MASTER EN SCIENCES INFORMATIQUES

Analyse des besoins d'extensibilité des métaCASE

Bonnoron, Nicolas

Award date:
2009

Facultés Universitaires Notre-Dame de la Paix, Namur

Institut d'Informatique

Année académique 2008 - 2009

Analyse des besoins d'extensibilité des métaCASE

Nicolas Bonnoron

Mémoire présenté en vue de l'obtention du grade de licencié en informatique

Résumé

Ce mémoire analyse les besoins d'extensibilité des métaCASE.

A cette fin, un état de l'art des métaCASE existants est établi et des scénarios d'extensibilité sont mis en place.

Une liste de besoins d'extensibilité est ensuite développée à partir de ces deux aspects.

Puis, l'étude va faire ressortir l'incapacité de l'état de l'art à répondre à ces besoins, et enfin émettre des propositions pour compléter les architectures existantes.

Abstract

This thesis analyses the needs in extensibility of the metaCASE.

For this purpose, an inventory of the current situation of existing metaCASE is established and extensibility scenarios are set up.

From these two aspects, a list of needs for extensibility is developed.

The study will then emphasize on the incapacity of the current situation to respond to those needs and will put forward proposals to complete the existing architectures.

Remerciements

Je tiens à remercier tout particulièrement Vincent Englebert, mon promoteur, pour ses conseils, sa disponibilité et ses encouragements constructifs.

Je remercie aussi Gwenaëlle, mon amie, pour son aide active lors des dernières étapes de la rédaction, son soutien au quotidien, notamment dans la logistique, et son dynamisme stimulant.

Ma famille m'a également soutenu par sa présence réconfortante, ses corrections et sa confiance.

Enfin, mes collègues de la Banque Nationale de Belgique m'ont encouragé par leurs marques d'intérêt.

Table des matières

Première partie

Introduction

Concevoir un logiciel répondant parfaitement aux besoins des utilisateurs : tel est le challenge que les concepteurs s'efforcent de relever.

Cependant, ces besoins sont souvent hétérogènes et peuvent varier d'un utilisateur à l'autre. Ainsi, lors de la création d'un logiciel, les concepteurs doivent limiter l'étendue des besoins qui seront couverts et prendre en compte ceux qui satisferont les exigences du plus grand nombre. Les cas particuliers des différents utilisateurs ne pourront donc pas être considérés.

Comment adapter le mieux possible les logiciels aux différentes exigences des différents utilisateurs ? La solution consiste à permettre l'adaptation ou l'ajout de fonctionnalités par l'utilisateur lui-même.

Pour ce faire, deux possibilités sont envisageables :
- La première est de pouvoir configurer le logiciel afin d'acquérir la fonctionnalité auprès d'un autre logiciel.
- La seconde est de permettre à l'utilisateur d'ajouter la fonctionnalité dans le logiciel même, pour adapter ce dernier à ses besoins. Cette adaptation passe par le remplacement ou l'ajout de briques logicielles, désignées sous le terme de *plugin*.

La capacité du logiciel à être étendu par des plugins et à s'interfacer avec d'autres logiciels s'intitule *extensibilité*.

Tous les logiciels n'ont pas le même besoin d'extensibilité. L'objet de ce mémoire est d'étudier ce besoin dans le cas particulier des métaCASE. Ceux-ci sont des logiciels servant à élaborer d'autres logiciels, en permettant au concepteur non plus de créer en programmant des lignes de code, mais en manipulant les concepts fonctionnels du domaine d'application.

L'analyse des besoins d'extensibilité des métaCASE s'inscrit dans le projet de recherche de Vincent Englebert et de son équipe, qui développent un métaCASE aux Facultés Universitaires Notre-Dame de la Paix à Namur. Ce métaCASE, appelé MetaDone, n'est pas abordé dans ce mémoire.

L'originalité de ce mémoire est d'envisager l'étude du besoin d'extensibilité sous deux approches :
- L'approche "Bottom-Up" consistant à identifier les fonctionnalités que les architectures existantes peuvent mettre en oeuvre.
- L'approche "Top-Down" consistant, à partir de fonctionnalités souhaitées, à identifier les architectures capables de les mettre en oeuvre.

Le déroulement de cette étude se divise en trois parties. Après une explication succincte des outils CASE et métaCASE (première partie), un état de l'art étudie les métaCASE existants et identifie leurs différentes fonctionnalités, en approchant leur architecture en terme d'extensibilité. De ces études individuelles sont ensuite extraits des critères de comparaison transversaux, afin de comparer entre elles les différentes architectures (deuxième partie).

Ensuite, dans une troisième partie, ces fonctionnalités sont complétées par d'autres fonctionnalités souhaitées, à travers la présentation de scénarios d'extensibilité. Ces scénarios découlent des architectures étudiées dans l'état de l'art et d'un apport personnel. Cette étape correspond au point de rencontre des approches "Bottom-Up" et "Top-Down". Finalement, des propositions personnelles sont avancées afin de compléter les architectures existantes et de rendre les scénarios possibles.

Chapitre 1

Outils CASE et MétaCASE

1.1 Les outils CASE

Les outils CASE (*Computer-Aided Software Engineering*) sont des applications permettant la création de logiciels à partir de concepts fonctionnels. L'élaboration d'un logiciel se fait directement avec les concepts du domaine, et non plus en langage purement informatique. Les concepts du domaine sont agencés et paramétrés dans des modèles servant de base à la génération automatique de logiciels finaux. Le passage du modèle fonctionnel vers le code informatique se faisant automatiquement, les erreurs d'interprétation et de transcription manuelle sont évitées. De plus, les modèles sont directement compréhensibles par l'utilisateur final, facilitant son dialogue avec l'informaticien.

Outre la manipulation directe des concepts du domaine, les outils CASE permettent aux entreprises d'automatiser leur production de logiciels, mettant à disposition un environnement de développement adapté au domaine.

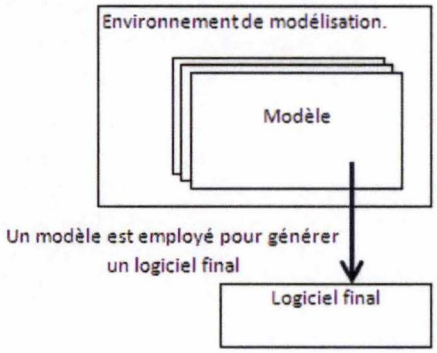

FIGURE 1.1 – Architecture d'un outil CASE

L'emploi des outils CASE a montré son efficacité dans le domaine industriel, où la vitesse de développement de certains logiciels a été multipliée jusqu'à dix fois [29].

Les outils CASE présentent cependant des limitations importantes [29, 63] :

- Ils utilisent des méthodologies figées s'appliquant chacune à un domaine précis, et limitées à produire un certain type de logiciel. Les entreprises qui développent des logiciels sont cependant fort différentes les unes des autres, et évoluent au cours du temps.

- Les concepts du domaine ne sont pas toujours adaptés aux besoins spécifiques d'une entreprise, et ne permettent pas de définir les modèles avec la précision voulue, faute de notations ou de concepts adéquats. Dans ce cas de figure, les utilisateurs finaux peuvent trouver difficile de lire et comprendre les modèles, les concepts employés ne correspondant pas tout-à-fait aux concepts réels.

- Outre les concepts du domaine, l'outil CASE détermine la façon dont un modèle peut être créé, vérifié, et le code final généré. La génération efficace de code à partir des modèles est possible quand les générateurs et le langage employés correspondent aux besoins de l'entreprise, et produisent du code compatible avec leur architecture, leurs protocoles etc. Les générateurs de code des outils CASE standards tentent de rencontrer la majorité des besoins, et produisent ainsi soit des squelettes de programmes, soit du code lourd et peu performant.

Les outils CASE ne fournissent, au final, pas de solution réaliste et suffisamment flexible pour s'adapter à l'évolution des entreprises et de leurs besoins.

1.2 Les métaCASE

Références bibliographiques : [12, 13, 19, 29, 61, 63, 65]

Les métaCASE sont une évolution des outils CASE, et donnent la solution au manque de flexibilité de ces derniers : ils permettent la personnalisation de la méthode de modélisation. Celle-ci peut être librement modifiée, et même être remplacée par une nouvelle.

Les métaCASE présentent une architecture à deux niveaux, illustrée dans la figure 1.2 :

- Le niveau de modélisation, également présent dans les outils CASE, fournit un environnement pour manipuler les concepts d'un domaine précis et les agencer, pour créer des modèles et en générer des logiciels automatiquement.

- Le niveau de métamodélisation fournit un environnement pour définir des domaines, aussi appelés métamodèles. Un métamodèle définit les règles de syntaxe, les règles sémantiques, les concepts du domaine, les relations qui peuvent se nouer entre ces concepts, la façon dont ils seront représentés dans les modèles. Un métamodèle peut par exemple définir des concepts comme classe et héritage, définir comment ils sont reliés et comment ils sont représentés. Il définit également la manière dont le code final est généré. Dans le cas d'un outil CASE, ce métamodèle est unique et ne peut pas changer : le domaine est définit une fois pour toute. Le métaCASE, quant à lui, permet de définir autant de métamodèles que souhaité, et ainsi définir plusieurs domaines et les faire évoluer. Chaque métamodèle sert à générer un environnement (niveau "Modèle") dans lequel des modèles pourront être créés à l'aide des concepts du domaine.

Pour permettre la définition de domaines variés, le métaCASE doit disposer d'un langage suffisamment riche. Ce langage est la partie fixe du métaCASE, et est utilisé pour définir les métamodèles. La littérature emploie parfois les termes méta-langage ou "méta-métamodèle", c'est-à-dire modèle servant à modéliser des métamodèles.

