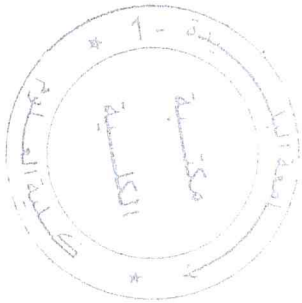
MA-004-428-1

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Saad DAHLEB de Blida



Faculté des Sciences

Département d'Informatique

Thème : Aide à la spécification des architectures logicielles dynamiques

Rapport présenté par : **HAMIDI ABDERRAHMANE**
HANECHE MOHAMED

En vue d'obtenir le diplôme de Master

Domaine : Mathématiques et Informatique
Filière : Informatique
Spécialité : Systèmes Informatiques et Réseaux

Promotrice : Mme Guessoum Dalila
Devant les jurys : Professeur Abed Hafida
Monsieur Hammouda Mohamed
Soutenu le : 30/06/2018
Année Universitaire : 2017 /2018

Remerciements

جامعة البليدة 1 - مكتبة الكلية - كلية العلوم

Tout d'abord, Nous tenons à remercier DIEU le tout puissant de nous avoir donné la force, le courage et la patience pour accomplir notre travail.

Nous tenons à remercier nos très chers parents pour leur tendresse, leur encouragement et leur soutien moral et matériel dans le but d'assurer notre réussite.

Nous tenons à remercier vivement notre promotrice madame **Guessoum Dalila** pour sa précieuse aide, sa patience, sa générosité, et surtout son soutien moral et sa disponibilité tout au long de ce projet.

Enfin un grand merci à nos collègues et tous ceux qui ont contribué de près ou de loin à la réalisation de ce mémoire.

Résumé

L'approche orientée composants a été l'approche du génie logiciel la plus prometteuse pour développer des systèmes logiciels complexes. Cependant, avec le développement technologique, la demande d'approches de spécifications dynamiques et flexibles est devenue essentielle. Notre travail s'inscrit dans ce domaine, il consiste à proposer une aide à la spécification des architectures dynamiques. Pour se faire nous avons proposé tout d'abord un méta-modèle qui permet de décrire l'architecture logicielle et son évolution. Ensuite nous avons proposé pour chaque élément de notre méta-modèle une description graphique pour aider l'architecte à décrire son architecture. Nous avons ensuite développé un éditeur graphique selon notre méta-modèle. Nous avons aussi proposé un processus pour la gestion des architectures dynamiques que nous avons incorporé dans ce qu'on a appelé le gestionnaire d'évolution. Une étude de cas a été utilisée pour valider l'outil proposé.

Mots clés

Architecture logicielle dynamique, évènement interne, évènement externe, méta-modèle, éditeur graphique.

الملخص

لقدكانالنهجالموجهللمكونأحدأكثرالمناهجالواعدةفيمجالهندسةالبرمجياتلتطويرأنظمةبرمجيةمعقدة.

رغمذلك،ومعالتطورالتكنولوجيالحاصل،أصبحالطلبعلىنهجوصفالديناميكيةوالمرونةضروريًا.

عملنافيهذاالمجاليتمثلفياقتراحمساعدةلوصفالمعمارياتالديناميكية.

للقيامبذلك،اقترحناأولاًنموذجًافوقيايصفبنيةالبرامجوتطورها.

ثماقترحناالكلعنصرمنعناصرنموذجناالفوقيوصفًارسوميًالمساعدةالمهندسفيوصفبنيتهالمعمارية.

ثمقمنابتطويرمحررسوماتوفقاًللنموذجناالفوقي.

كمااقترحناأيضًاعمليةلإدارةالمعمارياتالديناميكيةالتيأدخلناهافيماأطلقناعليهاسممديرالتطوير.

أخيرا،وللتحققمنصحةالأداةالمقترحةقمنابتطبيقهذاالنموذجعلىأحدالحالات.

الكلمات المفتاحية

معمارية البرامج الديناميكية، حدث داخلي، حدث خارجي، نموذج فوقي، محرر رسومي.

.

Abstract

The component-oriented approach has been the most promising software engineering approach to develop complex software systems. However, with the technological development, the need of dynamic and flexible specification approaches has become essential. Our work aims to contribute in this research field, it consists to assist the architect in the specification of dynamic architectures. To do so, we first proposed a meta-model that describes the software architecture and its evolution. Then we proposed for each element of our meta-model, a graphical description to help the architect to describe its architecture. Then we developed a graphical editor according to our meta-model. We also proposed a process for the management of dynamic architectures that we incorporated in what we called the evolution manager. A case study was conducted to validate the proposed tool

Keywords

Dynamic software architecture, interne event, externe event, meta-model, graphical editor.

Table des Matières

Chapitre 2 :

La méta modélisation

Chapitre 3 :

Méta-modèle pour la Spécification des Architectures Logicielles Dynamiques

Chapitre 4 :

Implémentation

Liste des figures

Liste des tableaux

Introduction générale

Contexte

De nos jours, les architectures logicielles sont devenues une partie intégrante du développement de logiciels, elles sont caractérisées par des changements fréquents et imprévisibles dans l'environnement dans lequel ils fonctionnent et les exigences qu'ils doivent respecter. Pour faire face à ce problème de l'évolution du logiciel, il est nécessaire de considérer l'évolution de ces systèmes au niveau architectural.

En effet, quand un système évolue, son architecture est influencée et vice-versa, l'architecture logicielle est apparue comme un outil puissant pour guider et planifier l'évolution des logiciels. Il offre un niveau abstraction élevé pour la description des systèmes. Cela permet aux développeurs de ne pas tenir compte des détails inutiles et de se concentrer sur la structure globale du système.

Problématique

Cependant, le caractère dynamique dans les architectures logicielles introduit des difficultés supplémentaires concernant la description. En effet, Dans le cas d'une architecture statique, la description consiste à "simplement" énumérer les différents composants et connexions qui la constituent. Cette approche est inadaptée pour la description des architectures dynamiques dont la configuration, par définition, change pendant l'exécution et dont les composants et les connexions peuvent être introduits et supprimés tout au long du cycle de vie de l'application. De plus, les architectes ont peu d'outils pour les aider à planifier et à exécuter l'évolution des architectures logicielles.

Objectifs

Il est question dans ce projet de proposer un méta-modèle décrivant les concepts relatifs à la spécification de l'évolution (dynamisme) des architectures. Ce méta-modèle décrit et recense l'ensemble des événements qui peuvent se produire pendant le déploiement de l'architecture logicielle. Ces événements peuvent être un événement simple, tel qu'une reconfiguration manuelle du logiciel déclenchée par un utilisateur ou un événement déclenché par un autre événement. Le méta-modèle proposé permettra de prendre en compte des informations

historiques simples pour faciliter l'évolution des systèmes, pour ce faire il est nécessaire de stocker les connaissances en termes d'actions appropriées qui ont été prises par des types spécifiques en réponse aux événements standards.

Sur la base de ce méta-modèle il est demandé en second lieu de concevoir une interface graphique permettant la modélisation graphique du modèle proposé. Cet outil sera développé avec GMF (Graphical Modeling Framework), qui permet de générer des éditeurs graphiques. Il sera implémenté avec Java sous Eclipse.

Organisation de mémoire

Notre mémoire se compose de quatre chapitres organisés comme suit :

- Le premier chapitre présente une étude sur les architectures logicielles. Il est partitionné en deux sections, la première section introduit les architectures logicielles et leurs principaux concepts comme leurs éléments, leurs styles architecturaux et leurs langages de description. La deuxième section est consacrée à l'étude des architectures dynamiques et ses différents types.
- Le deuxième chapitre présente la notion de méta-modélisation et les méta-modèles ainsi que les différents concepts liés à la méta-modélisation.
- Le troisième chapitre décrit notre méta-modèle proposé pour décrire la dynamique des architectures logicielles. Ainsi, il introduit des notations graphiques pour réifier ce méta-modèle, dans ce chapitre un cas d'étude est aussi présenté pour illustrer notre approche.
- Le quatrième chapitre présente l'implémentation de notre méta-modèle avec le framework GMF (Graphical Modeling Framework) qui permet de générer notre éditeur graphique.

1 Introduction

L'architecture logicielle est une partie intégrante du développement de logiciels, la gestion de son évolution est devenue la préoccupation de la plupart des chercheurs en architecture.

Dans ce chapitre nous allons aborder les parties les plus importants pour bien comprendre le contexte dans lequel s'intègre notre travail. Nous commençons tout d'abord par présenter les concepts de base de l'architecture logicielle, puis nous abordons par la suite la problématique des architectures logicielles dynamiques.

2 Architecture logicielle

L'architecture logicielle se définit comme une spécification abstraite d'un système en termes de composants logiciels ou modules qui le constituent, des interactions entre ces composants à travers des connecteurs et d'un ensemble de règles configuration qui gouvernent cette interaction [1].

Les composants encapsulent typiquement l'information ou la fonctionnalité. Tandis que les connecteurs assurent la communication entre les composants. Cette architecture possède, généralement, un ensemble de propriétés d'ordre topologique ou fonctionnelle qu'elle doit respecter tout au long de son évolution.

3 Les éléments d'une architecture logicielle

L'architecture est une vue abstraite d'un système en termes d'éléments architecturaux. Ces éléments sont :

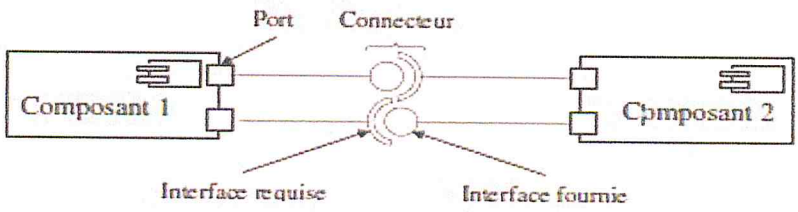


Figure 1. 1 Les éléments d'une architecture logicielle[22].

3.1 Le Composant

Un composant est une unité de calcul ou de stockage. Il peut être primitif ou composé. On parle dans ce dernier cas de composite. Sa taille peut aller de la fonction mathématique à une application compl²e. Deux parties définissent un composant. Une première partie, dite externe, comprend la descr0iption des interfaces fournies et requises par le composant. Elle définit les interactions du composant avec son environnement.

La seconde partie, dite interne, correspond à son contenu et permet la description du fonctionnement interne du composant [2].

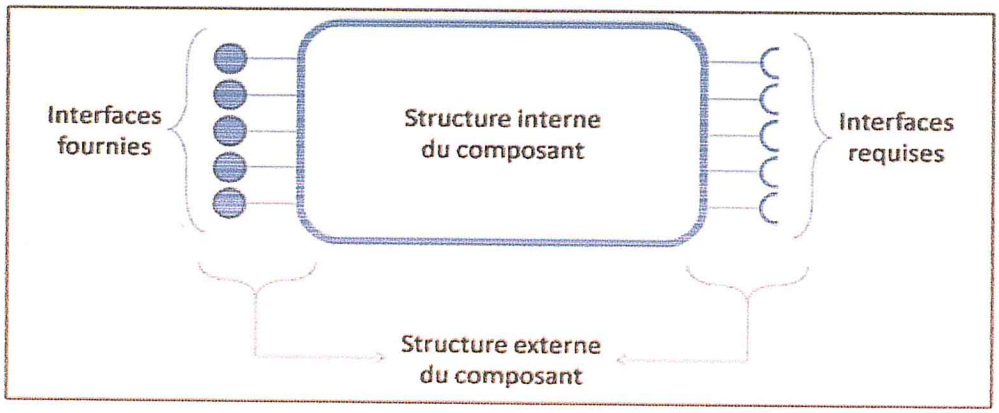


Figure 1. 2 Structure d'un composant [4].

3.1.1 Structure externe et interne d'un composant

- La structure externe d'un composant est généralement caractérisée par :
  - Les interfaces du composant : c'est la spécification des services fournis et requis par le composant.
  - Les propriétés du composant : elles servent à documenter l'architecture en décrivant les aspects relevant de la conception ou de l'analyse du composant.
- Il existe deux types de structure interne d'un composant :
  - Composant atomique : comporte la description de l'implémentation des fonctionnalités du composant.
  - Composant composite : comporte des composants internes pouvant être aussi composite ou atomique.

3.2 Port

Un composant interagit avec son environnement par l'intermédiaire de points d'interactions appelés ports. Les ports (et par conséquent les interfaces) peuvent être fournis ou requis. Un port représente un point d'accès à certains services du composant [3]. Le comportement interne du composant ne doit être ni visible, ni accessible autrement que par ses ports.

3.3 Le connecteur

Les connecteurs gèrent les interactions entre les composants, c'est-à-dire, ils établissent les règles qui gouvernent les interactions entre composants et spécifient tous les mécanismes auxiliaires nécessaires [4].

Les connecteurs sont des entités architecturales de communication qui modélisent de manière explicite les interactions (transfert de contrôle et de données) entre les composants. Ils contiennent des informations concernant les règles d'interaction entre les composants. Ainsi, l'objectif des connecteurs est d'atteindre une meilleure réutilisabilité lors de l'assemblage des composants.

En effet, la raison de l'existence des connecteurs est de faciliter le développement d'applications à base de composants logiciels. Les composants s'occupent du calcul et stockage tandis que les connecteurs s'occupent de gérer les interactions (communication/coordination) entre les composants.

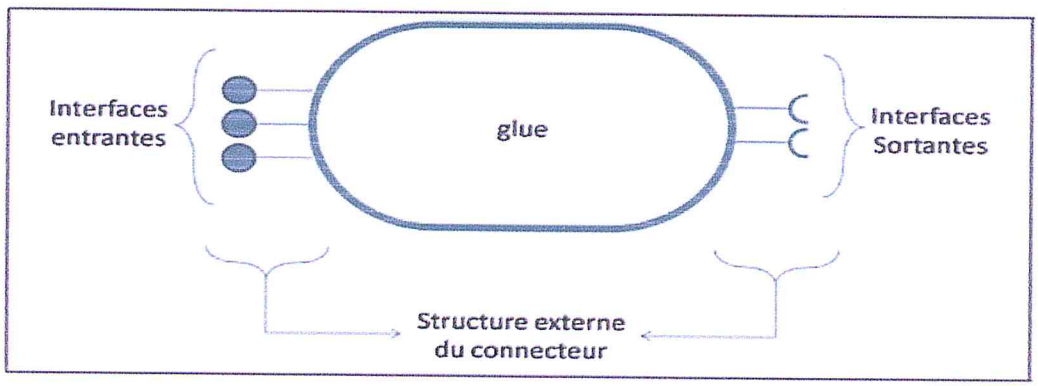


Figure 1. 3 Structure d'un connecteur [4].

3.3.1 Structure externe d'un connecteur

La structure externe d'un connecteur est généralement caractérisée par :

- Les interfaces du connecteur :
  - Constituent les points de connexion entre les connecteurs et les composants.
  - A la différence des composants, les interfaces ne décrivent pas de services fonctionnels mais des mécanismes de connexion.
- Les propriétés du connecteur :
  - Les propriétés non fonctionnelles d'un connecteur : représentent les informations additionnelles nécessaires pour l'implémentation correcte d'un connecteur (performance, sécurité, …).
  - Les contraintes : permettent d'assurer les protocoles d'interaction prévus, d'établir les dépendances intra-connecteur et de fixer les conditions d'utilisation des connecteurs. Comme pour les contraintes portant sur les composants, ces contraintes doivent être vérifiées pour que le système soit considéré comme cohérent.

3.3.2 Structure interne d'un connecteur

Il existe deux types de structure interne d'un connecteur :

- Connecteur atomique :
  - Décrit le protocole de communication entre les interfaces, points d'accès des composants vers le connecteur.
  - La structure interne des connecteurs atomiques est appelée **glue.**
- Connecteur composite :
  - Possède une structure interne composée de composants, de connecteurs et d'une configuration.

3.4 Configuration

La configuration définit la structure de l'architecture. Pour cela, la configuration d'une architecture présente la topologie des connexions entre les composants et les connecteurs ainsi que les propriétés de cette topologie. Par conséquent, la configuration est caractérisée par deux éléments : la structure de la configuration, représentant la topologie des connexions entre composants et connecteurs, et les propriétés de la configuration [4].

3.4.1 Structure de la configuration

La structure de la configuration représente la topologie des connexions entre les composants et les connecteurs. Elle vérifie la correspondance entre les interfaces des composants et des connecteurs.

3.4.2 Les propriétés de la configuration

Les propriétés de la configuration sont similaires à celles des composants et des connecteurs. Comme pour les autres éléments architecturaux, ces propriétés sont de deux types:

- **Propriétés non fonctionnelles :** certaines propriétés non fonctionnelles ne peuvent pas être exprimées au niveau des composants ou des connecteurs. Il faut donc exprimer ces propriétés au niveau de la configuration. Ces propriétés concernent par exemple l'environnement de déploiement de l'architecture.
- **Contraintes :** les contraintes portant sur la configuration s'ajoutent à celles portant sur les autres éléments architecturaux. Elles permettent d'exprimer des contraintes portant sur plusieurs éléments architecturaux. Par exemple, une contrainte peut exprimer une relation entre deux composants.

  Les contraintes de la configuration peuvent également être globales et porter sur l'ensemble des éléments architecturaux.

4 Style architectural

Dans le domaine des architectures logicielles, une des manières, est de classer les architectures par catégories et de définir leurs caractéristiques communes. En effet, un style architectural définit une famille d'architectures logicielles qui sont caractérisées par des propriétés structurelles et sémantiques communes [5].

Un style n'est pas une architecture mais une aide générique à l'élaboration de structures architecturales [6]. Un style architectural inclut une spécification statique et une spécification dynamique. La partie statique englobe l'ensemble des éléments (composants et connecteurs) et des contraintes sur ces éléments. La partie dynamique décrit l'évolution possible d'une architecture en réaction à des changements prévus ou imprévus de l'environnement. Un style architectural est précisément défini par un ensemble de caractéristiques telles que : le vocabulaire utilisé (les types de composants et de connexions),

Les contraintes de configuration (contraintes topologiques appelées aussi patrons structurels), les invariants du style, les exemples communs d'utilisation, les avantages et inconvénients d'utilisation de ce style et les spécialisations communes du style. Un style architectural spécifie aussi les types d'analyse que l'on peut faire sur ses instances (systèmes construits conformément à ce style) [7]. Parmi les principaux styles architecturaux, nous citons le style "client-serveur", le style "publier-souscrire" et le style "pipe and filtre".

4.1 Style "client-serveur"

Il est sans doute le style le plus connu de tous [8, 9]. Comme le montre la figure 1.4, il se base sur deux types de composants : un composant de type serveur, offrant un ensemble de services, écoute des demandes sur ses services. Un composant de type client, désirant qu'un service soit assuré, envoie une demande (requête) au serveur par l'intermédiaire d'un connecteur. Le serveur rejette ou exécute la demande et envoie une réponse de nouveau au client. La contrainte qui s'impose dans ce type est qu'un composant ne peut-être qu'un fournisseur de services ou un demandeur de services. Le style client-serveur consiste alors à structurer un système en termes d'entités serveurs et d'entités clientes qui communiquent par l'intermédiaire d'un protocole de communication à travers un réseau informatique.

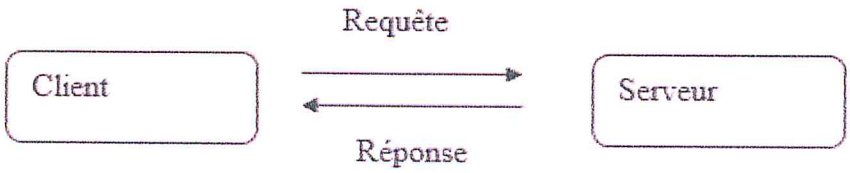


Figure 1. 4 Le style client-serveur [22].

4.2 Style "publier-souscrire"

Ce style, comme le montre la figure 1.5, se base sur trois composants. Un composant **producteur**qui va produire des informations. Un composant **consommateur** qui va les consommer et un composant **service d'événement** qui va assurer l'échange d'informations entre les producteurs et les consommateurs. Ces derniers ne communiquent donc pas directement et ne gardent même pas les références des uns et des autres. De plus, les producteurs et les consommateurs n'ont pas besoin de participer activement à l'interaction selon un mode synchrone. Le producteur peut publier des événements pendant que le consommateur est déconnecté, et réciproquement, le consommateur peut être notifié à propos d'un évènement pendant que le producteur, source de cet événement, est déconnecté.

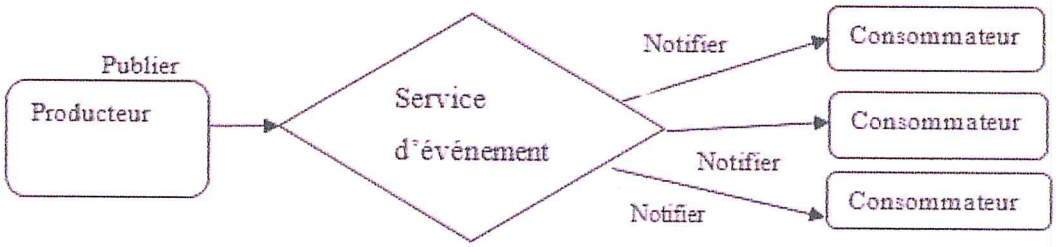


Figure 1. 5 Le style publier-souscrire [22].

4.3 Style "pipes and filters"

Dans ce style, comme le montre la figure 1.6, un composant reçoit un ensemble de données en entrée et produit un ensemble de données en sortie. Le composant, appelé **filtre**, lit continuellement les entrées sur lesquelles il exécute un traitement ou une sorte de "filtrage" pour produire les sorties [7].

Les spécifications des filtres peuvent contraindre les entrées et les sorties. Un connecteur, quant à lui, est appelé **pipe** puisqu'il représente une sorte de conduite qui permet de véhiculer les sorties d'un filtre vers les entrées d'un autre.

Le style "pipes and filters" exige que les filtres soient des entités indépendantes et qu'ils ne connaissent pas l'identité des autres filtres. De plus, la validité d'un système conforme à ce style ne doit pas dépendre de l'ordre dans lequel les filtres exécutent leur traitement.

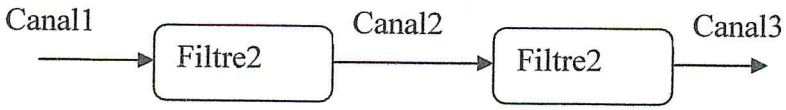


Figure 1. 6 Le style pipes and filters [22].

5 Les langages de description d'architectures (ADL : Architecture Description Language)

Les ADLs sont une famille de langages élaborés dans les années 90 dans le domaine de l'ingénierie du logiciel. Ils ont en commun une capacité à décrire l'architecture des systèmes mais diffèrent largement quant à leurs motivations et leurs formalismes d'écriture. Ils sont en général graphiques et textuels et possèdent des outils associés [10]. Le tour d'horizon qui suit donne une indication sur la diversité des concepts et des usages couverts par ce type de langage :

- **Aesop :** Développement d'applications fondé sur la notion de style architectural.
- **Darwin :** Configuration et instanciation dynamique des systèmes distribués.
- **C2 :** Architectures évolutives ou dynamiques, C2 s'appuie sur un style d'architecture logiciel en couche et des communications par échange de message.
- **Rapide :** Modélisation, simulation et analyse du comportement dynamique d'un système à l'exécution. Rapide se fonde sur le concept de posets (Partial Ordered event Sets).
- **UniCon :** Construction et vérification d'architectures à partir d'éléments architecturaux prédéfinis.
- **Wright :** Vérification formelle, absence de blocage, Wright décrit formellement le comportement d'une architecture à l'aide de CSP (Communicating Sequential Process).
- **SADL** : Structural Architecture Description Langage Raffinement formel (traçabilité des variantes et des évolutions).
- **Acme :** Interchange, Acme a été conçu comme langage passerelle pour permettre aux différents ADLs d'interopérer. Cette vocation l'a conduit à se définir comme langage d'architecture générique.
- **Olan :** Développement, configuration, déploiement et administration d'applications réparties.

