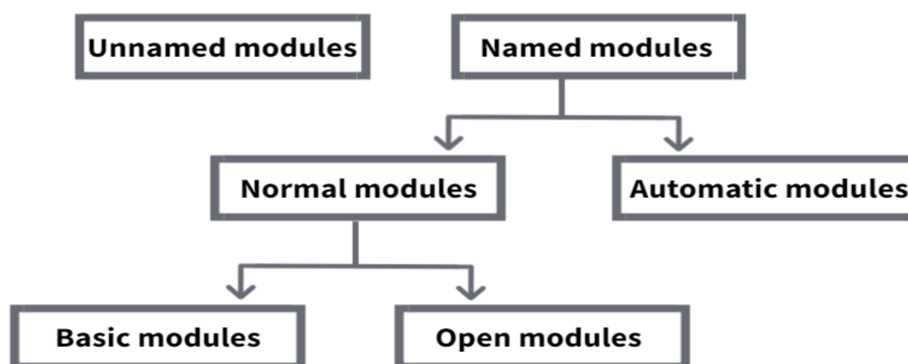


FIGURE 1.8 – Classification des modules dans le JPMS [2].

- Named Modules : Les modules nommés comprennent tous les modules du système de modules à l'exception du module sans nom. Il y a deux choses importantes qui distinguent un module sans nom d'un module nommé. Premièrement, le module sans nom vit sur le chemin de classe, tandis que les modules nommés vivent sur le chemin de module. Deuxièmement, le module sans nom n'a pas de nom, alors que chaque module nommé a un nom. Un module nommé peut être un module normal ou un module automatique. Les modules nommés sont des modules déclarés à l'aide d'un nom dans le fichier descripteur de module module-info.java. Chaque module qui a une déclaration de la forme module <module-name> dans son module-info.java est un module nommé. C'est la seule condition qui doit être remplie pour qu'un module soit classé comme module nommé. Des exemples de modules nommés incluent tous les modules de la plate-forme, mais nos propres modules peuvent également être inclus dans cette catégorie s'ils respectent la condition unique qui vient d'être mentionnée. Transformer

un fichier JAR en un module nommé est possible en y ajoutant simplement un fichier module-info.class.

- Normal Modules : La notion de modules "normaux" n'existe pas officiellement. Nous utilisons ce terme pour définir un module nommé qui n'est pas automatique. La principale différence entre un module normal et un module automatique est qu'un module normal a un descripteur de module module-info.java, alors qu'un module automatique n'en a pas. De plus, un module normal est explicitement déclaré par les développeurs, qui déclarent les dépendances du module dans le descripteur de module du module. Le descripteur de module d'un module automatique n'est pas fourni par les développeurs. Un module normal est déclaré à l'aide du mot clé module suivi du nom du module. Tous les modules que nous avons présentés jusqu'à présent dans ce chapitre étaient des modules normaux. Un module normal n'exporte aucun de ses packages par défaut. En outre, ses clauses d'exportation doivent être explicitement spécifiées. Les clauses exports exportent les packages au moment de la compilation ainsi qu'au moment de l'exécution. Un module normal comprend à la fois des modules de base et des modules ouverts.

- Automatic Modules : Un module automatique est un module créé après avoir placé un fichier JAR sur le chemin du module. En comparant un module automatique à un module normal, deux distinctions importantes émergent :

- Un module automatique nécessite par défaut tous les modules existants du système, qui comprennent tous nos propres modules, plus tous les modules de l'image JDK, plus tous les autres modules automatiques
- Un module automatique exporte tous ses packages par défaut.

Un module automatique peut accéder aux types sur le chemin de classe et est particulièrement utile pour le code tiers. Les modules automatiques sont utilisés pour migrer des applications existantes vers Java 9.

- Basic Module : Nous appelons chaque module nommé qui n'est pas un module ouvert un module de base. Cependant, le terme module "de base" n'existe pas officiellement dans JDK 9. Nous l'utilisons pour définir un module nommé qui n'est ni automatique ni ouvert. Un module de base a le même ensemble de caractéristiques qu'un module

normal, sauf qu'il n'est pas ouvert pour une réflexion profonde.

- Open Modules : À l'intérieur d'un module, les packages ne sont pas accessibles au code d'un autre module au moment de la compilation, même lors de l'utilisation de la réflexion approfondie. Cependant, de nombreuses bibliothèques et frameworks tiers utilisent la réflexion pour accéder aux éléments internes du JDK lors de l'exécution. Par conséquent, tous ces frameworks ne fonctionnent pas dans JDK 9 à moins que l'accès réflexif ne soit accordé. Dans JDK 9, l'accès réflexif est accordé uniquement par code dans des modules nommés pour coder à partir du chemin de classe. Il n'est pas accordé par défaut par le code dans les modules nommés au code dans d'autres modules nommés. Par conséquent, si les bibliothèques ou frameworks tiers se trouvent sur le chemin de classe, ils disposent par défaut d'un accès réflexif dans le JDK. S'ils vivent sur le chemin du module, ils n'ont pas d'accès réflexif dans le JDK. Mais pour accorder un accès réflexif à tous les packages d'un module, le module doit être déclaré ouvert.

1.8 Avantages d'utilisation des modules Java

L'utilisation de modules Java apporte divers avantages aux développeurs Java. Voici quelques-uns des principaux avantages qui font des modules Java une excellente option pour créer des applications.

- Distribution plus facile des applications : JPMS divise toutes les API de la plate-forme Java en modules distincts. Il permet aux développeurs de spécifier les modules dont l'application aura besoin. Avec ces informations, Java peut empaqueter votre application en incluant uniquement les modules API Java nécessaires.
Avant JPMS, vous auriez dû empaqueter toutes les API avec votre application. Les API inutilisées rendaient souvent l'application inutilement volumineuse et difficile à distribuer.

- Meilleure encapsulation des packages internes : Un module Java doit spécifier explicitement quels packages Java à l'intérieur du module peuvent être visibles par les autres utilisateurs de modules Java. Les modules cachés ne peuvent être utilisés qu'en interne dans le module Java. Cela rend les paquets très bien encapsulés.

- Détection des classes manquantes au démarrage : À partir de Java 9 et au-delà, les applications Java sont désormais également packagées dans des modules Java. Par conséquent, chaque module d'application doit spécifier les modules d'API Java dont il a besoin.

1.9 Conclusion

Dans ce chapitre , nous introduisons les concepts de modularité qui apparaissent dans la 9 ème version de Java, ce qui permet aux développeurs de travailler de manière indépendante sur leurs domaines d'application. Nous présentons les modules et leur déclarations , leur structure ce qui nous aide à les implémenter dans les chapitres suivants.

Chapitre 2

Architectures logicielles et évolution logiciel

2.1 Introduction

Les architectures logicielles ont apporté une réelle contribution dans le développement de les systèmes logiciels.Leurs principales caractéristiques résident d'une part dans leur pouvoir de gérer les abstractions et les niveaux d'expressivité d'un système, et d'autre part dans leur capacité à prendre en compte la modélisation de la structure et du comportement d'un système. Les architectures logicielles modélisent un système en terme de composants représentant les fonctionnalités de ce système et des connecteurs décrivant les interactions entre ces composants.

2.2 Architectures logicielles

Dans cette section, nous présentons les différentes définitions de l'architecture logicielle, ses modèles les plus connus et son importance.

2.2.1 Définition de l'architecture logicielle

Dans cette section nous allons donner des définitions des architectures logicielles.
" Un programme ou un système logiciel est composé de composants logiciels. L'architecture de ce programme ou d'un système est une ou plusieurs structures décrites par les propriétés externes et visibles des composants logiciels et les liens entre ces composants" [11].

Alors architecture logicielle est :

- Une représentation abstraite d'un système exprimée essentiellement à l'aide de composants logiciels en interaction via des connecteurs.
- Décompose de façon logique un système en sous-systèmes.
- Notions et concepts de découpage en couches, modules, composants, design patterns et Frameworks.
- Contrairement aux spécifications produites par l'analyse fonctionnelle :
  - Ne décrit pas ce que doit réaliser un système mais comment il doit être conçu pour à répondre aux spécifications.
  - L'analyse fonctionnelle décrit le " quoi faire " alors que l'architecture décrit le " comment le faire ".

L'architecture logiciel : fait référence aux structures fondamentales d'un système logiciel et à la discipline de création de ces structures et systèmes. Chaque structure comprend des éléments logiciels, des relations entre eux et des propriétés à la fois des éléments et des relations.L' architecture d'un système logiciel est une métaphore, analogue à l' architecture d'un bâtiment. Il fonctionne comme un modèle pour le système et le projet en développement, définissant les tâches nécessaires à exécuter par les équipes de conception.
L'architecture logicielle consiste à faire des choix structurels fondamentaux qui sont coûteux à modifier une fois mis en œuvre. Les choix d'architecture logicielle incluent des options structurelles spécifiques à partir des possibilités de conception du logiciel . Par exemple, les systèmes qui contrôlaient le lanceur de la navette spatiale devaient être très rapides et très fiables. Par conséquent, un langage de calcul en temps réel approprié devra être choisi. De plus, pour satisfaire le besoin de fiabilité, le choix pourrait être fait d'avoir plusieurs copies redondantes et produites indépendamment du programme, et d'exécuter ces copies sur un matériel indépendant tout en recoupant les résultats.
La documentation de l' architecture logicielle facilite la communication entre les parties prenantes , capture les premières décisions concernant la conception de haut niveau et permet la réutilisation des composants de conception entre les projets [12].

2.2.2 Activités d'un architecte

Un architecte logiciel effectue de nombreuses activités. Un architecte logiciel travaille généralement avec des chefs de projet, discute des exigences architecturales importantes avec les parties prenantes, conçoit une architec-

ture logicielle, évalue une conception, communique avec les concepteurs et les parties prenantes, documente la conception architecturale et plus encore.Il y a quatre activités principales dans la conception d'architecture logicielle.Ces activités d'architecture de base sont exécutées de manière itérative et à différentes étapes du cycle de vie initial du développement logiciel, ainsi qu'au cours de l'évolution d'un système [12].

- L'analyse architecturale est le processus consistant à comprendre l'environnement dans lequel un système proposé fonctionnera et à déterminer les exigences du système. L'entrée ou les exigences de l'activité d'analyse peuvent provenir de n'importe quel nombre de parties prenantes et inclure des éléments tels que :
- ce que le système fera lorsqu'il sera opérationnel (les exigences fonctionnelles).
- dans quelle mesure le système exécutera-t-il les exigences non fonctionnelles d'exécution telles que la fiabilité, l'opérabilité, l'efficacité des performances, la sécurité, la compatibilité.
- temps de développement des exigences non fonctionnelles telles que la maintenabilité et la transférabilité.
- les exigences commerciales et les contextes environnementaux d'un système qui peuvent changer au fil du temps, tels que les préoccupations juridiques, sociales, financières, concurrentielles et technologiques.

Les sorties de l'activité d'analyse sont les exigences qui ont un impact mesurable sur l'architecture d'un système logiciel, appelées exigences architecturalement significatives.

- La synthèse architecturale ou la conception est le processus de création d'une architecture. Compte tenu des exigences architecturales importantes déterminées par l'analyse, de l'état actuel de la conception et des résultats de toutes les activités d'évaluation, la conception est créée et améliorée.

- L'évaluation de l'architecture est le processus consistant à déterminer dans quelle mesure la conception actuelle ou une partie de celle-ci satisfait aux exigences dérivées lors de l'analyse. Une évaluation peut avoir lieu chaque fois qu'un architecte envisage une décision de conception, elle peut avoir lieu après qu'une partie de la conception a été achevée, elle peut avoir lieu après l'achèvement de la conception finale ou après la construction du système. Certaines des techniques

d'évaluation d'architecture logicielle disponibles

- L'évolution de l'architecture est le processus de maintien et d'adaptation d'une architecture logicielle existante pour répondre aux changements d'exigences et d'environnement. Comme l'architecture logicielle fournit une structure fondamentale d'un système logiciel, son évolution et sa maintenance auraient nécessairement un impact sur sa structure fondamentale. En tant que telle, l'évolution de l'architecture concerne l'ajout de nouvelles fonctionnalités ainsi que le maintien des fonctionnalités existantes et du comportement du système.