Les métaCASE permettent donc de réaliser des modèles avec des concepts spécifiques à l'entreprise, et de générer du code suivant ses standards.

Exemple de création d'un logiciel à l'aide d'un métaCASE

La figure 1.3 montre la conception d'un logiciel représentant un automate fini [61].

Le développeur définit tout d'abord le métamodèle, en décrivant :

- les différents concepts d'un automate fini : état initial, état terminal, état intermédiaire, transition ;
- la représentation des concepts : un état est représenté par un rond, une transition par une flèche ;
- la façon dont s'agencent les concepts : deux états peuvent être reliés par une transition, etc ;
- le générateur de code.

Le métaCASE génère ensuite l'outil CASE sous-jacent, à partir du métamodèle. Le développeur y manipule les concepts de l'automate fini, et crée le modèle d'un automate fini particulier.

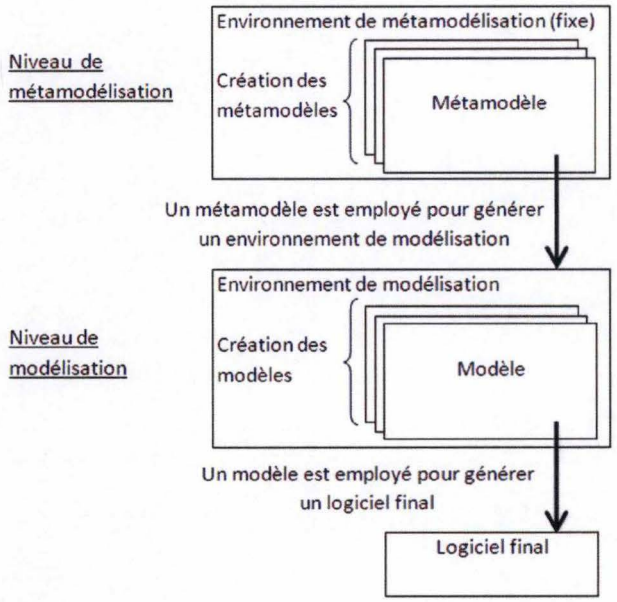

FIGURE 1.2 – Architecture d'un métaCASE

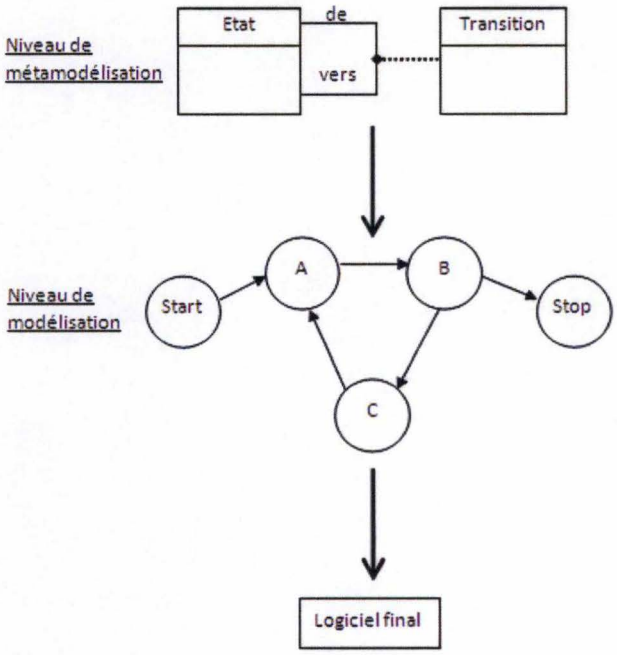

FIGURE 1.3 – Conception d'un automate fini avec un métaCASE

L'outil CASE génère finalement le logiciel représentant cet automate fini. Le générateur de code employé est celui défini dans le métamodèle.

Deuxième partie

Etat de l'art

Chapitre 2

Etude de métaCASE et architectures existantes

Ce chapitre passe en revue cinq logiciels existants, en envisageant leurs architectures en terme d'extensibilité. L'étude de ces architectures permet d'identifier un premier panel de besoins d'extensibilité, ainsi que les fonctionnalités permettant de les mettre en oeuvre.

Protégé, GME, MetaEdit+ et le Meta-Environment sont les quatre premiers logiciels étudiés, et sont des métaCASE. Le cinquième, Eclipse, n'est pas un métaCASE mais présente une architecture élaborée, basée sur les plugins.

2.1 Protégé

2.1.1 Présentation générale

Protégé est un logiciel de gestion de bases de connaissances développé à l'université de Stanford en Californie, dans le département de recherche informatique biomédicale [53]. Son but premier est la modélisation de larges bases de connaissances médicales, pour répertorier les pathologies et aider à la pose de diagnostics [36].

Développé en Java, gratuit et open source, il peut tourner sur une grande variété de plateformes et son interface peut être étendue par des extensions sur-mesure. Il peut interagir avec des repositories standards tels que des bases de données relationnelles et XML. La communauté des utilisateurs de Protégé compte plus de 100.000 membres et il a déjà été employé par des centaines de groupes de recherche.

Protégé modélise ses bases de connaissances sous forme d'ontologies : une ontologie organise un domaine de connaissances sous forme d'entités (les objets du domaine), structurées par des relations. Les taxonomies, les bases de données, les catalogues de produits en ligne (Amazone), les annuaires (catégories Yahoo !) sont des exemples d'ontologies.

L'exemple suivant présente une ontologie simplifiée décrivant les concepts entrant en jeu dans la conception de cartes électroniques [67] :

- Une carte électronique est un ensemble de composants.
- Un composant peut être soit un condensateur, soit une résistance, soit une puce.
- Une puce peut être soit une unité de mémoire, soit une unité de calcul.
- Une carte électronique qui contient une unité de calcul contient aussi au moins une unité de mémoire.

Protégé contient un éditeur d'ontologies, qui permet de décrire un domaine donné. Les instances de cette ontologie sont acquises grâce à des formulaires personnalisables [43]. Un grand nombre de plugins permet d'étendre Protégé, pour par exemple raisonner sur les connaissances, les visualiser de différentes manières, en déduire de nouveaux faits par application de règles d'inférences , etc.

2.1.2 L'architecture de plugins

Protégé a une architecture modulaire et extensible. Son comportement et ses fonctionnalités peuvent être modifiés et étendus. Il est entre autres possible de configurer l'interface utilisateur, ce qui facilite beaucoup l'acquisition de connaissances pour des domaines précis.

Cette flexibilité est rendue possible grâce à l'architecture de plugins de Protégé. Protégé est lui-même écrit comme une collection de plugins et peut être complété. Les plugins natifs de Protégé peuvent également être remplacés. L'interface et le comportement de Protégé peuvent ainsi être complètement personnalisés.

Les types de plugins

Il existe six types techniques de plugins, chacun remplissant un rôle différent [44] :

1. Le plugin de type "Tab widget" est un tab d'interface utilisateur. Il est affiché parmi les tabs standards qui constituent l'interface utilisateur de Protégé, et sert de conteneur pour des applications personnalisées (voir figure 2.1). Ces applications bénéficient d'une connexion vers la base de connaissance, et sont partie intégrante de l'environnement d'édition.

Si la base de connaissance contient par exemple des vins et des aliments, ainsi que les relations appropriées entre les vins et les aliments, on peut imaginer une application qui donne une suggestion de vin pour un menu dans lequel intervient différents aliments. Cette application fera évidemment usage de la base de connaissance, mais aura aussi sa propre logique pour analyser les différentes suggestions. Elle aura sa propre interface utilisateur, mais sera complètement intégrée dans Protégé et pourra interroger et modifier la base de connaissances [38].

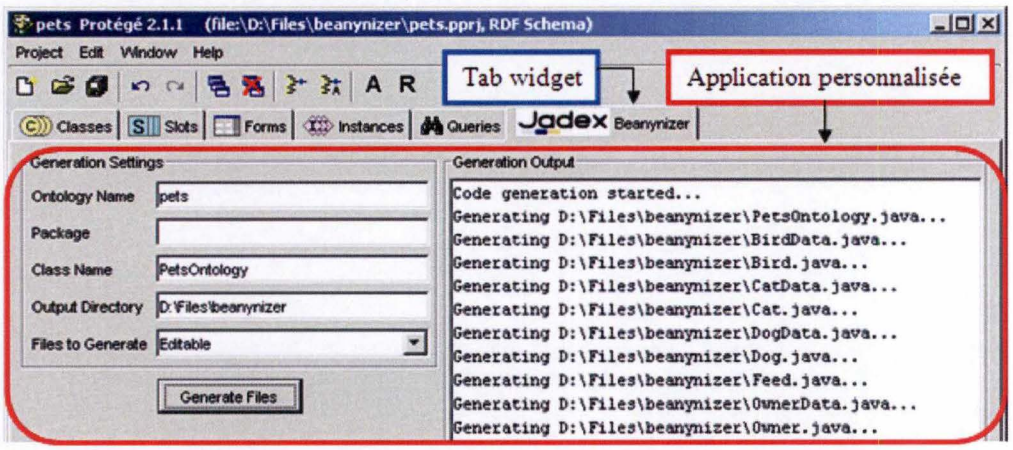

FIGURE 2.1 – Le plugin tab widget

On trouve dans la documentation [43] un classement en deux catégories plus fonctionnelles des tab widgets : la catégorie "utility functions" et la catégorie "end-user applications". La distinction majeure se situe au niveau de l'interaction avec d'autres applications : les tabs "end-user applications" restent au niveau de Protégé, tandis que les "utility functions" interagissent avec des applications externes.