6 Avantages des architectures logicielles

L'architecture logicielle est une partie intégrante du développement de logiciels, elle permet entre autres :

La compréhension du système : l'architecture fournie une représentation d'un système à un haut niveau d'abstraction. Cette vue synthétique du système met en valeur la plupart des décisions de conception ainsi que les conséquences de ces mauvaises décisions.

- **La réutilisation :** les descriptions architecturales favorisent la réutilisation à plusieurs niveaux. (Réutilisation des bibliothèques de composants). La conception architecturale supporte la réutilisation de grands composants (large components), ainsi que les Framework ou les composants peuvent être intégrés.
- **L'évolution :** l'architecture fournie un squelette du système. Ce squelette permet d'identifier les parties fortement utilisées ainsi que les parties potentiellement fragiles. L'architecture permet ainsi de mettre en valeur les parties nécessitant une attention particulière lors de l'évolution du système. Mais l'architecture permet également de révéler une image précise des dépendances entre les composants. Cette image est nécessaire pour connaitre les impacts de la modification d'un composant sur les autres composants du système et donc les conséquences des différentes évolutions.
- **L'analyse :** la vue abstraite fournie par l'architecture permet de mesurer différents attributs tels que la consistance du système, son respect du style architectural ou encore d'autres attributs de qualité. Elle permet également de vérifier que les changements prévus dans le système sont conformes au style et aux objectifs de qualité fixés à la conception.
- **La gestion de projets :** l'architecture permet une gestion plus précise des coûts et des risques de modifications, en particulier en soulignant les dépendances entre les composants. Elle permet également une évaluation des qualités du système dans son ensemble, mais aussi des qualités de chaque composant.

7 Architecture logicielle dynamique

Les architectures logicielles, comme nous l'avons présenté précédemment, peuvent être décrites comme étant un ensemble de composants et de connecteurs dans un cadre commun.

Toutefois, cette approche est une approche statique. En effet, la structure du système, Une fois son architecture définie, n'évolue pas dans le temps. Ce type d'architecture commence à perdre de l'ampleur et céder la place au concept d'architecture dynamique suite à de

nombreuses raisons détaillées plus tard. Dans la suite, nous présentons une définition et quelques motivations qui sont derrière la naissance de ce concept.

7.1 Définition et motivation de la dynamique des architectures

Depuis quelques années, la dynamique des architectures est devenue une activité qui commence à prendre de l'ampleur. Les architectures des systèmes logiciels tendent à devenir de plus en plus dynamiques. En effet, durant son cycle de vie, ce type d'applications peut être amené à évoluer de différentes manières : évolution de l'architecture, ajout de fonctions, remplacements de certains composants en cours d'exécution par de nouveaux développés selon de nouvelles technologies de programmation ou de communication, etc.

7.2 Raison derrière la dynamique des architectures

La dynamique des architectures peut être réalisée pour différentes raisons. Ces raisons peuvent être classées en quatre catégories [11] :

- **Dynamique correctionnelle** : dans certains cas, on remarque que l'application en cours d'exécution ne se comporte pas correctement comme prévu. La solution est d'identifier le composant de l'application qui pose problème et le remplacer par une nouvelle version supposée correcte. Cette nouvelle version fournit la même fonctionnalité que l'ancienne, elle se contente simplement de corriger ses défauts.
- **Dynamique adaptative** : même si l'application s'exécute correctement, parfois l'environnement d'exécution, comme le système d'exploitation, les composants matériels ou d'autres applications ou données dont elle dépend, changent. L'architecture doit donc s'adapter et évoluer soit en ajoutant des nouveaux composants/connexions soit en remplaçant des composants/connexions par d'autres.
- **Dynamique évolutive** : au moment du d'enveloppement de l'application, certaines fonctionnalités ne sont pas prises en compte. Avec l'évolution des besoins de l'utilisateur, l'application doit être étendue avec de nouvelles fonctionnalités. Cette extension peut être réalisée en ajoutant un ou plusieurs composants/connexions pour assurer les nouvelles fonctionnalités ou même en gardant la même architecture de l'application et étendre simplement les composants existants.
- **Dynamique perfective** : l'objectif de ce type d'adaptation est d'améliorer les performances du système. A titre d'exemple, on se rend compte que l'implémentation d'un composant n'est pas optimisée. On décide alors de remplacer l'implémentation du composant en question. Un autre exemple peut être celui d'un composant qui reçoit beaucoup de requêtes

et qui n'arrive pas à les satisfaire. Pour éviter la dégradation des performances de l'application, on diminue la charge de ce composant en installant un autre qui lui partage sa tâche.

7.3 Les types des architectures dynamiques

Pour soutenir le développement des architectures dynamiques, plusieurs chercheurs se sont concentrés sur le développement d'approches et de formalismes fournissant des systèmes d'architecture dynamique. Suite à ces recherches, différents types de dynamiques sont apparus. Parmi ces types nous trouvons :

- **Dynamique d'implémentation** : ce type de dynamique permet de modifier la mise En œuvre d'un composant, c'est-à-dire la manière avec laquelle un composant a été développe (le code du composant) sans changer ni les interfaces ni les connexions. Ce type de changement permet de faire évoluer un composant d'une version à une autre (voir figure 1.7).

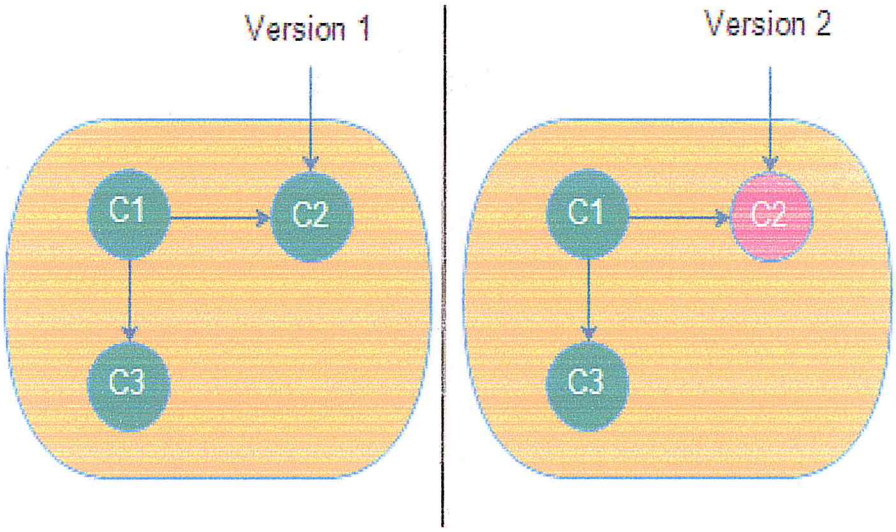


Figure 1. 7 Exemple de changement d'implémentation

- **Dynamique d'interfaces** : comme nous l'avons détaillé précédemment, un composant fournit ses services à travers des interfaces. La dynamique d'interfaces consiste à modifier les services fournis par un composant au biais de ses interfaces. Ceci, se traduit soit par la modification de l'ensemble des services fournis soit par la modification de la signature d'un service (voir figure 1.8).

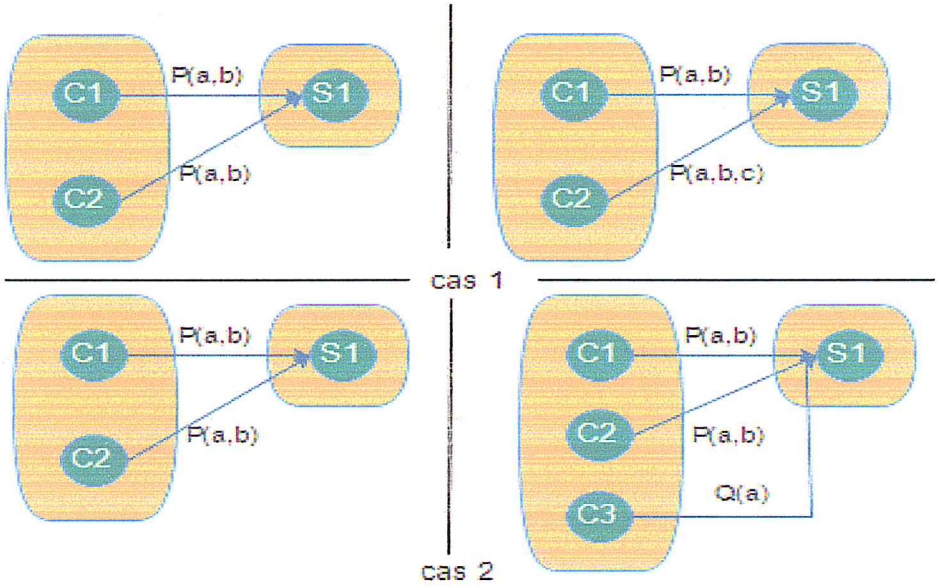


Figure 1. 8 Exemple de changement d'interface.

❖ **Dynamique de géométrie** : elle est appelée aussi dynamique de localisation ou encore de migration. La dynamique de géométrie modifie la distribution géographique d'une application. En effet, ce type de dynamique altère uniquement l'emplacement des composants.

Ainsi, un composant peut changer de localisation en migrant d'un site vers un autre. Comme le montre la figure 1.9, le composant C4 est déplacé du site 2 vers le site 3.

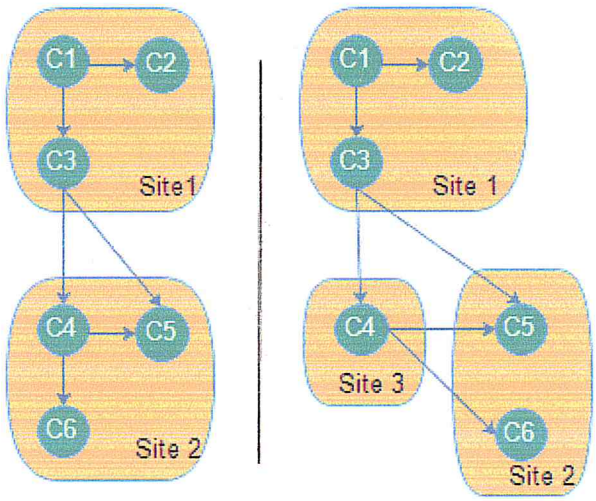


Figure 1. 9 Exemple de changement géométrique.

❖ **Dynamique de structure** : elle modifie la topologie de l'application. C'est-`a-dire dans ce type de dynamique on s'intéresse uniquement au changement de la structure de l'application en termes de composants et de connexions. Cette dynamique se réalise à l'aide de quatre opérations de base [12] : ajout d'un composant, suppression d'un composant, ajout d'une

connexion et suppression d'une connexion. Comme le montre la figure 1.10, La connexion T1 est redirigée vers le composant S1.

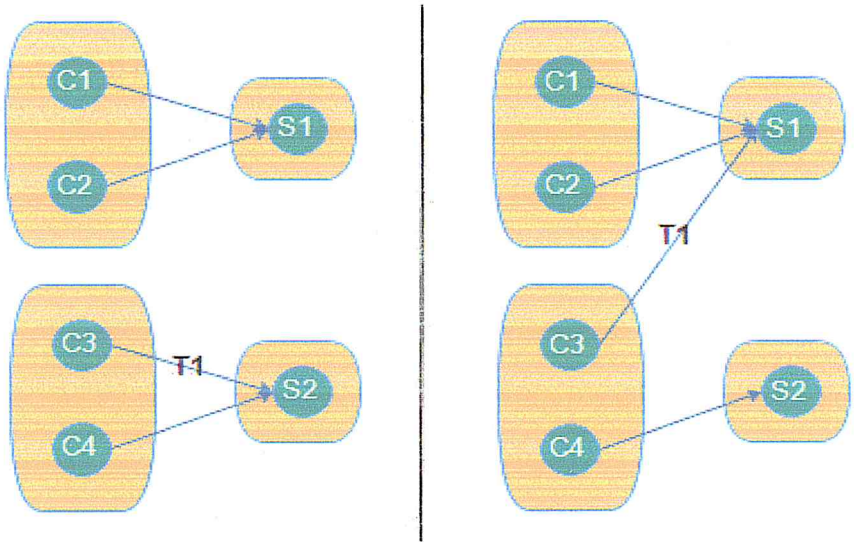


Figure 1. 10 Exemple de changement de structure.

8 Les langages de description des Architectures logicielles dynamiques

La plupart des ADLs proposés se sont limités à une description statique des architectures logicielles. D'autres sont allés plus loin, en abordant la problématique des architectures dynamiques. Parmi les plus représentatifs on peut citer :

8.1 Dynamic Wright

Dynamic Wright [13] prend en charge la description de l'architecture à la fois point de vue structurel (statique) et comportemental (dynamique). Sa description structurelle contient les éléments suivants : composant, interface du composant nommé port, connecteur et interface de connecteur (rôle). En ce qui concerne la spécification de dynamisme, le comportement du système est spécifié séparément dans un configurateur. Le configurateur est chargé de reconfigurer le flux de travail de l'architecture en utilisant des instructions de rattachement et de détachement.

8.2 Plastik

Plastik [14] a les éléments structurels suivants : composant, connecteur et le port. En ce qui concerne les éléments dynamiques pour décrire le comportement (une configuration spécifique), l'expression "on condition do operations" est utilisée pour basculer entre différents choix au moment d'exécution. Pour remplacer une instance de composant lors de l'exécution, les instructions de détachement et de rattachement sont utilisées dans la partie des opérations

afin de dissocier et de lier respectivement les composants et de remplacer ainsi une instance de composant lors de l'exécution.

8.3 Koala

Koala [15] est un exemple de modèle de composant, où toutes les reconfigurations au moment d'exécution sont prédéfinies au moment du design-time. Le dynamisme de ce langage est limité à la "commutation" entre composants selon les règles affinées pour lier le composant sélectionné à l'exécution.

8.4 Darwin

Darwin [16] a été développé au "Distributed Software Engineering Group" de l'Imperial Collège de Londres. Il propose un modèle de composant pour la construction d'applications distribuées. Sa particularité est de permettre la spécification d'une partie de la dynamique d'une application en termes de schéma de création de composants logiciels avant, pendant ou après l'exécution d'une application.

8.5 ADL Fractal

Le modèle de composant Fractal réalisé dans le cadre du consortium *ObjectWeb* par France Telecom R&D et par l'INRIA. Fractal vise à autoriser une définition, une configuration et une reconfiguration dynamique d'une architecture à base de composants, ainsi qu'une séparation claire des préoccupations fonctionnelles et non-fonctionnelles [17].

8.6 SafArchie

SafArchie (*Safe Architecture*) s'appuie sur une description étendue, à la fois structurelle et comportementale, des interfaces des composants afin de permettre une analyse de l'assemblage. Il est basé sur deux modèles de composant abstraits permettant de spécifier une architecture logicielle typée. Ces deux modèles nommés respectivement **modèle de type** et **modèle logique** permettent de séparer la spécification des invariants du système de sa dynamique. Dans SafArchie, les cinq éléments qui composent une architecture logicielle sont le composant primitif, le composite, le port, l'opération et la liaison [18].

8.7 Soadl

Soadl [19] est un langage de description d'architecture orientée service qui est utilisé pour modéliser l'architecture orientée services dans un niveau abstrait. Techniquement, SOADL adopte la notation XML, et est donc indépendant de la plateforme et des technologies. Il spécifie l'architecture en termes de services, d'interfaces, de comportement, de sémantique et de propriétés de qualité. Il prend également en charge la composition de services basée sur

l'architecture. En observant la syntaxe du pseudo-schéma de SOADL, nous pouvons distinguer quatre parties principales : **Port**, **Le comportement, partie subArchitecture** (comprend trois parties : La partie dépendante, La partie configurateur, La partie contrainte), **La partie Propriétés**.

Cependant, dans SOADL, la reconfiguration dynamique des services ne traite que de la substitution des instances de service en cas d'indisponibilité d'un service principal. Les services de substitution sont définis statiquement au moment du design dans la partie configurateur de l'élément SubArchitecture.

8.8 π-ADL for WS-Composition

π-ADL pour WS-Composition [20] est une composition ADL pour service Web (WS) orientée service qui a les mêmes racines que π-ADL et qui repose fortement sur la notation visuelle de BPMN. Il décrit formellement des architectures dynamiques orientées services à partir de points de vue structurels et comportementaux. π-ADL pour WS-Composition est considéré comme ADL dynamique car certains services tiers peuvent être découverts et liés au service broker lors de l'exécution alors que d'autres services sont déjà liés au moment du design. La définition de l'architecture est divisée en deux parties : **Comportement**, **Définition de la structure**.

8.9 Rapide

Rapide [21] est un langage orienté objet, concurrent, basé sur les événements, spécialement conçu pour les architectures de prototypage de systèmes distribués. Rapide permet la simulation et l'analyse comportementale des architectures de tels systèmes à un stade précoce du processus de développement du système.

Le tableau ci-dessous spécifie si les ADLs étudiés ont un outil de spécification graphique ou pas.

ADLs Dynamiques	Editeur Graphique
Dynamic Wright	Non
Plastik	Non
Koala	Non
DARWIN	Non
ADL Fractal	Non
SafArchie	Non
Soadl	Non
Π -ADL	Non
Rapide	Non

Tableau 1. 1Classification d'ADLs étudiés en termes d'éditeurs graphiques.

D'après cette classification on a conclure que les éditeurs d'ADLs mentionné précédemment ne supporte que la partie statique.

9 Conclusion

Nous avons présenté, dans ce chapitre, le domaine auquel s'intègre notre problématique à savoir le domaine d'architecture logicielle et le plus précisément les architectures logiciels dynamiques. Nous avons constaté que les architectures dynamiques présentent des difficultés supplémentaires dans leur spécification surtout que les ADLs dynamiques proposées dans la littérature ne sont pas simples à utiliser même pour les spécialistes du domaine. De plus, le dynamisme de ces langages est limité parfois qu'aux substitutions des composants et la spécification du dynamisme est généralement définis statiquement au moment du design. Pour remédier à ces problèmes, notre objectif dans ce mémoire est de proposer une aide à ces spécialistes pour décrire les architectures dynamiques. Pour décrire notre approche nous devront tout d'abord proposer un modèle de description.

Le chapitre qui suit sera consacré pout présenter l'ingénierie dirigée par les modèles, sur laquelle on s'est basé pour décrire notre modèle de description.

Chapitre 2 :

La méta modélisation

1 Introduction

Puisque notre objectif est de proposer un méta-modèle pour la spécification des architectures logicielles dynamiques, pour cela nous allons aborder dans ce chapitre la méta-modélisation, ainsi que les principaux concepts de cette dernière.

Dans ce chapitre nous commençons par introduire les notions fondamentales de la méta-modélisation, par la suite nous présenterons les différents concepts suivis durant notre projet pour la création d'un méta-modèle selon la norme MOF1.4.

2 L'ingénierie dirigée par les modèles

L'ingénierie dirigée par les modèles IDM (ou aussi Model Driven Engineering MDE) est une pratique d'ingénierie des systèmes utilisant les capacités des technologies informatiques pour décrire au travers de modèles, concepts, et langages, à la fois le problème posé (besoin) et sa solution. Cette pratique est construite autour de deux notions fondamentales que sont les modèles et les transformations. L'objectif principal est que tout processus de production peut être vu comme un ensemble de modèles reliés par des transformations. En effet, un modèle est une représentation abstraite de la réalité. Il sert à représenter de façon schématique et simplifiée certains concepts d'un logiciel de façon à mieux en comprendre le fonctionnement et l'architecture. Pour comprendre ce que contient un modèle et les informations qu'il représente, il faut auparavant s'être entendu sur la définition de son contenu. C'est ce travail que l'on effectue lorsque l'on parle de méta-modélisation. Un méta-modèle est donc le modèle d'un modèle, plus précisément un méta-modèle décrit les relations et les concepts pour construire un modèle [23].

2.1 L'architecture dirigée par les modèles MDA

Après la technologie procédurale, la technologie objet et la technologie des composants, l'approche MDA (Model Driven Architecture) est un processus de l'ingénierie dirigée par les modèles IDM (ou aussi MDE Model Driven Engineering). Proposée par l'OMG (Object Management Group) en 2000, l'approche MDA est basée sur la séparation des préoccupations. Elle permet prendre en compte, séparément, aspect métier et aspect technique d'un logiciel, grâce à la modélisation. L'implémentation d'un logiciel est obtenue par génération automatique à partir des modèles d'un logiciel. Les modèles ne sont plus seulement un élément visuel ou de communication, mais sont, dans l'approche MDA, un élément productif et le pivot du processus MDA [23].

2.2 Hiérarchie Modélisation à 4 Niveaux de MDA

L'OMG (Object Management Groupe) définit 4 niveaux de modélisation comme indiqué sur la table ci-dessous [24] :

Les niveaux des méta	Les normes OMG
M3 – Niveau méta-méta-modèle	MOF
M2 – Niveau méta-modèle	UML, CWM, SPEM, ..etc.
M1 – Niveau modèle	UML
M0 – Niveau instance	Informations réelles (Objets)

Tableau 2. 1 Hiérarchie selon une méta-modélisation par objets [24].

- M0 : système réel, système modélisé.
- M1 : modèle du système réel défini dans un certain langage.
- M2 : méta-modèle définissant ce langage.
- M3 : méta-méta-modèle définissant le méta-modèle.
  - Le niveau M3 est le MOF (Meta-Object Facility).
  - Dernier niveau, il est méta-circulaire : il peut se définir lui-même.

3 La méta-modélisation

3.1 Modèle

Un modèle est un langage artificiel qui peut être utilisé pour exprimer l'information des systèmes dans une structure qui est définie par un ensemble cohérent de règles. Les règles sont utilisées pour interpréter la signification des composants dans la structure. Le modèle peut être visuel et/ou textuel.

Les modèles visuels se basent sur des diagrammes, des symboles qui représentent le nom des concepts, des lignes qui relient ces symboles et qui représentent des relations ainsi que diverses autres annotations graphiques pour représenter des contraintes. Les modèles textuels utilisent généralement des mots-clés standardisés accompagnés par des paramètres afin que les expressions soient interprétables.

3.2 Méta-modèle

Selon MOF [25], un méta-modèle définit la structure que doit avoir tout modèle conforme à ce méta-modèle. Autrement dit, tout modèle doit respecter la structure définie par son méta-modèle. Par exemple, le méta-modèle UML définit que les modèles UML contiennent des packages, leurs packages des classes, leurs classes des attributs et des opérations, etc. La figure 2.1 illustre la relation entre un méta-modèle et l'ensemble des modèles qu'il structure. Les méta-

modèles fournissent la définition des entités d'un modèle, ainsi que les propriétés de leurs connexions et de leurs règles de cohérence. MOF les représente sous forme de diagrammes de classes [26].

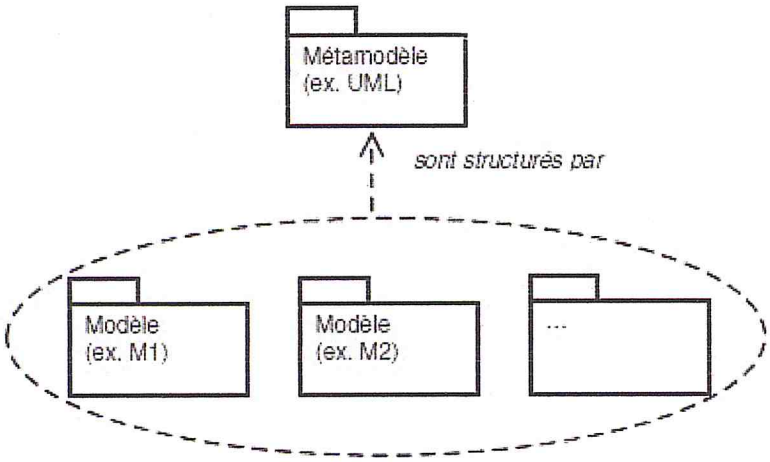


Figure 2. 1 Relation entre modèles et Méta-modèles [26].