L'architecture nécessite des activités de soutien critiques. Ces activités de soutien ont lieu tout au long du processus d'architecture logicielle de base. Ils comprennent la gestion et la communication des connaissances, le raisonnement de conception et la prise de décision, et la documentation.

2.2.3 Concepts de base de l'architecture logiciel

Les architectures logicielles sont le modèle de construction et d'évolution du système logiciel. Ils décrivent la structure de haut niveau du système logiciel et exposent les dimensions selon lesquelles il est censé évoluer . Une architecture logicielle capture les principales décisions de conception prises concernant le système, telles que sa structure, son comportement fonctionnel, son interaction et ses propriétés non fonctionnelles [11].

Trois types d'éléments architecturaux ont été identifiés qui peuvent être combinés en deux concepts architecturaux principaux ; Composants ou "components" en anglais et connecteurs :

- Les éléments de traitement sont des composants qui traitent les données.
- Les éléments de données sont des composants qui contiennent des données à traiter.
- Les éléments de connexion sont des médiateurs qui maintiennent les connexions entre les différents composants.

Dans la suite, nous définissons et détaillons les notions de composants et de connecteurs.

2.2.3.1 Composants

Plusieurs définitions des composants logiciels existent dans la littérature. Nous donnons deux définitions largement citées qui disent l'essentiel de ce qu'est un composant. La première définition a été donnée par Taylor et al.[13] comme suit :

"Un composant logiciel est une entité architecturale qui (1) encapsule un sous-ensemble de fonctionnalités et/ou de données du système, (2) restreint l'accès à ce sous-ensemble via une interface explicitement définie et (3) a des dépendances explicitement définies sur son contexte d'exécution requis."

En d'autres termes, un composant est une unité de calcul qui encapsule des données, fournit des services pour les traiter et peut nécessiter des services d'autres composants pour son exécution. Selon l'architecture, un composant peut être aussi simple qu'une opération ou aussi complexe qu'un système entier. La deuxième définition est celle de Clemens Szyperski [14] qui a défini un composant logiciel comme suit :

"Un composant logiciel est une unité de composition avec des interfaces spécifiées contractuellement et des dépendances de contexte explicites uniquement. Un composant logiciel peut être déployé indépendamment et est sujet à composition par des tiers."

En gros, un composant est une "boîte noire" où les détails du code et de l'implémentation sont entièrement cachés et où les données ne sont accessibles que via des interfaces. Les composants adhèrent donc aux principes du génie logiciel d'encapsulation, d'abstraction et de modularité. Ce sont des entités découplées développées pour être réutilisées. Un composant comprend deux parties : un ensemble d'interfaces, une implémentation.

A- Interfaces. Une interface est le point de communication qui gère l'interaction d'un composant avec son environnement, c'est-à-dire les autres composants [14]. Les composants sont rendus abstraits grâce à leurs interfaces qui masquent les détails d'implémentation et facilitent la réutilisation. Les interfaces peuvent être fournies ou requises ; Figure 2.1 [15] :

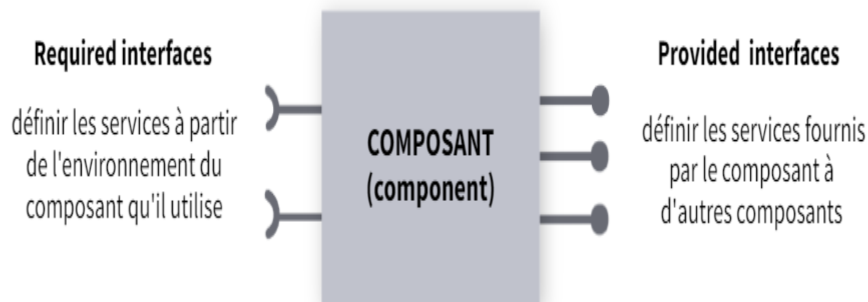


FIGURE 2.1 – Interfaces de composant.

- Une interface fournie expose l'ensemble des services fournis par le composant à d'autres composants. Le point de communication permet de recevoir des invocations de service d'autres composants.
- Une interface requise définit les services requis par le composant à partir d'autres composants. Le point de communication permet d'envoyer des demandes de service à d'autres composants.

B- Implémentation. Au contraire, une implémentation fait référence à la définition concrète et interne d'un composant (boîte blanche), c'est-à-dire son code source. Une fois le composant développé, sa mise en œuvre est masquée et découplée grâce aux interfaces. Les composants peuvent ensuite être (ré)utilisés pour construire des architectures sans aucune connaissance requise sur leur implémentation interne.

2.2.3.2 Connecteurs

Un autre aspect fondamental des systèmes logiciels est la gestion de l'interaction entre ses blocs de construction, c'est-à-dire les composants. Alors que les composants logiciels implémentent les fonctionnalités du système, les connecteurs logiciels lient les composants entre eux et agissent comme des médiateurs entre eux.
Ceci sépare le souci de calcul géré par les composants du souci d'interaction géré par les connecteurs, renforçant ainsi la réutilisation. L'interaction peut

même devenir plus difficile que la sélection des composants appropriés. C'est souvent le cas lorsque les systèmes sont construits à partir d'un grand nombre de composants complexes répartis sur plusieurs hôtes et mis à jour sur de longues périodes. Un connecteur logiciel peut fournir plusieurs services dans l'architecture.

2.2.4 Styles d'architecture logicielle

Les styles d'architecture logicielle représentent les modèles développés à une ascendance pour les systèmes afin d'apporter des solutions incontinentes aux problèmes qui surviennent dans le processus du cycle de vie du logiciel, comme la maintenance. Il détermine le vocabulaire des composants et des connecteurs qui peuvent être utilisés dans les instances de ce style, ainsi qu'un ensemble de contraintes sur la façon dont ils peuvent être combinés.
L'avantage des styles architecturaux est qu'ils offrent un niveau de représentation supérieur indépendant de la technologie. Il existe différents styles d'architecture logicielle et parmi les plus populaires figurent les architectures orientées objets (OOA), les architectures à base de composants (CBA) et les architectures orientées services (SOA).

2.2.4.1 Style d'architecture orientée objets

L'architecture orientée objet est un paradigme de conception basé sur la division des responsabilités d'une application ou d'un système en objets individuels réutilisables et auto suffisants, chacun contenant les données et le comportement correspondant à l'objet. Les objets sont discrets, indépendants et faiblement couplés ; ils communiquent via des interfaces, en appelant des méthodes ou en accédant à des propriétés dans d'autres objets, et en envoyant et en recevant des messages [16].

2.2.4.2 Style d'architecture basée sur les composants

La norme Service Component Architecture (SCA) a été publiée pour la première fois par la collaboration OSOA (Open Service Oriented Architecture). Il s'agit d'un modèle de composant hiérarchique qui est défini pour répondre à l'unification des architectures à base de composants et de services optimisés. Le but de ce modèle est de pouvoir réaliser la composition de services indépendamment des technologies utilisées pour la réalisation de ces services et indépendamment de la plate-forme implémentant la spécification SCA (Service Component Architecture).

Avantages de l'architecture basée sur les composants

- **Facilité de déploiement :** Un composant d'une ancienne version peut être remplacé par une nouvelle version sans impact sur les autres composants ou sur le système dans son ensemble.
- **Coût réduit :** L'utilisation de composants tiers permet de réduire les coûts de développement et de maintenance.
- **Facilité de développement :** les composants implémentent des interfaces pour fournir des fonctionnalités, cela permet un développement sans impact sur les autres parties du système.
- **Réutilisable :** L'utilisation de composants réutilisables réduit les coûts de développement et de maintenance des systèmes logiciels.

2.2.4.3 Style d'architecture orienté services

L'architecture orientée services (SOA) est un style architectural qui prend en charge l'orientation des services, il rend la fonctionnalité de l'application à fournir sous la forme d'un ensemble de composants appelés services qui peuvent être invoqués, publiés et découverts via un contrat.

2.2.5 Construction de l'architecture

La construction de l'architecture est l'activité consistant à documenter un ou plusieurs aspects de l'architecture d'un système à l'aide d'une notation particulière. Un modèle architectural est donc un artefact qui capture tout ou partie des décisions de conception comprises dans l'architecture logicielle. Les modèles d'architecture sont parfois appelés descriptions d'architecture. Une notation de modélisation architecturale est un langage ou un moyen utilisé pour modéliser des architectures logicielles.

2.2.6 Récupération de l'architecture logicielle

La restauration de l'architecture logicielle peut être divisée en deux phases :

1. identification et extraction des artefacts du code source, y compris les éléments architecturaux.

2. analyse des artefacts source extraits pour en déduire une vue de l'architecture implémentée.

Les artefacts source extraits forment un modèle source, qui comprend une collection d'éléments (par exemple, des fonctions, des fichiers, des variables, des objets, etc.), un ensemble de relations entre les éléments (par exemple, "la fonction appelle la fonction", l'objet A a une instance") et un ensemble d'attributs de ces éléments et relations, pour représenter le système.

2.2.7 Rétro-ingénierie

la maintenance est la modification d'un logiciel après la livraison, pour corriger les fautes, améliorer la performance ou les autres attributs, ou encore adapter le produit à un environnement modifié. La phase de maintenance est donc une étape du cycle de vie du logiciel comme on a déjà dit. Cette étape a pour objectif de maintenir le logiciel conforme aux besoins des utilisateurs. Mais Cette phase est la plus longue du cycle de vie puisque elle débute après le développement et ne finit que avec la fin du logiciel.
Pour répondre à ces problématiques, de nombreux domaines sont apparus. Le rétro-ingénierie est l'un des plusieurs de ces domaines.
Ce processus vise à identifier les structures du système et à fournir une représentation du logiciel à des niveaux d'abstraction plus élevés que celui de l'implémentation. Ces techniques sont passives et permettent uniquement une meilleure visualisation. Ce domaine inclut l'extraction d'architectures, la re-documentation et l'exploration du code.

2.3 Évolution de logiciel

À mesure que le logiciel vieillit, son architecture doit évoluer pour faire face à tous les changements qui surviennent au cours du cycle de vie du logiciel. L'évolution de l'architecture est considérée comme l'une des tâches les plus difficiles de l'ingénierie logicielle à base de composants. Pour mieux identifier les motivations et les enjeux de l'évolution de l'architecture, cette section donne un aperçu rapide de l'évolution logicielle en général et de l'évolution centrée sur l'architecture en particulier.

Depuis les premières années du génie logiciel, il y avait une prise de conscience que la partie la plus coûteuse et la plus difficile du développement logiciel est sa phase de maintenance. il a été prouvé que les coûts de maintenance représentent environ 60% des coûts globaux de production d'un logiciel. La maintenance est la dernière phase du cycle de vie d'un système logiciel qui vient après son déploiement pour corriger les bogues et apporter quelques ajustements. Cette vision classique a longtemps régi la pratique et est encore largement utilisée dans l'industrie aujourd'hui. Il est également devenu une partie de la norme de [3] (qui décrit un processus itératif de gestion et d'exécution des activités de maintenance logicielle) pour la maintenance logicielle qui définit la maintenance comme suit :

"La maintenance logicielle est la modification d'un produit logiciel après livraison pour corriger des défauts, améliorer les performances ou d'autres

attributs, ou pour adapter le produit à un environnement modifié."

Un intérêt particulier pour l'évolution du logiciel a été commencé par Manny Lehman lorsqu'il a énoncé les fameuses " Lois de l'évolution du logiciel " [17]. Lehman a défini l'évolution du logiciel comme suit :

"L'évolution logicielle est l'ensemble des activités de programmation destinées à générer une nouvelle version d'un logiciel à partir d'une version opérationnelle plus ancienne. Si ces activités peuvent être effectuées au moment de l'exécution sans qu'il soit nécessaire de recompiler ou de redémarrer le système, cela devient une évolution logicielle dynamique."

La particularité de la définition de Lehman est qu'elle traite de l'évolution des systèmes et pas seulement de l'évolution du code. Les recherches ultérieures sur l'évolution des logiciels ont eu recours à des logiciels évolutifs au niveau architectural plutôt qu'au code source.