- Les plugins "end-users applications" servent à acquérir et manipuler des ontologies et des bases de connaissances directement dans Protégé. Ce sont par exemple :
  - des formulaires permettant l'introduction de données, leur vérification et leur sauvegarde dans une base de connaissances ;
  - des graphiques, des tableaux permettant de visualiser le contenu d'une base de connaissances ;
  - des applications qui manipulent les données d'une base de connaissances, raisonnent dessus, en tirent des conclusions, des alertes, ...

- Les plugins "utility functions" servent à s'interconnecter vers des bases de connaissances externes. Les utilisateurs de Protégé peuvent donc, via ces plugins, consulter et importer des arbres de connaissances à partir de bases de connaissances distantes. Pour permettre la communication avec l'application distante (qui expose par exemple des services web ou une API java), le plugin doit jouer le rôle d'interface entre celle-ci et Protégé. Protégé peut ainsi être en interconnexion directe avec d'autres applications, d'autres bases de connaissances.

2. Le plugin de type "Slot widget" sert à acquérir des données d'un certain type (par exemple une date, un texte) et à les valider. La validation se fait en accord avec les définitions des données

dans l'ontologie (les "slots"). Il est intéressant de pouvoir définir ses propres "slot widgets", ce qui permet d'adapter la saisie à des domaines spécifiques ; certaines connaissances sont parfois plus faciles à encoder ou à représenter sous forme de graphiques, de tableaux, etc.

La figure 2.2 montre deux slot widgets pour acquérir une teneur en sucre [38]. L'un présente une liste déroulante avec plusieurs valeurs (non sucré, peu sucré, ..., fort sucré), tandis que l'autre présente une aiguille mobile. Le choix de l'un ou l'autre widget dépend du domaine d'application.

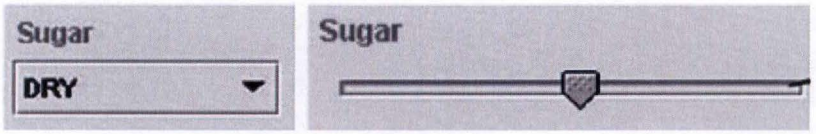

FIGURE 2.2 – Deux plugins slot widget

3. Le plugin de type "Back-end" [38] permet de définir de nouveaux formats de sauvegarde, en plus du format texte par défaut de Protégé. Le plugin back-end va assurer le mapping entre le format interne d'une base de connaissances Protégé avec le format externe souhaité. Le nouveau format de fichier a le même statut que le format natif de Protégé, et un utilisateur peut au choix opter pour l'un ou l'autre format de sauvegarde. On peut ainsi rendre Protégé compatible avec d'autres applications.

4. et 5. Les plugins de type "Createproject" et "Export" sont des variantes du type back-end. Ils sont plus légers et destinés à exporter des bases de connaissances dans des formats utilisables par d'autres applications, sans souci d'optimisation des performances ni d'exhaustivité (tous les éléments de Protégé qui ne sont pas supportés par le format cible sont perdus). A l'inverse, un type back-end est destiné à sauver la totalité de la base de connaissances. En pratique, les plugins de type Export/CreateProject sont plus faciles à implémenter que ceux de type Back-End. Le lecteur peut se référer au site de Protégé où un comparatif entre ces trois types est donné [47].

6. Le plugin de type "Project" est appelé à différents moments du cycle de vie d'un projet Protégé. Ce type de plugin est peu documenté [48].

Créer un plugin d'un certain type indique à Protégé où il doit l'incorporer dans l'interface utilisateur et à quel moment il doit l'appeler. Par exemple, un plugin de type tab widget sera incorporé dans la fenêtre principale de Protégé au démarrage de l'application ; un nouveau format défini par un plugin back-end sera ajouté aux différents formats de sauvegarde disponibles.

Chaque plugin est chargé par un chargeur de classes Java [54]. Ces chargeurs de classes sont des objets Java responsables du chargement des classes dans l'environnement d'exécution du programme. Ils localisent le code de la classe, et le chargent pour qu'il puisse être exécuté. Habituellement, les fichiers de classe se trouvent sur le file system local de la machine, dans un ou plusieurs répertoires définis dans une variable d'environnement appelée le classpath. Cependant, certaines classes peuvent provenir d'autres endroits, par exemple n'exister qu'en mémoire car construites dynamiquement par une application, ou se trouver sur un réseau (cas des applets Java). Il faut alors employer des chargeurs de classes appropriés pour les charger.

A la lecture de la documentation, Protégé ne permet apparemment que de charger des plugins locaux, dont les classes sont stockées dans le répertoire /plugin, ce qui exclus les plugins distants.

Dépendances entre plugins

Il est possible de développer un plugin qui dépend d'autres plugins Protégé [50]. Il suffit d'indiquer dans un fichier de configuration (plugin.properties) quels sont les plugins dont il est dépendant. Un plugin A peut accéder aux méthodes publiques d'un plugin B si B est référencé dans le plugin.properties de A (le "classpath" de A). Ce système permet à un plugin d'utiliser des services offerts par d'autres plugins. Par exemple un plugin de type tab-widget pourrait instancier un plugin de type slot-widget, pour l'afficher dans son interface.

Exemple de plugin : le tab widget

L'architecture de plugins de Protégé est propriétaire et n'est pas détaillée dans la documentation. Toutefois, tous les plugins sont écrits en Java, et sont des sous-classes de l'API java de Protégé.

Le plugin tab widget est le type le plus simple. Il a deux exigences [49] :
- avoir pour parent la classe `AbstractTabWidget` ;
- implémenter la méthode `initialize()`.

Le code du tab widget de la figure 2.3 [52] affiche une information provenant de la base de connaissances dans l'interface utilisateur.

```
public class FrameCounter extends AbstractTabWidget {
    private JTextField field = new JTextField(10);
    public void initialize() {
        // initialisation du label du tab
        setLabel("Frame Counter");
        setIcon(Icons.getInstanceIcon());
        // récupère une information dans la base de connaissances
        int count = getKnowledgeBase().getFrameCount();
        field.setText(String.valueOf(count));
        // ajoute la zone de texte dans l'interface du tab
        setLayout(new FlowLayout());
        add(field);
    }
}
```

FIGURE 2.3 – Le code d'un plugin tab widget

L'intégration du plugin dans Protégé est simple :
- Une fois compilées, les classes doivent être copiées dans un sous-répertoire du répertoire /plugin de Protégé.
- La classe principale du plugin doit être référencée dans le fichier manifest de Protégé. Dans notre exemple, les deux lignes suivantes sont ajoutées :

```
Name: examples/tabwidget/FrameCounter.class
Tab-Widget: True
```

– Au démarrage suivant de Protégé, le plugin est disponible moyennant une activation via le menu Configuration.

Cet exemple montre un élément important : le label et l'icône du tab sont définis dans le code source du plugin. La classe principale du plugin est donc chargée en mémoire dès le démarrage de Protégé, et sa méthode initialize() exécutée.

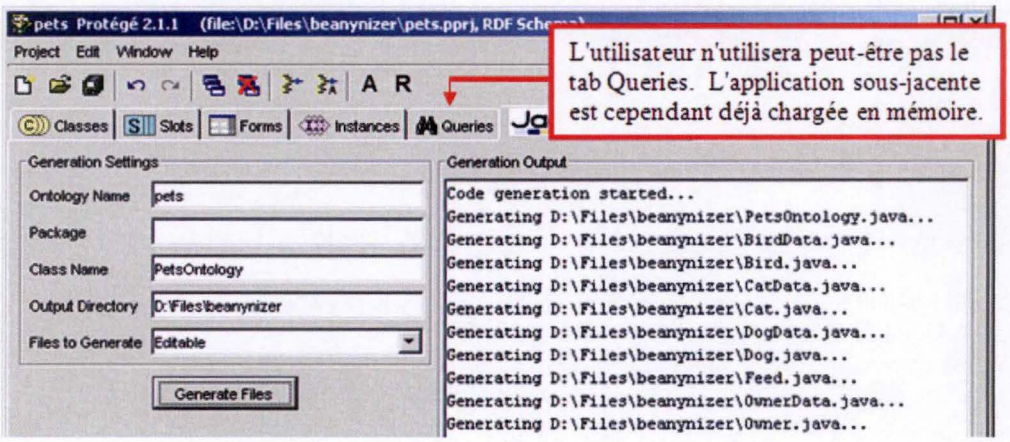

FIGURE 2.4 – Les plugins sont chargés en mémoire dès le démarrage de Protégé

A la lecture de la documentation, il semble que les plugins ne peuvent pas être chargés lorsque Protégé est déjà démarré. Tous les exemples disponibles sur le site web de Protégé demandent un redémarrage de l'application.

2.1.3 L'API

Protégé dispose d'une API (Application Programming Interface) packagée sous forme d'une archive java (fichier JAR), servant à accéder aux bases de connaissances. Protégé lui-même utilise cette API pour y accéder. De même, les plugins développés pour Protégé sont des sous-classes de cette API. Elle peut également être utilisée directement par des applications Java externes pour accéder aux bases de connaissances, sans démarrer le client Protégé même [46].

L'API permet de consulter et de modifier les bases de connaissances. Elle peut aussi générer des formulaires pour l'introduction de données. Elle permet apparemment d'accéder des bases de connaissances distantes : la javadoc de l'API [51] montre que la localisation d'une base de connaissance peut être donnée sous la forme d'une URI [45], laissant supposer un accès distant possible.