Rappelons que les diagrammes de classes permettent de représenter les notions d'un domaine et leurs propriétés, que ces notions ou entités soient organisées ou non sous forme d'objets. Cette utilisation des diagrammes de classes au double avantage de permettre de définir très précisément les méta-modèles et de les rendre eux aussi pérennes et productifs [26]. Un méta-modèle est donc une sorte de diagramme de classes qui définit la structure d'un ensemble de modèles. La section suivante de ce chapitre donne un exemple de méta-modèle [26].

3.3 Exemples de méta-modèles (Méta-modèle de diagramme de classes)

Afin d'illustrer l'utilité des méta-modèles et leurs élaborations conceptuelles, nous allons développer un exemple d'une version allégée d'UML des diagrammes de classes simplifiés (package, classes, attributs). L'information qui nous intéresse est la suivante :

« Un diagramme de classes contient des **packages.** Un package à un nom et contient des **classes.** Un package peut importer un autre package. Une classe a un nom et peut contenir des **attributs.** Une classe peut aussi hériter d'une autre classe. Un attribut a un nom et une visibilité qui peut être soit public soit privât. Un attribut a un **type** qui peut être soit un **type de base (string, integer, boolean),** soit une classe du diagramme. » La figure 2.2 illustre le méta-modèle des diagrammes de classes. Il est composé de huit classes. La classe package contient un attribut nommé nom, dont le type est une chaîne de caractères. La classe class contient un attribut

nommé nom, dont le type est une chaîne de caractères. La classe class hérite de la classe type. La classe attribute contient un attribut nommé nom, dont le type est une chaîne de caractères, et un attribut nommé visibilité, dont le type est une énumération (public ou private). Les classes string, integer et boolean héritent de la classe basicType, qui hérite de la classe type. Il existe une association nommée import, qui a pour source et pour cible la classe package, ainsi qu'une association nommée super, qui a pour source et pour cible la classe class, une association nommée type, entre les classes attribute et type, et deux associations d'agrégation entre les classes package et class et entre les classes class et attribute. Ce méta-modèle définit que les modèles conformes ne peuvent contenir que des packages qui contiennent des classes contenant des attributs. Package, classes et attributs ont des noms. Les packages peuvent importer d'autres packages et les classes hériter d'autres classes, tandis que le type d'un attribut peut être soit un type de base, soit une classe. Ce méta-modèle définit donc bien la structure des diagrammes de classes telle que nous l'avons énoncée en langage naturel précédemment. Ces deux exemples nous ont permis de démystifier les méta-modèles. Nous avons vu qu'un méta-modèle était une sorte de diagramme de classes permettant de représenter les concepts fondamentaux d'un langage de modélisation sous forme de classes et de représenter les relations existantes entre ces concepts sous forme d'associations [26].

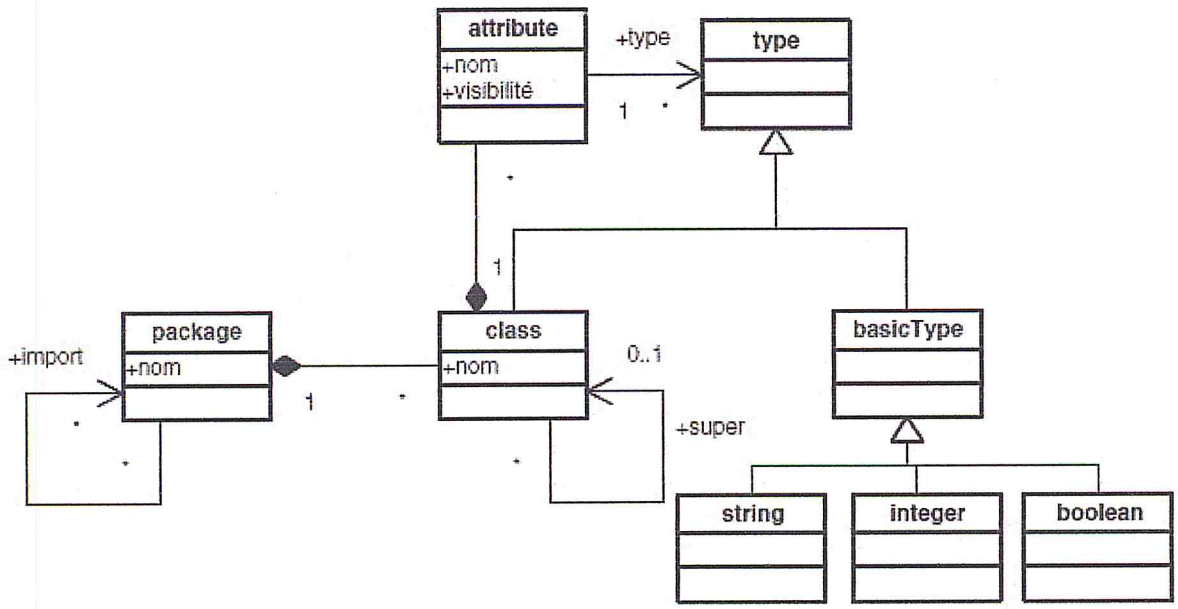


Figure 2. 2 Méta-modèle des diagrammes de classes [26].

4 Création de méta-modèle selon MOF 1.4

Un méta-modèle définit la structure d'un ensemble de modèles. Cette structuration est semblable à la structuration orientée objet. Les méta-modèles MOF1.4 sont donc définis sous forme de classes. Les modèles conformes aux méta-modèles sont considérés comme des instances de ces classes. Afin de discerner les classes constituantes d'un méta-modèle des autres classes, telles que les classes Java, MOF1.4 propose d'utiliser le terme méta-classe. Un méta-modèle est ainsi constitué d'un ensemble de méta-classes. De même, afin de discerner les objets instances des méta-classes des autres objets, MOF1.4 propose d'utiliser le terme méta-objet. Ainsi, un modèle est constitué d'un ensemble de méta-objets instances de méta-classes [26].

Une méta-classe a un nom et contient des attributs et des opérations, aussi appelés méta-attributs et méta-opérations. Un méta-attribut représente une propriété d'un élément du modèle. Une méta-opération représente un traitement applicable à un élément du modèle. Pour typer les attributs et les paramètres des opérations, MOF1.4 propose le concept de type de donnée (dataType). Un type de donnée permet de spécifier un type qui n'est pas un type d'objet. Les types tels que les booléens, les chaînes de caractères, les tableaux et les structures sont des types de données. Pour ce qui concerne l'expression des relations entre méta-classes, MOF1.4 propose le concept de méta-association. Une méta-association est une association binaire entre deux méta-classes. Une méta-association a un nom, et chacune de ses extrémités peut porter un nom de rôle et une multiplicité. Pour grouper entre eux les différents éléments d'un méta-modèle, MOF1.4 propose le concept de package. Un package, aussi appelé méta-package, est un espace de nommage servant à identifier par des noms les différents éléments qui le constituent. Pour résumer, on peut dire que les concepts de base de MOF1.4 sont les suivants (en appellation anglaise) :

- **Class :** Une méta-classe permet de définir la structure de méta-objets. Un ensemble de méta-objets constitue un modèle. Une méta-classe contient des méta-attributs et des méta-opérations.
- **Data Type :** Un type de donnée permet de spécifier le type non-objet d'un méta-attribut ou d'un paramètre d'une méta-opération.
- **Association :** Une méta-association permet de spécifier une relation binaire entre deux métas classes.
- **Package :** Un méta-package permet de regrouper sous un même espace de nommage différents éléments d'un méta-modèle.

4.1 Méta-Classe (class)

Comme expliqué précédemment, un méta-classe sert à décrire la structure d'un ensemble de méta-objets. Un méta-classe a les propriétés suivantes [26] :

- Possède un nom.
- Peut hériter d'une ou de plusieurs autres méta-classes. L'héritage signifie que la sous-classe possède tous les méta-attributs et toutes les méta-opérations de la superclasse. Si un méta-classe hérite de plusieurs méta-classes, il est impératif que ces méta-classes n'aient pas de méta-attributs ni de méta-opérations de mêmes noms.
- Peut-être abstraite. Si tel est le cas, aucun méta-objet ne peut être une instance directe de ce méta-classe.
- Peut être considérée comme feuille (leaf) ou racine (root) d'un arbre d'héritage. Si le méta-classe est feuille, cela signifie qu'aucun autre méta-classe ne peut en hériter. Si le méta-classe est racine, elle ne peut hériter d'un autre méta-classe.
  - **Méta-attribut *(attribute)* :** Un méta-classe peut posséder zéro ou plusieurs méta-attributs. Un méta-attribut a les propriétés suivantes [26] :
    - Possède un nom.
    - Possède une portée (scope). Peut-être de niveau méta-objet ou méta-classe. Si laportée est de niveau méta-objet (instance_level), cela signifie que le méta-objet porte la valeur du méta-attribut. Si la portée est de niveau méta-classe(classifier_level), cela signifie que le méta-classe porte la valeur du méta-attribut pour tous les méta-objets instances.
    - Possède un type. Peut-être un méta-classe ou un type de donnée (voir la section suivante sur les types de données).
    - Peut-être ou non modifiable (isChangeable). S'il est non modifiable, cela signifie en quelque sorte que le méta-attribut a une valeur constante.
    - Peut être considéré comme dérivé (isDerived). Si le méta-attribut est dérivé, cela signifie que sa valeur peut être calculée, par exemple, grâce aux valeurs d'autres méta-attributs du méta-classe.
    - Possède une multiplicité (multiplicity). La multiplicité est spécifiée par trois informations. La première contient les bornes maximale (upper) et minimale (lower)de la multiplicité. Par exemple, un méta-attribut ayant 0 comme borne minimale et 1 comme borne maximale est un méta-attribut optionnel, tandis qu'un attribut ayant 1 comme borne minimale et l'infini (représenté par le caractère *)

comme borne maximale est un méta-attribut obligatoire pouvant avoir une infinité de valeurs. Les deux autres informations ne concernent que les méta-attributs ayant une borne maximale supérieure à 1. L'information « ordonné » (is_ordered) permet de spécifier que l'ensemble des valeurs du méta-attribut est ordonné et l'information « unique » (is_unique) que l'ensemble des valeurs du méta-attribut ne doit pas contenir de doublon.

- **Méta-opération (operation) :** Un méta-classe peut contenir zéro ou plusieurs méta-opérations. Une méta-opération a les propriétés suivantes [26] :
  - Possède un nom.
  - Possède une portée. Peut-être de niveau méta-objet ou méta-classe. Si la portée est de niveau méta-objet, il est possible de demander à un méta-objet de réaliser le méta-opération. Si la portée est de niveau méta-classe, il est possible de demander directement au méta-classe de réaliser la méta-opération.
  - Peut contenir zéro ou plusieurs métas paramètres d'entrée. Un méta-paramètre, comme un méta-attribut, a un nom, un type et une multiplicité. De plus, il a une direction (direction). Cette direction est soit monodirectionnelle, soit bidirectionnelle. Si elle est monodirectionnelle, elle peut être soit de l'appelant vers l'appelé (in), l'appelant donnant la valeur au méta-paramètre, soit de l'appelé vers l'appelant (out), l'appelé donnant la valeur au méta-paramètre. Si elle est bidirectionnelle (in_out), c'est indifféremment l'appelant ou l'appelé qui donne la valeur au méta-paramètre.
  - Peut optionnellement définir un type de retour. Ce dernier peut être soit un méta-classe, soit un type de donnée. Si une méta-opération n'a pas de type de retour, on considère qu'elle ne retourne rien et non pas qu'elle retourne un ensemble vide (void).
  - Peut jeter une ou plusieurs exceptions. Une exception a un nom et peut contenir zéro ou plusieurs attributs permettant de la renseigner.

4.2 Les Types de Données (data Type)

Les types de données permettent de définir des types non-objet. MOF1.4 permet de définir deux sortes de types de données : les types primitifs et les types construits. Concernant les types primitifs, MOF1.4 définit les types suivants [26] :

- Boolean : la valeur est soit vrai (true), soit faux (false).
- Integer : la valeur est un entier compris-en entre – 2^31 et + 2^31–1.

- Long : la valeur est un entier compris entre – 2^61 et + 2^61 – 1.
- Float : la valeur est définie par la norme ANSI/IEEE 754-1985.
- Double : la valeur est définie par la norme ANSI/IEEE 754-1985.
- String : la valeur est une chaîne de caractères encodée en 16 bits.

Concernant les types construits, MOF1.4 propose les énumérations, les alias et les structures. Lorsqu'on élabore un méta-modèle, on peut soit utiliser les types primitifs existants, soit construire ses propres types grâce aux constructions proposées par MOF1.4.

4.3 Méta-association (association)

Une méta-association permet de définir une relation entre deux méta-classes. En fait, une méta-association permet de définir la structure des liens entre les méta-objets instances des méta-classes reliées par la méta-association. Une méta-association a les propriétés suivantes [26]:

- Possède un nom.
- Contient obligatoirement deux extrémités (associationEnd). Ce sont les extrémités d'une méta-association qui sont reliées aux méta-classes
  - **Extrémité d'association (associationEnd) :** Une extrémité de méta-association a les propriétés suivantes :
    - Possède un type. Le type de l'extrémité identifie la méta-classe reliée par la méta-association.
    - Possède un nom. Correspond au nom du rôle que joue la méta-classe reliée.
    - Possède une multiplicité (même multiplicité que les méta-attributs). Cette multiplicité permet de spécifier le nombre d'instances de la méta-classe identifiée par le type de l'extrémité pouvant être liées à exactement une instance de la méta-classe de l'autre extrémité de l'association. La méta-association cas relie les méta-classes acteur et cas d'utilisation. Une extrémité de cette méta-association a pour type cas d'utilisation. La multiplicité de cette extrémité est *. Cela signifie que plusieurs instances de cas d'utilisation peuvent être liées à une instance d'acteur (rappelons que * signifie zéro ou plusieurs). Pour savoir combien d'instances d'acteur peuvent être liées à une instance de cas d'utilisation, il faut regarder l'autre extrémité de l'association (de nouveau *). Une extrémité de méta-association dispose d'une spécification d'agrégation. Celle-ci peut être soit composite (composite), soit non existante (non-aggregate). Une spécification d'agrégation composite entraîne que les méta-objets (instances de la méta-classe identifiée par le

type de l'extrémité) contiennent les méta-objets instances de la méta-classe identifiée par le type de l'autre extrémité (extrémité opposée). Concrètement, cela signifie que si le méta-objet est détruit, tous les méta-objets qu'il contient sont aussi détruits. L'extrémité de la méta-association qui relie la méta-classe système à la méta-classe cas d'utilisation et qui a pour type la méta-classe système a une spécification d'agrégation composite (illustrée par le losange noir). Dans cet exemple, cela signifie que les méta-objets instances de la méta-classe système contiennent les méta-objets de la méta-classe cas d'utilisation. Si un méta-objet instance de la méta-classe système est détruit, tous les méta-objets instances de la méta-classe cas d'utilisation qu'il contient doivent être détruits. Les contraintes suivantes doivent être respectées lorsqu'il existe des extrémités de méta-association ayant des spécifications d'agrégation composite :

- Il ne faut pas que les deux extrémités d'une méta-association aient une spécification d'agréation composite. Cela empêche que deux méta-objets soient composés l'un de l'autre.
- Il ne faut pas qu'une méta-classe soit reliée par deux méta-associations dont les extrémités opposées (celles dont les types sont les autres méta-classes) aient des politiques d'agrégation composite. Cela empêche qu'un méta-objet soit inclus dans deux méta-objets instances de deux méta-classes différentes.
- Il faut que la multiplicité de l'extrémité ayant une spécification d'agrégation composite ait 1 comme maximum. Cela empêche qu'un méta-objet soit inclus dans deux méta-objets instances de la même méta-classe.

Une extrémité de méta-association peut être ou non modifiable (isChangeable). Si l'extrémité est changeable, cela signifie qu'il est possible de modifier n'importe quand les liens des méta-objets correspondant aux extrémités des méta-associations.

Une extrémité de méta-association peut être ou non navigable (isNavigable). Si l'extrémité est navigable, cela signifie qu'il est possible d'atteindre les valeurs des méta-attributs du méta-objet lié et de lui appeler ses méta-opérations. Bien sûr, l'accès n'est possible que pour les méta-objets liés par les extrémités opposées de la méta-association. Dans l'exemple du méta-modèle des diagrammes de cas d'utilisation, la méta-association cas à son extrémité dont le type est cas d'utilisation, qui est navigable (symbolisé par une flèche). Cela signifie que les méta-objets instances de la méta-classe acteur peuvent atteindre les valeurs des méta-attributs et appeler les

méta-opérations des méta-objets instances de la méta-classe cas d'utilisation avec lesquels ils sont liés.

- **Référence (reference)** : Afin de faciliter les navigations entre méta-objets via les méta-associations dont les extrémités sont navigables, MOF1.4 propose le concept de référence (reference). Une référence est une sorte de méta-attribut permettant essentiellement de naviguer via les méta-associations. Comme un méta-attribut, une référence à un nom, un type, une multiplicité, etc. La différence avec les méta-attributs réside dans le fait que le type de la référence ne peut être qu'une méta-classe reliée par une méta-association avec la méta-classe qui contient la référence. De plus, l'extrémité qui pointe vers l'autre méta-classe (celle qui ne contient pas la référence) doit être navigable. La multiplicité de la référence doit être la même que la multiplicité de cette extrémité. La référence est en fait une sorte d'alias pour atteindre les méta-objets liés.

4.4 Package

MOF1.4 propose le concept de *package,* qui permet de grouper différents éléments d'un même méta-modèle. L'objectif est, d'une part, de regrouper les éléments d'un méta-modèle portant sur un même domaine, par exemple, les éléments relatifs aux diagrammes de classes, et, d'autre part, de gérer plus facilement les méta-objets instances d'un ensemble de méta-classes. Il est ainsi possible de stocker dans un même espace mémoire des méta-objets dépendant les uns des autres.

Un package aux propriétés suivantes :

- Possède un nom.
- Contient zéro ou plusieurs méta-classes, méta-associations reliant ces classes et types de données nécessaires.
- Peut contenir zéro ou plusieurs autres packages.
- Lorsqu'un élément est dans un package (méta-classe, méta-association, type de donnée, Package), il dispose d'un nom complet. Ce nom correspond au nom complet du package, suivi du caractère «. », suivi du nom de l'élément. Si, par exemple, la méta-classe C est contenue dans le package B, qui est lui-même contenu dans le package A, le nom complet de la méta-classe est A.B.C.
- Un package peut hériter d'un ou de plusieurs autres packages. Dans ce cas, le souspackage acquiert tous les éléments du superpackage (méta-classes, méta-associations, types de données, packages).

- Un package peut importer un autre package. Le package qui importe l'autre package (le package importé) peut utiliser tous les éléments du package importé.

5 EMF (Eclipse Modeling Framework)

EMF permet de créer deux types de modèles, d'un côté des modèles définissant des concepts, souvent nommé le méta-modèle, et de l'autre des modèles instanciant ces concepts. À titre d'exemple, on peut définir un méta-modèle définissant des concepts tels que « Classe », « Operation » et « Attribut » et ensuite utiliser ce méta-modèle pour définir des modèles contenant par exemple un élément de type « Classe » nommé « Voiture » avec une « Operation » nommée « rouler » et un « Attribut » nommé « consommation ». Tout modèle EMF est une instance d'un modèle EMF avec pour racine commune le modèle Ecore fourni par EMF. EMF permet non seulement de créer un méta-modèle représentant les concepts désirés par l'utilisateur mais il permet ensuite à l'utilisateur de créer des modèles issus de ce méta-modèle et de les manipuler avec un outillage adapté [27].

5.1 Ecore

Tous les modèles créés avec EMF sont liés d'une manière ou d'une autre avec Ecore. Ecore est un modèle EMF définissant les concepts manipulables dans EMF. Ces concepts, toujours préfixés par un "E", sont les suivants : EAttribute, EAnnotation, EClass, EDataType, Enum etc…

Ecore contient aussi de nombreux types de données afin d'initialiser des modèles EMF sans besoin de redéfinir les types primitifs courants (EString, EBoolean, EInt, EFloat, etc.) [27].

Dans notre projet nous utilisons Ecore pour développer notre méta-modèle proposé pour spécifier les architectures logicielles dynamiques.

6 Conclusion

Dans ce chapitre nous avons présenté les concepts fondamentaux de la méta-modélisation qui vont nous servir à proposer notre méta-modèle des architectures logicielles dynamiques.

Dans le chapitre suivant nous allons décrire notre méta-modèle servant à modéliser les architectures logicielles dynamiques.

Chapitre 3 :

Méta-modèle pour la Spécification des Architectures Logicielles Dynamiques

1 Introduction

Après l'étude des différentes techniques utilisées pour décrire les architectures logicielles dynamiques et leur évolution, nous présentons notre modèle baptisée ALD (modèle des Architectures Logicielles Dynamiques) comme un nouveau modèle de spécification orienté vers les architectures logicielles à base de composants. Dans notre approche, nous nous concentrons sur la modélisation de la dynamique de l'architecture logicielle au niveau conceptuel. Nous traitons essentiellement le problème de l'évolution suites un changement dans l'activité soutenue par le système modélisé. Nous associons des opérations d'évolution architecturale à ces situations pour adapter le système à ces changements.

Dans ce chapitre nous commençons en premier lieu par décrire le méta-modèle proposé, qui est divisé en deux parties ainsi que la description graphique pour chaque élément de notre modèle. En dernier lieu, nous présentons un cas d'étude pour illustrer notre approche.

2 Description d'ALD (Architectures Logicielles Dynamiques)

Avant de présenter le modèle d'ALD, nous rappelons notre perception de l'évolution d'une architecture logicielle dynamique. Nous considérons que l'évolution d'une architecture logicielle dynamique se reflète par les différents changements dans sa structure, son comportements et/ou dans celle de ses éléments constitutifs en réponse à des évènements internes ou externes. Un changement dans une architecture est toujours engendré par une ou plusieurs opérations appliquées à cette architecture ou à l'un de ses éléments : ajouter un composant, supprimer un connecteur, modifier une interface, etc. Ces opérations expriment en effet les besoins d'évolution qui doivent être appliqués sur l'architecture.

Pour caractériser ainsi une évolution au sein d'une architecture logicielle et donc pour pouvoir la réifier, il faut identifier, au minimum, l'élément architectural sur lequel porte l'évolution, l'opération à appliquer sur cet élément, les impacts de cette opération sur l'élément architectural considéré et sur les autres éléments concernés de l'architecture.

Le modèle d'évolution d'ALD offre ainsi, au vu de ces caractéristiques précédemment citées, des concepts définis au sein de son méta-modèle permettant à la fois la spécification et la gestion de l'évolution d'une architecture logicielle dynamique ainsi qu'un processus opératoire décrivant les étapes à suivre pour mener une évolution.

2.1 Conception d'ALD

Les systèmes développés évoluent ainsi que leur architecture, nous pouvons avoir besoin, par exemple pour ajouter de nouveaux composants, de modifier les composants existants ou de

de modifier les connexions entre ces composants. Cette évolution doit être identifiée et gérée pour maintenir la cohérence architecturale du système évalué. L'objectif est de permettre au concepteur de modifier la structure et le comportement de l'architecture logicielle en fonction de ses besoins (ajout ou suppression de composant, modification ou substitution du connecteur ...) pour maintenir la cohérence de l'architecture après cette modification. Nous proposons dans cette section notre modèle comme solution pour faire face à cette problématique. Nous nous intéressons plus précisément à l'évolution structurelle et comportementale de l'architecture. Pour cela notre modèle doit :

- Supporter l'évolution statique (au moment de la spécification de l'architecture) ainsi que l'évolution dynamique (au moment de l'exécution de l'application).
- Définir un mécanisme de description et de gestion de l'évolution indépendamment des éléments architecturaux et de leurs langages de description.
- Soutenir la réutilisation de ces mécanismes d'évolution dans plusieurs cas d'évolution.
- Etre ouvert à l'ajout de nouvelles évolutions.