2.4 Conclusion

Ce chapitre a introduit le contexte Architectures logicielles et Leur évolution et mis en évidence les enjeux des deux domaines.Les architectures logicielles peuvent être représentées dans différents styles tels que l'architecture orientée objet (OOA), l'architecture basée sur les composants (CBA), l'architecture orientée service (SOA). il a présenté la construction et la récupération de l'architecture logiciel et la rétro-ingénierie et terminé par les concepts d'évolution de logiciel.

Chapitre 3

Travaux connexes

3.1 Introduction

Nous présentons dans ce chapitre l'état de l'art des approches d'évolution dynamique des architectures logicielles. Ensuite, nous mentionnons les travaux existants d'architecture logicielle à évolution statique et dynamique. Dans ce chapitre, nous présentons l'état de l'art des approches de récupération d'architecture et d'évolution d'architecture logicielle dynamique, car nous nous intéressons à l'évolution logicielle dynamique pour les systèmes basés sur des modules dans ce travail. Par conséquent, nous avons rassemblé quelques-uns des travaux qui se concentrent sur la récupération d'architecture à partir d'applications orientées objet pour les présenter. Ensuite, nous parlons des travaux basés sur le style de l'architecture récupérée.

3.2 Récupération de l'architecture basée sur les composants

De nombreux travaux ont été proposés pour récupérer des architectures orientées services à partir d'applications orientées objets pour aider à la migration vers SOA (Service-Oriented Architecture). Nous avons pris : [4], [5], [6] et[7]] comme exemples de ces travaux :

Alae-Eddine El Hamdouni et al., [4] ont proposé une approche de récupération d'architecture qui vise à extraire l'architecture à base de composants d'un système orienté objet, par un processus d'exploration semi-automatique afin d'identifier les composants architecturaux par l'analyse de concept relationnel (RCA). L'approche RCA se présente comme une méthode

complémentaire pour lever certaines limites de l'implémentation existante de "ROMANTIC" basée sur un algorithme de recuit simulé. Les composants architecturaux de cette approche sont identifiés à partir de concepts dérivés en exploitant toutes les relations de dépendance existantes entre les classes du système orienté objet.

Hong Mei et Jian Lü [5] ont présenté une approche pour récupérer l'architecture logicielle à partir de systèmes à base de composants au moment de l'exécution et modifier les systèmes d'exécution en manipulant l'architecture logicielle récupérée dans laquelle peut décrire avec précision et en profondeur les états et comportements réels du système d'exécution. Afin de maintenir à tout moment la mise à jour de l'architecture logicielle récupérée et de modifier le système d'exécution en manipulant son architecture logicielle récupérée, les éléments de l'architecture logicielle récupérée sont implémentés sous la forme d'un ensemble de méta-objets créés au moment de l'exécution.

Thibaud Lutellier et al., [6] ont travaillé pour améliorer les études précédentes sur la récupération d'architectures logicielles à partir des implémentations logicielles en étudiant l'impact de l'exploitation des dépendances de symboles précis sur la précision des techniques de récupération d'architecture. En outre, ils ont évalué d'autres facteurs des dépendances d'entrée tels que le niveau de granularité ou précision et la construction de graphes de liaisons dynamiques, dans lesquels ils ont évalué neuf techniques de récupération d'architecture. Les résultats suggèrent que : i) l'utilisation de dépendances de symboles précises a une influence majeure sur la qualité de la récupération. ii) des techniques de récupération plus précises sont nécessaires. En outre, ils ont développé une nouvelle technique basée sur des sous-modules pour récupérer des versions préliminaires d'architectures de vérité terrain.

Kerdoudi Mohamed Lamine, et al., [7] ont travaillé sur systèmes logiciels à base de composants/services, ils ont proposé un méta-modèle pour les architectures de gammes de produits logiciels basés sur les composants/services, ils ont ajouté la conception d'un adaptateur d'un processus générique SPL-RE, pour la construction de modèles d'architecture (modèles SAPL) en analysant les variantes de produits ,et une implémentation de cet adaptateur spécifique aux applications basées sur OSGi, et une expérimentation de ce processus de récupération sur un ensemble de versions d'Eclipse. L'expérimentation qu'ils ont menée a permis d'évaluer l'efficacité du processus à identifier les bonnes caractéristiques, par rapport à celles identifiées/construites par des experts. De plus, il a permis de mesurer la précision des architectures de produits dérivés du SAPL récupéré.

3.3 Récupération de l'architecture orientée services

De nombreux travaux ont été proposés aussi pour récupérer des architectures orientées services à partir d'applications orientées objets pour aider à la migration vers SOA (Service-Oriented Architecture). Nous avons pris : [18] et [19]] comme des examples de ces traveaux :

Zhang et al., [18] ont proposé un processus d'identification de service basé sur l'architecture. Cela commence par la récupération des composants et des connecteurs à l'aide de techniques de rétro-ingénierie et d'analyses statiques et dynamiques en fonction des fonctionnalités du système hérité (legacy). Différentes métriques sont utilisées pour identifier les éléments architecturaux tels que la cohésion, le couplage, la fiabilité et les mesures de maintenance. Les informations architecturales récupérées sont représentées à l'aide d'un langage de description d'architecture. L'identification du service est réalisée en deux étapes. La première étape est l'analyse du domaine. Le modèle de domaine est utilisé pour identifier les services logiques qui représentent les fonctionnalités commerciales qui doivent être fournies en tant que services.

Marvin Greiger et al., [19] a proposé une approche semi-automatique pour identifier les services composites. L'approche suppose l'existence d'une conception de service initiale qui est construite en cartographiant chaque module à un service métier et à un composite dédié. La conception initiale du service est améliorée progressivement grâce à un regroupement "*clustering*" hiérarchique et de partitionnement. La première étape consiste à appliquer le regroupement hiérarchique sur les services métier. Les groupes "*clusters*" résultants regroupent les services liés en termes de processus métier. Le *clustering* hiérarchique est effectué de manière itérative, à chaque itération, de nouveaux groupes sont calculés pour une couche hiérarchique et des services composites sont introduits pour implémenter ces clusters.
Dans la deuxième étape, les clones de logiciels sont supprimés à l'aide d'une technique de partitionnement pour partitionner les fonctionnalités de plusieurs services en parties individuelles, et les parties communes sont déplacées dans un seul service. Le processus décrit a été appliqué manuellement à une application pour la reconstruction d'architecture et la migration d'un système hérité d'entreprise. Les auteurs travaillent sur un outil qui implémente le processus proposé pour réduire les erreurs manuelles et augmenter l'automatisation.

3.4 Évolution statique de l'architecture logicielle

Dans cette section, nous présentons des travaux d'évolution statique d'architecture logicielle existants dans la littérature tels que [20]et [21]] :

David Garlan and Bradley Schmerl [20] ont développé un outil de planification et d'analyse de l'évolution de l'architecture qui permet aux architectes de planifier les modifications évolutives d'un système logiciel d'un point de vue architectural. Les architectes peuvent définir les modifications à apporter à chaque étape de l'évolution et peuvent explorer plusieurs de ces voies d'évolution. L'outil fournit un cadre de plug-in permettant des analyses afin qu'un architecte puisse comparer et échanger plusieurs chemins d'évolution possibles. Ces analyses peuvent être adaptées à des domaines d'évolution particuliers et à des environnements d'affaires particuliers intéressant l'architecte.

Jeffrey Barnes et al.,[21] ont décrit une approche de planification et de raisonnement sur l'évolution de l'architecture. L'approche vise à fournir aux architectes les moyens de modéliser des voies d'évolution prospectives et à soutenir l'analyse pour sélectionner parmi ces voies candidates. Ils ont caractérisé les modèles récurrents comme un ensemble de chemins liés, que nous appelons styles d'évolution. De tels styles peuvent être formellement caractérisés, permettant un support par des outils. Ils ont évalué leur approche pour la modélisation de l'évolution de l'architecture logicielle par deux méthodes très différentes. Dans la première méthode, ils ont évalué la complexité de calcul des contraintes de chemin d'évolution du vérification de modèle. Cette étude théorique a montré que l'approche de vérification de la validité du chemin est réalisable sur le plan informatique.

3.5 Évolution dynamique de l'architecture logicielle

Dans cette section, nous présentons des travaux d'évolution dynamique d'architecture logicielle existants dans la littérature tels que [22], [5] et [23]] :

Jennifer Pérez et al.,[22] ont présenté une solution au problème d'évolution des architectures logicielles fournies par "PRISMA". "PRISMA" est

une approche de modélisation d'architecture qui intègre les avantages du développement logiciel basé sur les composants et du développement logiciel orienté aspect, il se présente comme un cadre pour faire évoluer les architectures orientées aspect et basées sur les composants par une évolution axée sur les exigences. L'évolution s'appuie sur un méta-niveau et les propriétés réflexives de "PRISMA" qui ont été implémentées en middleware (logiciel intermédiaire). Il est également démontré comment les services d'évolution du méta-niveau "PRISMA" permettent l'évolution à l'exécution des architectures logicielles à l'aide d'un cas d'étude industriel, le robot "TeachMover".

Hong Mei and Jian Lü [5] ont présenté une approche de récupération de l'architecture logicielle à partir de systèmes à base de composants au moment de l'exécution et de modification des systèmes d'exécution via la manipulation de l'architecture logicielle récupérée dans laquelle peut décrire avec précision et en profondeur les états et comportements réels du système d'exécution.

Adel Hassan et al., [23] ont proposé un style d'évolution dynamique pour l'évolution dynamique de l'architecture logicielle, et de fournir un style suffisamment riche pour modéliser les changements dynamiques dans l'architecture logicielle d'un système temps réel et pour pouvoir représenter les manières potentielles d'effectuer ces changements. Pour cela, ils ont intégré les concepts de comportement des changements dynamiques dans le Méta-Évolution Style (MES) afin qu'ils puissent avoir une bonne compréhension des problèmes d'évolution dynamique (arrêt sûr des artefacts en cours d'exécution, transfert d'état, gestion des changements et planification de l'évolution dynamique) et des contraintes. , condition préalable au développement d'un environnement de modélisation prenant en charge les styles d'évolution dynamique.

3.6 Conclusion

Dans ce chapitre, nous avons présenté l'état de l'art des approches d'évolution dynamique des architectures logicielles. Nous avons d'abord présenté quelques travaux de récupération d'architecture logicielle. Puis nous avons présenté des travaux d'architecture logicielle à évolution statique et dynamique. Nous nous concentrons sur l'évolution dynamique des logiciels pour les systèmes à base des modules Java 9.

Chapitre 4

Reconstruction des architectures logicielles à partir des systèmes basés sur les modules

4.1 Introduction

Dans ce chapitre, nous présentons notre approche proposée pour l'évolution basée sur l'architecture logicielle des systèmes établi sur les modules Java . Nous commençons par une présentation de l'approche proposée, puis des détails sur chaque étape. Nous présentons également dans ce chapitre notre proposition de méta-modèle des modules java avec une explication de ses éléments.

4.2 Processus générale

Nous introduisons d'abord une vue d'ensemble de l'approche proposée pour l'évolution basée sur l'architecture logicielle pour les systèmes basés sur des modules, puis nous présentons notre proposition de méta-modèle .