Exemple d'usage de l'API

L'API permettant à des applications externes de se connecter à la base de connaissances, toutes sortes de modèles peuvent être imaginés. Ainsi, on peut par exemple concevoir une petite application, externe à Protégé, qui expose des services web au monde extérieur et traduit les requêtes web reçues en appels vers l'API de Protégé [10]. Ce modèle rend l'API de Protégé interopérable avec des applications qui ne sont pas écrites en Java.

Cette API rend Protégé flexible, pour autant qu'on programme les interfaces appropriées.

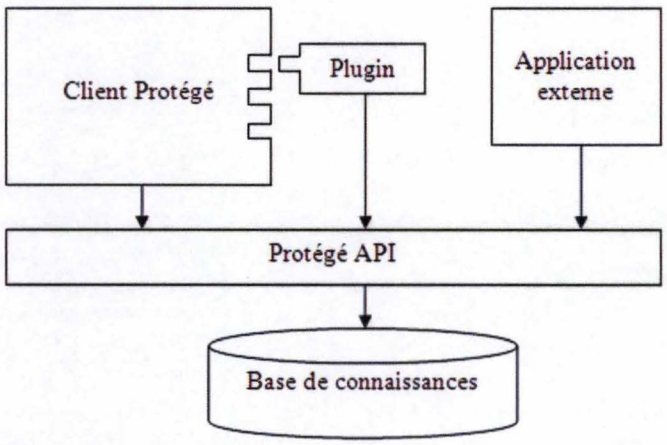

FIGURE 2.5 – Architecture de Protégé

2.2 GME

2.2.1 Présentation générale

GME (Generic Modeling Environment) [59] est un métaCASE gratuit et open source, développé à l'université Vanderbilt au Tennessee.

GME est composé de deux grands modules, GMeta (aussi appelé Meta-GME) et GModel (aussi appelé GME) :

1. GMeta permet de construire des méta-modèles qui décrivent chacun un domaine précis. Ces méta-modèles pourront être chargés dans GModel. Le méta-langage utilisé est basé sur la notation du diagramme de classes UML, et complété par des contraintes OCL (Object Constraint Language [37, 66]).
2. GModel est l'environnement de modélisation, qui est configuré et adapté à partir des méta-modèles définis dans GMeta. Ces méta-modèles sont chargés dans GModel, et mettent à disposition tous les concepts et relations qui peuvent être employées pour construire les modèles spécifiques au domaine. Ces modèles servent, in fine, à générer automatiquement des applications.

2.2.2 Microsoft COM

Références bibliographiques : [33, 34, 42, 64]

L'architecture extensible et modulaire de GME est basée sur Microsoft COM (Component Object Model) [59]. COM est une architecture mature permettant à des applications d'utiliser des composants fournis par différents vendeurs. Elle est très largement employée dans les applications Windows, et des milliers de composants COM sont disponibles et utilisables pour créer rapidement des applications élaborées.

Les composants COM peuvent être de trois types [42] :
- In-process : s'exécutent dans le même processus que l'application qui les utilise, et sont implémentés sous forme de librairies dll (dynamic linked library).
- Local : s'exécutent sur la même machine mais dans un processus différent.
- Remote : s'exécutent sur un ordinateur distant.

Une application qui utilise un composant COM n'a pas besoin de savoir de quel type est ce composant. Distant ou non, l'accès au composant est géré par l'infrastructure COM et est transparent pour l'application. L'instanciation d'un composant se produit quand l'application décide de l'accéder.

L'interface d'un composant COM est décrite dans un langage générique : IDL (Interface Definition Language). Avec IDL, le développeur d'un composant COM peut décrire les interfaces, méthodes et propriétés qui sont supportées par son composant. Les applications clientes se réfèrent à la définition IDL du composant COM sans se soucier des détails spécifiques à l'implémentation du composant, comme son langage de programmation.

Les composants externes peuvent être écrits dans tout langage qui supporte COM, dont C++, Visual Basic, C#, Python et Java. L'usage de COM en Java nécessite cependant d'utiliser la Virtual Machine de Microsoft, qui n'est supportée que sous Windows.

Il existe une portabilité de COM sous Linux et MAC, mais dans les faits l'interopérabilité n'est pas garantie.

2.2.3 L'architecture de GME

GME est écrit sous forme de composants qui communiquent entre eux via des interfaces COM. Tous ces composants internes sont accessibles par des composants externes, qui peuvent accéder à toutes leurs fonctionnalités, et réaliser tout ce qu'il est possible de faire à travers l'interface utilisateur de GME [24].

La figure 2.6 montre l'architecture de GME. Les traits noirs terminés par un rond blanc symbolisent chacun une interface COM. GModel et GMeta sont les deux composants principaux, qui permettent la construction des modèles et méta-modèles. GModel utilise les services de GMeta pour obtenir les méta-spécifications sur lesquelles construire ses modèles. Outre l'usage direct de COM, les composants

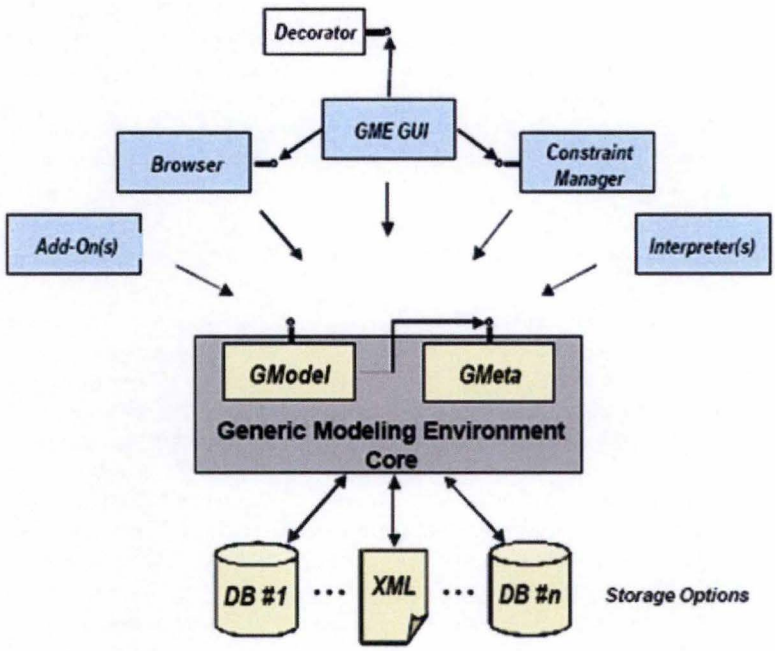

FIGURE 2.6 – Architecture de GME

externes écrits en C++ peuvent utiliser l'interface BON (Builder Object Network) mise à disposition par GME [58]. Cette interface écrite en C++ est basée sur COM, et permet aux composants externes de ne plus se soucier de la mise en oeuvre de COM. Une version Java du BON existe également, mais n'est utilisable qu'avec la Virtual Machine de Microsoft.

Cette architecture permet d'envisager quelques scénarios d'extension de GME :

- avoir une application externe qui gère ses propres traitements mais qui affiche ses résultats dans l'interface utilisateur de GME [57, 59] ;
- avoir une application externe qui est une nouvelle interface utilisateur pour GME, et affiche les modèles créés par GMeta et GModel ;
- avoir un composant qui communique avec GMeta et GModel, et qui affiche ses résultats dans l'interface utilisateur de GME.

Tous ces composants restent complètement indépendants de GME.

2.2.4 Les composants de GME

La documentation distingue trois types de composants qui contribuent à la flexibilité de GME [58]. La différence entre les deux premiers types de composants est purement fonctionnelle. Le troisième type diffère quant à lui par l'utilisation d'événements. On distingue :

- les interpréteurs, qui utilisent les interfaces de GModel et GMeta pour accéder aux modèles, à leurs attributs et à leurs relations. Les interpréteurs "interprètent" les modèles pour générer des schémas de base de données, du code source, etc.
- les plugins, qui fournissent des fonctionnalités additionnelles et facilitent le travail dans GME. Ils sont très similaires aux interpréteurs, mais ne sont pas dédiés à l'analyse des modèles : ils peuvent ajouter de nouvelles fonctionnalités pour manipuler les modèles, les visualiser, faire des recherches, etc.
- les add-ons[24, 59], qui réagissent aux événements émis par GModel et GMeta. GModel et GMeta exposent une série d'événements, comme "Objet supprimé", "Attribut modifié", etc. Les composants externes peuvent s'enregistrer auprès de GModel et GMeta pour capter un ou des événements. Ils sont automatiquement appelés quand l'événement se produit. Les add-ons sont très utiles pour étendre les fonctions de l'interface utilisateur : un add-on peut par exemple réagir à chaque fois qu'un attribut est ajouté par l'utilisateur, et vérifier que cet attribut répond à certains critères (nom, type, etc).

  Un exemple d'add-on est le vérificateur de syntaxe OCL, développé par l'équipe de GME [25] :

  Lors de l'élaboration d'un méta-modèle, l'utilisateur peut définir des contraintes OCL qui devront être respectées par les modèles dérivés de ce méta-modèle. Cependant, il se peut que ces contraintes définies dans le méta-modèle contiennent des erreurs de syntaxe. Comme la version de base de GME ne vérifie pas cette syntaxe, ces erreurs passent inaperçues lors de la définition du méta-modèle, et apparaissent seulement lors de leur exécution dans les modèles dérivés du méta-modèle. Pour intercepter ces erreurs de syntaxe lors de la définition de la contrainte OCL, l'add-on vérificateur de syntaxe s'enregistre auprès de l'événement "onChange" de l'expression OCL. L'add-on est ainsi averti dès que l'expression est changée, et peut ainsi vérifier sa syntaxe et donner un feedback à l'utilisateur.