Pour atteindre ces objectifs, notre modèle doit prendre en compte tous les éléments architecturaux proposés par les ADLs à savoir : le composant, le connecteur et les ports.

3 Le méta-modèle ALD proposé

Pour la description des concepts du Méta-modèle ALD nous allons se baser sur une modélisation UML présentée dans le chapitre précèdent. Nous utiliserons alors le formalisme du diagramme de classe de la notation Ecore, plus précisément les notions clés de classe, l'héritage, la composition et les associations, ainsi que la notion de multiplicité sur les relations.

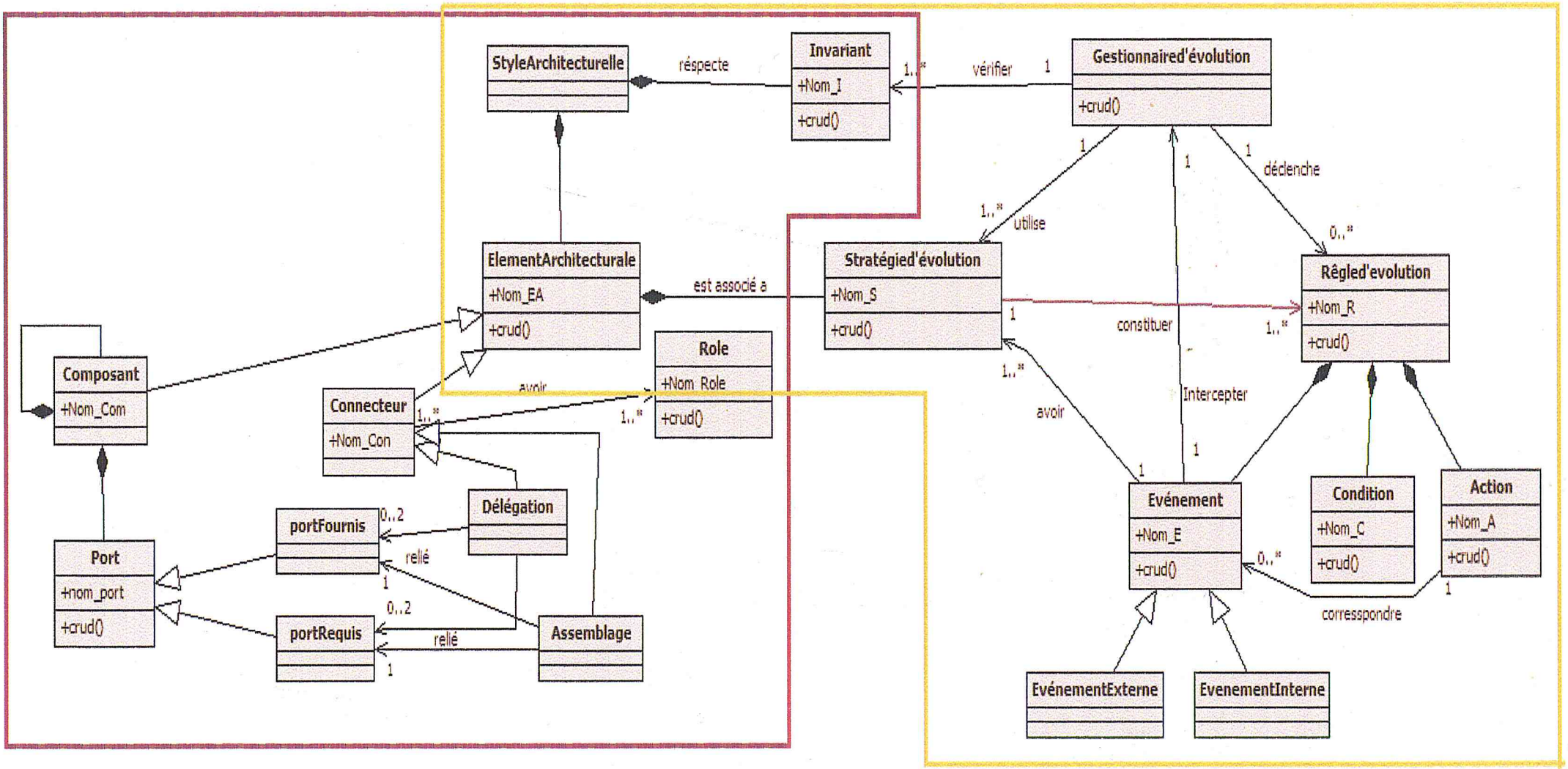


Figure 3. 1 Méta-modèle ALD proposé.

Notre approche, basée sur un profil UML, permet de décrire la dynamique des architectures logicielles. Elle est basée sur deux méta-modèles, comme le montre la figure 3.2 Le premier (cadre rouge) permet de décrire l'architecturale statique (Style Architecturale) d'une architecture logicielle. Il décrit l'ensemble des types de composants qui constituent le système, les types de connexions entre eux, ainsi que les contraintes d'ordre architectural. Le deuxième (cadre vert) permet de décrire la dynamique et l'évolution de l'architecture en termes d'opérations d'évolution que peuvent avoir une architecture tout au long de son cycle de vie.

Remarque : il y a une annexe contient les méta-modèles de l'implémentation en format Ecore, cette dernier nous a permet de implémenter notre méta-modèle dans l'application.

3.1 Partie de style architecture

3.1.1 Méta-modèle de style architectural

La structure du méta-modèle du style architecturalest décrite par la figure 3.3.

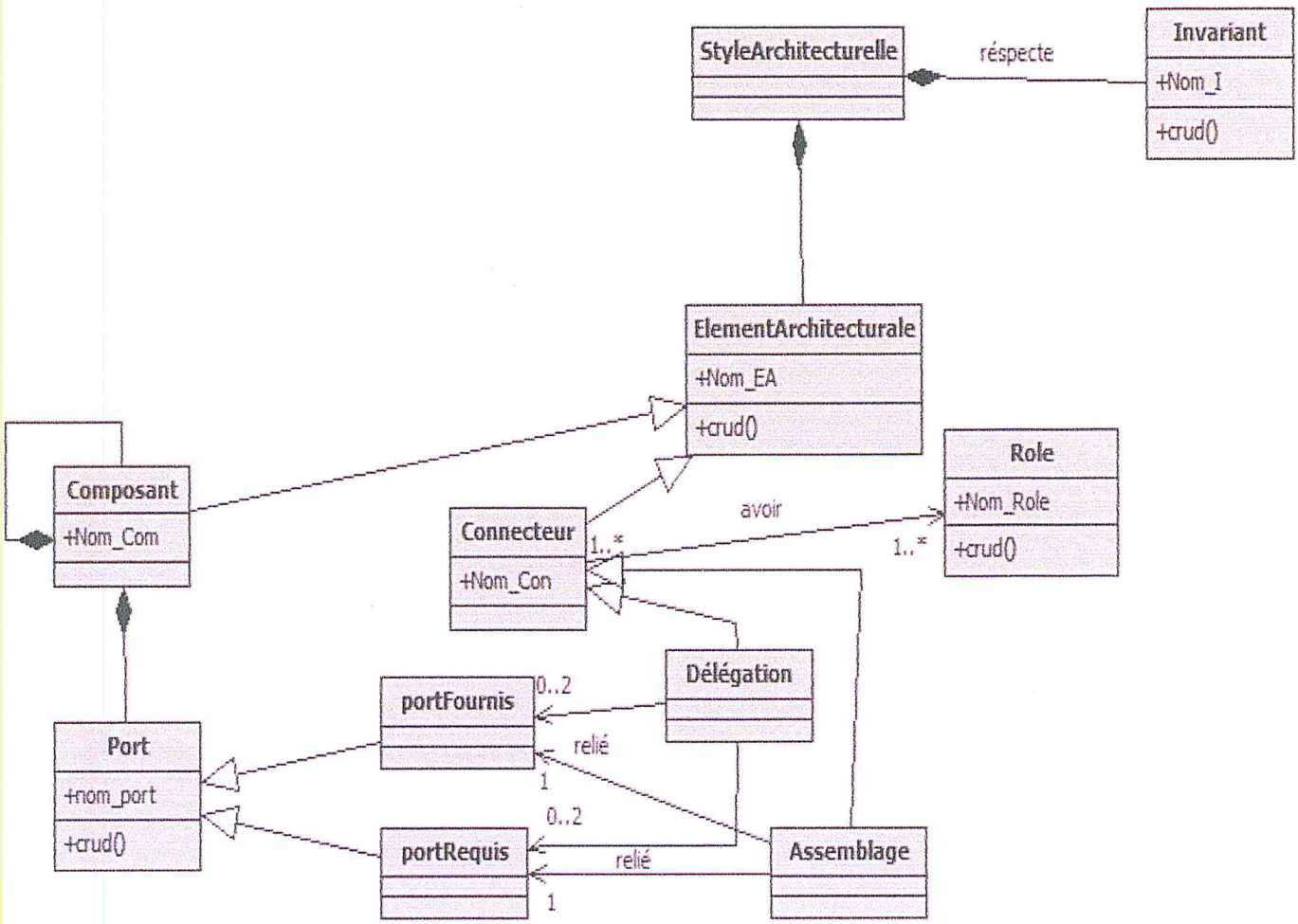


Figure 3. 2 Méta-modèle du Style Architectural proposé.

Le méta-modèle du style Architectural se compose de deux classe Invariant et ElementArchitecturale. La classe Invariant décrivent les contraintes architecturales que le système doit respecter au cours de son évolution {'les Invariants c'est l'ensemble des contraintes structurelles et comportementales sur un ou plusieurs éléments architecturaux. Ces contraintes doivent être respecté durant tout le cycle de vie de l'élément architectural, quelle que soit l'action ou l'opération d'évolution qui affecte sur cet élément architectural.et aussi la modification dans l'architecture logicielle doit garantir la stabilité de chaque invariant pour sauvegarder la cohérence de l'architecture.'}. La classe ElementArchitecturale est composée des classes Composent et Connecteur. Il spécifie tous les types de composants et de connecteurs qui constituent le style architectural d'un système {'un élément architectural permet de regrouper tous les éléments significatifs dans une architecture logicielle. Il représente tous les éléments de l'architecture à évoluer (composant, connecteur ou interface).'}. La classe Composent, composée éventuellement de plusieurs composants, Chaque composant a une ou plusieurs ports fournis et/ou requises exposées à travers les ports. La classe Port représente le point d'interaction pour un composant.

La classe Connecteur définit un lien qui permet la communication entre deux composants ou plus, il existe deux types de connecteurs définis par les classes ConnecteurDelegation et ConnecteurAssemblage. Un ConnecteurDelegation exprime un lien entre deux composants d'une interface requise à une interface requise ou d'une interface fournie à une interface fournie. Un ConnecteurAssemblage exprime un lien entre deux composants d'un port requise à un port fournie. Le rôle d'un connecteur décrit la fonctionnalité que le connecteur devra fournir à son environnement.

3.1.2 Notation graphique de style architectural

Pour décrire le style architectural, nous proposons une nouvelle notation graphique décrite dans le tableau 3.1.

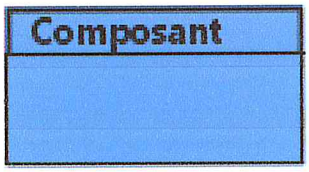
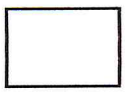
Notation sémantiques	Notation graphiques
Composant	
Port fournis	
Port requis	
Connecteur Assemblage	
Connecteur Délégation	
Invariant	

Tableau 3. 1 Notation graphique du style architectural.

3.2 Partie de gestion d'évolution

3.2.1 Méta-modèle du la gestion d'évolution

La structure du méta-modèle de la gestion d'évolutionest décrite par la figure 3.4.

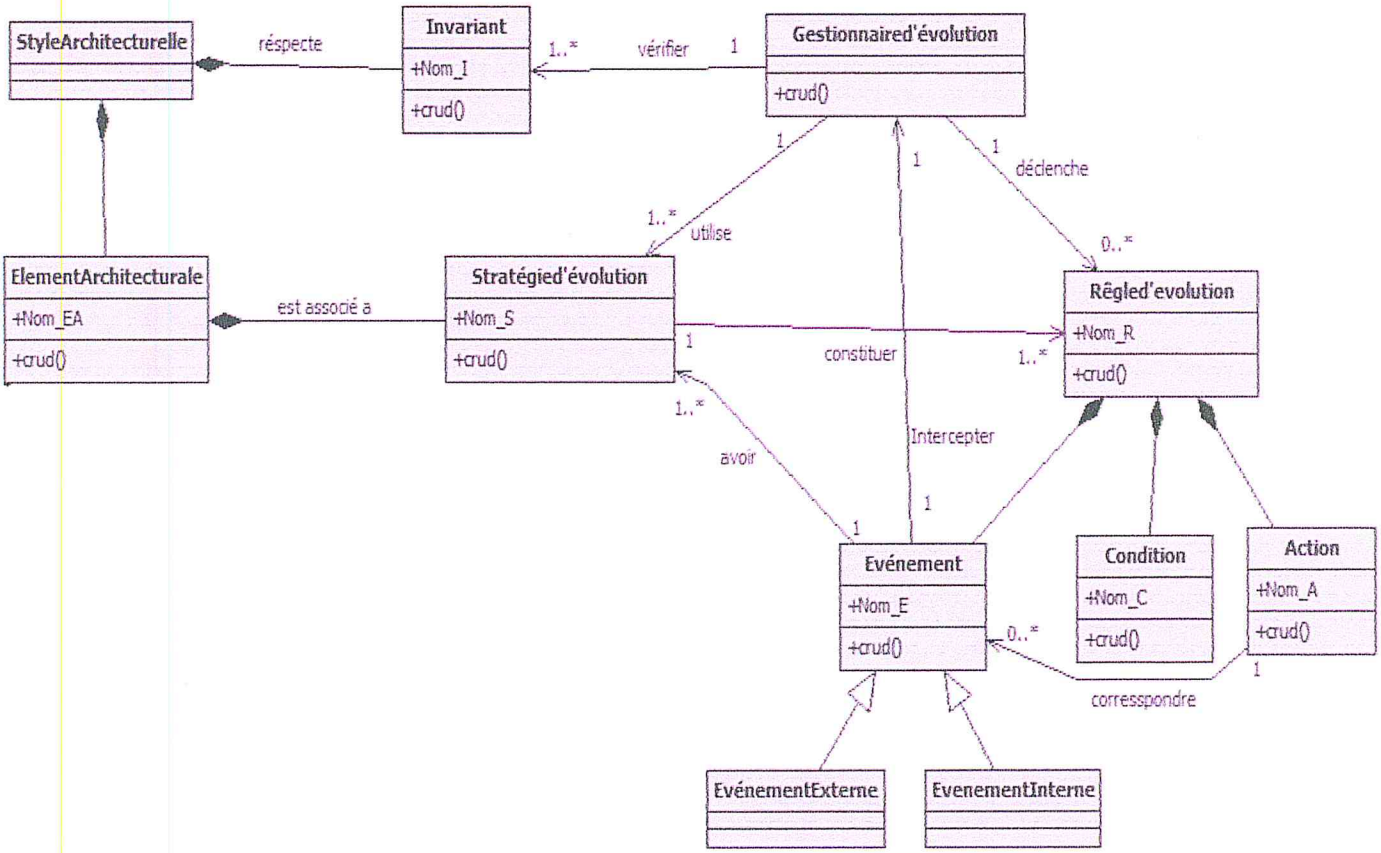


Figure 3. 3 Méta-modèle de la gestion d'évolution.

Le méta-modèle proposé se base sur trois notions (le gestionnaire d'évolution, la stratégie d'évolution et les règles d'évolution).

Le gestionnaire d'évolution est responsable de l'interception des évènements en cas d'une évolution sur un élément architectural donné (composant ajouter, composant supprimer …), ensuite déclenche la stratégie d'évolution à appliquer. Une stratégie d'évolution est constituée d'un ou plusieurs règles d'évolution. Une règle d'évolution est composée d'événement, de condition et d'action (règle ECA). Les règles ECA [28] effectuent automatiquement des actions en réponse à des événements à condition que les conditions énoncées soient respectées.

❖ **Gestionnaire d'évolution**

Tous les concepts précédemment définis ont un rôle descriptif. Le gestionnaire d'évolution quant à lui est chargé de gérer l'évolution d'une architecture logicielle donnée au vu des stratégies d'évolution définies et de leurs règles d'évolution. Son rôle est d'intercepter tout évènement d'évolution interne ou externe émanant du concepteur ou des règles d'évolution ou par les propriétés non fonctionnelles des composants vers l'architecture ou l'un de ses éléments, puis d'identifier la stratégie d'évolution correspondante. Suivant cette stratégie, il déclenche la ou les règles d'évolution associées à l'évènement reçu. Il est chargé ensuite de propager les impacts des règles d'évolution exécutées. Le gestionnaire d'évolution doit vérifier aussi le maintien des invariants. L'ensemble de ces tâches du gestionnaire d'évolution sera détaillé dans le mécanisme opératoire d'ALD décrit dans la section processus d'évolution.

❖ **Stratégie d'évolution**

Une stratégie d'évolution encapsule ce qui permet de décrire et d'appliquer les différentes évolutions possibles sur un élément architectural. Une stratégie d'évolution regroupe les règles permettant de spécifier l'évolution d'un élément architectural.

La stratégie d'évolution contient une ou plusieurs règles d'évolution. Au moins quatre stratégies sont associées à chaque élément architectural :

Ajout, Suppression, Modification et substitution …etc.

- Stratégie d'ajout : inclut toutes les règles décrivant comment ajouter un élément architectural.
- Stratégie de suppression : inclut toutes les règles décrivant la suppression d'un élément architectural.
- Stratégie de modification : inclut toutes les règles décrivant la modification du comportement d'un élément architectural.
- Stratégie de substitution : inclut toutes les règles décrivant la substitution d'un élément architectural.

Nous notons que d'autres stratégies peuvent être construites, si des besoins de spécifier des opérations d'évolution plus complexes sont apparus.

❖ **Règle d'évolution**

Le concept de règle d'évolution permet d'exprimer et de spécifier les évolutions qui peuvent être menées sur une architecture logicielle. Une règle d'évolution permet la

description de l'application d'une opération d'évolution (ajout/suppression /modification/substitution) sur un élément architectural, en spécifiant les conditions nécessaires pour le faire, ainsi que les éventuels impacts qu'elle peut engendrer sur les autres éléments architecturaux. En effet, la propagation des impacts est nécessaire notamment pour sauvegarder la cohérence de l'architecture ayant évolué. Les impacts d'une évolution peuvent concernés les éléments du même niveau ou les éléments des niveaux inférieurs quand ils existent.

Nous avons choisi de modéliser les règles d'évolution par le formalisme ECA (évènement/ Condition/Action). Nous jugeons que ce formalisme est adapté pour permettre à une architecture logicielle d'être réactive, c'est à dire de réagir automatiquement à des évènements d'évolution qu'elle peut recevoir.

En effet, grâce à cette approche événementielle, la règle à déclencher est élue dynamiquement dès la réception de l'évènement qui lui est associé. Le formalisme ECA permet aussi de spécifier le déclenchement automatique de la propagation des impacts d'une évolution. Chaque règle d'évolution d'ALD est ainsi composée :

- **D'un évènement :** Une évolution peut être induite par un événement, le concept d'événement spécifie le signal qui déclenche l'invocation de la règle. Plusieurs événements peuvent se produire lors du déploiement ou lors de l'évolution du logiciel. De nombreux événements ne peuvent pas être connus à l'avance et beaucoup d'autres peuvent dépendre du comportement de certains composants. Les événements peuvent être soit un simple événement tel qu'une 'évolution manuelle du logiciel déclenchée par un utilisateur, soit un événement déclenché par un autre événement. Le gestionnaire d'évolution est responsable de l'interception de tout événement. Généralement, les événements qui se produisent pendant le déploiement du logiciel sont divisés en deux catégories :
  - Événement externe : correspondent aux opérations d'évolution possibles de l'architecture du logicielle : création des composants, modifications du cycle de vie (démarrage ou arrêt du composant), modification des paramètres de configuration, ajout ou suppression de sous-composants et création et destruction de liaisons. Un utilisateur ou un administrateur système initie généralement ces événements. Il peut déclencher aussi par les propriétés non fonctionnelles des éléments architecturales comme par exemple le temps de réponse ou la consommation de la mémoire etc. Un évènement externe peut déclencher un ou plusieurs stratégies d'évolution.

- Événement interne : dépend du comportement des composants en général, il peut correspondre à la réception d'un message, à la réponse réussie d'un message et à un lancement d'exception, et aussi à l'impact généré par les actions effectuées sur les éléments constructifs de l'architecture

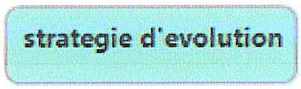
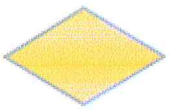
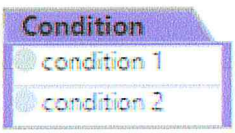
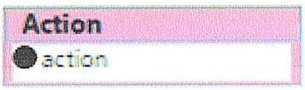
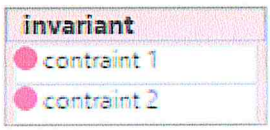
➢ **D'une partie condition :** ce sont des contraintes que doit nécessairement vérifier par le système pour exécuter une règle d'évolution afin de maintenir la cohérence de système. Elle peut correspondre à une seule condition qui porte sur un élément architectural ou plus qu'une condition qui porte sur un ou plusieurs éléments architecturaux.

➢ **D'une partie action :** la partie action décrit les changements appliqués sur l'élément architectural auquel est associée la règle d'évolution, ainsi une action peut déclencher une autre action sur les autres éléments architecturaux. La propagation des impacts d'une évolution est exprimée sous forme d'invocation d'autres règles. Quand l'évènement a eu lieu, et si les conditions sont satisfaites, alors la partie action est exécutée apportant une réponse appropriée à l'évènement. Ainsi, une action dans une règle d'évolution peut correspondre à :

- Un évènement interne (une référence vers une autre règle), dans ce cas il sera redirigé par le gestionnaire d'évolution vers d'autres règles d'évolution. Le but de l'évènement est alors de déclencher d'autres règles d'évolution pour propager les impacts de l'évolution traitée. Par exemple, une règle de suppression d'un port peut déclencher la règle de suppression de l'attachement associé à ce port ;
- Un événement externe (une référence vers une autre stratégie), dans ce cas il sera redirigé par le gestionnaire d'évolution vers d'autres stratégies d'évolution. Le but de l'évènement alors de déclencher d'autres stratégies d'évolution pour propager les impacts de l'évolution traitée. Par exemple, une règle de suppression d'un serveur peut déclencher la stratégie de taux de temps de réponse lent.

3.2.2 Notation graphique de gestion d'évolution

Pour décrire la gestion de l'évolution, nous proposons une nouvelle notation graphique décrite dans le tableau 3.2.