4.2.1 Présentation de l'approche proposée

Le processus global de notre approche de l'évolution basée sur l'architecture logicielle pour les systèmes basés sur les modules est présenté dans La Figure 4.1

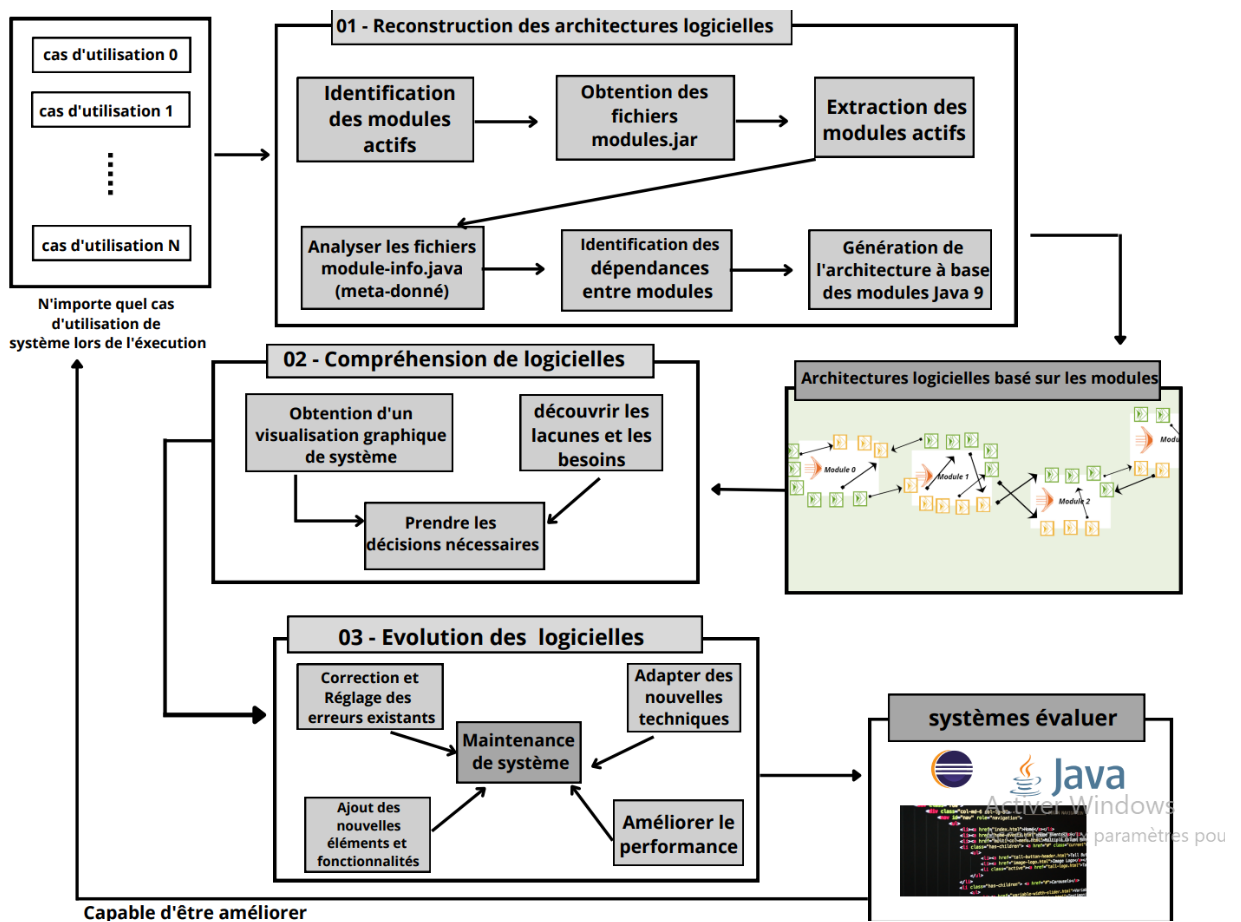


FIGURE 4.1 – Approche proposée.

Notre processus composé de trois grande phases : i) La reconstruction de l'architecture logicielle, ii) La compréhension de système logiciel obtenue et finalement iii) L'évolution de logiciel grâce à la bonne représentation correcte de le système.

1- **Reconstruction de l'architecture logicielle** cette phase passe à plusieurs étapes. Commençant par identifier les modules actives lors de l'exécution de système, et après avec l'obtention de tout les fichiers modules.jar , et extraire les modules actifs, ensuite on va analyser le fichier module-info.java de chaque actif module, qui nous permet de identifier toute les dépendances entre les modules, ce qui nous permet de faire la création de l'architecture logiciel et de passer à l'étape suivant.

2- Compréhension de système logiciel la deuxième phase consiste et visé à Obtenir la visualisation graphique de système pour comprendre son fonctionnement et découvrir les lacunes et tous les besoins et aussi les éléments insuffisants ou endommagés, qui nous permet de prendre des bonnes décisions pour modifier et changer n'importe quelle partie de l'ensemble du système.

3- Évolution de logiciel la dernière phase vise à obtenir la version évoluée du système donné. Après l'avoir clarté de l'architecture de système, n'importe quel développeur peut donc faire la Correction et le réglage des erreurs existants, adapter des nouvelles techniques, ajout ou supprimer ou modifier des éléments et les services et aussi les fonctionnalités de système, et améliorer le performance et alors et atteindre la maintenance requise que le client souhaitait.

4.2.2 Méta-modèle pour les modules Java 9

La Méta-modélisation c'est la représentation d'un domaine pour faciliter la création et la manipulation des modèles.

- Modèle.
- C'est une instance d'un méta-modèle.
- Une syntaxe abstraite de l'entité modélisée (le système).
- Il se conforme à son méta-modèle.
- On passe d'un modèle à un autre par transformation.
- Peut être exprimé avec différents niveau d'abstraction / raffinement.

- Méta-modèle.
- C'est un modèle.
- C'est la définition des concepts et des relations des instances qui lui sont conformes.
- Il ressemble à la définition d'une grammaire.

La Figure 4.2 présenté notre méta-modèle des modules. C'est une description de notre modèle, il montre tous les éléments architecturaux du système logiciel dynamique basé sur des modules. Notre méta-modèle basé sur un méta-modèle OSGi existant dans la littérature qui est présenté dans [7].
Dans notre méta-modèle, nous avons ajouté tous les éléments associés à un modèle java construit avec les modules tels que la classe ModuleElement elle-même avec ses attributs, ainsi que RequireElement et ExportElement.

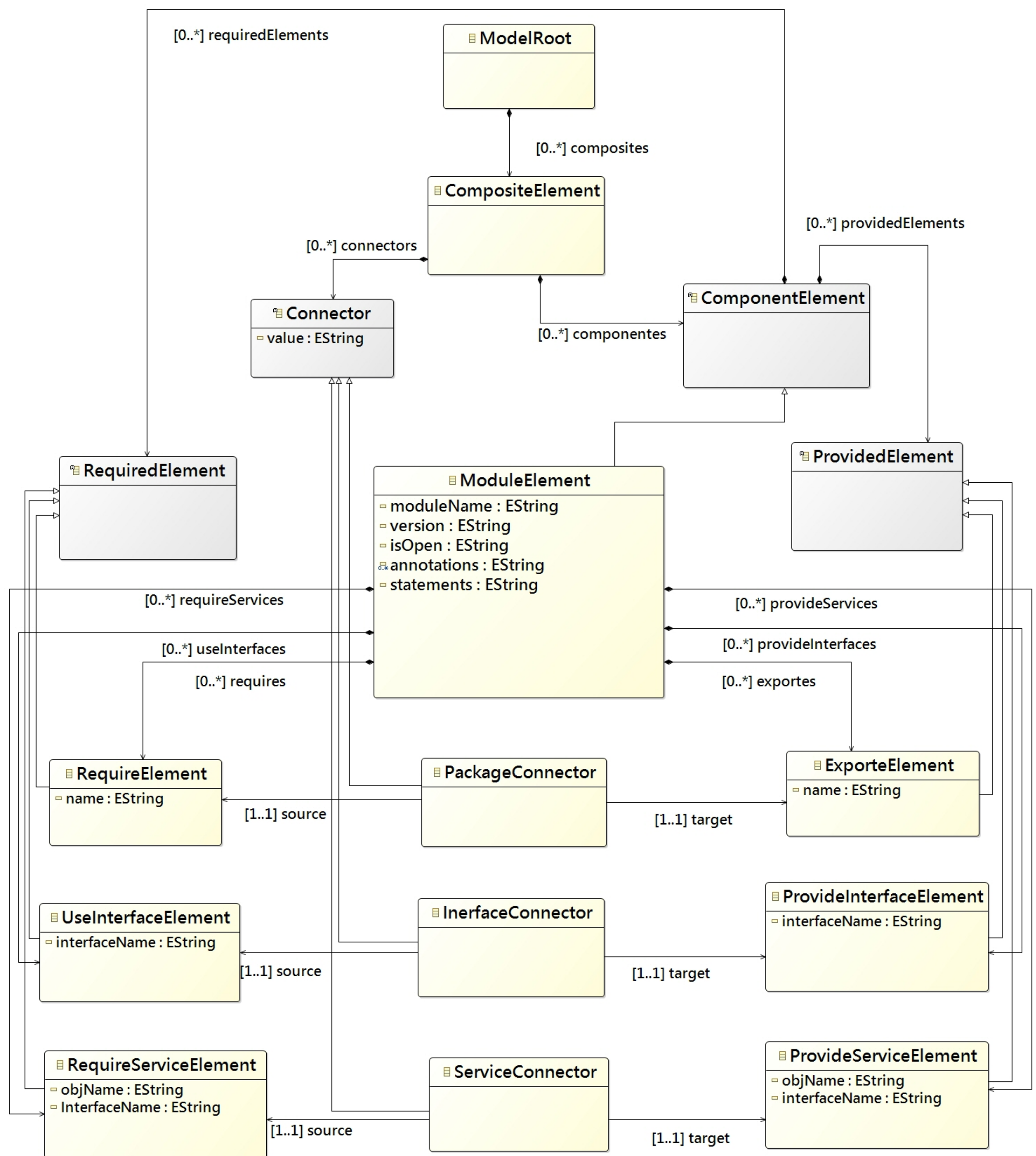


FIGURE 4.2 – Méta-modèle

- Notre méta-modèle pour les modules Java 9 est composé d'un ensemble de méta-classes, et nous allons toutes les définir ensuite :

- **ModelRoot :** Élément racine de notre méta-modèle, il représente le conteneur de tous les composants de l'architecture, il est composé de nombreux composants et connecteurs.

- **CompositeElement :** le conteneur qui contient tous les composants de l'architecture. Il est composé de plusieurs composants, connecteurs et de ses propres interfaces fournies et requises. L'attribut *CompositeName* du composite est le nom de l'application.

- **ComponentElement :** Une méta-classe, qui représente l'entité composant. Un composant est composé d'un ensemble de classes et de ses interfaces fournies/requises.

- **Connector :** Une méta-classe, elle représente une connexion entre le *ProvidedElement* et le *RequiredElement*. Il s'agit également de la classe de base de *PackageConnector*, *InterfaceConnector* et *ServiceConnector*.

- **RequiredElement :** Méta-classe, elle représente les éléments requis du composant dans l'architecture logicielle. Il est géré par *ComponentElement* et constitue la classe de base de *RequireElement*,*UseInterfaceElement* et *RequireServiceElement*.

- **ProvidedElement :** Méta-classe qui représente les éléments fournis du composant dans l'architecture logicielle. Il est géré par *ComponentElement* et constitue la classe de base de *ExporteElement*, *ProvidedInterfaceElement* et *ProvideServiceElement*.

- **ModuleElement :** Une méta-classe, elle représente le composant Module (un package pour les plug-ins), elle étend de ComponentElement. Il est composé d'un certain nombre de packages (*ExporteElement* et *ProvideElement*), d'interfaces (*ProvideInterfaceElement* et *UseInterfaceElement*) et de services (*ProvideServiceElement* et *RequireServiceElement*). En outre, il possède un ensemble d'attributs qui sont :

  - *moduleName :* le nom de l'interface fournie.
  - *version :* le nom de l'interface fournie.
  - *isOpen :* les package privés sont accessibles ou non
  - *annotations :* Les modules peuvent être annotés. Plusieurs annotations par défaut faisant partie de la plate-forme Java peuvent être appliquées aux modules.

- *statements :* une liste qui contient (Module Exports Statement, Module Opens Statement, Module Provides Statement, Module Requires Statement, Module Uses Statement) : qui sont des déclarations et des directives pour chaque type de packages et de services et d'interfaces dans module-info.java.

- **RequireElement :** Méta-classe, elle représente l'élément de package importé, les packages dont dépend ce module sans identifier explicitement leur module d'origine. L'attribut "*name*" est le nom du package importé.

- **ExporteElement :** Une méta-classe, elle représente les éléments de packages exportés, les packages que le module expose aux clients. L'attribut "*name*" est le nom du package exporté.

- **ProvideInterfaceElement :** Une méta-classe, elle représente les éléments d'interface fournis, les interfaces que ce module a fournies à d'autres modules. Il a un attribut :
  - *interfaceName :* le nom de l'interface fournie.

- **UseInterfaceElement :** Une méta-classe, elle représente les interfaces requise éléments, les interfaces que ce module a utilisées à partir d'autres modules. Il a un attributs :
  - *interfaceName :* le nom de l'interface utilisée.