  La documentation ne donne pas d'informations sur la synchronisation des événements. Si le système est synchrone, GME attend que l'add-on ait traité l'événement pour continuer son process ; s'il est asynchrone, l'événement est transmis et GME continue son traitement sans attendre de réaction de l'add-on. Un système asynchrone cause moins de ralentissements et est plus performant. Il peut être construit au-dessus d'un système synchrone, mais demande une rigueur dans la programmation des add-ons : le traitement de chaque événement reçu par l'add-on doit être délégué dans un thread asynchrone.

Dans notre optique, ces 3 types de composants sont des plugins. Ils permettent d'interagir avec GME, modifier ses données, son interface, réagir à des événements.

La documentation ne renseigne pas la façon dont les composants exposent leur interface utilisateur dans GME. Un exemple donné dans la documentation de GME [60] montre un interpréteur affichant son résultat dans un popup.

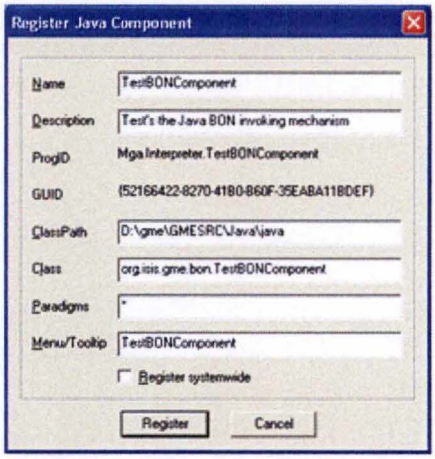

FIGURE 2.7 – Enregistrement d'un composant dans GME

Les décorateurs graphiques

Le dessin d'un modèle (ou d'un méta-modèle) dans GME se fait grâce à des composants appelés décorateurs graphiques [35]. Les décorateurs graphiques sont des plugins, au sens donné par la terminologie des composants GME (page 22). Il existe des décorateurs par défaut, qui peuvent être remplacés par des décorateurs sur-mesure. La visualisation graphique peut alors différer d'un modèle à l'autre, et être en accord avec le domaine représenté par le modèle. Par exemple, dans un modèle hydraulique, deux objets "vannes" pourraient être visuellement interconnectés par le dessin d'un tuyau à la place d'une simple flèche.

Un décorateur graphique doit implémenter l'interface COM `IMgaDecorator`, définie dans le fichier IDL MgaDecorator.idl. Cette interface impose l'implémentation de la fonction `draw()`, qui sera appelée par l'élément décoré. Les éléments d'un modèle qui ont un pendant graphique présentent tous un attribut `decorator`, dans lequel on peut spécifier le décorateur choisi ainsi que ses paramètres. Les paramètres du décorateur servent par exemple à déterminer la couleur ou la taille du graphique.

Un décorateur graphique pourrait réagir à des événements ; rien ne l'empêche d'implémenter plusieurs interfaces COM et de s'enregistrer auprès de GModel pour capter ses événements. L'apparence du graphique qu'il dessine pourrait varier en fonction d'événements extérieurs, comme faire passer la couleur d'un tuyau du bleu au rouge si la source d'eau à laquelle il est relié passe du statut eau froide vers le statut eau chaude.

Le mécanisme des décorateurs graphiques permet de personnaliser l'affichage grâce à des composants externes. Si GME ne proposait pas ce système, on pourrait néanmoins toujours créer un composant qui dessine lui-même toute l'interface représentant le modèle, puisque les composants ont accès aux modèles et qu'ils peuvent réagir à des événements. C'est évidemment beaucoup plus lourd que de créer un simple décorateur.

Enregistrement des composants

Les interpréteurs et les plugins à inclure dans l'interface de GME doivent être enregistrés via un outil de GME (figure 2.7) [60]. Une fois enregistrés, ils sont disponibles dans l'interface, dans une liste qui reprend les composants [58]. Pour les add-ons, la documentation [58] renseigne qu'ils sont associés à un projet (les modèles développés dans GME sont repris dans des projets, sortes d'unités de classement).

Elle ne renseigne pas la façon dont ils sont associés au projet, mais il paraît plausible que cela se fasse d'une façon analogue à celle des interpréteurs et plugins. Interpréteurs, plugins et add-ons peuvent être des composants complètement externes, alors démarrés par leur propre applicatif.

Les décorateurs graphiques sont enregistrés en les renseignant dans l'attribut `decorator` de l'élément à décorer.

2.3 MetaEdit+

2.3.1 Présentation générale

MetaEdit+ est un métaCASE développé par la société MetaCase, en Finlande [31].
Il a été employé avec succès par plusieurs entreprises, dont Nokia, Siemens et Panasonic [32].
Son architecture est propriétaire et multiplateformes.

2.3.2 Les services web

MetaEdit+ ne dispose pas d'une architecture de plugins ; il n'y a donc pas moyen de l'étendre avec de nouvelles fonctionnalités.

Il dispose cependant d'une API [30], basée sur les services web, permettant à des outils externes d'interagir avec lui. Le concept de service web [62, 68] signifie qu'un serveur propose des fonctions (les services) qui peuvent être appelées à distance à travers le web, afin de déclencher un traitement sur le serveur distant et/ou obtenir une information de ce serveur.

Les services web ne sont qu'une des multiples variantes d'appels de procédures distantes, déjà implémentés dans des standards middelwares comme CORBA ou COM.

La nouveauté apportée par les services web est que ces appels de procédures distantes se font comme des appels de page web, c'est-à-dire sur le protocole HTTP. "Je demande une URL, je reçois une page" devient "Je demande un service, je reçois une réponse". La grande force des services web est d'utiliser des standards ouverts et reconnus (HTTP, XML). L'utilisation de ces standards permet d'écrire des services web dans plusieurs langages et de les utiliser sur des systèmes d'exploitation différents. Un autre avantage d'employer HTTP est de ne pas nécessiter l'ouverture de ports spécifiques sur les firewalls, ce qui est généralement apprécié dans une entreprise.

Le service web et le programme qui l'invoque sont légèrement couplés, et peuvent être modifiés indépendamment l'un de l'autre. Contrairement à une composante logicielle qui serait fortement couplée, il n'est pas nécessaire de connaître la machine, le langage, le système d'exploitation ou tout autre détail, pour qu'une communication puisse avoir lieu. De plus, la localisation du service web a lieu lors de l'exécution du programme.

La mise en oeuvre des services web peut s'appuyer sur différents standards (SOAP, XML-RPC, etc). La combinaison la plus courante est :

- le SOAP (Simple Object Access Protocol) pour l'échange de messages ;
- le WSDL (Web Service Description Language) pour la description des services web, leur localisation, leurs opérations, les messages utilisés, les types de données utilisées, les protocoles utilisés ;
- les annuaires UDDI (Universal Description, Discovery and Integration) pour la publication des descriptifs de services web (matérialisés par les fichiers wsdl).

Le scénario de base pour une application qui désire appeler un service web est le suivant :

- connaissant le nom logique du service web, l'application interroge l'annuaire UDDI ;
- l'annuaire renvoie le fichier wsdl associé au service web ;
- l'application y trouve toutes les informations nécessaires pour accéder le service : l'URL du service, son protocole de communication (ex : SOAP) etc ;
- l'application prépare une requête HTTP à destination du service, avec le message SOAP comme charge utile.

2.3.3 L'API de MetaEdit+

MetaEdit+ met en oeuvre ses services web en se basant sur SOAP et WSDL [27]. Il n'y a pas d'annuaire UDDI, l'unique fichier wsdl qui reprend le descriptif de tous les services web de l'API est disponible directement dans MetaEdit+.

Pour accéder l'API, il faut tout d'abord démarrer le serveur web de MetaEdit+ [27]. L'application externe, implémentant un client SOAP, s'occupera d'établir la connexion avec le serveur et d'accéder les services web.

L'API permet à une application externe de créer, mettre à jour, lire et manipuler les modèles de MetaEdit+ [30]. Certaines fonctionnalités de MetaEdit+ ne sont disponibles qu'à travers son API, comme animer en temps réel des diagrammes pour effectuer des simulations ou du debuggage visuel [30, 56]. Par exemple, une application développée à partir d'un modèle de MetaEdit+ peut, lors de son exécution, montrer la séquence d'actions qu'elle suit en animant le modèle sur lequel elle a été construite.

La communication entre une application externe et MetaEdit+ ne va cependant que dans un seul sens : l'application doit prendre l'initiative de la communication. MetaEdit+ se contente de répondre à des instructions, mais ne transmet de lui-même aucune information et n'émet aucun événement.

Les générateurs de code

MetaEdit+ permet de générer du code à partir des modèles [28]. Un éditeur de générateurs est disponible dans MetaEdit+, et permet de définir des générateurs pour n'importe quel langage de programmation.

La documentation de MetaEdit+ mentionne également qu'il est possible d'intégrer son propre générateur [28]. Cela se fait cependant uniquement via l'API, qu'un générateur externe pourra employer pour accéder au descriptif des modèles. Le générateur n'est donc pas intégré à MetaEdit+ comme on aurait pu l'espérer.

MetaEdit+ permet aussi d'exporter ses modèles sous forme de documents XML [30]. Avec le parser adéquat, un générateur externe pourra extraire le modèle et générer le code ad hoc [28].