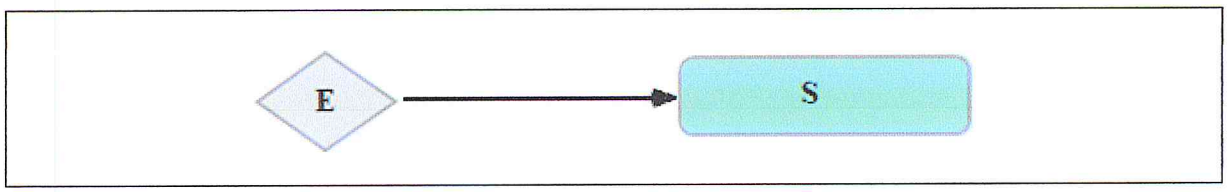
Notation sémantiques	Notation Graphique
Gestionnaire d'évolution	
Stratégie d'évolution	
Règle d'évolution	
Evènement externe	
Evènement interne	
Condition	
Action	
Invariant	
Arc include	
Arc extend	

Arc vérifier	
Arc intercepter	
Arc constituer	
Arc composer action	
Arc composer condition	
Arc composer évènement	

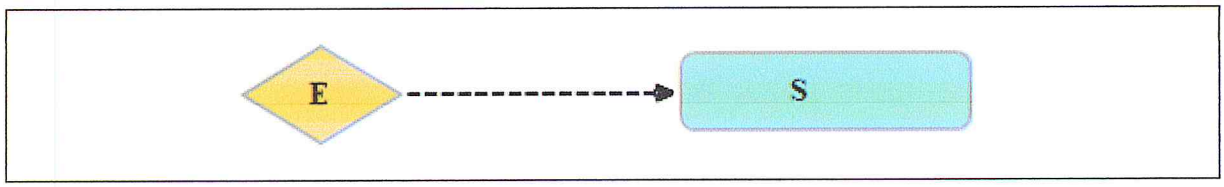
Tableau 3. 2 Notation Graphique du la gestion de l'évolution.

❖ **Description des arcs :**

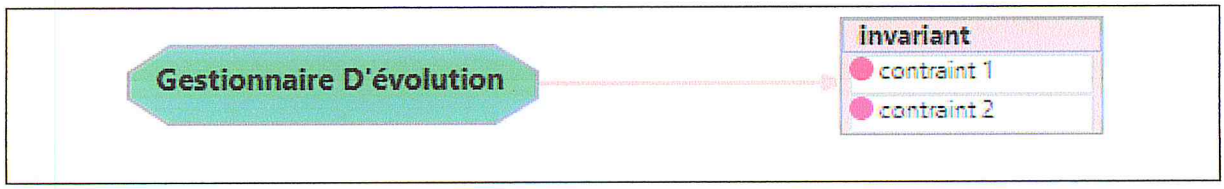
➢ **Arc include :** ce lien décrit que la relation entre les évènements et les stratégies est obligatoire (à l'arrivée d'un évènement **E**, la stratégie **S** doit être exécutée)



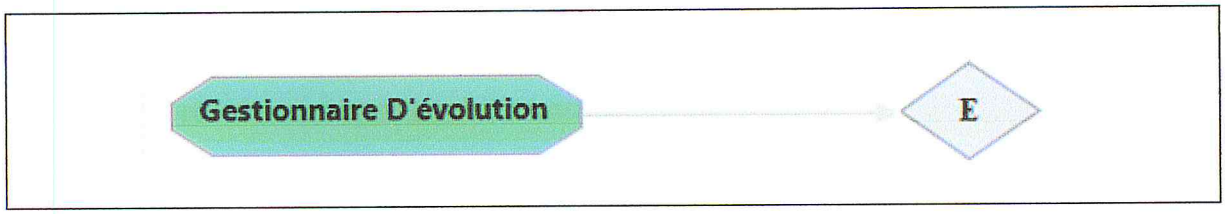
➢ **Arc extend :** ce lien décrit que la relation entre les évènements et les stratégies est optionnelle (à l'arrivée d'un évènement **E**, la stratégie **S** peut être exécutée).



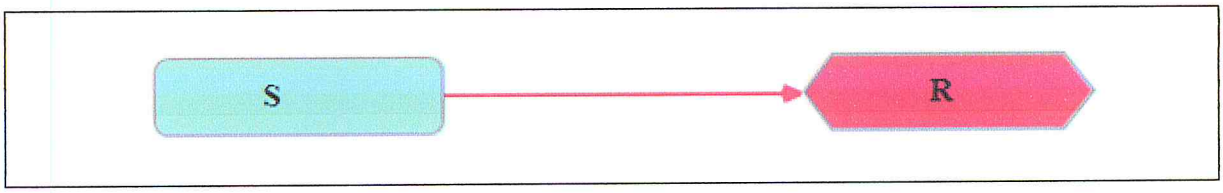
➢ **Arc vérifier :** ce lien décrit que le gestionnaire d'évolution doit vérifier les invariant (le gestionnaire doit vérifier les contraintes dès que l'évolution du ALD est terminer).



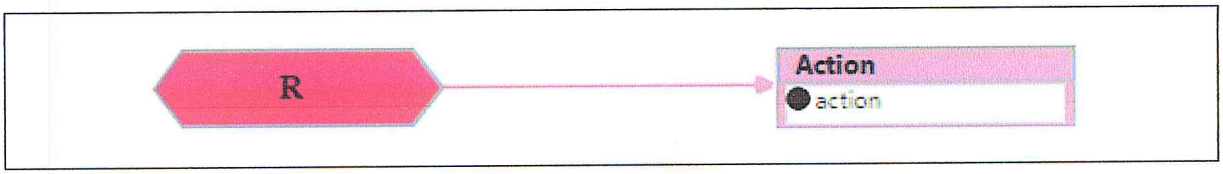
- **Arc intercepter :** ce lien décrit la relation entre le gestionnaire d'évolution et l'événement externe (le gestionnaire intercepte l'événement E)



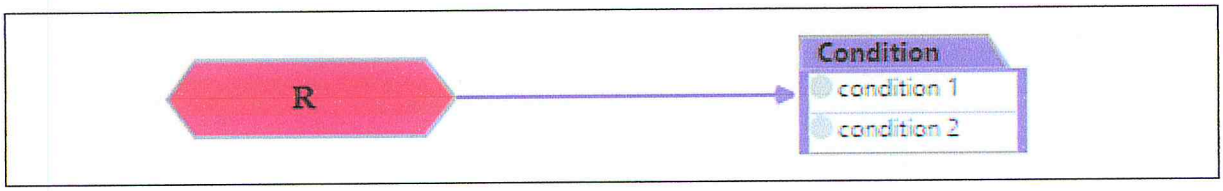
- **Arc constituer :** ce lien décrit la relation entre les stratégies et ses règles d'évolution (une stratégie peut constituer une ou plusieurs règles).



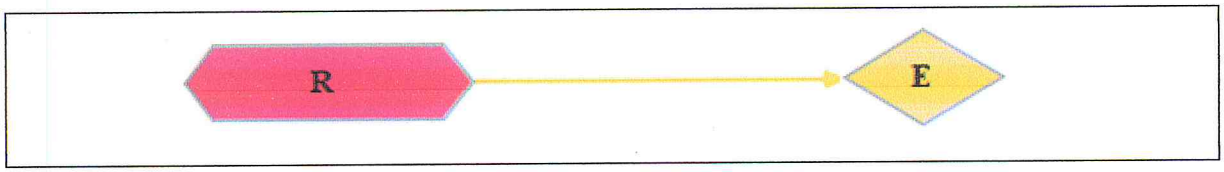
- **Arc composer action :** ce lien décrit la relation entre la règle d'évolution et leur partie actions.



- **Arc composer condition :** ce lien décrit la relation entre la règle d'évolution et leur partie conditions.



- **Arc composer évènement :** ce lien décrit la relation entre la règle d'évolution et leur partie évènements.



3.2.3 Processus d'évolution proposé

Le gestionnaire d'évolution est responsable de la gestion de l'évolution de l'architecture logicielle en fonction des stratégies d'évolution définies et de leurs règles d'évolution. Son rôle est d'intercepter tout événement d'évolution émis par le concepteur ou les règles d'évolution vers l'architecture ou l'un de ses éléments, puis il identifié la stratégie d'évolution correspondante, suivant cette stratégie, il déclenche les règles d'évolution associées à l'événement reçu. Le gestionnaire d'évolution doit également vérifier la maintenance des invariants. Le gestionnaire d'évolution utilise une ou plusieurs stratégies d'évolution et peut déclencher une ou plusieurs règles d'évolution.

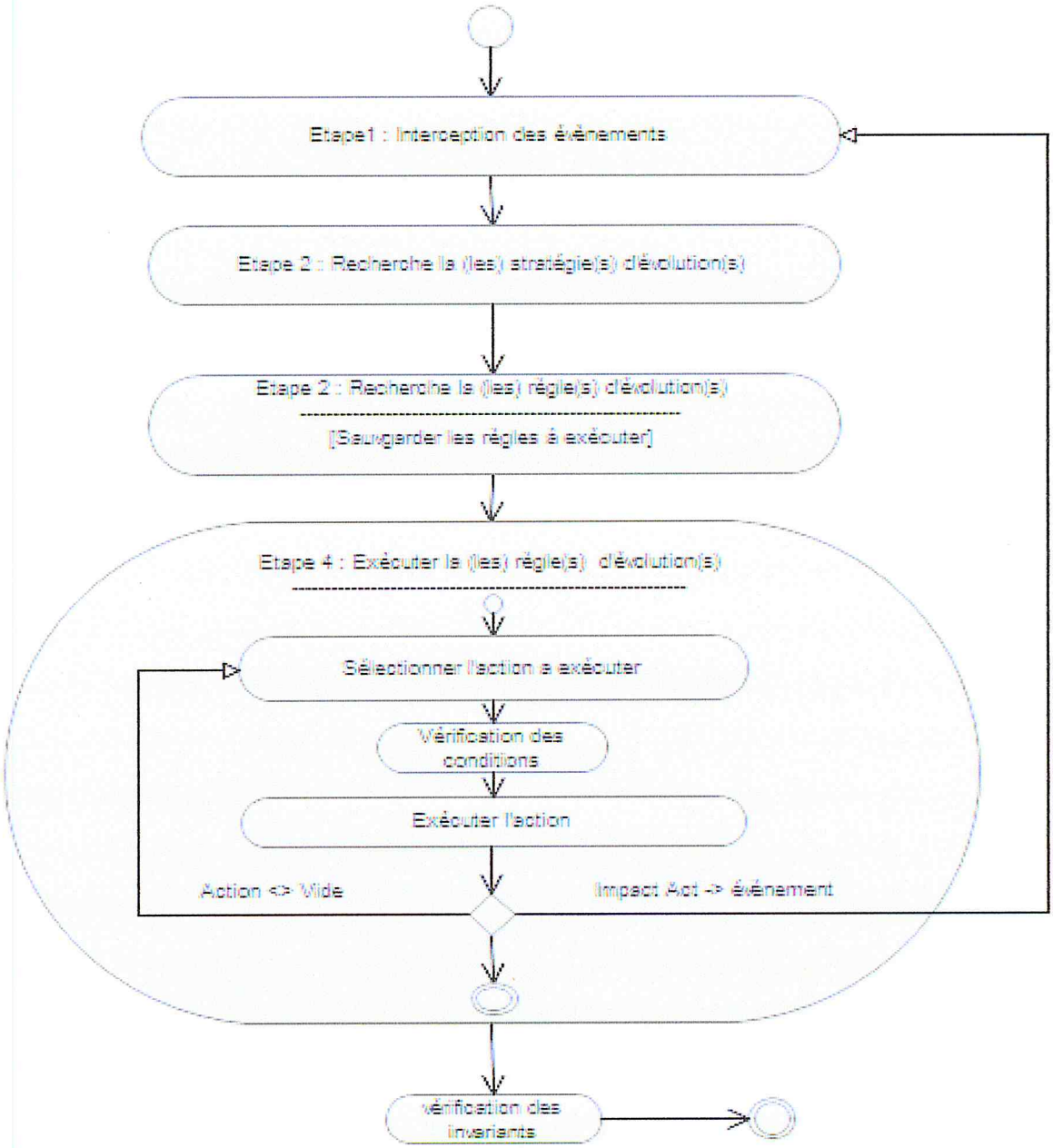


Figure 3. 5 Processus des gestions d'évolutions.

Etape1 : Interception des évènements : Le gestionnaire d'évolution doit intercepter deux types d'évènements : les évènements internes et les évènements externes. ❖ Les évènements externes : ce sont les évolutions où le concepteur doit sélectionner l'élément architectural qu'il souhaite faire évoluer ou bien les évènements émanant par les propriétés non fonctionnelles des éléments architecturales. ❖ Les évènements internes : l'exécution d'une règle d'évolution peut invoquer d'autres évènements pour propager l'impact qu'elle provoque. Ces évènements sont également interceptés par le gestionnaire d'évolution.
Étape 2 : Recherche de la stratégie d'évolution : A l'interception d'un évènement, le gestionnaire recherche la stratégie correspondante à l'évènement intercepté.
Étape 3 : Recherche de la règle d'évolution à exécuter Après que la stratégie d'évolution identifiée, le gestionnaire recherche la ou les règles d'évolution à appliquer. Toutes les règles dont la partie événement correspondent à l'évènement reçu, sont sélectionnées. Si le gestionnaire d'évolution sélectionne une seule règle d'évolution, dans ce cas il déclenche son exécution. Sinon s'il sélectionne plusieurs règles d'évolution, Le gestionnaire doit sauvegarder ces règles pour les exécuter.
Étape 4 : Exécution des règles d'évolution L'exécution d'une règle a appliquée par sa partie action sachant que les conditions sont vrai. Si une action introduit un nouvel évènement (impact), ce dernier sera alors intercepté et une nouvelle règle sera exécutée de la même façon.
Étape 5 : vérification des invariants A chaque évolution traité le gestionnaire d'évolution doit vérifier les invariants de système pour garder la cohérence de l'architecture.

Tableau 3. 3 Explication du Processus d'évolution.

3.3 Les évènements

Nous présentons dans la table 3.3 et la table 3.4 les événements que nous avons pris en compte dans notre travail. Ces événements sont devisés en deux types : les événements internes et les évènements externes.

3.3.1 Les évènements interne

Évènement	Action d'évolution
Générés par les composants :	**Actions appliquées sur les composants :**
Composant Ajouté	Ajouter composant
Composant supprimé	Supprimer composant.
Composant activé	Activer composant.
Composant désactivé	Désactiver composant.
Générés par les connecteurs :	**Actions appliquées sur les connecteurs :**
Composant Ajouté	Ajouter connecteurs
Composant supprimé	Supprimer connecteurs
Composant activé	Activer connecteurs
Composant désactivé	Désactiver connecteurs
Générés par l'interface	**Actions appliquées sur l'interface**
Interface Activée	Activer interface
Interface désactivée	Désactiver interface
Interface démarrée	Démarrer interface
Interface arrêtée	Arrêter interface

Tableau 3. 4 Table des évènements Interne.

3.3.2 Les événements Externes

Evènement	Stratégie d'évolution
Générés par l'architecture	Appliquées sur l'architecture
Taux de temps de réponse lent	Stratégie taux de temps de réponse lent
Stockage mémoire en rupture	Stratégie mémoire en rupture
Performance basse	Stratégie performance basse
Générés par les composants	Appliquées sur les composants
Composant en panne	Stratégie de panne

Tableau 3. 5 Table des évènements externes.

4 Cas d'étude << znn.com >>

4.1 Présentation du cas d'étude

Afin d'illustrer notre approche nous présentons l'exemple Znn.com [29]. Znn.com est un site de nouvelles qui diffuse du contenu multimédia à ses clients. Du point de vue architectural, Znn.com est un système client-serveur basé sur le Web qui se conforme à un style N-tiers. Comme illustré dans la figure 3.6, Znn.com utilise un balanceur de charge pour équilibrer les demandes sur un pool de serveurs répliqués, dont la taille est ajustée dynamiquement pour équilibrer l'utilisation du serveur et le temps de réponse du service. Un ensemble de processus client (représentés par le composant Client) envoie des demandes de contenu sans état à l'un des serveurs. Supposons que nous puissions surveiller le système pour des informations telles que la charge du serveur et la bande passante des connexions serveur-client. Supposons en outre que nous pouvons modifier le système, par exemple, pour ajouter plus de serveurs au pool ou pour changer la qualité du contenu.

Les objectifs commerciaux de Znn.com consistent à diffuser du contenu d'information à ses clients dans un délai de réponse raisonnable tout en maintenant le coût du pool de serveurs dans son budget de fonctionnement. De temps en temps, en raison d'événements très populaires, Znn.com subit des pics dans les demandes de nouvelles qu'il ne peut pas servir de manière adéquate, même à la taille maximale du pool. Pour éviter les latences inacceptables, Znn.com choisit de proposer un contenu textuel minimaliste pendant ces heures de pointe, au lieu de fournir à ses clients un service inégalé. Les administrateurs système de Znn.com (sys-admins) adaptent le système en utilisant deux actions : ajuster la taille du pool de serveurs ou changer de mode de contenu. Lorsque le système est soumis à une charge élevée, les administrateurs système peuvent augmenter la taille du pool de serveurs jusqu'à ce qu'un maximum déterminé par les coûts soit atteint, à partir duquel l'administrateur système fait basculer les serveurs pour servir le contenu textuel.

La décision d'adaptation est déterminée par des observations du temps de réponse moyen global par rapport à la charge du serveur. Plus précisément, quatre adaptations sont possibles, et le choix dépend non seulement des conditions du système, mais aussi des objectifs de l'entreprise :

- Passer du mode multimédia au mode textuel.
- Basculer le mode de contenu du serveur du texte au multimédia.
- Augmenter la taille du pool de serveurs.

- Décrémenter la taille du pool de serveurs.

Le znn.com est un système évolutif possédant les propriétés architecturales suivantes :

- Le nombre maximal des serveurs égale à 5.
- Un balanceur servis au maximum 3 serveurs.
- Le nombre maximal des balanceur égale à 2.
- Le temps de réponse moyen inférieur ou égale à 2 secondes.
- La capacité du stockage minimal (rupture) égale à 10 TB.

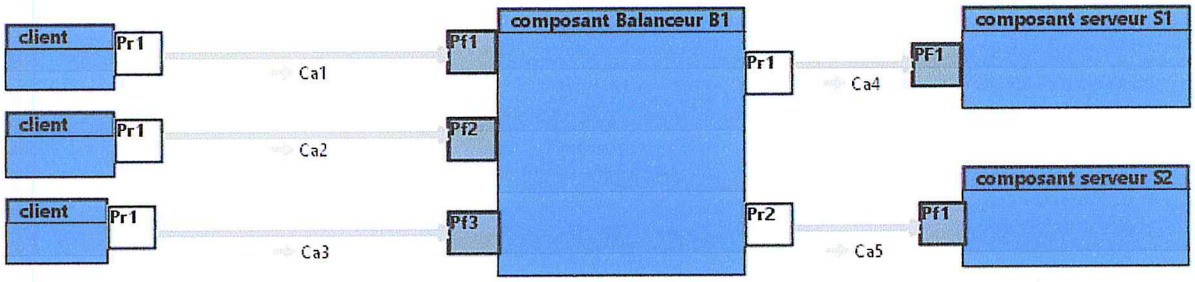


Figure 3. 6 Style architecturale de Znn.com.

4.2 Modélisation de l'étude de cas

Notre objectif est d'aider Znn.com à Modéliser la gestion du système pour ajuster la taille du pool de serveurs par rapport au contenu du serveur entre les modes multimédia et textuel. Pour cela nous avons proposé une approche qui va aider znn.com à modéliser la gestion de système.

4.2.1 Modélisation d'un évènement externe « Serveur Ajouté »

Nous présentons dans la figure 3.7 et la figure 3.8 un exemple d'un ajout d'un serveur émanant par le concepteur (événement externe). Dans cet exemple on va présenter les changements qui vont se produire dans notre architecture.

4.2.1.1 Modélisation du Style architectural

En se basant sur la notation proposée dans la table 3.1 pour décrire le style architectural en cas d'un ajout d'un serveur. Nous présentons dans la figure 3.7 le style architectural du znn.com après l'ajout de serveur S3.

Notre système est composé de trois types composant (Client, Balanceur, Serveur). Chaque type de composant contient des ports fournies et requis connecté entre eux par des connecteurs assemblage. Dans la partie invariant nous avons spécifiées les propriétés architecturales que le système doit respecter.

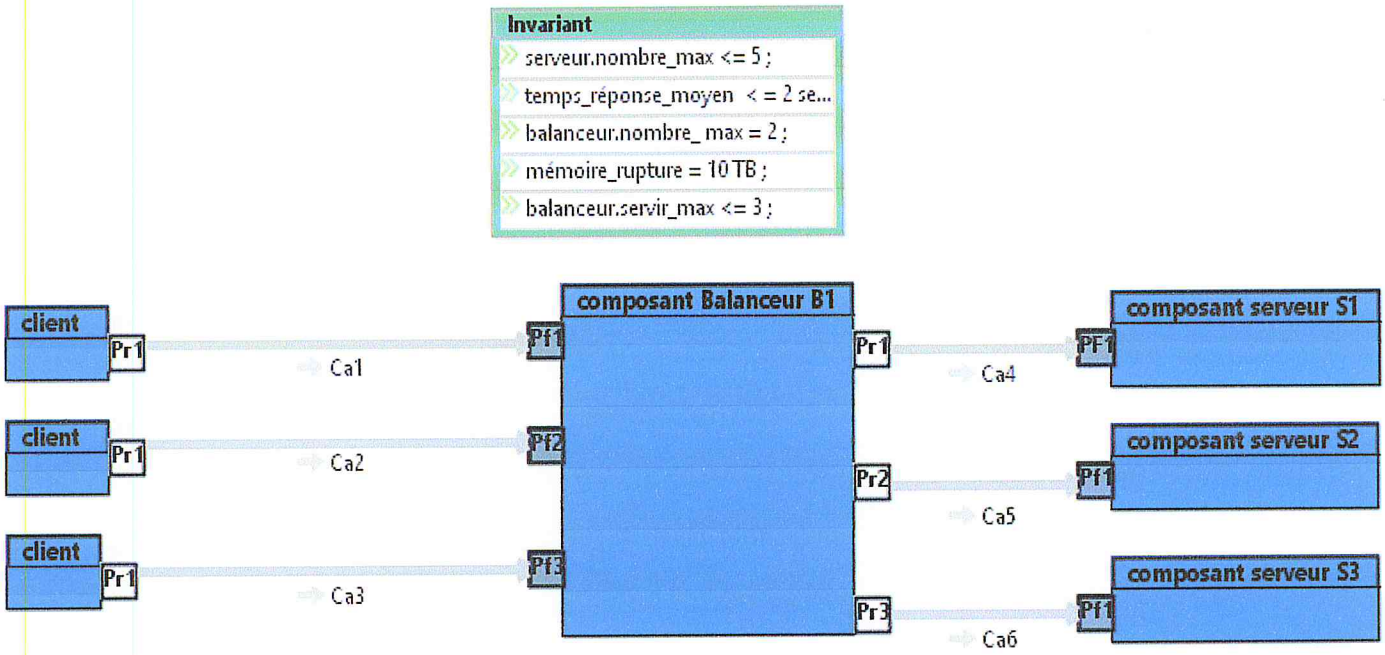


Figure 3. 7 Structure du Style Architectural après l'ajout du serveur S3.

4.2.1.2 Modélisation de la gestion d'évolution

En se basant sur la notation proposée dans la Table 3.2 pour décrire cette opération d'évolution qui permet d'ajouter un composant de type serveur nommée **S3**. La modélisation de cette opération d'évolution avec notre nouvelle notation est donnée par la figure 3.8.

A l'interception de l'évènement **serveur_ajouté (S3)**, le gestionnaire d'évolution recherche la stratégie d'évolution qui correspond à l'évènement intercepté. Puisque nous n'avons qu'une seule stratégie, qui est la **stratégie_ajouter_serveur (S3)**, on choisit cette stratégie pour l'appliquer. La **stratégie_ajouter_serveur (S3)** est constituée par les trois règles d'évolution suivantes :

1. **R1 : ajouter_serveur (S3) :** décrit l'ajout d'un serveur nommé **S3**. Elle est également composée d'un événement, d'une condition et une action :
    a. L'événement **serveur_ajouté (S3)** : décrit que le serveur S3 a bien été ajouté.
    b. La condition **S3.nom != AutreServeur.nom :** exprime que le serveur S3 doit avoir un nom différent que les autres serveurs.
    c. L'action **ajouter_serveur (S3)** : invoque le code source qui implémente l'opération d'ajout de serveur S3.