- **RequireServiceElement :** Une méta-classe, elle représente les éléments de services enregistrés, les services enregistrés par ce module. Il a deux attributs :
  - *objName :* est le nom de l'objet de service enregistré par ce module.
  - *interfaceName :* le nom de l'interface requis.

- **ProvideServiceElement :** Une méta-classe, elle représente les éléments de services consommés, les services que ce module utilise. Il a deux attributs :
  - *objName :* est le nom de l'objet de service enregistré par ce module.
  - *interfaceName :* le nom de l'interface fournis.

- **PackageConnector :** Une méta-classe, elle représente une connexion entre *RequireElement* (source) et *ProvideElement* (cible).

- **InterfaceConnector :** Une méta-classe, elle représente une connexion entre *UseInterfaceElement* (source) et *ProvideInterfaceElement* (cible).

- **ServiceConnector :** Une méta-classe, elle représente une connexion entre *RequireServiceElement* (source) et *ProvideServiceElement*(cible).

4.3 Reconstruction de l'architecture logicielle

La première étape de notre approche est considéré comme la plus importante. Elle permet de récupérer l'architecture logicielle. On doit d'abord choisir n'importe quel cas d'utilisation lors de l'exécution de l'application (en gardant à l'esprit que le "Use Case 0" est le lancement de l'application).

- **Identification des modules actifs :**
Cela parmi sélectionner juste les modules actifs lors de l'exécution dans le cas d'utilisation choisie.

- **Obtention les fichiers modules.jar :**
Ici on va obtenir tout les fichiers modules.jar qui contiennent tout les informations sur les modules.

- **Extraction des modules actifs :**
Après avoir les fichiers.jar on va extraire les modules actifs qui vont construire l'architecture.

- **Analyser module-info.java :**
L'analyse de le fichier module-info.java de chaque module est important pour obtenir les packages et les services et les interfaces fournis et requis et les relations entre eux, qui vont être des composants de notre architecture de logiciel.

- **Extraction des dépendances entre les modules :**
Cette étape permet d'identifier les éléments architecturaux et les liens entre eux, à partir des modules applicatifs,qui vont nous aider à récupérer les éléments associés avec eux.
  - Récupération les packages exportés et les packages fournis.
  - Récupération des services requis et des services fournis.
  - Récupération les interfaces utilisées et les interfaces fournies.

- **Génération de l'architecture à base des module Java 9 :**
Ici on génère l'architecture basée modules à partir tous ce que on a récupérer et on fait la reconstruction.

4.4 Architecture logicielle basée sur les modules

Ce partie est considéré comme un résultat de la phase précédente.
Après la génération de notre architecture, on est besoin de réaliser une visualisation graphique pour mettre l'image de l'architecture claire et facile à comprendre.

4.5 Compréhension de système logiciel

Dans cette phase, nous pouvons dire que nous arrivons au but de notre travail, qui est la compréhension de la fonctionnalité d'un système sans même savoir comment il a été construit depuis le début, ni connaître tous ses composants.

- **Obtention d'un visualisation graphique :**
cette phase prend le résultat de la dernière étape en entrée, et donne la visualisation graphique de l'architecture logicielle en sortie.
Cela nous aide donc à visualiser graphiquement l'architecture et à en faciliter la compréhension grâce à la présentation claire et bien organisée que l'éditeur va montrera.

- **Découvrir les besoins :**
Après la visualisation de notre système basé sur des modules, nous pouvons comprendre son architecture et cela pourrait aider tout développeur à détecter facilement tous les besoins du logiciel, ce qui nous amène à la phase suivante qui contient tous les cas pour évaluer le système.

4.6 Évolution de système logiciel

Cette phase comporte de nombreuses parties qui visent à obtenir un système évolué.

- **Correction des erreurs :**
On parle ici de trouver les erreurs qui corrompent le bon fonctionnement du système, qui n'ont pas pu être trouvées avant la visualisation graphique de l'architecture du système, puis de les corriger.

- **Adapter des nouvelles techniques :**
La visualisation graphique permet aux développeurs d'éditer facilement l'architecture logicielle récupérée, donc il est possible d'avoir une interaction visuelle entre les développeurs et l'architecture logicielle facilement et de leur donner différentes options pour mettre à jour cette architecture.

- **Ajout des nouvelles fonctionnalités :**
Dans cette étape, il faut être prudent car l'ajout de nouvelles fonctionnalités dépend de beaucoup de règles de modification, car tout ajout ou changement peut :

  - Ruiner et endommager les fonctionnalités existantes.
  - Réduire les performances du logiciel.
  - Ajoutant de nouvelles erreurs au système qui causent beaucoup de nouveaux problèmes.

  L'ajout de nouvelles fonctionnalités consiste donc à suivre le style d'architecture qui fait que cette partie de l'évolution fonctionne bien.

- **Améliorer le performance :**
Cela par :
  - Ajouter de nouveaux composants : tels que l'ajout de nouveaux modules, de nouveaux services, interfaces, packages et relations entre eux.
  - Mettre à jour les composants : la possibilité de mettre à jour différents composants, comme la possibilité d'arrêter ou de démarrer des modules ou d'activer les services et les interfaces nécessaires et de modifier les noms, les états et les versions des composants.
  - Supprimer les composants : supprimer tout composant inutilisé ou endommagé de l'architecture logicielle (modules, services, packages ou interfaces).

Après la mise à jour et la correction de l'architecture logicielle récupérée, nous sommes maintenant dans la dernière étape du processus d'évolution de l'architecture logicielle qui génère automatiquement la nouvelle version mise à jour de l'architecture logicielle, avec notre système évolué.

4.7 Conclusion

Dans ce chapitre, nous avons présenté le processus général de notre approche proposée pour l'évolution de l'architecture logicielle pour les systèmes basés sur des modules, nous avons également présenté notre méta-modèle et expliqué tous ses éléments.

Chapitre 5

Outil ArchBaseDevModules et Editeur pour les applications basées module Java 9

5.1 Introduction

Dans ce chapitre,nous présentons la mise en œuvre de notre processus d'approche proposé, les cadres et les outils que nous utilisons, et les étapes que nous prenons pour créer notre application.Nous présentons également l'architecture globale et détaillée de notre outil.

5.2 Environnement et outils de développement

Dans cette section, nous présentons la configuration du système, les outils, le langage et les environnements que nous avons utilisés pour implémenter notre outil.

1. **Configuration du système :** Les expériences sont réalisées sur CPU 1,80 GHz Intel Core i3, avec 4 Go de mémoire.
2. **Langage de programmation :** Nous avons utilisé Java pour implémenter les algorithmes proposés dans notre travail car il fournit les API nécessaires pour analyser le code source des applications.
3. **Plate-forme Eclipse :** La plate-forme Eclipse est composée d'un ensemble de plug-ins et est conçue pour être extensible à l'aide de plug-ins supplémentaires. La plate-forme Eclipse peut être utilisée pour

développer des applications clientes riches, des environnements de développement intégrés et d'autres outils. Dans notre travail, nous utilisons "Eclipse Modeling Tools" (version Neon 3) comme plate-forme pour développer notre outil et notre application RCP, en plus nous avons utilisé :

- Outil de développement Java (JDT) pour développer l'analyseur de composants .
- Eclipse Modeling Framework (EMF) pour créer notre méta-modèle de composants.
- Graphical Modeling Framework (GMF) pour créer une application RCP pour y intégrer notre outil.

5.3 Notre outil *ArchBaseDevModules*

Dans cette section, nous présentons l'architecture globale et détaillée de l'application RCP qui nous avons créé, avec toutes les étapes que nous suivons pour l'y développer et implémenter sur Eclipse Neon. Nous avons utilisé EMF pour définir notre méta-modèle et GMF pour construire notre outil. Nous avons également ajouté une action qui invoque et lance *ArchBaseDevModules*.

5.3.1 Architecture de l'outil *ArchBaseDevModules*

Nous avons réalisé la figure pour représenter l'architecture globale de *ArchBaseDevModules*.

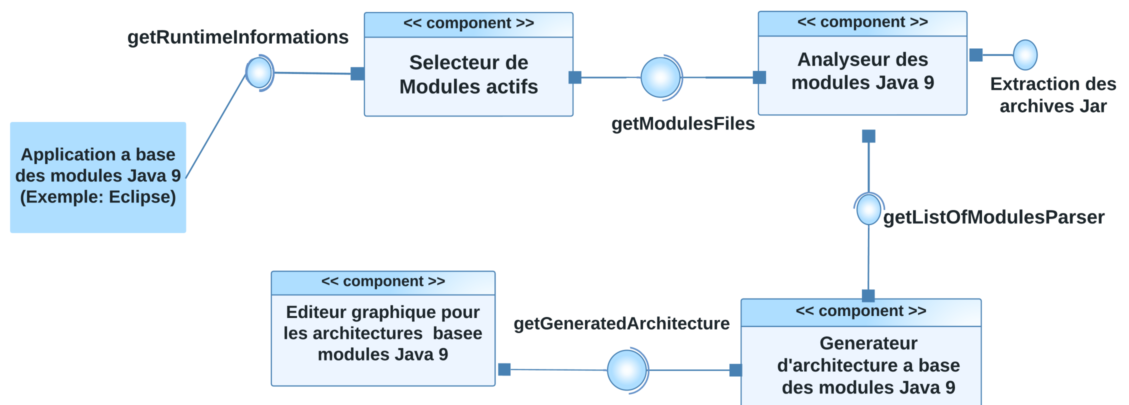


FIGURE 5.1 – Architecture de l'outil *ArchBaseDevModules*

1- Sélecteur de modules actifs composant : Ce composant permet d'identifier les modules actifs au moment de l'exécution d'après les informations récupérées de l'application (Eclipse).
2- Analyseur des modules Java 9 : Le composant analyseur fait l'analyse des fichiers modules.jar et les fichiers module-info.java et nous donne une liste des modules actifs analyser.
3- Générateur d'architecture a base des modules Java9 : Ce composant nous génère l'architecture basée sur les modules Java 9 sous un fichier XMI à partir les donnés récupérées par l'analyseur.
4- Éditeur graphique pour les architectures basée modules Java 9 : Le dernier composant de notre architecture globale fait un lecture de fichier XMI et nous offre la visualisation graphique de l'architecture logiciel à base des modules de l'application donnée .

5.3.2 Framework de modélisation de Eclipse (EMF)

Le projet EMF est un cadre de modélisation et une installation de génération de code pour la construction d'outils et d'autres applications basées sur un modèle de données structuré. À partir d'une spécification de modèle décrite dans XMI, EMF fournit des outils et un support d'exécution pour produire un ensemble de classes Java pour le modèle, ainsi qu'un ensemble de classes d'adaptateur qui permettent l'affichage et l'édition basée sur des commandes du modèle, et un éditeur de base.

Le méta-modèle EMF se compose de deux parties : les fichiers de description Ecore et genmodel.

- Le fichier Ecore contient les informations sur les classes définies.
- Le fichier genmodel contient des informations supplémentaires pour la génération de code.
- Le fichier genmodel contient également le paramètre de contrôle de la manière dont le code doit être généré.

5.3.3 Éditeur graphique pour l'architecture logicielle

Nous vous présentons ici l'éditeur graphique d'architecture logicielle avec toutes les étapes que nous suivons pour le développer.

5.3.3.1 Création de méta-modèle Ecore

1- Créer un nouveau projet EMF vide : "File" → "New" → "Other" → "Eclipse Modeling Framework" → "Empty EMF Project".