Un problème de l'API

Un problème de l'API [56] est que ses méthodes n'offrent pas d'accès direct aux objets et aux modèles. Retrouver les propriétés d'un objet implique le scénario suivant :

1. un appel à l'API pour récupérer la référence de l'objet ;
2. un appel à l'API pour récupérer les références des propriétés de l'objet ;
3. un appel à l'API pour chaque propriété dont on veut récupérer la valeur.

On obtient donc vite un grand nombre d'appels à l'API, ce qui augmente la taille du programme et ralentit son exécution. Ceci est d'autant plus sensible que les services web sont une technologie d'appel distant plus lente que d'autres standards comme COM ou CORBA, à cause du protocole HTTP.

2.4 Meta-Environment & Toolbus

2.4.1 Présentation générale

Le Meta-Environment [2] est un outil servant à créer des langages de programmation et à manipuler du code source. Il est développé par l'équipe de recherche de Paul Klint [20], au CWI Centrum Wiskunde & Informatica aux Pays-Bas, et est soutenu par une communauté Open Source.

Le Meta-Environment sert à [9] :
- concevoir et implémenter des langages de programmation ;
- analyser, transformer, générer du code source ;
- générer de la documentation à partir de code source ;
- extraire une description formelle de la syntaxe et de la sémantique de langages de programmation
- générer des environnements de programmation à partir de ces descriptions formelles, etc.

Il est composé d'outils d'analyse syntaxique, sémantique, d'outils de transformation et d'un environnement de développement interactif.

2.4.2 L'architecture du Meta-Environment

Le Meta-Environment est un logiciel composé d'outils qui communiquent entre eux via un bus logiciel : le Toolbus. Le Toolbus a été développé pour répondre à trois problématiques majeures des

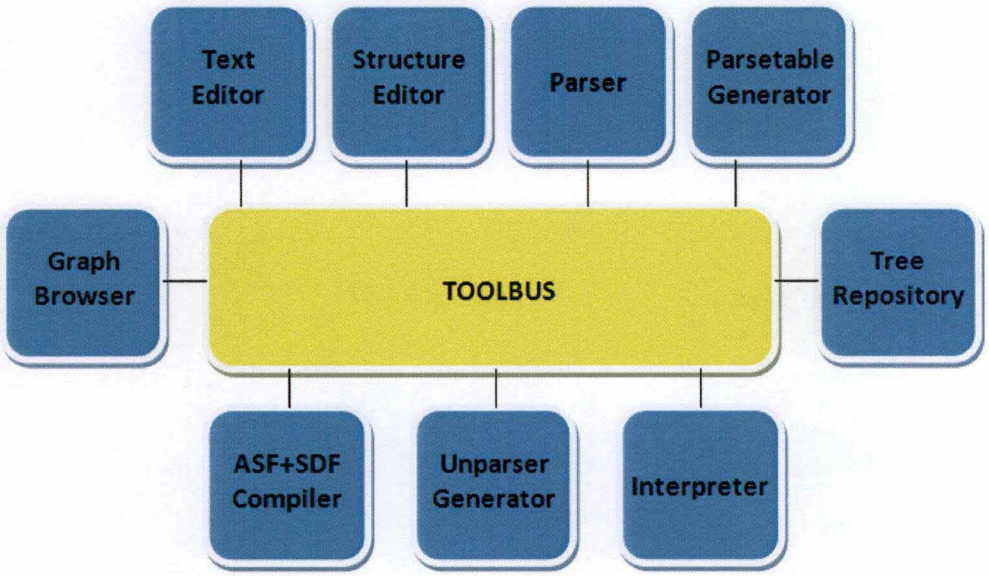

FIGURE 2.8 – Architecture du Meta-Environment [9]

systèmes actuels : leur taille, leur hétérogénéité, leur fractionnement sur réseau [18].

- La taille : les systèmes deviennent de plus en plus larges parce que la complexité des tâches à automatiser augmente.
- L'hétérogénéité : les systèmes larges peuvent être construits en réutilisant des composants existants, potentiellement écrits en différents langages et pour des plateformes différentes.

- Le fractionnement sur réseau : les systèmes deviennent distribués et se composent d'éléments dispersés sur différentes machines.

Le but du Toolbus est d'intégrer des composants écrits en différents langages et tournant sur différentes machines. Pour remplir cet objectif, trois aspects des systèmes distribués et hétérogènes doivent être considérés : la coordination, le calcul et la représentation [18].

- La coordination est la façon dont les composants interagissent et se coordonnent entre eux.
- Le calcul est le travail accompli par un composant.
- La représentation est le format des données échangées entre composants.

L'hypothèse à la base de l'architecture du Toolbus est qu'une séparation rigoureuse de la coordination et du calcul est la clé d'un système flexible et réutilisable. Le Toolbus interdit donc la communication directe entre composants (les *tools*) et gère lui-même leur coordination et leurs interactions.

Les composants communiquent avec le Toolbus via un adaptateur qui joue le rôle d'interface pour la représentation des données et le protocole d'échange imposés par le Toolbus. L'adaptateur dépend du langage de programmation du composant. Dans le Toolbus même, des process écrits en Tscript gèrent les interactions entre composants. Cette architecture est représentée sur la figure 2.9 [18].

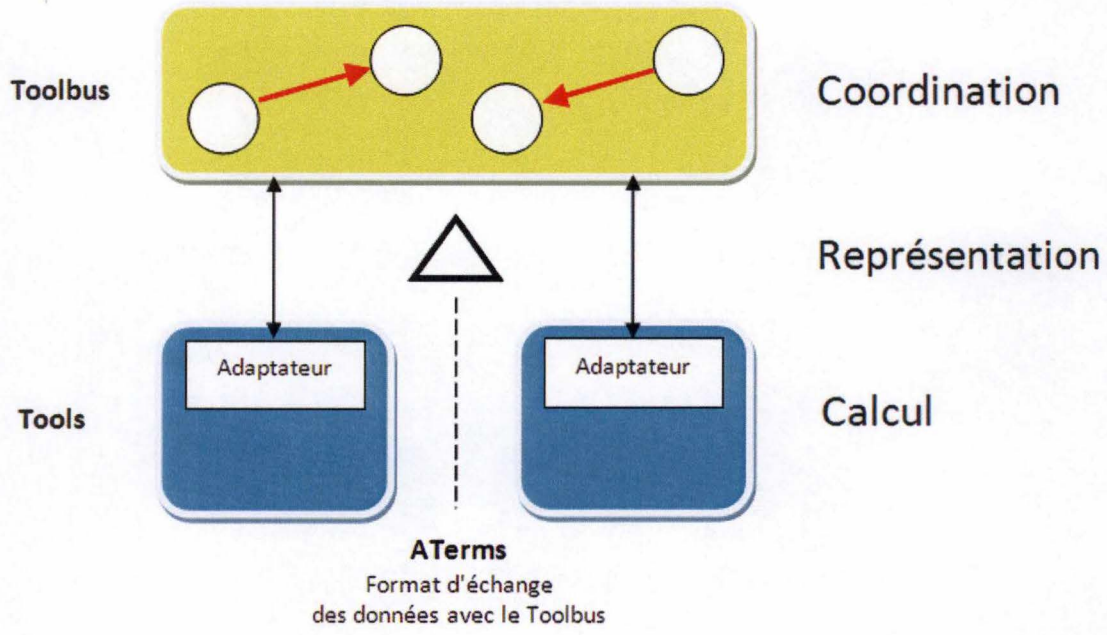

FIGURE 2.9 – Architecture du Toolbus

2.4.3 Les Tscripts

Les Tscripts [17]décrivent comment les composants d'une application coopèrent entre eux. Ils définissent chacun un process communiquant avec les composants et les autres process.

Un scénario typique où deux composants doivent coopérer est illustré dans la figure 2.10 [18] : un utilisateur pousse sur un bouton dans l'interface utilisateur (UI), et déclenche une action dans la base de donnée (DB) dont le résultat s'affiche dans l'interface utilisateur. Dans une approche traditionnelle, l'action de la base de données est directement reliée au bouton de l'interface, ce qui implique que l'interface connaisse la base de données (où la trouver, quelle fonction appeler, ...). Dans l'approche Toolbus, les deux composants sont complètement découplés : l'appui sur le bouton engendre uniquement un événement qui est traité par un process du Toolbus. Cet événement est conduit jusqu'au composant base de données par le process, qui récupère la réponse et la transmet à l'interface utilisateur par le chemin inverse. Le choix du composant à appeler ainsi que son emplacement est entièrement géré dans le Tscript. Ainsi, c'est uniquement dans le Toolbus que ces deux composants sont configurés pour travailler ensemble.