2. **R2 : ajouter_port (Pf1, S3)** : décrit l'ajout d'un port nommé **Pf1** qui se situe dans le serveur **S3**. Elle est également composée d'un événement, d'une condition et une action :
    a. L'événement **port_ajouté (Pf1, S3)** : décrit que le port Pf1 a bien été ajouté sur le serveur S3.
    b. La condition **Pf1.type = Pbalanceur.type :** exprime que le type de port Pf1 est le même type de port de balanceur qui se connecte avec le serveur S3.
    c. L'action **ajouter_port (Pf1, S3) :** invoque le code source qui implémente l'opération d'ajout de port Pf1.
3. **R3 : ajouter_connecteur (Ca6, Pf1, S3) :** décrit l'ajout d'un connecteur assemblage nommé **Ca6** qui se connecte avec le port **Pf1** du serveur **S3**. Elle est également composée d'un événement, d'une condition et une action :
    a. L'événement **connecteur_ajouté (Ca6, Pf1, S3)** : décrit que le connecteur Ca6 a bien été ajouté sur le port Pf1
    b. La condition **Ca6.BandePassante > = 1 GB** : exprime que le connecteur Ca6 doit avoir une bande passante supérieure ou égale 1 Giga byte.
    c. L'action **ajouter_connecteur (Ca6, Pf1, S3)** : invoque le code source qui implémente l'opération d'ajout de connecteur Ca6.

Après le traitement de cette évolution, le gestionnaire d'évolution doit vérifier les invariants pour valider l'évolution et maintenir la cohérence de l'architecture.

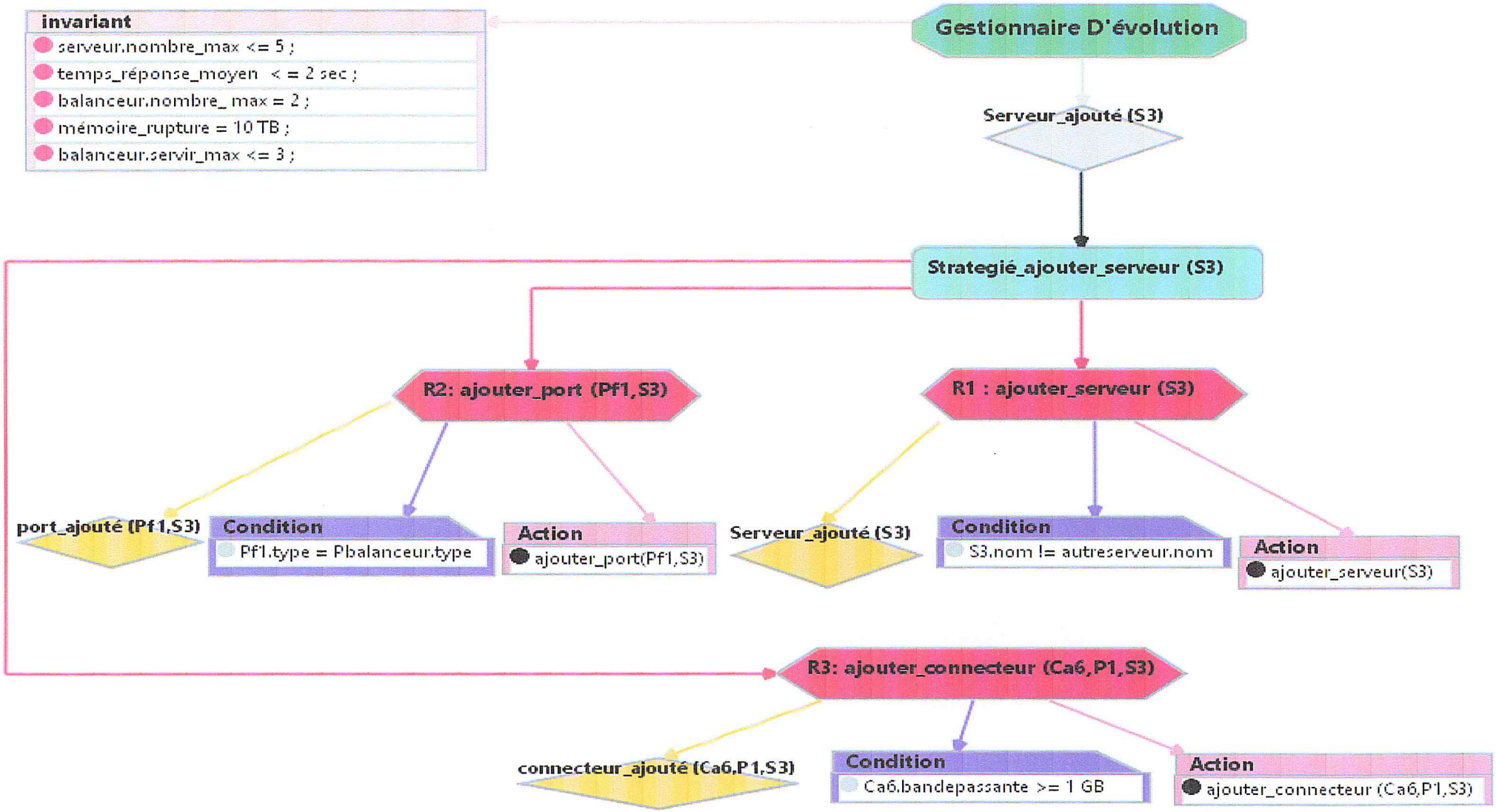


Figure 3. 8 Diagramme de gestion d'évolution en cas d'ajout d'un serveur.

4.2.2 Modélisation de l'évènement taux de temps de réponse lent

Nous présentons dans ce cas le diagramme de la gestion d'évolution lors de l'interception de l'évènement **taux de temps de réponse lent** (le temps de réponse moyen est supérieur à la limite imposée 2 s). Le style architectural aura été selon la stratégie choisie.

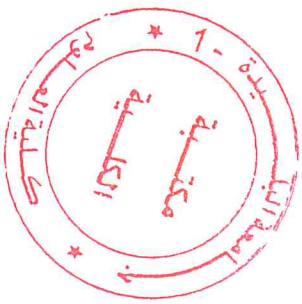
4.2.2.1 Modélisation de la gestion d'évolution

En se basant sur la notation proposée dans la Table 3.2 pour décrire le diagramme de la gestion d'évolution lors de l'interception de l'événement **taux de temps de réponse lent** dans la figure 3.9.

A l'interception de l'évènement **taux_temps_réponse_lent**, le gestionnaire d'évolution recherche toutes les stratégies d'évolutions qui correspond à l'évènement intercepté. Puisque nous avons deux stratégies, qui sont les **(stratégie_temps_reponse_lent1 (S3) ou stratégie_modifier_serveur (S3))**, la gestionnaire doit choisir une seule stratégie parmi les deux pour l'appliquer.

On commence d'abord par la **stratégie_temps_reponse_lent1 (S4).** Cette dernière, elle a le même concept de la **stratégie_ajouter_serveur (S4)** que nous avons détaillé dans l'exemple précédent qui consiste à ajouter un serveur pour diminuer le taux de temps de réponse.

La deuxième stratégie **stratégie_modifier_serveur (S3)** consiste à modifier le comportement de serveur **S3** vers le mode textuel sachant que le serveur est modifiable. Cette stratégie est composée d'une seule règle **R1 : modifier_serveur (S3)** qui décrit la modification de serveur S3. La règle R1 est composée d'un événement interne décrit que le serveur S3 a été bien modifié, une condition décrit que le serveur doit être modifiable et une action qui invoque le code source qui implémente l'opération de modification de serveur **S3** ver le mode textuel.



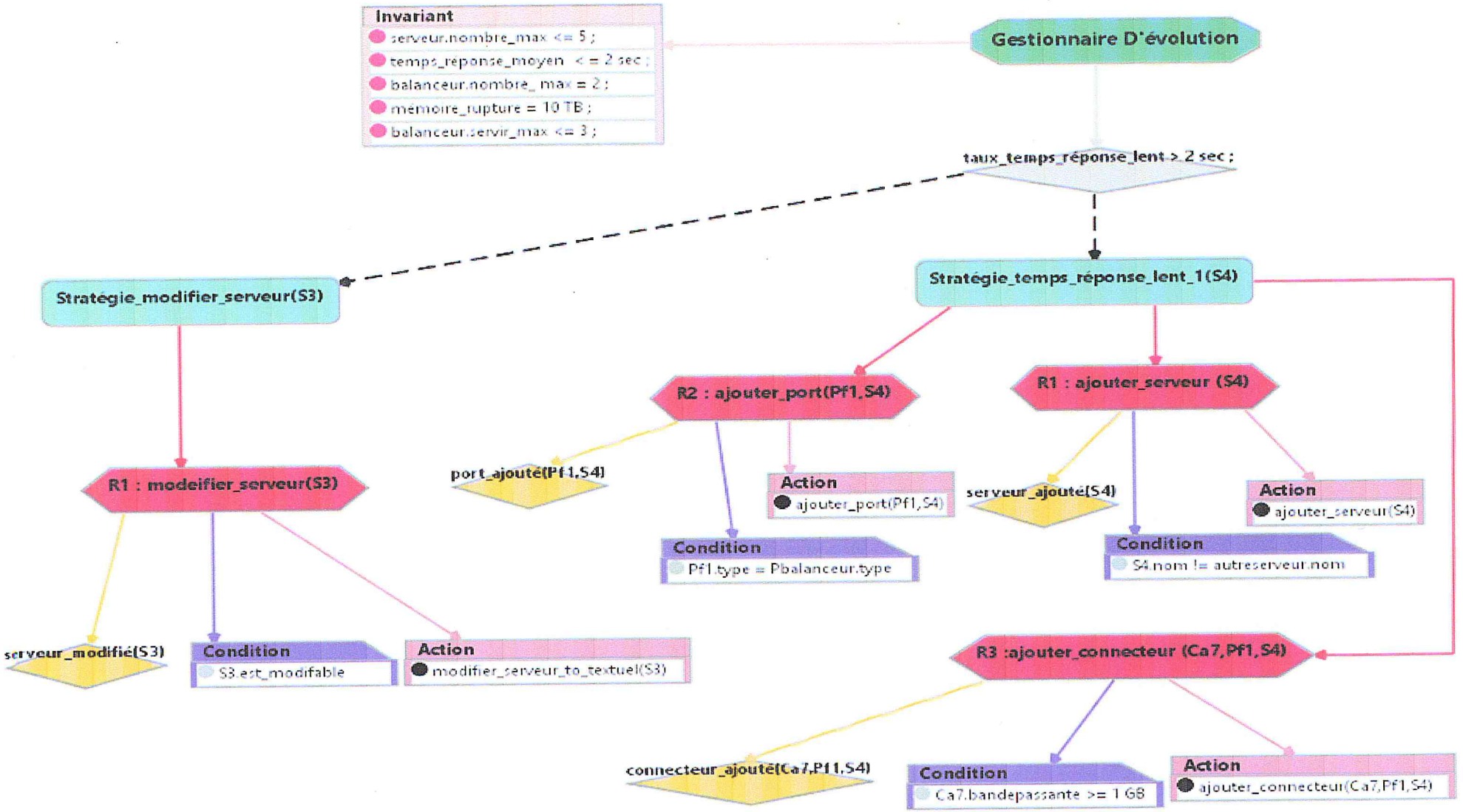


Figure 3. 9 Diagramme de gestion d'évolution en cas d'intercepter un évènement taux de temps de réponse lent.

4.2.2.2 Modélisation du style architectural en cas du choix de la stratégie taux de temps de réponse lent (S4)

La figure 3.10 décrit le Style Architecturale possible en cas du choix de la stratégie taux de temps de réponse lent (S4) lors d'intercepter l'événement taux de temps réponse lent > 2 secondes.

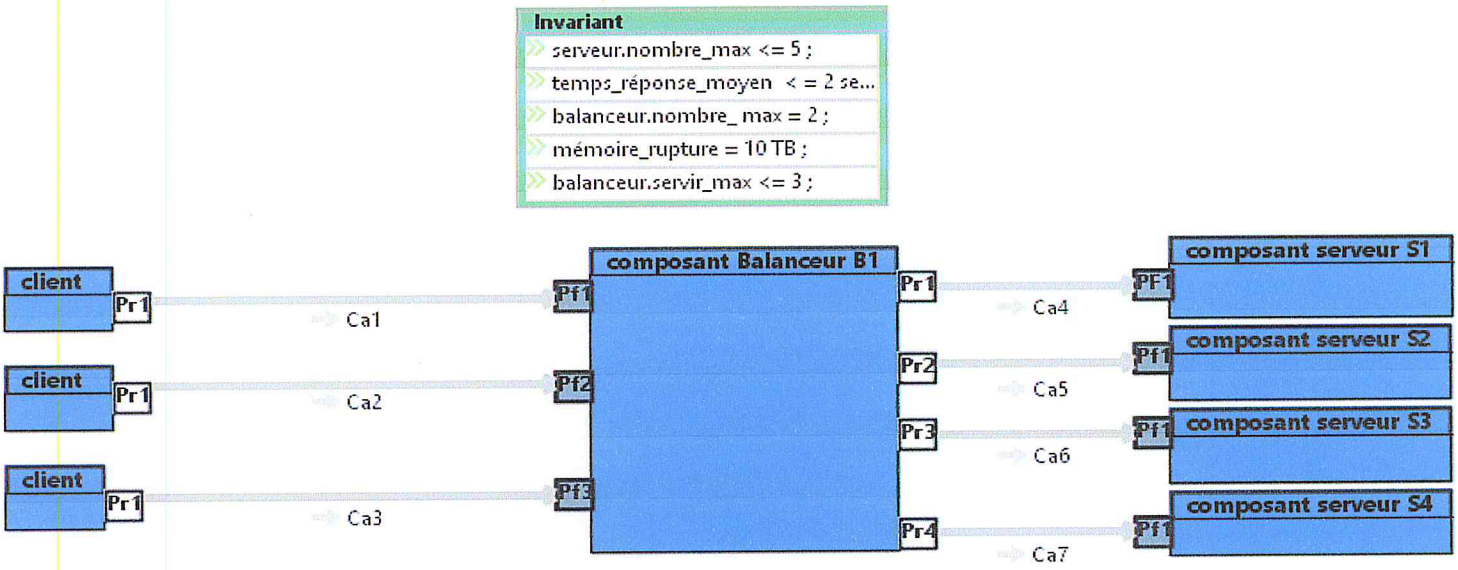


Figure 3. 10 Structure du style architectural en cas du choix de la stratégie taux de temps réponse lent (S4).

4.2.2.3 Modélisation du style architectural en cas du choix de la stratégie modifier serveur (S3).

La figure 3.11 décrit le Style Architecturale possible en cas du choix de la stratégie modifier serveur (S3) lors d'intercepter l'événement taux de temps réponse lent > 2 seconde.

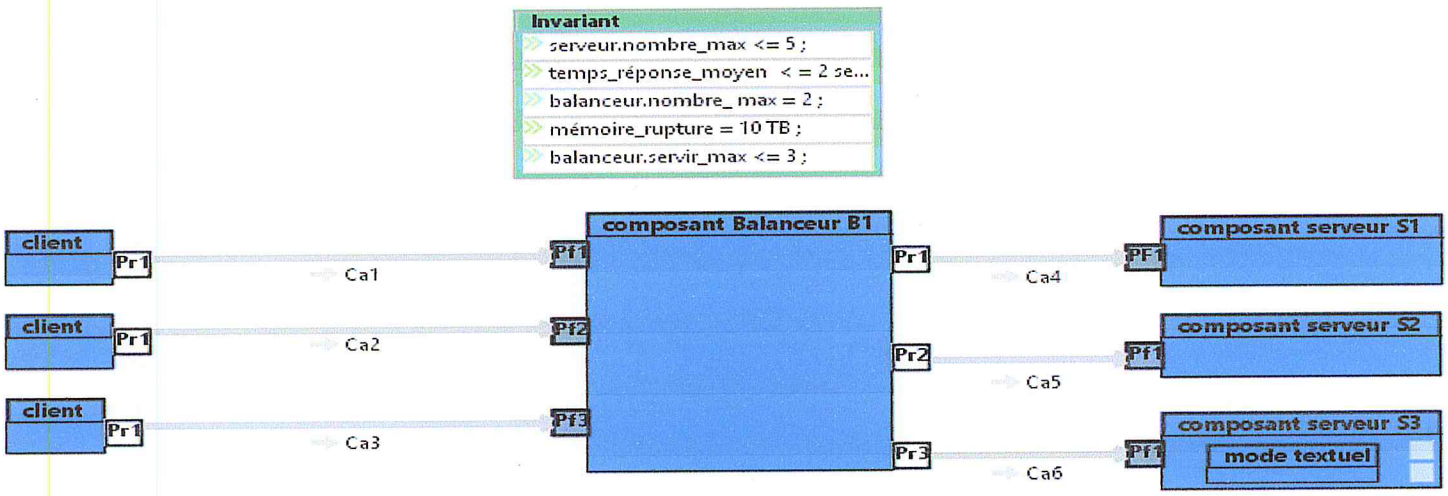


Figure 3. 11 Structure du style architectural en cas du choix de la stratégie modifier serveur (S3).

4.2.3 Modélisation de l'événement stockage mémoire en rupture

Nous présentons dans ce cas le diagramme de la gestion d'évolution et le style architectural lors de l'interception de l'évènement **stockage mémoire en** (Décrit que le stockage mémoire est inférieur à 10 TB.). Le style architectural sera adapté selon la stratégie choisie.

4.2.3.1 Modélisation de la gestion d'évolution

En se basant sur la notation proposée dans la Table 3.2 pour décrire le diagramme de la gestion d'évolution lors de l'interception de l'événement **stockage mémoire en rupture** dans la figure 3.12.

A l'interception de l'évènement **stockage_mémoire_en_rupture**, le gestionnaire d'évolution recherche tous les stratégies d'évolutions correspondantes à l'évènement intercepté. Puisque nous avons trois stratégies, qui sont les **(stratégie_ajouter_disk_dur (D1), stratégie_ajouter_serveur (S4) et stratégie_substituer_serveur (S1))**, la gestionnaire doit choisir une seule stratégie parmi les trois pour l'appliquer.

La première stratégie **stratégie_ajouter_disk_dur (D1)** consiste à ajouter le disque dur **D1** dans le serveur **S3**. Cette stratégie est composée d'une seule règle **R1 : ajouter_disk_dur (S3)** qui décrit l'ajout du disque dur **D1** dans le serveur **S3**. La règle **R1** est composée d'un événement interne décrit que le disque dur **D1** a été bien ajouté, une condition décrit que le disque dur **D1** doit avoir un stockage mémoire supérieur à 5 TeraByte et une action qui invoque le code source qui implémente l'opération de l'ajout du disque dur **D1** dans **S3**.

La deuxième stratégie **stratégie_ajouter_serveur (S4).** Cette dernière, elle a le même concept de la **stratégie_ajouter_serveur (S4)** que nous avons détaillé dans l'exemple précédent qui consiste à ajouter un serveur pour augmenter l'espace de stockage mémoire.

La troisième stratégie **stratégie_substituer_serveur (S1)** consiste à substituer le serveur **S1** par un autre serveur **S4** qui a des capacités de stockage mémoire plus développé que la capacité du serveur **S1**. Cette stratégie est composée de six règles, les trois premières règles décrivent la suppression du serveur S1 et les autres trois règles décrivent l'ajout du serveur S4.

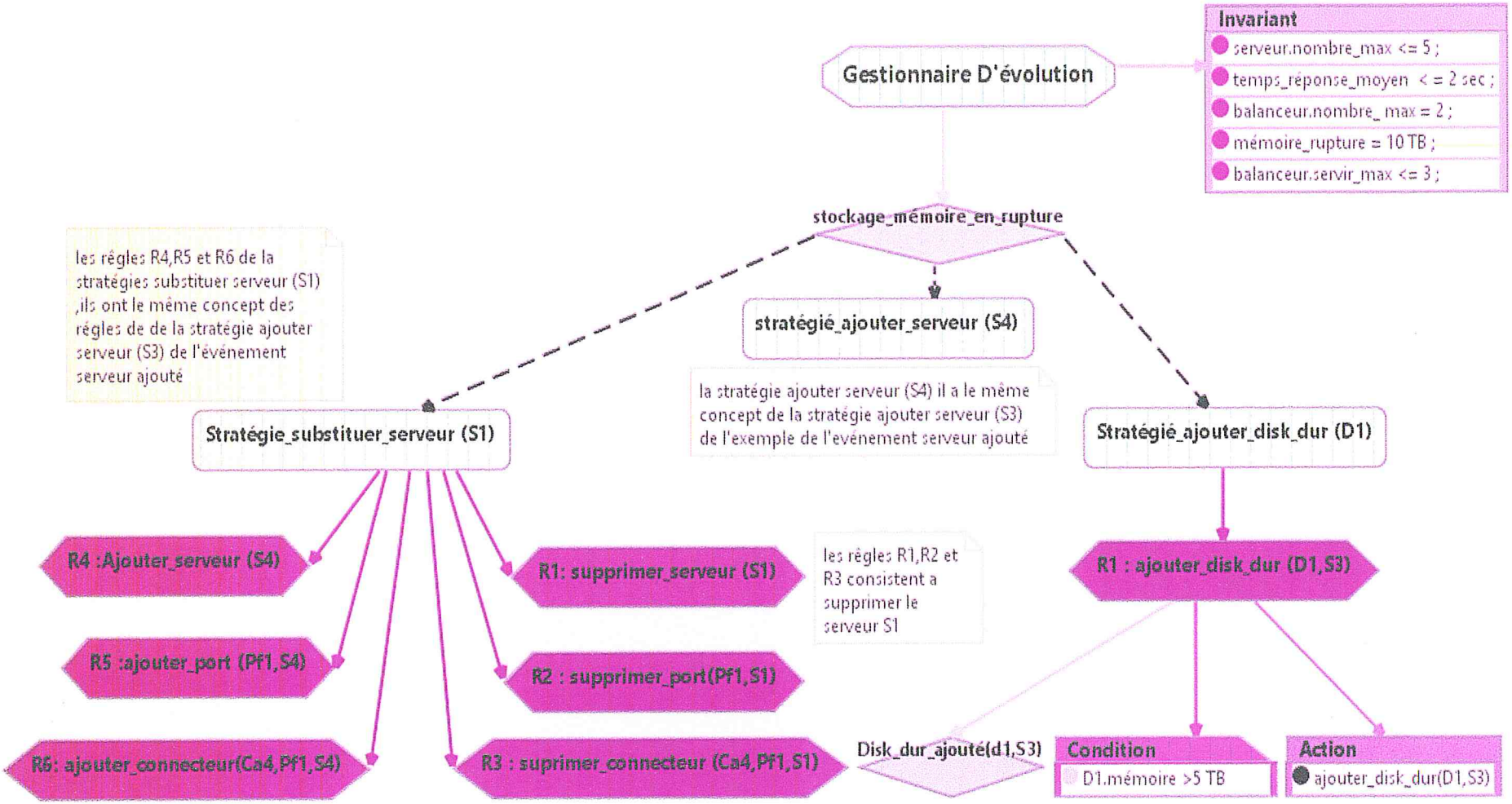


Figure 3. 11 Diagramme de gestion d'évolution en cas d'intercepter l'évènement stockage mémoire en rupture.

4.2.3.2 Modélisation du style architectural en cas du choix de la stratégie ajouter disque dur (D1)

La figure 3.13 décrit le Style Architecturale possible en cas du choix de la stratégie ajouter disque dur (D1) lors d'intercepter l'événement stockage mémoire en rupture.