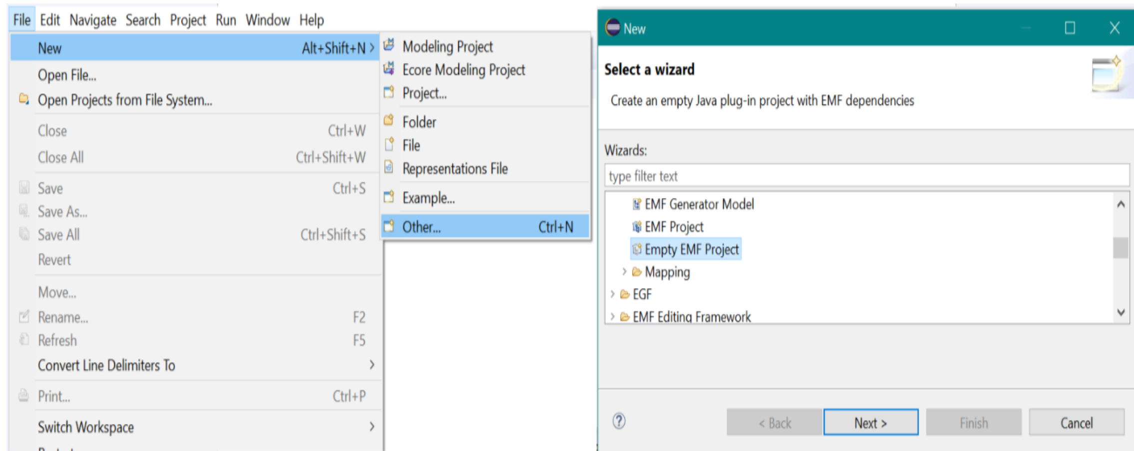


FIGURE 5.2 – Création un nouveau projet EMF vide.

- Et donnez-lui un nom, et clique sur "Finish".

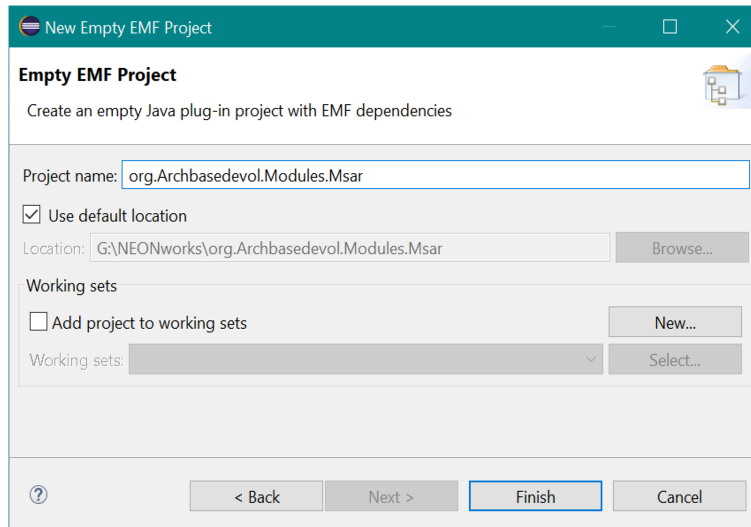


FIGURE 5.3 – Création un nouveau projet EMF vide

2- Ensuite, nous allons créer le fichier Ecore. Tout d'abord, dans la hiérarchie du projet nous faisons un clic droit sur le dossier "model".
Puis nous choisissons "New" → "Other", et en choisissant parmi Eclipse Modeling Framework "Ecore Model", on clique ensuite sur "Next".

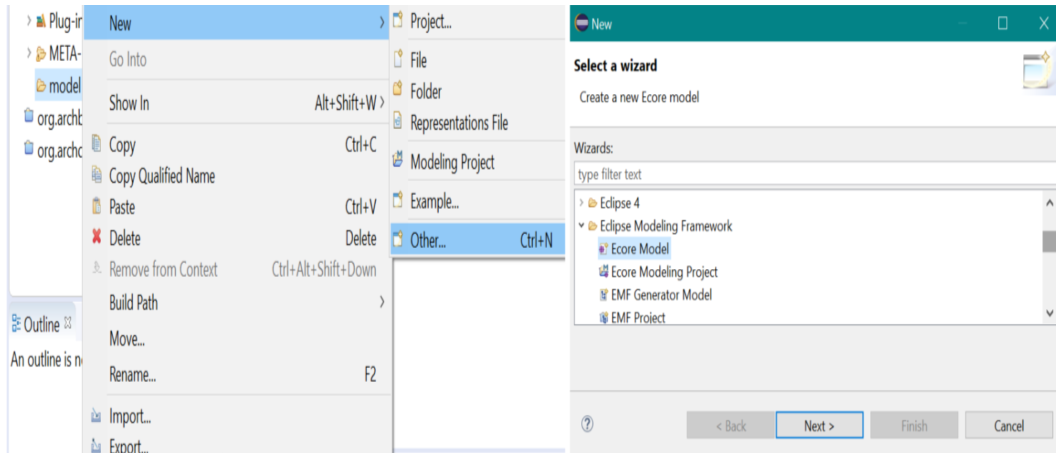


FIGURE 5.4 – Étapes de création du méta-modèle Ecore.

- Puis on donne un nom et on clique "Finish".

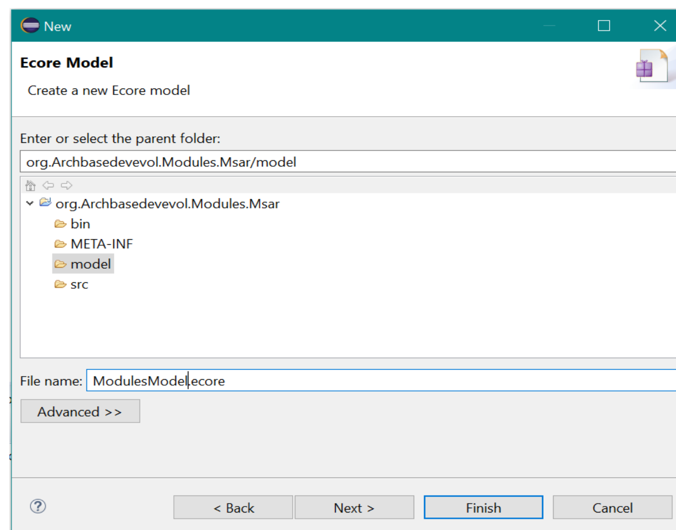
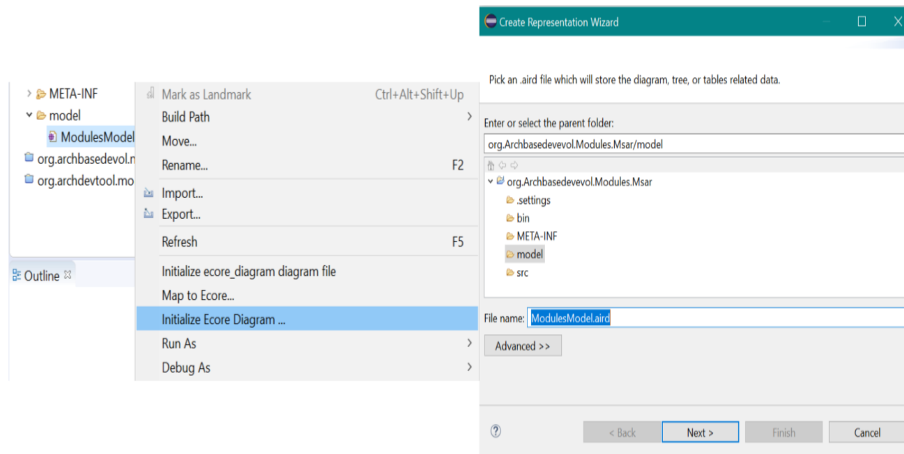


FIGURE 5.5 – Étapes de création du méta-modèle Ecore.

3- Pour construite notre méta-modèle graphiquement, on clique sur le fichier Ecore et nous choisissons "Initialize Ecore Diagram ..." → "Next" →

"Next" → "Finish".



- On peut donc ajouter les classes et les relations à partir de "Palette" pour construite notre méta-modèle correctement, et modifier les propriétés de n'importe quelle élément et nous changeons quoi que ce soit là-bas.

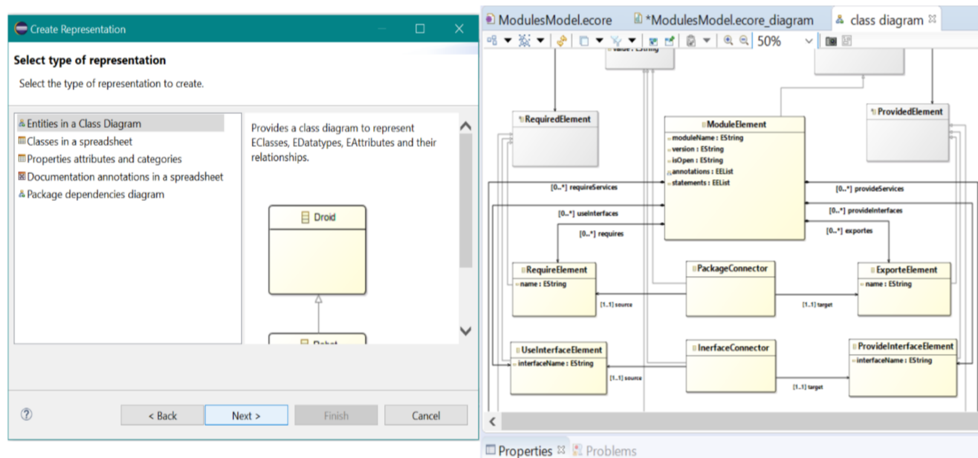


FIGURE 5.6 – Visualisation graphique du méta-modèle Ecore diagramme.

4- Dans la fenêtre centrale, ouvrir l'onglet du méta-modèle Ecore créé et remplir les champs du package créé par défaut et sans nom :

- Name : le nom du package.
- NS Prefix : mettre la même valeur que le nom du package.

- NS URI : mettre un URI de la forme suivante, par exemple : "platform :/resource/Archbasedevevol.Modules.Msar/model/ModulesModel.ecore". Cet URI est l'adressage absolu de votre méta-modèle au sein de votre workspace Eclipse.

Properties ☒ Problems

Property	Value
Name	ModulesModel
Ns Prefix	ModulesModel
Ns URI	platform:/resource/org.Archbasedevevol.Modules.Msar/model/ModulesModel.ecore

FIGURE 5.7 – les champs du package.

5- Maintenant, après avoir terminé l'organisation de nos méta-classes de modèle Ecore (*CompositeElement*, *ModuleElement*, *Connector*,...). Le modèle Ecore final "ModulesModel.ecore" est illustré à La Figure 5.8.

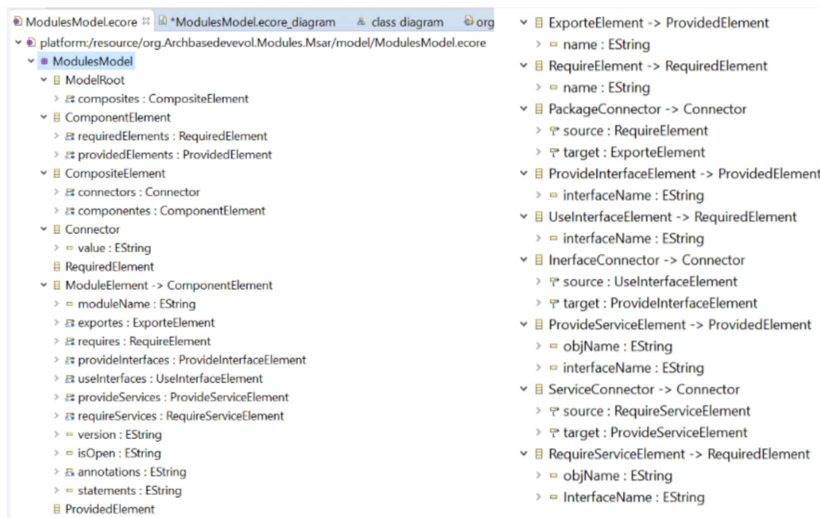


FIGURE 5.8 – Modèle Ecore.

5.3.3.2 Création de GenModel

Pour générer des entités et de code à partir du fichier Ecore que nous avons créé, nous créons d'abord un modèle de générateur,nous allons travailler avec "GMF Dashboard" . Depuis le GMF Dashboard, on clique sur "Select" de

"Domain Model" puis on choisit le modèle Ecore créé → "Drive". Nous donnons un nom à ce genmodel "ModulesModel.genmodel", en clinquant sur "Next" → "Ecore model" → "Next" → "Load" → "Finish".