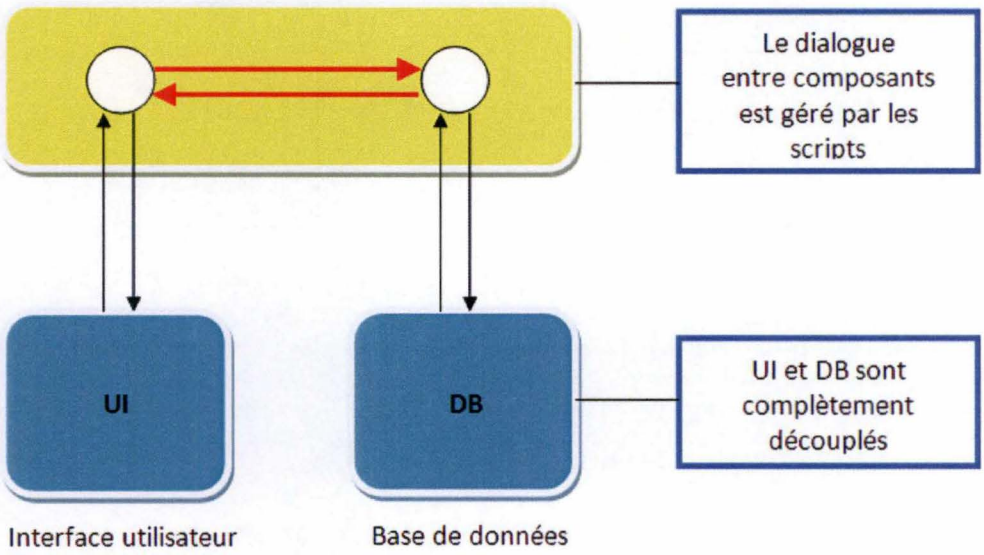

FIGURE 2.10 – Un scénario typique de coopération

Inévitablement les Tscripts deviennent le noeud central de coordination, leur taille et leur complexité peuvent devenir importantes dans de grosses applications comme le Meta-Environment [22]. Ils sont cependant l'emplacement unique où cette information de composition réside.

2.4.4 La communication dans le Toolbus

Il y a deux mécanismes de communication inter-process disponibles dans le Toolbus : les notes et les messages [18].

- Les notes : un process (émetteur) peut envoyer une note (snd-note) vers les process intéressés. Le process émetteur n'attend pas de réponse des autres process, qui lisent la note de manière asynchrone. Les notes servent à communiquer aux autres process des changements d'état d'un process. Les process reçoivent uniquement les notes auxquelles ils ont souscrit.

- Les messages : un process A peut envoyer un message (snd-msg) à un process B pour lui adresser une demande, qui est intercepté dans une fonction rec-msg. Quand le process B a terminé, il informe le process A au moyen d'un autre message.

La documentation indique que les messages sont transmis de façon synchrone, ce qui laisserait supposer que le process A reste en attente tant que le process B n'a pas envoyé son message de réponse. Cependant, en lisant les exemples de code [18], c'est apparemment par une autre instruction que le process A se met en attente (rec-msg).

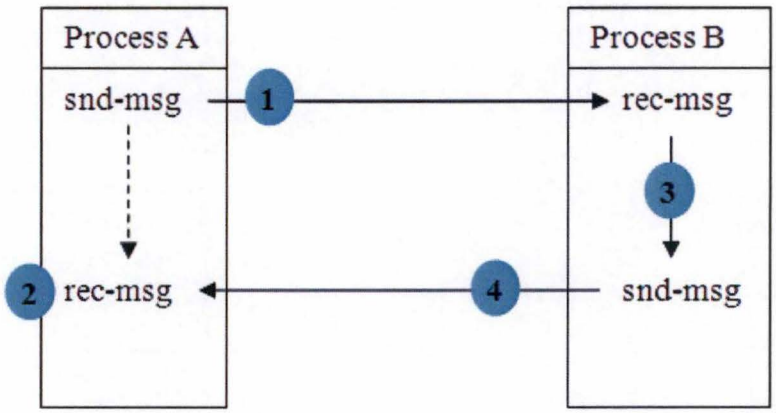

FIGURE 2.11 – Communication par messages entre deux process

La figure 2.11 illustre une communication par messages entre deux process. Le process A envoie un message au process B, qui le récupère via sa fonction rec-msg (1). Le process A se met ensuite en attente d'une réponse (2), pendant que le process B exécute la demande (3). Le process B envoie ensuite la réponse au process A (4). La communication par message semble donc être asynchrone, mais pourvue de points de synchronisation (rec-msg).

2.4.5 La communication entre le Toolbus et les composants

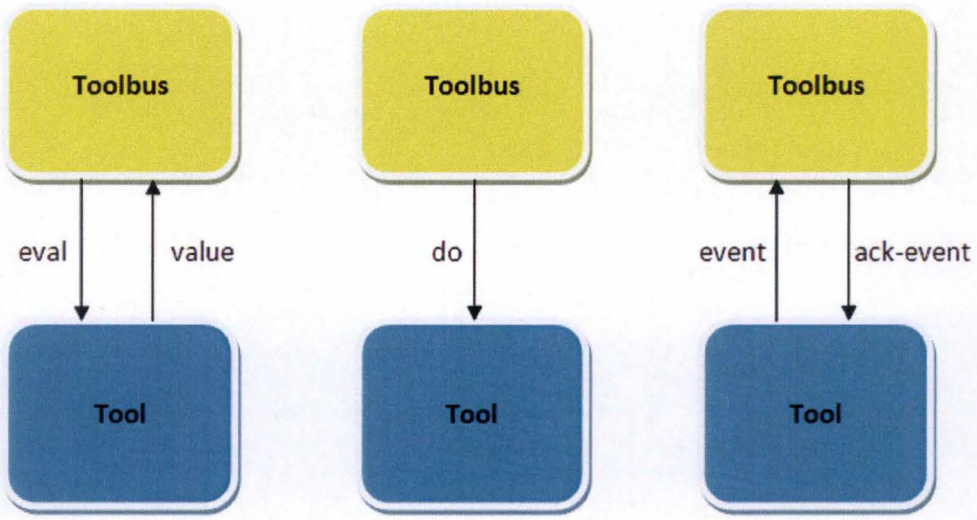

FIGURE 2.12 – Communication entre le Toolbus et les composants

La figure 2.12 montre les différents protocoles de communication entre le Toolbus et les composants. On distingue :

- une requête du Toolbus en attente d'une réponse du composant, par exemple un calcul ;
- une requête du Toolbus sans attente de réponse du composant, par exemple une demande d'affichage ;
- un événement déclenché par un composant, auquel le Toolbus répond par un accusé de réception lorsque le process déclenché par l'événement est terminé .

Il y a deux scénarios possibles pour le cycle de vie d'un composant [18] :

- Dans le premier scénario, le composant est démarré par le Toolbus, il reçoit ensuite un certains nombre de requête ou génère un certains nombre d'événements, et finalement le Toolbus décide de terminer l'exécution du composant. Le composant pourrait également décider de se déconnecter du Toolbus et de continuer son exécution à part.
- Dans le deuxième scénario, le composant est déjà démarré et démarre une coopération avec le Toolbus en lui demandant une connexion. Une fois connecté, le composant suit les mêmes étapes que dans le premier scénario.

Les composants peuvent être contrôlés par plusieurs process, de même qu'un groupe de composants peut être contrôlé par un seul process [18].

2.4.6 Le Toolbus et les plugins

Le Toolbus s'apparente plus à une architecture middelware, gérant des interactions entre composants, qu'à une architecture de plugins. Cependant, son bus actif permet d'envisager quelques scénarios de plugins. Un plugin serait composé d'un process écrit en Tscript à déployer dans le Toolbus, et/ou de composants munis des adaptateurs ad hoc.

Scénarios

- Un nouveau process intercepte toute une série de notes d'autres process et les transmet à un nouveau composant de suivi d'activité.

- Un nouveau process combine des composants existants avec un nouveau composant, qui s'occupe de l'affichage ou rajoute une étape de calcul, de sauvegarde.

- Un nouveau process fait uniquement appel à des process et des composants existants pour créer une nouvelle fonctionnalité.

- Autres scénarios, plus éloignés de l'idée de plugin :

  - Création d'un nouveau composant pour remplacer un composant existant, sans changer le process existant.

  - Modification d'un process existant pour inclure des instructions vers un nouveau process ou un nouveau composant. Ceci revient à modifier le code existant d'un process du Toolbus, ce qui est déjà loin d'un plugin stand-alone. Néanmoins on ne modifie pas le code des composants existants, on change uniquement la séquence d'appel entre ces composants. Un exemple donné dans la documentation [18] montre un bouton qui provoque l'affichage d'une zone de saisie quand on clique dessus. Bouton et zone de saisie sont gérés par le même composant. Dans les faits, la pression sur le bouton déclenche un événement du composant vers le process du

Toolbus, qui renvoie ensuite au composant un ordre d'affichage de la zone de saisie. Si on modifie le process, il pourrait envoyer son ordre vers notre nouveau composant.

Ces deux scénarios-ci changent le comportement de l'application, plutôt que d'en ajouter un nouveau.

Le Toolbus permet de créer un nouveau composant qui fait usage de l'existant, mais ne permet pas de rajouter une fonctionnalité qui s'intègre dans l'existant. On n'a donc pas la possibilité de modifier une interface graphique, ou de rajouter un correcteur orthographique dans un éditeur existant, sans modifier les process.

2.5 Eclipse

2.5.1 Présentation générale

La plateforme Eclipse [4] est un environnement de développement intégré regroupant une série d'outils pour le développement de logiciels : éditeur, compilateur, debugger, etc.

A l'origine, Eclipse a été développé par IBM, qui l'a mis en Open Source en 2001 afin de l'imposer auprès de la communauté.

Eclipse est un logiciel libre, extensible, universel et polyvalent, permettant de construire des environnement de développement pour n'importe quel langage de programmation. Plusieurs logiciels commerciaux sont basés sur Eclipse, par exemple IBM Rational Application Developer et IBM WebSphere Studio Application Developer.

Le projet Eclipse est soutenu par une grande communauté de développeurs et d'entreprises. Eclipse.org est un consortium (une collaboration temporaire entre plusieurs acteurs à un projet déterminé) d'entreprises de logiciels de développement, qui ont formé une communauté pour créer des produits facilement interopérables et basés sur la technologie du plugin.