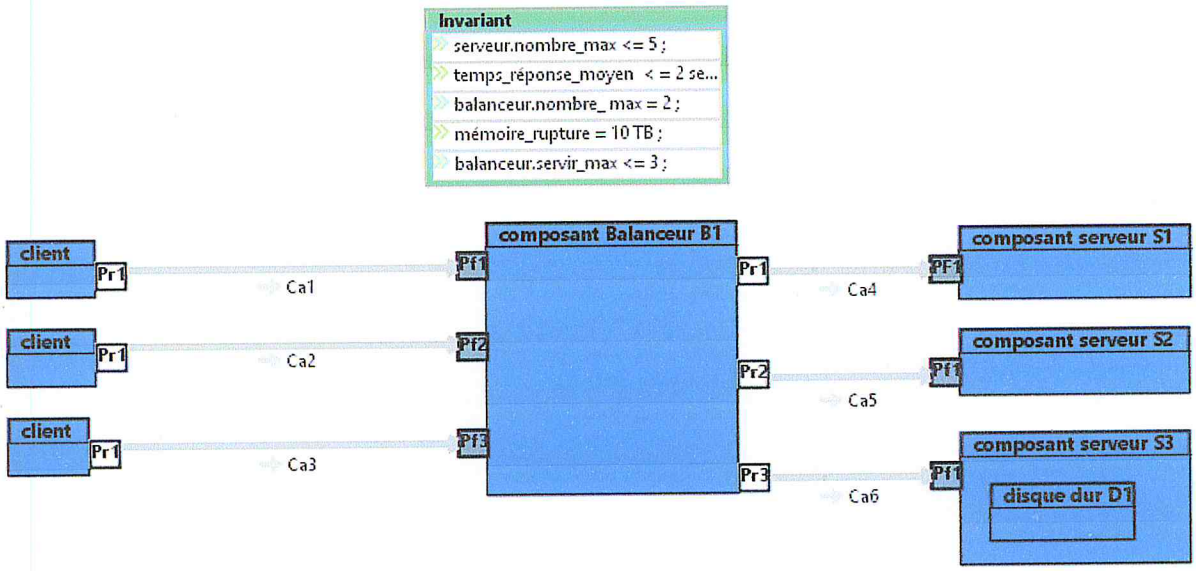


Figure 3. 13 Structure du style architectural en cas du choix de la stratégie ajouter disque dur (D1).

4.2.3.3 Modélisation du style architectural en cas du choix de la stratégie ajouter serveur (S4)

La figure 3.14 décrit le Style Architecturale possible en cas du choix de la stratégie ajouter serveur (S4) lors d'intercepter l'événement stockage mémoire en rupture.

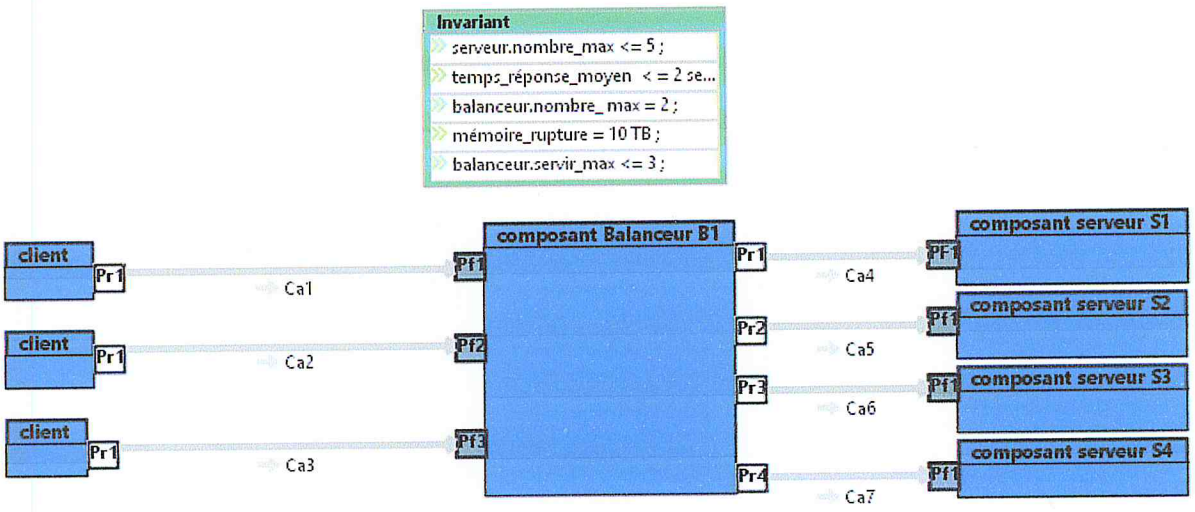


Figure 3. 14 Structure du style architectural en cas du choix la stratégie ajouter serveur (S4).

4.2.3.4 Modélisation du style architectural en cas du choix de la stratégie substituer serveur (S1)

La figure 3.15 décrit le Style Architecturale possible en cas du choix de la stratégie substituer serveur (S1) lors d'intercepter l'événement stockage mémoire en rupture.

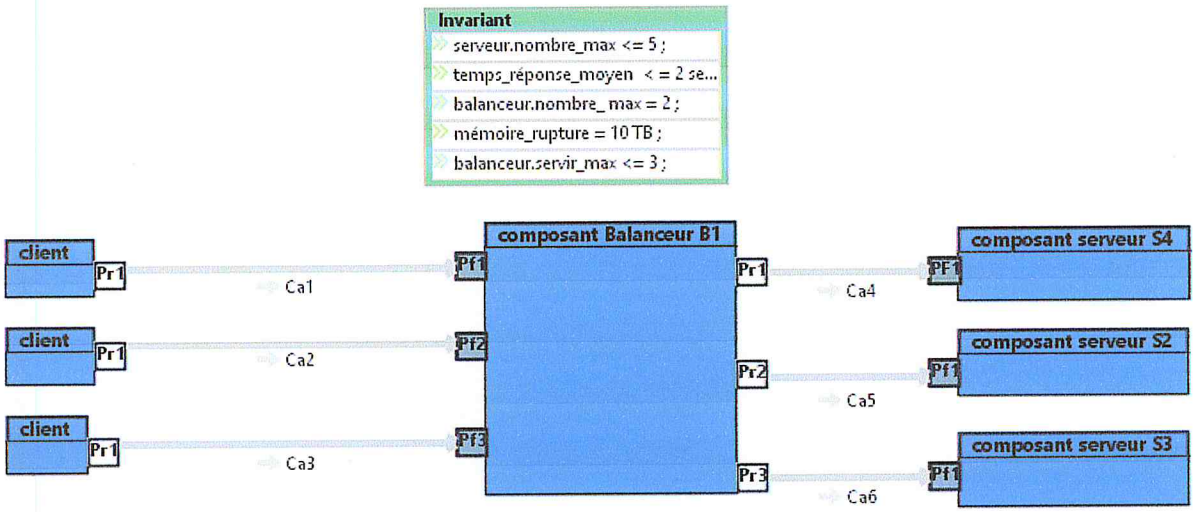


Figure 3. 15 Structure du style architectural en cas du choix de la stratégie substituer serveur (S1).

5 Conclusion

Dans ce chapitre nous avons présenté notre méta-modèle ALD à travers un profil Ecore de EMF, la description de ce méta-modèle est composée de deux phases, dans la première phase nous avons décrit le style Architectural et en deuxième phase nous avons décrit la gestion d'évolution, à partir de cette dernière nous avons proposé des notations graphiques correspondantes aux éléments développées dans les deux phases, enfin, nous avons illustré des opérations d'évolution par des diagrammes en utilisant les notations graphiques.

Le chapitre qui suit sera consacré à l'implémentation de notre éditeur graphique. Nous présentons aussi l'environnement de travail, ainsi que les outils de développement utilisés.

Chapitre 4 :

Implémentation

1 Introduction

Après avoir proposé notre méta-modèle et la notation graphique correspondante, nous allons présenter dans ce chapitre l'implémentation graphique de ces dernies.

Nous commençons tout d'abord, par présenter l'environnement de travail ainsi que les outils de développement utilisés. Nous terminons ce chapitre par la présentation de notre éditeur.

2 Environnement de travail

Dans notre application nous avons utilisé la plate-forme Java 8 comme environnement de développement et d'exécution ainsi que le package de modélisation Eclipse Modeling Tools avec ces différents éléments nécessaires tels qu'Eclipse Modeling Framework (EMF) et le Graphical Modeling Framework (GMF), que nous allons présenter ci-dessous.

2.1 Langages utilisés

2.1.1 Java 8

Java 8 est la dernière version de Java. Elle offre de nouvelles fonctionnalités, des performances accrues et des corrections de bug pour améliorer l'efficacité de développement et d'exécution des programmes Java. La nouvelle version de Java est d'abord mise à disposition des développeurs afin qu'ils disposent du temps adéquat pour effectuer les opérations de test et de certification. Java 8 comprend des fonctionnalités pour la productivité, la facilité d'utilisation, la programmation polyglotte améliorée, la sécurité et les performances améliorées. [30].

2.1.2 Ecore

Ecore est un langage de méta-modélisation permet de définir la structure d'une série d'instances d'un modèle. C'est-à-dire, nous spécifions les types d'objets qui composent les instances de ce modèle, les données qu'ils contiennent et les relations entre eux [31].

2.2 Utilitaires utilisés

2.2.1 Eclipse Neon.3

Pour la réalisation de notre implémentation nous avons utilisé Eclipse IDE avec la version Neon.3, cette dernière comporte le package de modélisation "Eclipse Modeling Tools" qui fournit des outils et des runtimes pour la construction d'applications basées sur des modèles [32].

2.2.2 EMF (Eclipse Modeling Framework)

Eclipse Modeling Framework (EMF) est un ensemble de plugins Eclipse qui peut être utilisé pour modéliser un modèle de données et générer un code ou une autre sortie en fonction de ce modèle. EMF a une distinction entre le méta-modèle et le modèle actuel. Le méta-modèle décrit la structure du modèle. Un modèle est une instance concrète de ce méta-modèle. EMF permet au développeur de créer le méta-modèle par différents moyens, par exemple XMI, annotations Java, UML ou un schéma XML. Il permet également de persister les données du modèle [31].

2.2.3 GMF (Graphical Modeling Framework)

GMF est un Framework de l'environnement de travail Eclipse. Il fournit une infrastructure permettant l'exécution d'éditeurs graphiques basés sur les Framework EMF. Le projet GMF fournit une approche axée sur le modèle pour générer des éditeurs graphiques dans Eclipse. En définissant une boite d'outils graphique, on peut générer un éditeur graphique entièrement fonctionnel basé sur le temps d'exécution de GMF [33].

3 Les étapes suivies pour construire notre application

Après l'installation de Eclipse Modeling Neon.3 qui est déjà porté Eclipse Modeling Framework, nous installons aussi les plugins GMF Tooling, GMF Notation et GMF Runtime qui nous a permis de créer un éditeur graphique. Pour atteindre cet objectif, il faut suivre les étapes suivantes :

- **Modèle de Domaine (Domain Model Definition)**

  Dans cette étape nous créons le méta-modèle que nous voulons utiliser pour créer l'éditeur graphique. Pour ce méta-modèle, nous avons le choix entre plusieurs types de méta-modèles : code Java annoté, modèle Ecore, modèle de classe Rose, modèle UML ou schéma XML). Dans ce projet, nous avons utilisé le méta-modèle Ecore.

  GMF fournit le tableau de bord GMF (GMF Dashboard), ce tableau de bord sert comme un moyen facile pour passer par le processus de génération d'un éditeur graphique. Comme on peut le voir dans la figure 4.1, tout commence à partir du modèle de domaine au centre. De cela, nous pouvons dériver trois modèles. Le modèle de génération de domaine (Domain Gen Model), le modèle de définition graphique (Graphical Definition model) et le modèle de définition d'outillage (Tooling Definition Model). Dans ce qui suit nous allons décrire l'utilité de chaque modèle :

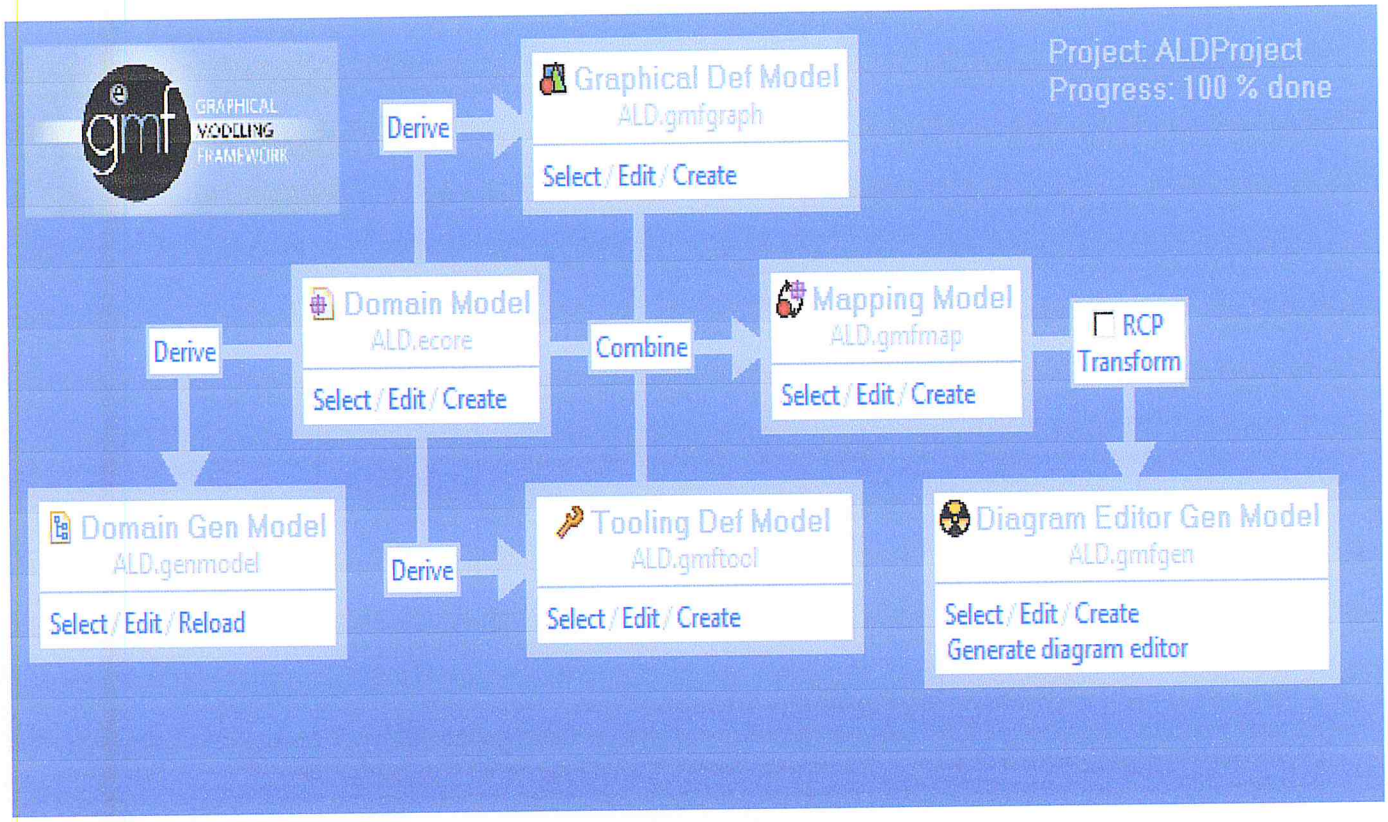


Figure 4. 1 Tableau de Bord de GMF

- **Modèle de Génération de Domaine (Domain Gen Model)**

  Cette étape est utilisée pour générer le code du modèle de domaine avec EMF

- **Modèle de Définition Graphique (Graphical Definition model)**

  Le modèle de définition graphique est utilisé pour définir les figures, les nœuds, les liens…etc, que nous allons afficher sur notre diagramme.

- **Modèle de définition d'outillage (Tooling Definition Model)**

  Le modèle de définition d'outillage décrit la palette et les outils, qui nous allons utiliser pour la création des éléments du diagramme.

- **Modèle de Définition de Mappage (Mapping Model)**

  Le modèle de définition de mappage nous permettra de lier les trois modèles que nous avons jusqu'à présent : le modèle de domaine, le modèle de graphique et le modèle d'outillage. En langage clair, le modèle de définition de mappage indique qu'une ellipse dans le modèle graphique est la représentation d'une instance d'une classe dans le modèle de domaine et est créée à l'aide d'un outil de création décrit dans le modèle d'outillage.

❖ **Modèle de générateur d'éditeur de diagramme (Diagram Editor Gen Model)**

Cette dernière étape nous a permis de générer notre éditeur graphique GMF. Vous pouvez générer deux types d'éditeur graphique avec GMF : un éditeur graphique de plugin intégré à Eclipse ou une application RCP (Rich Client Platform) qui consiste en une application autonome. Pour générer une application RCP, cliquez simplement sur RCP sur le tableau de bord GMF. Dans ce projet, nous allons créer un éditeur GMF en tant que RCP application.

En fin, nous exécutons notre projet pour obtenir notre nouvel éditeur graphique

4 Présentation de l'application

4.1 Créer un diagramme

Cette interface présente la barre d'outils de l'éditeur graphique. Dans cet éditeur on a deux types de diagramme **ALD Diagram** (Diagramme de Gestion d'évolution) et **Archstyle Diagram** (Diagramme du style Architectural).

Pour créer un diagramme on clique sur File → New → ALD Diagram ou Archstyle Diagram, comme nous montre la figure 4.2.

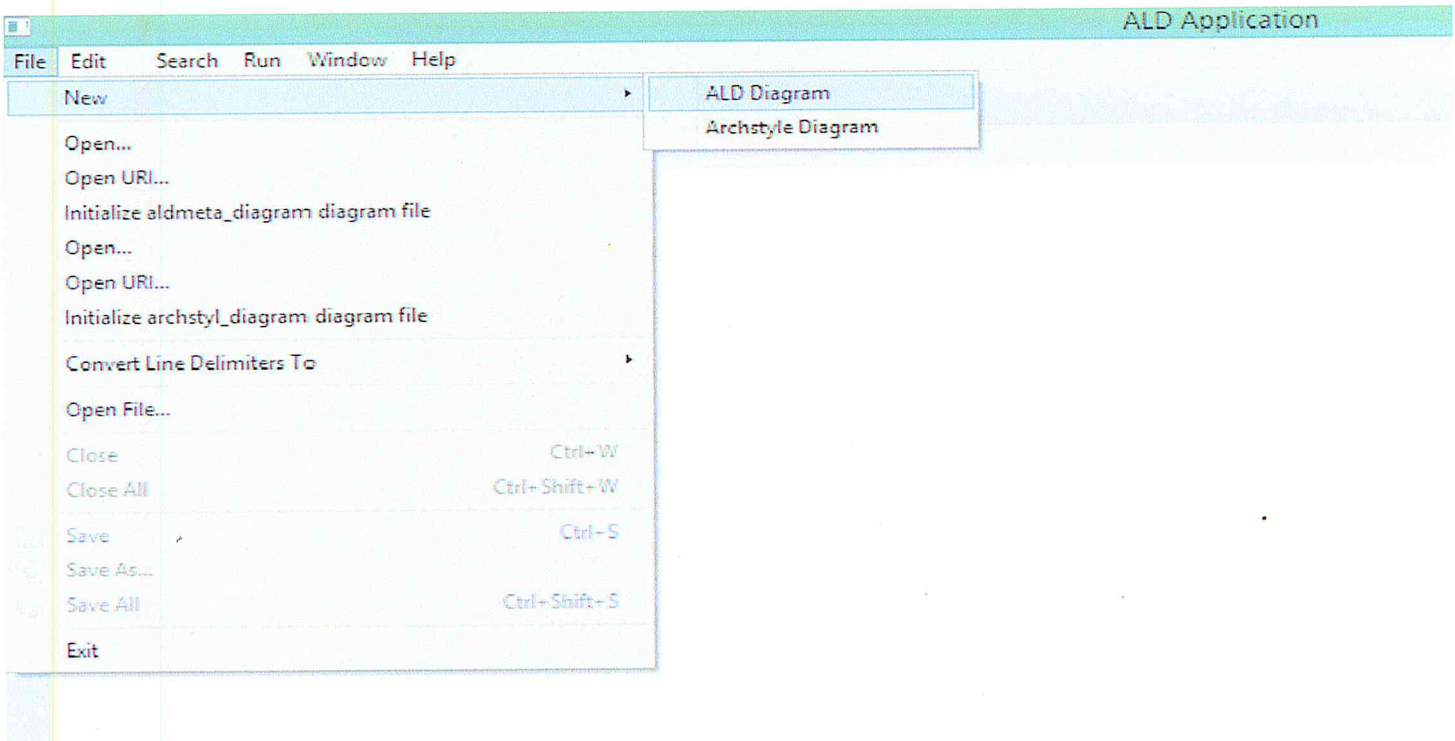


Figure 4. 2 Création d'un nouveau Diagramme.

4.2 Dessiner un diagramme du Style Architectural

La figure 4.3 illustre comment dessiner un diagramme du Style Architectural proposé, il y a deux parties dans la figure 4.3, la partie 1 c'est notre palette qui contient trois groupes d'outils,

le premier groupe présente les composants (composant, port fournis et port requis), le deuxième groupe présente les connecteurs (connecteur délégation et connecteur d'assemblage) et le troisième groupe présente les invariants. Pour dessiner on doit glisser l'un des éléments de la palette vers la partie 2 (la zone de modélisation).

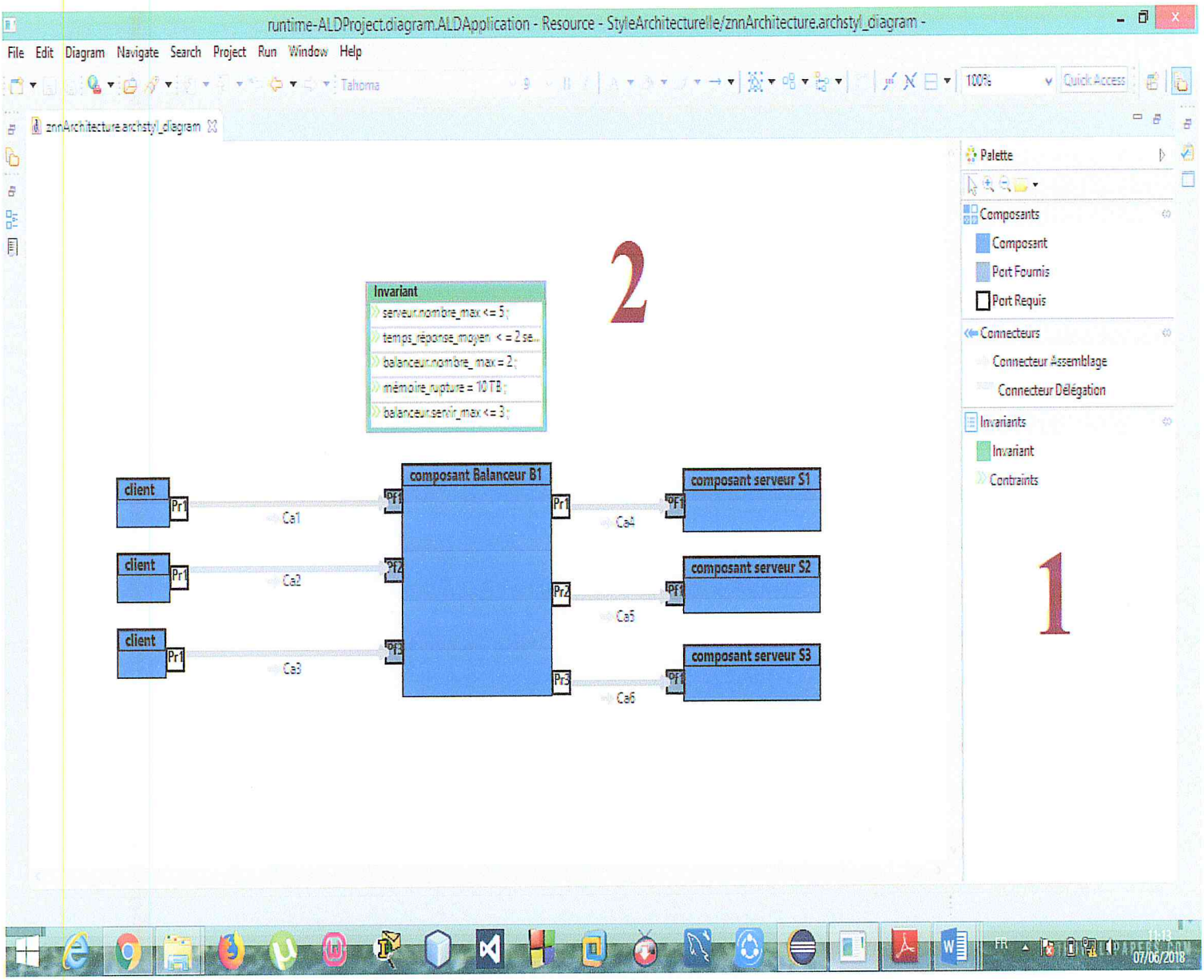


Figure 4. 3 dessiner un Diagramme du Style architectural.