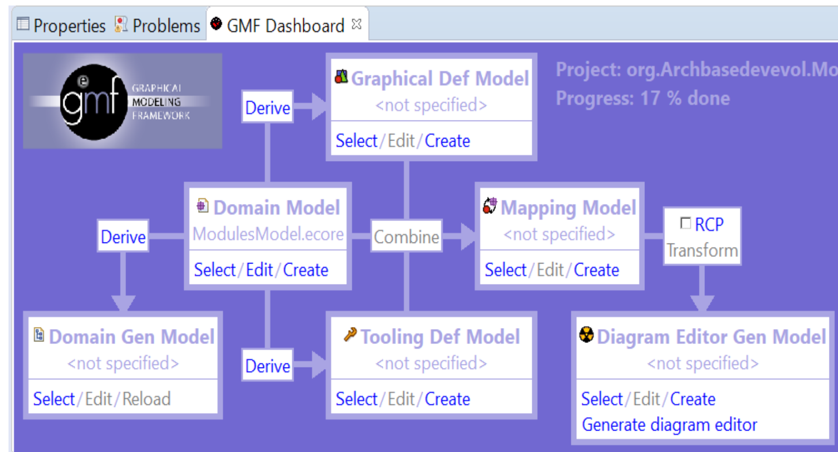


FIGURE 5.9 – GMF Dashboard.

- Dans le nœud racine du modèle générateur, nous allons modifier les paramètres par défaut afin de pouvoir générer une application RCP à partir de notre méta-modèle EMF :

  - Définissez la propriété "Rich Client Platform" sur "true".
  - Définissez le niveau de conformité sur "6.0".

- A partir du générateur de modèle , nous pouvons maintenant générer le code source. EMF nous permet de générer quatre différents plug-ins pour un modèle défini : (Modèle, Édition, Éditeur, Test).

- Pour générer ces plug-ins, nous faisons un clic droit sur le nœud racine du modèle de générateur et sélectionnons le plug-in. Pour notre projet, nous avons besoin de tous, nous sélectionnons donc "générer tout".

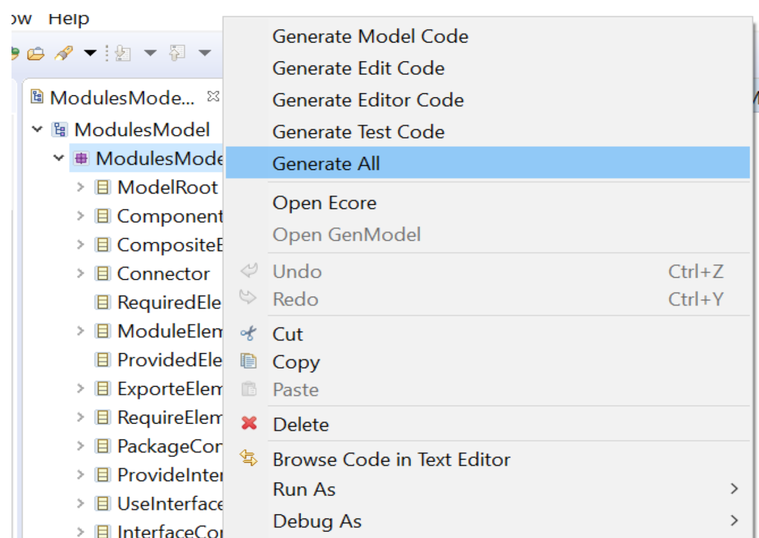


FIGURE 5.10 – Génération des plug-ins de "ModulesModel.genmodel".

5.3.3.3 Création de l'éditeur graphique

- Le Cadre de Modélisation Graphique (GMF) est un framework au sein de la plate-forme eclipse. Il fournit un composant génératif et une infrastructure d'exécution pour le développement d'éditeurs graphiques basés sur Eclipse Modeling Framework (EMF) and Graphical Editing Framework (GEF).

- Nous utilisons GMF pour créer notre éditeur graphique afin de visualiser l'architecture récupérée. suivant ces étapes :

- **Générer "graphical definition model" :** Fichier avec l'extension ".gmfgraph", contient les éléments de surface de l'éditeur graphique.
- **Générer "tooling definition model" :** Fichier avec l'extension ".gmftool", les éléments de menu et la palette de l'éditeur.
- **Générer "mapping model" :** Les éléments de la définition du diagramme d'EMF (nœuds et liens) sont mappés au modèle de domaine "domain model" et assigné et aux éléments d'outillage affectés "tooling" dans un fichier avec l'extension ".gmfmap".
- **Générer "generator model" et le "diagram code" :** Fichier avec l'extension ".gmfgen" qui nous permet de générer un diagramme plugin.

Génération de "Graphical definition model"

- Pour créer le "graphical definition model",qui définit les éléments de surface de l'éditeur graphique.

- Nous cliquons d'abord sur "Drive" de "Graphical Def Model" sur la vue du "dashboard", puis nous donnons "ModulesModel.gmfgraph" comme nom → "Next" → "Load" → nous sélectionnons la racine (CompositeElement) → "Next" → maintenant vérifier nos éléments qui doivent être visualisés sur la surface de l'éditeur → "Finish".

- La Figure 5.11 affiche les éléments de la class "ModulesModel.gmfgraph".

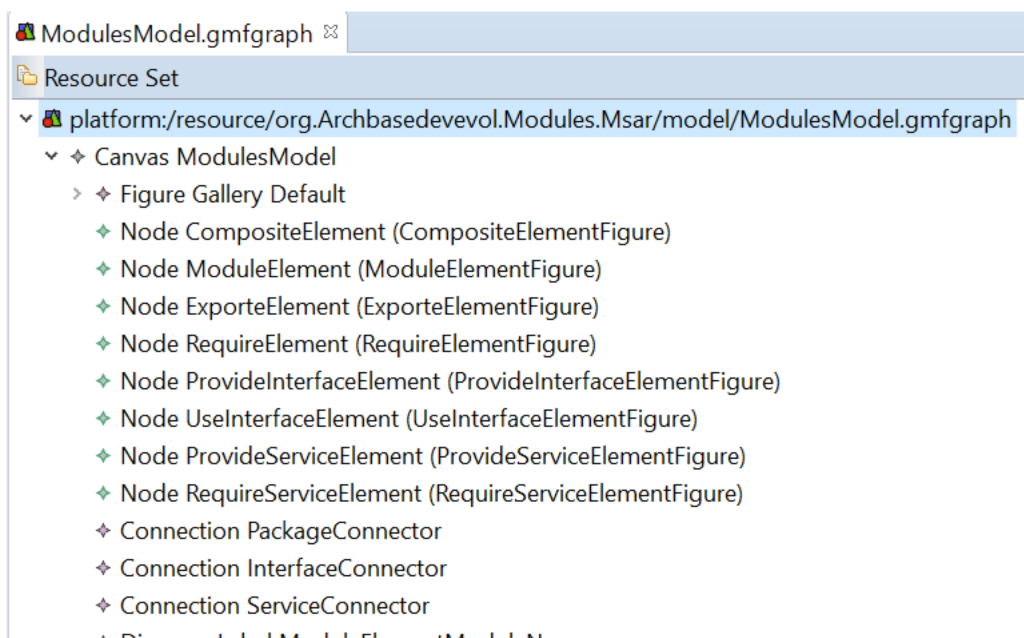


FIGURE 5.11 – "ModulesModel.gmfgraph".

Génération de "tooling definition model"

- Ce modèle présente la palette et les éléments de menu de l'éditeur graphique. Pour créer le "tooling definition model".

- Nous cliquons d'abord sur "Drive" de "Tooling Def Model" sur la vue du tableau de bord, puis nous donnons "ModuleModel.gmftool" comme nom → "Next" → "Load".
Nous sélectionnons la racine, dans notre cas c'est "CompositeElement" →

"Next" → puis on vérifier les éléments de la palette → "Finish".

- L'organisation des "Tool Groups" est affiché dans La Figure 5.12.

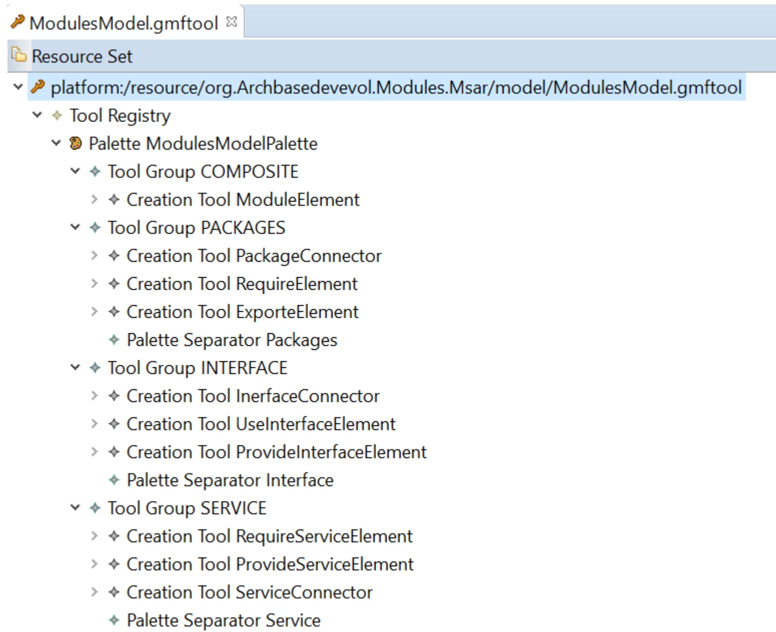


FIGURE 5.12 – "ModuleModel.gmftool".

- Nous pouvons changer les icônes de la palette en plaçant les nouvelles dans le dossier "icons" dans le plug "edit" - en sélectionnant le floder "edit" → "icons" → "full" → "obj16", et il devrait avoir le même nom et avec l'extension "GIF".

- Ils seront modifiés dans la palette de l'éditeur après l'exécution de l'éditeur.

Génération de "mapping model"

- Ce modèle lie les trois modèles précédents : le "domain", le "graphical definition", et le "tooling definition".

- Pour créer le "mapping model", nous cliquons d'abord sur "Combine" dans le tableau "dashboard", puis donnons "ModuleModel.gmfmap" comme nom → "Next" → "Load" → nous sélectionnons l'élément racine (CompositeElement) → "Next" → "Load" → "Next" → "Load" → "Next".

- Dans cette étape nous devons réglé les Éléments de modèle les nœuds et les liens (Nodes and Links), comme La Figure 5.13 est affichée.

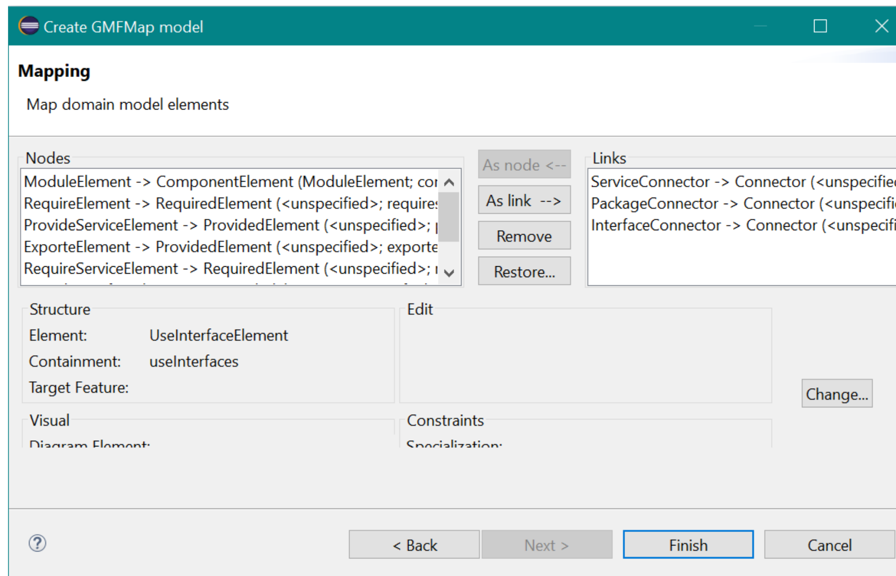


FIGURE 5.13 – Éléments de modèle "map domain".

- Avant de clique sur "Finish", il faut vérifier les propriétés de chaque lien, et on fait ça à partir le button "Change", et on définie les sections suivantes : "source feature" et "target feature", "diagram link", "tool", et on clique sur "Finish", Figure 5.14.

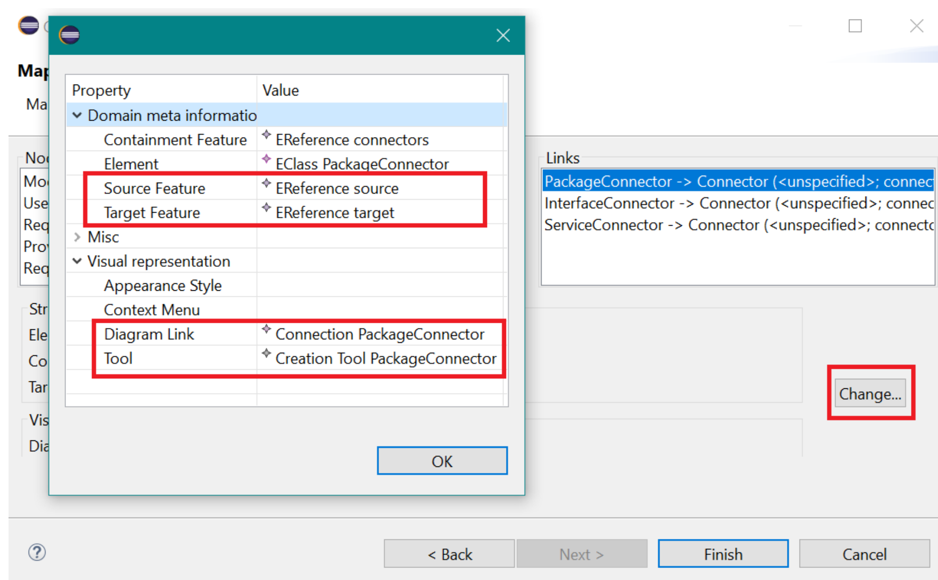


FIGURE 5.14 – Propriétés des connecteurs de "map domain".

- La Figure 5.16, affiche l'organization des noueds de le fichier "ModuleModel.gmfmap".

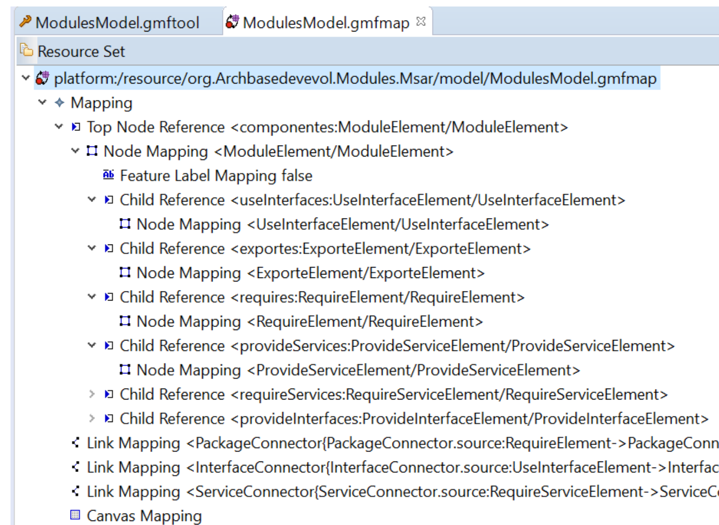


FIGURE 5.15 – le fichier "ModuleModel.gmfmap".

- Pour chaque "Child Refrence", il faut vérifier les propriétés "Containment Feature" et "Refrenced Child", et pour les "Node Mapping" il faut vérifier les propriétés "Diagram Node" et "Tool".

Génération de "generator model" et "diagram code"

- Cette étape est la dernière dans la création de l'éditeur graphique. Tout d'abord, nous allons dans le "dashboard" et on cocher la case "RCP" → cliquer sur "Transforme", cela donnera le modèle "ModulesModel.gmfgen" → cliquer sur "Gen Editor Generation" cela nous donner un nouvelle model avec l'extention "gmfgen".

- Nous revenons au la vue "dashboard" et on clique sur "Generate Diagram Editor" pour générer le "diagram" du plug-in.

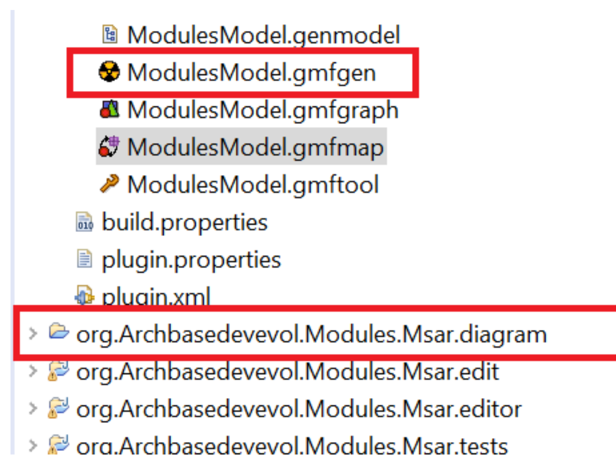


FIGURE 5.16 – "generator model" et "diagram code" .

5.4 Exécution de notre outil :

Cette section présent les étapes de l'exécution de l'outil.

5.4.1 Configuration de l'exécution

Pour exécuter notre application, nous devons d'abord créer une nouvelle configuration..

- Dans "Exécuter les configurations", nous faisons un clic droit sur "Eclipse Application" → "New" → définir le nom de la configuration → "Apply" → "Run".

- Après avoir cliqué sur "Run", "Validation" s'affiche, nous sélectionnons "org.apache.xmlrpc" → "OK".

- Lorsque nous cliquons sur "OK", l'éditeur graphique s'ouvre.

5.4.2 Visualisation graphique et récupération de l'architecture

Ces étapes sont exécutées lors de l'architecture logicielle au moment de l'exécution est récupéré. Notre outil génère un fichier XMI pour chaque type d'éléments d'architecture. Après la récupération de l'architecture logicielle et la génération de sa version XMI, maintenant pour présenter graphiquement

cette architecture faites ceci :

- Copie du fichier XMI dans la vue "Project Explore" en changeant son extension en ".modulesmodel".

- En cliquant sur "File" → "Initialize modulesmodel-diagram diagram file" → en sélectionnant le fichier XMI que nous voulons faire évoluer → "Finish".

- Maintenant, nous pouvons voir notre architecture logicielle récupérée et tous ses éléments au moment de l'exécution.

5.5 Conclusion

Dans ce chapitre, nous avons présenté l'architecture globale et détaillée de notre outil "*ArchBaseDevModules*". Nous avons également présenté les outils et les cadres de développement que nous avons utilisés pour mettre en œuvre notre processus d'approche proposé. Ensuite, nous avons montré les étapes que nous avons suivies pour créer et développer notre application.

Conclusion générale

Aujourd'hui, la récupération de l'architecture logicielle est presque indispensable pour comprendre le fonctionnement d'un logiciel. Elle joue un rôle important lors de la tâche de maintenance et d'évolution des systèmes logiciels, ce qui permet d'économiser le temps, l'argent et l'effort et de réduire la complexité.

Dans ce travail, nous nous concentrons sur l'évolution dynamique de l'architecture logicielle pour aider les développeurs à faire évoluer les systèmes volumineux et complexes pendant l'exécution.

Nous avons proposé un méta-modèle pour les à systèmes à base de module Java 9. Ce méta-modèle est basé sur un méta modèle OSGI existé. Nous avons également proposé un processus qui contient trois sous-processus, le premier visant à récupérer l'architecture logicielle d'un système logiciel au moment de l'exécution, il commence par récupérer l'architecture basée sur les modules à partir des applications au moment de l'exécution. Cela commence par l'identification de tous les modules actifs de l'application et de leurs relations les uns avec les autres. Cela est faite par l'obtention des fichiers modules.jar et l'analyse des fichiers module-info.java de chaque module sélectionné. Après cela, nous générons l'architecture des modules sous forme de fichier XMI et par la création d'une instance de notre méta-modèle des modules Java 9. L'architecture sera plus tard visualisée en utilisant notre éditeur graphique. La visualisation graphique est considéré dans la deuxième sous-processus de l'architecture logicielle aide les développeurs à comprendre le système pour commencer la tâche d'évolution et de maintenance qui est la dernière étape dans notre processus. A la fin, on obtient une nouvelle architecture logicielle et un système bien évolué.

Nous visons dans le future de :

- Découvrir notre application avec d'autres plates-formes pour démontrer son efficacité.
- Inclure l'intelligence artificielle afin de trouver un intelligent pour organiser les énormes composants de l'architecture logicielle afin de simplifier la visualisation des grands systèmes.

- Inclure l'intelligence artificielle dans la maintenance où la machine peut prendre les décisions pour faire évoluer le système.

Bibliographie

[1] Sander Mak and Paul Bakker. *“Java 9 Modularity” :Patterns and Practices for Developing Maintainable Applications.* 2016.

[2] Alexandru Jecan. “Java 9 Modularity Revealed ” :Project Jigsaw and Scalable Java Applications. page 236.

[3] Mira Kajko-Mattsson. Applicability of ieee 1219 within correctiev maintenance. In *2006 International Conference on Software Engineering Advances (ICSEA'06)*, pages 13–13. IEEE, 2006.

[4] Alae-Eddine El Hamdouni, Abdelhak-Djamel Seriai, and Marianne Huchard. "Component-based architecture recovery from object oriented systems via relational concept analysis" . (University of Sevilla) [In *CLA : Concept Lattices and their Applications*]. page 259–270, 2010.

[5] Hong Mei and Jian Lü. “Runtime recovery and manipulation of software architecture of component-based systems“. (Springer) [In *Internetware*]. page 115–138, 2016.

[6] Thibaud Lutellier et al. “Measuring the impact of code dependencies on software architecture recovery techniques“ . (University of Sevilla) [In *IEEE Transactions on Software Engineering*]. page 159–181, 2017.

[7] Mohamed Lamine Kerdoudi, Tewfik Ziadi, Chouki Tibermacine, and Salah Sadou. Recovering software architecture product lines. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 226–235. IEEE, 2019.

[8] Amani Khadraoui. *Software Architecture-Based Evolution for Component/Service-Oriented Systems.* PhD thesis, Université Mohamed Khider - BISKRA, 2020.

[9] OpenJDK Developer(s) :Oracle and Java Community. projects jigsaw, 2017.

[10] Sander Mak and Paul Bakker. *Java 9 Modularity : Patterns and Practices for Developing Maintainable Applications.* " O'Reilly Media, Inc.", 2017.

[11] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.

[12] Nassima Sadou. *Evolution Structurelle dans les Architectures Logicielles à base de Composants*. PhD thesis, Université de Nantes ; Ecole Centrale de Nantes (ECN), 2007.

[13] Richard N Taylor. Software architecture : many faces, many places, yet a central discipline. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 303–304, 2009.

[14] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component software : beyond object-oriented programming*. Pearson Education, 2002.

[15] Ian Sommerville. Software testing. *Software Engineering*, 8 :537–65, 2010.

[16] Microsoft Patterns. *Microsoft application architecture guide*. Microsoft Press, 2009.

[17] Meir M Lehman. Program evolution. *Information Processing & Management*, 20(1-2) :19–36, 1984.

[18] Z. Zhang, R. Liu, and H. Yang. "Service identification and packaging in service oriented reengineering" . ((SEKE) [In *the 17th International Conference on Software Engineering and Knowledge Engineering*]. page 620–625, 2005.

[19] Marvin Grieger, Stefan Sauer, and Markus Klenke. "Architectural restructuring by semi-automatic clustering to facilitate migration towards a serviceoriented architecture" [In *2nd Workshop Model-Based and Model-Driven Software Modernization.*]. page 44–45, 2014.

[20] David Garlan and Bradley Schmerl. "Ævol : A tool for defining and planning architecture evolution" [In *2009 IEEE 31st International Conference on Software Engineering*]. page 591–594, IEEE. 2009.

[21] Jeffrey M Barnes, David Garlan, and Bradley Schmerl. "Evolution styles : foundations and models for software architecture evolution" [In *Software and Systems Modeling 13.2*]. page 649–678, 2014.

[22] Jennifer Pérez et al. "Evolution style : framework for dynamic evolution of real-time software architecture" [In *European Conference on Software Architecture.*]. pages 59–76, 2005.

[23] Adel Hassan, Audrey Queudet, and Mourad Oussalah. "Evolution style : framework for dynamic evolution of real-time software architecture" [In *European Conference on Software Architecture.*]. page 166–174, 2016.