4.3 Dessiner un diagramme de la gestion d'évolution

La figure 4.4 illustre comment dessiner un diagramme du Style Architectural, la figure 4.4, présente deux parties : la partie 1 décrit notre palette qui contient trois groupes d'outils, le premier groupe présente les nœuds de notre modèle comme le gestionnaire d'évolution, les stratégies, les règles…etc., le deuxième groupe présente les liens qui relient les nœuds et le

troisième groupe présente les attributs des neouds. Pour dessiner les éléments proposés, il suffit de glisser l'un des éléments de la palette vers la partie 2 (la zone de modélisation).

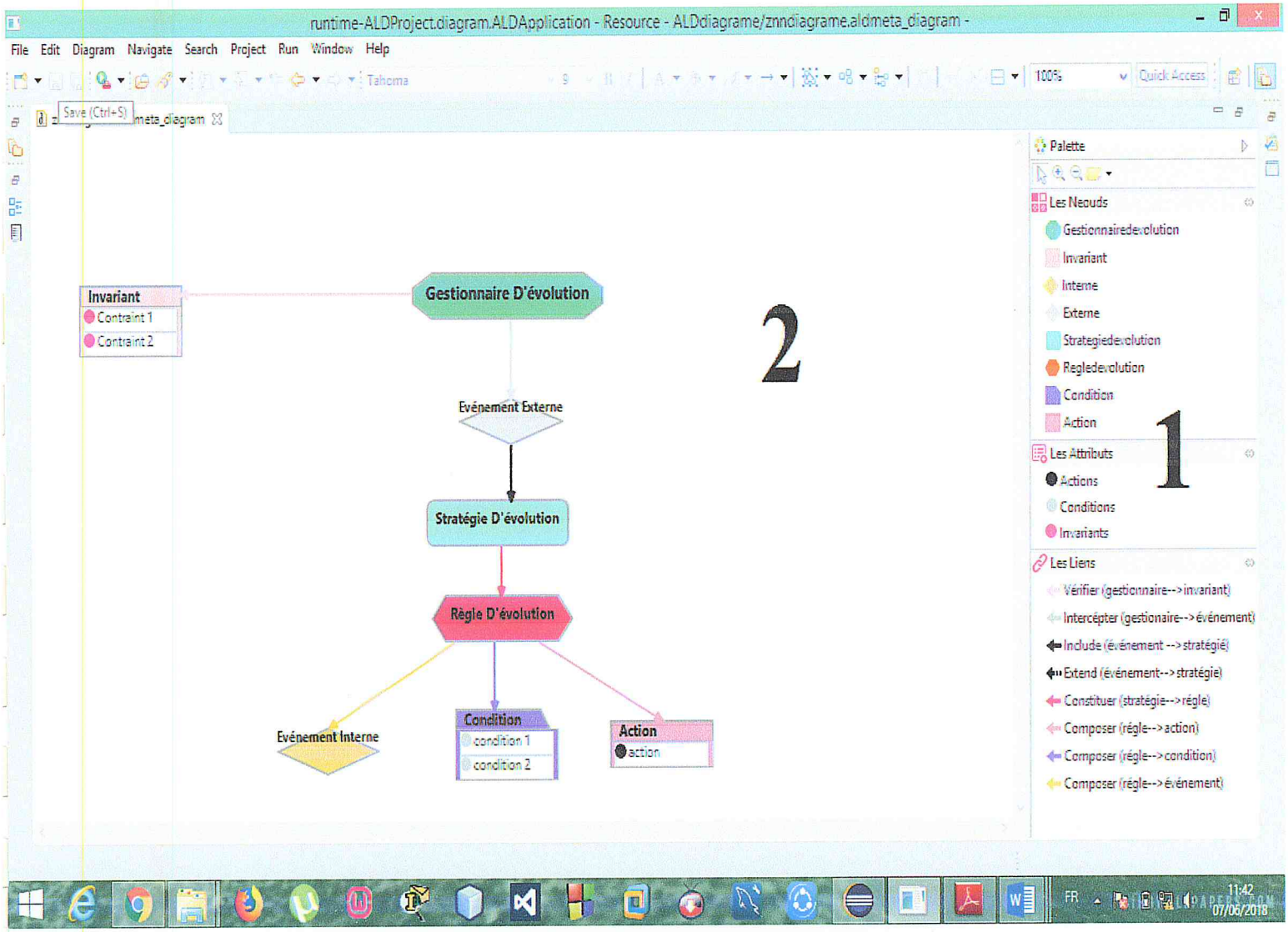


Figure 4. 4 Dessiner un Diagramme de Gestion d'évolution.

4.4 Interface d'historique du processus d'évolution

La figure 4.5 illustre l'interface d'historique de processus d'évolution lors de l'interception d'un événement. Cette interface est composée de cinq parties. La première partie (les événements) présente les événements interceptés par le système, la deuxième partie (les stratégies) présente les stratégies correspondantes à chaque événement, la troisième partie (la stratégie choisie) présente la stratégie qui a été choisie pour réagir à l'évènement intercepté, la quatrième partie (les propriétés de système) présente les propriétés du systèmes lors de l'interception d'un événement, la cinquième partie (les actions) présente les actions de chaque stratégie.

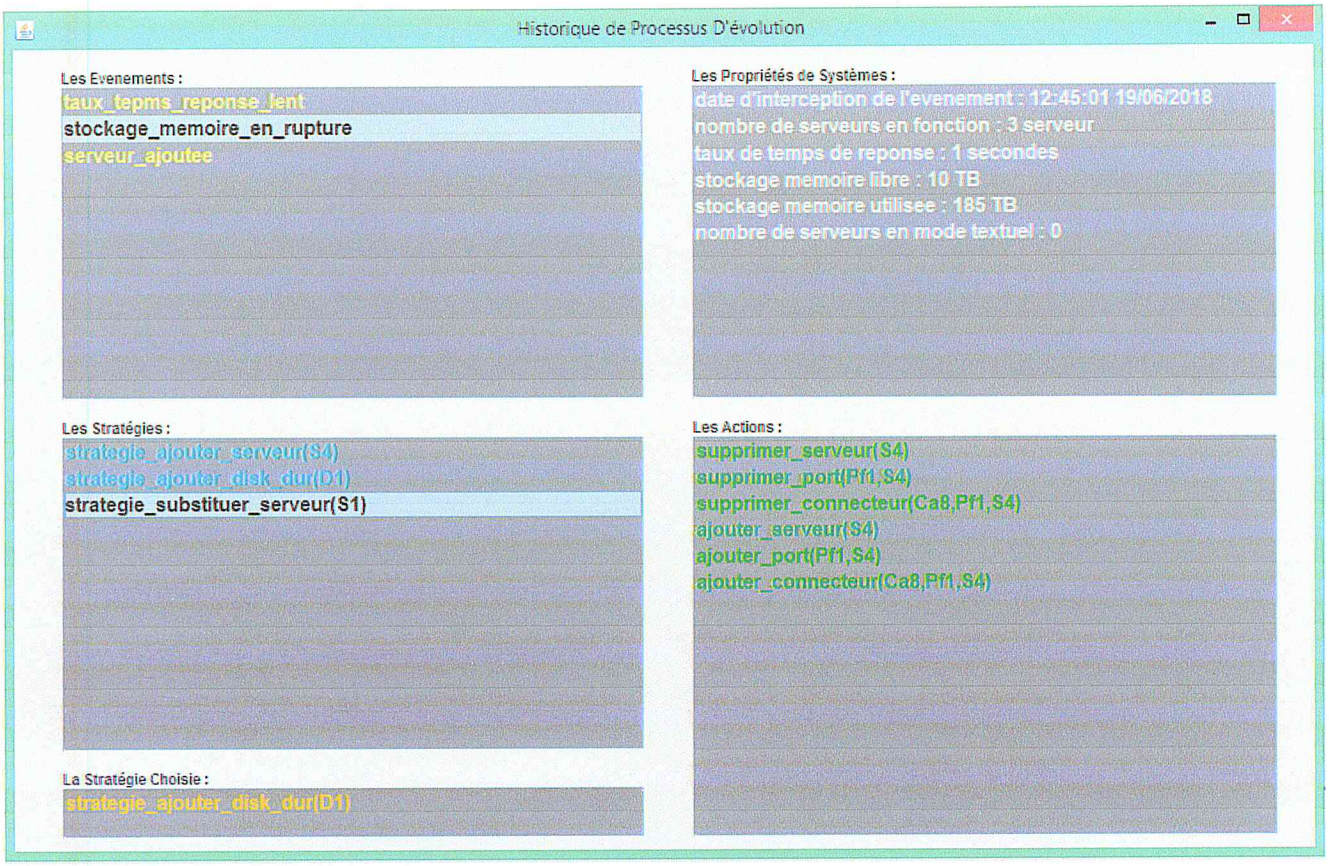


Figure 4. 5 Interface d'historique de processus d'évolution.

Dans cette interface nous avons utilisé un fichier XML pour stocker les pour sauvegarder l'historique des événements, comme le montre la figure 4.6.

```
<?xml version="1.0" encoding="UTF-8"?>
- <gestionnaire>
  - <evenement nom="taux_tepms_reponse_lent">
    - <strategie nom="strategie_temps_reponse_lent(S4)">
        <action>ajouter_serveur(S4)</action>
        <action>ajouter_port(Pf1,S4)</action>
        <action>ajouter_connecteur(Ca7,Pf1,S4)</action>
      </strategie>
    - <strategie nom="strategie_modifier_serveur(S3)">
        <action>modifier_serveur_to_textuel(S3)</action>
      </strategie>
    - <proprietee nom="">
        <pr> date d'interception de l'evenement : 12:45:01 19/06/2018</pr>
        <pr> nombre de serveurs en fonction : 3 serveur</pr>
        <pr> taux de temps de reponse : 3 secondes</pr>
        <pr> stockage memoire libre : 120 TB</pr>
        <pr> stockage memoire utilisee : 75 TB</pr>
        <pr>nombre de serveurs en mode textuel : 0 </pr>
      </proprietee>
      <strategiechoisie nom="">strategie_modifier_serveur(S3)</strategiechoisie>
  </evenement>
```

Figure 4. 6 partie du fichier XML.

5 Conclusion

Tout au long de ce chapitre, nous avons présenté notre application offrant un ensemble de fonctionnalités aidant l'architecte logicielle à modéliser des architectures logicielles dynamique. A ce stade, nous avons présenté l'environnement de travail, les outils de développement utilisés et les étapes suivies pour construire notre application.

Ainsi que les principales interfaces qui composent notre application.

Conclusion générale

Conclusion générale

Dans ce mémoire, nous avons présenté l'ensemble des travaux réalisés dans l'objectif de réaliser un éditeur graphique, cet éditeur graphique se base sur notre méta-modèle nommé ALD qui offre une aide à l'architecte des logiciels pour la spécification des architectures logicielles dynamiques.

Pour se faire, nous avons tout d'abord commencé par l'état de l'art. Ce dernier consiste à faire une étude approfondie de notre champ d'étude, afin de comprendre les principes généraux et les bases fondamentales des architectures logicielles dynamiques. Cette étude permet de recueillir les informations nécessaires pour entamer le travail demandé y compris les éléments architecturaux, les styles architecturaux, les langages utilisées pour la description des architectures logicielles et enfin un aperçu sur les architectures logicielles dynamiques.

Ensuite, puisque l'idée directrice de nos travaux a été de chercher à créer un méta-modèle pour aider à spécifier les architectures logicielles dynamiques. Il semble logique de connaitre la notion de méta-modélisation ainsi que les différents concepts liés. Pour cela nous avons présenté MOF 1.4 l'un des normes de modélisation le plus populaires de l'approche MDA de IDM.

Après nous nous sommes penchés sur les concepts clefs de l'IDM et de MDA, à savoir les modèles et les méta-modèles, il nous a paru évident d'apporter une contribution en créant un méta-modèle qui sert pour aider à spécifier les architectures logicielles dynamiques. Pour cela nous avons proposé notre méta-modèle baptisée ALD qui offre deux méta-modèles permettant décrire le style architectural et la gestion d'évolution des architectures logicielles dynamiques. Enfin, afin valider notre méta-modèle nous avons modéliser une étude de cas selon notre méta-modèle proposé.

Une fois que notre méta-modèle proposé a été définis, nous avons développé un éditeur graphique simple et facile à utiliser. A travers cet éditeur nous pouvons créer des styles architecturaux et de planifier des stratégies d'évolution et des règles d'évolution selon des contraintes architecturales.

Ce projet a fait l'objet d'une expérience intéressante, qui nous a permis d'améliorer nos connaissances et nos compétences dans le domaine de la modélisation et méta-modélisation ainsi que dans le domaine de développement des éditeurs graphiques. Ce dernier, n'a pas été une tâche facile, vu que GMF nécessite beaucoup d'effort pour sa prise en main.

Cependant, au cours de notre développement, nous avons pensé à plusieurs perspectives applicables, comme l'enrichissement de notre méta-modèle pour supporter plusieurs styles architecturaux et en termes d'impacts de changements. Une autre perspective serai de générer le code ADL textuel correspondant à la spécification graphiques à partir de l'éditeur.

Bibliographies

[1] Maarten, B. (1995). The artistry of software architecture. IEEE Software, 12(6) :13–16.

[2] Goaer, O. (2005). De l'adaptation des composants logiciels vers leur évolution. Mastersthesis, Laboratoire d'Informatique de Nantes Atlantique.

[3] Cyril, C. (2003). Contrats Comportementaux pour Composants. Phd thesis, L'école Nationale Supérieure des Télécommunications, Spécialiste : Informatique et Réseaux.

[4] Sylvain, CH. (2010). Extraction d'une architecture logicielle à base de composant depuis un système orienté objet. Phd thesis, Université de Nantes, Spécialiste : Informatique.

[5] Robert, T M., Andrew, K., Ralph, M., & David, G. (1997). Architectural styles, design patterns, and objects. IEEE Transactions on Software Engineering, 14(1) :43–52.

[6] Joëlle, C., & Laurence, N. (2001). Architecture logicielle conceptuelle des systèmes interactifs. Analyse et conception de l'IHM. Hermès, chapitre 7 :207–246.

[7] David, G., Robert, A., & John, O. (1994). Exploiting style in architectural design environments. In SIGSOFT'94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering, pages 175–188. New York, NY, USA.

[8] Andrew, D B., &Bruce J N. (1984). Implementing remote procedure call. ACM Transactions on Computer Systems, 2(1):39–59.

[9] Amjad, U. (1997). Object-oriented client/server Internet environments. Prentice Hall Press, Upper Saddle River, NJ, USA.

[10] Christophe, M. Adaptation d'un langage de description d'architecture à l'expression du comportement applications. ONERA-Supaéro. LIA 10 avenue Edouard Belin 31055 Toulouse cedex.

[11] Abdelmadjid, K., Noureddine, B., & Pierre-Yves, C. (2002). Adaptation dynamique concepts et expérimentations. In ICSSEA'02: Proceedings of the 15t International Conference Software, Systems and their Applications. CNAM Paris, France.

[12] Michel, W. (1999). Specification of software architecture reconfiguration. Phd thesis, Université Nova de Lisbon.

[13] Allen, R., Douence, R. & Garlan, D. (1998). Specifying and analyzing dynamic software architectures. Dans Proceedings of the Conference on Fundamental Approaches to Software Engineering, Lisbon, Portugal.

[14] Joolia, A., Batista, T., Coulson, G., Gomes & A T A. (2005). Mapping ADL Specifications to an Eficient and Reconfigurable Runtime Component Platform. In Software Architecture, 2005.WICSA 2005. 5thWorking IEEE/IFIP Conference on, pages 131_140.

[15] van Ommering, R., van der Linden, F., Kramer, J., & Magee., J. (2000). The Koala component model for consumer electronics software. Computer, vol. 33, no. 3, pages 78_85.

[16] Magee, J., Dulay, N., Eisenbach, S., & Kramer J. (1995). Specifying Distributed Software Architectures, Proc. Fifth European Software Eng. Conf. (ESEC '95).

[17] Bruneton, E., Coupaye, T., & Stefani, J B. (2004). The Fractal Component Model, Version 2.0-3.

[18] Barais, O., & Duchien, L. (2004). SafArch : Maîtriser l'Evolution d'une Architecture Logicielle, In Langages, Modèles et Objets, Journées Composants. Lille, France, Hermès Sciences, pp. 103-116.

[19] Xiangyang, J., Shi, Y., Honghua, C., & Xie, D. (2007). A New Architecture Description Language for Service-Oriented Architec. In Grid and Cooperative Computing, 2007. GCC 2007. Sixth International Conference on, pages 96_103.

[20] Flavio, O. (2008). π-ADL for WS-Composition: A Service-Oriented Architecture Description Language for the Formal Development of Dynamic Web Service Compositions. In Second Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS 2008), pages 1_14, Porto Alegre, Brazil.

[21] David, C L., John, J K., Larry, M A., James, V., Doug, B., & Walter M. (1995). Specification and analysis of system architecture using Rapide. IEEE Transactions on Software Engineering, vol. 21, pages 336_355.

[22] Seza, A., (2016). Describing Dynamic and Variable Software Architecture Based on Identified Services From Object-Oriented Legacy Applications. Software Engineering [cs.SE]. Université Montpellier; Lirmm, University of Montpellier.

[23] Généralités sur la proche MDA. Page consultée le 02 mai 2018, https://laine.developpez.com/tutoriels/alm/mda/generalites-approche-mda.

[24] Ingénierie des Modèles : Méta-modélisation. Page Consultée 04 mai 2018, http://ecariou.perso.univ-pau.fr/cours/idm/cours-meta.pdf .

[25] Meta Object Facility (MOF) Specification, Version 1.4.1. formal/05-05-05. Page Consultée 06 mai 2018, http://www.omg.org/spec/MOF/ISO/19502/PDF.

[26] Blanc, X. (2005). MDA en action : ingénierie logicielle guidée par les modèles. Editions Eyrolles.

[27] Eclipse Modeling Framework. Consulter le 10 mai 2018.https://fr.wikipedia.org/wiki/EclipseModelingFramework #Ecore.

[28] Dayal, U., Buchmman, A., & Mccarthy, D. Rules are objects too: a knowledge model for an active object-oriented database systems. Lecture Notes in Computer Science 334, pages 129–143.

[29] Znn.com example model. Consulter le 28 mai 2018, https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/model-problem-znn-com/ .

[30] Oracle corporation, Java 8. Consulter 02 juin 2018,http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html .

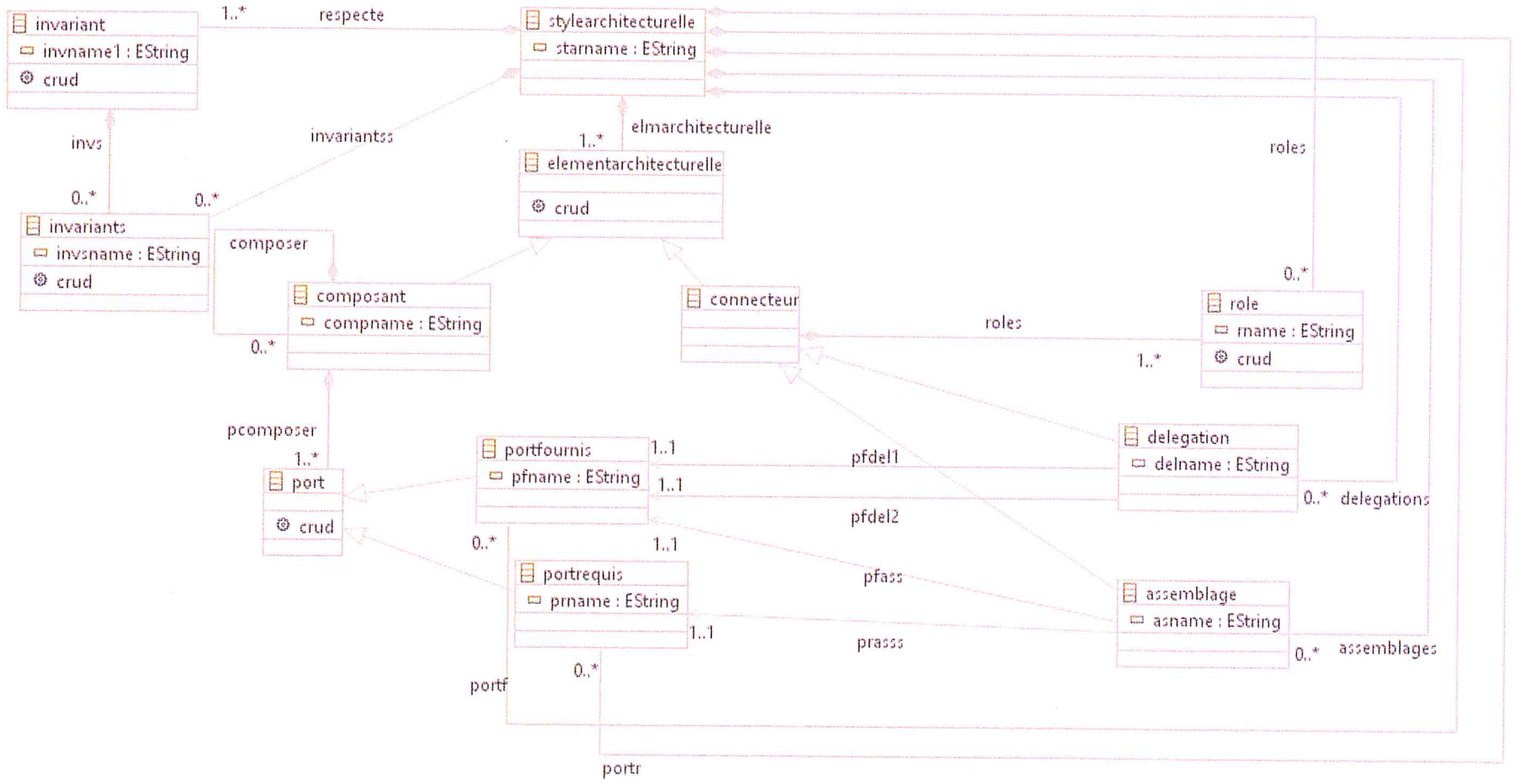
[31] Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). *EMF:* eclipse modeling framework. Pearson Education.

[32] Oracle corporation, Informations sur Java 8. Consulter 03 juin 2018, https://www.java.com/fr/download/faq/java8.xml.

[33] Eclipse Consortium. Eclipse Graphical Modeling Framework (GMF). Consulter 03 juin 2018, https://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial/Part_1.

Annexe A : méta-modèle de l'implémentation

Méta-modèle de style architectural



Annexe A : méta-modèle de l'implémentation

جامعة البليدة - 1 - مكتبة الكلية - كلية العلوم

Méta-modèle de style architectural