Eclipse est entièrement basé sur le principe de plugin. A part un petit noyau appelé Platform Runtime, qui sert à démarrer Eclipse et charger les plugins, toutes les fonctionnalités sont encapsulées dans des plugins. Eclipse n'est donc pas limité à construire des environnements de développement, mais peut être employé pour développer toutes sortes d'applications.

2.5.2 Les extensions et les points d'extension

Les plugins se connectent sur Eclipse et s'interconnectent entre eux grâce au mécanisme des extensions et points d'extension [16].

Le point d'extension est une déclaration faite par un plugin indiquant qu'il peut être étendu par de nouvelles fonctionnalités. L'extension est le pendant du point d'extension : c'est une déclaration faite par un plugin indiquant qu'il fournit une fonctionnalité pour étendre un autre plugin. Le point d'extension signale qu'il attend une fonctionnalité, et l'extension répond à cette demande.

Un exemple de point d'extension est la fenêtre Préférences d'Eclipse. Un plugin du noyau d'Eclipse s'occupe d'afficher la fenêtre principale, et expose des points d'extension pour permettre à d'autres plugins d'y ajouter leurs propres pages de préférences.

Contrat entre extension et point d'extension

Terminologie : Un plugin qui présente un point d'extension est appelé plugin hôte, celui qui fournit une extension est appelé plugin extension.

A chaque point d'extension est associé un contrat [7], sous forme d'un schéma xml, qui établit des obligations entre le plugin hôte et le plugin extension. Un contrat définit ce que le plugin hôte requiert pour être étendu, par exemple :

- L'id du plugin extension.
- Le chemin de fichiers de configuration.
- Des ressources, comme une icône, un texte.
- Un objet callback qui sera appelé si certaines conditions sont remplies.

L'objet callback est une exigence habituelle dans les contrats. Le plugin hôte s'engage à instancier et appeler cet objet sous certaines conditions précisées dans le contrat, et parfois suivant certains paramètres qui peuvent être spécifiés par le plugin extension. L'objet callback doit implémenter une interface Java fournie par le plugin hôte, tandis que son implémentation est fournie par le plugin extension, et est dédiée à une extension particulière. Vu que l'objet callback doit implémenter l'interface définie et packagée dans le plugin hôte, le plugin extension dépends lui aussi du plugin hôte. La figure 2.13 montre ces relations graphiquement.

Un point d'extension peut être étendu par plusieurs extensions. L'exemple typique est celui de l'interface graphique d'Eclipse : plusieurs points de menus peuvent y être ajouté. Eclipse propose un point d'extension "menu", auquel plusieurs autres plugins peuvent venir rajouter leurs extensions, c'est-à-dire de nouveaux points de menu.

A l'inverse, une extension n'est dédiée qu'à un seul point d'extension, vu qu'elle rempli un contrat spécifique défini par ce point d'extension. Il en découle également qu'une extension a toujours un point d'extension ad hoc. Un même plugin peut évidemment étendre plusieurs autres plugins, il lui suffit de fournir autant d'extensions.

Le registre des plugins

Le registre des plugins [7, 16] est au coeur du système. C'est l'endroit où les points d'extension sont mis en relation avec les extensions. Ce registre des plugins disponibles est construit en mémoire au démarrage d'Eclipse, et reprend l'emplacement de chaque plugin et de ses dépendances. Il est construit à partir des fichiers de configuration plugin.xml de chaque plugin. Une fois démarré, un plugin qui offre des points d'extension peut consulter le registre des plugins pour découvrir quels autres plugins lui fournissent des extensions.

Chaque plugin hôte est responsable du chargement des ressources et de l'instanciation de ses plugins extensions. La pratique recommandée est de ne charger un plugin en mémoire qu'au moment de son utilisation effective. Toutefois, un plugin peut très bien décider de charger toutes ses extensions directement, même s'il n'en a pas l'usage immédiat.

Un exemple d'extension : le menu du workbench

Terminologie : L'interface graphique d'Eclipse, appelée le Workbench dans le langage courant, est un des plugins de base d'Eclipse. Il est identifié par son package org.eclipse.ui, et est également dénommé plugin User Interface ou plugin UI.

Le plugin UI [7] expose un point d'extension `org.eclipse.ui.actionSets`, qui permet d'étendre ses menus et ses barres d'outils. Dans la documentation de ce point d'extension [26], on peut lire :

> *"Les plugins peuvent fournir des extensions* `actionSets`*, qui définissent des actions avec un id, un label, une icône, et une classe callback qui implémente l'interface* `IActionDelegate`*. L'interface utilisateur présentera ce label et cette icône à l'utilisateur, et quand l'utilisateur cliquera dessus, l'interface instanciera la classe callback en la castant vers l'interface* `IActionDelegate`*, et appellera sa méthode* `run()`".

La figure 2.13 montre cette relation graphiquement [26].
Le plugin UI expose un point d'extension `org.eclipse.ui.actionSets` et fournit une interface `IActionDelegate`.
Le plugin `MyPlugin` déclare une extension qui respecte le contrat du point d'extension `org.eclipse.ui.actionSets`, et dispose d'un objet de callback "`MyCallbackObject`" qui implémente `IActionDelegate`.

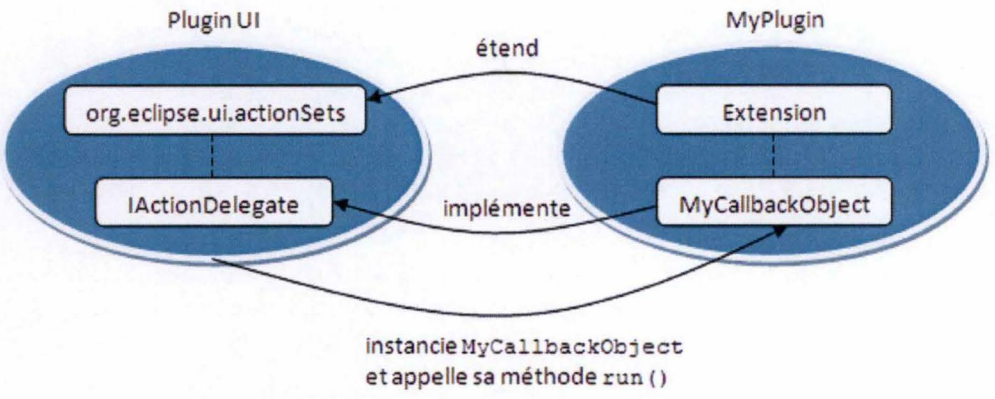

FIGURE 2.13 – Extension et point d'extension

Quand l'utilisateur clique sur le point de menu correspondant, l'UI plugin instancie `MyCallbackObject` et appelle sa méthode `run()`.

La relation entre une extension et un point d'extension est définie dans le fichier plugin.xml du plugin extension. Dans l'exemple illustré par la figure 2.14 [14], `MyPlugin` déclare une extension au point d'extension `actionSets`, définit où et comment son point de menu doit apparaître, et l'action à exécuter quand l'utilisateur clique dessus.

```
<extension point="org.eclipse.ui.actionSets">
<actionSet          ========================
   label="Sample Action Set"
   visible="true"
   id="com.example.myPlugin.actionSet">
   <menu
      label="Sample &Menu"
      id="sampleMenu">
      <separator name="sampleGroup">
      </separator>
   </menu>
   <action
      label="&Sample Action"
      icon="icons/sample.gif"
      class="com.example.myPlugin.myCallbackObject"
      tooltip="Hello"              ================
      menubarPath="sampleMenu/sampleGroup"
      toolbarPath="sampleGroup"
      id="com.example.myPlugin.myCallbackObject">
   </action>
</actionSet>
</extension>
```

FIGURE 2.14 – Le fichier plugin.xml met en relation une extension et un point d'extension

Cette relation relativement simple entre deux plugin est très puissante. Le plugin UI a permis à d'autres plugins de venir ajouter des points de menu dans l'interface utilisateur. De plus, tout est déclaratif : le plugin UI n'a pas besoin d'instancier les plugin extension pour en prendre connaissance, il lui suffit d'aller lire le registre des plugins. Ceci permet aux plugins de n'instancier leurs extensions qu'au moment de leur utilisation.

Les caractéristiques clés de ce modèle "extensions / points d'extension / registre des plugins" sont les suivantes :

- Les extensions et les points d'extension sont largement utilisés à travers Eclipse, pour ajouter des points de menu, des vues, des documents d'aide, des écrans de configuration, etc.
- Ce mécanisme peut être utilisé pour étendre un plugin avec du code (objet callback), ou des données (ex : document d'aide).
- Le mécanisme est déclaratif : les plugin sont connectés sans besoin de charger leur code.
- Le mécanisme est passif, c'est-à-dire qu'aucun code n'est chargé tant que ce n'est pas nécessaire. Dans notre exemple, `MyCallbackObject` n'est instancié que quand l'utilisateur clique sur le point de menu correspondant. Tant que l'utilisateur ne clique pas, la classe n'est pas chargée en mémoire.
- Cette approche permet d'avoir une multitude de plugins inclus dans Eclipse sans causer une surcharge du système.

Le Platform Runtime

Au départ, Eclipse est composé d'un noyau appelé le Platform Runtime [14], et de certains plugins de base qui sont présents dans toutes ses versions. L'accès à ces plugins de base est hardcodé dans le Platform Runtime ; ils sont activés d'office au démarrage d'Eclipse. Les autres plugins ajoutés à cette configuration de base ne sont activés que si nécessaire.

Tout plugin qui est ajouté à Eclipse s'ajoute en tant qu'extension d'un autre plugin. On a donc un empilement de plugins, tels qu'on peut le voir dans la figure 2.15.

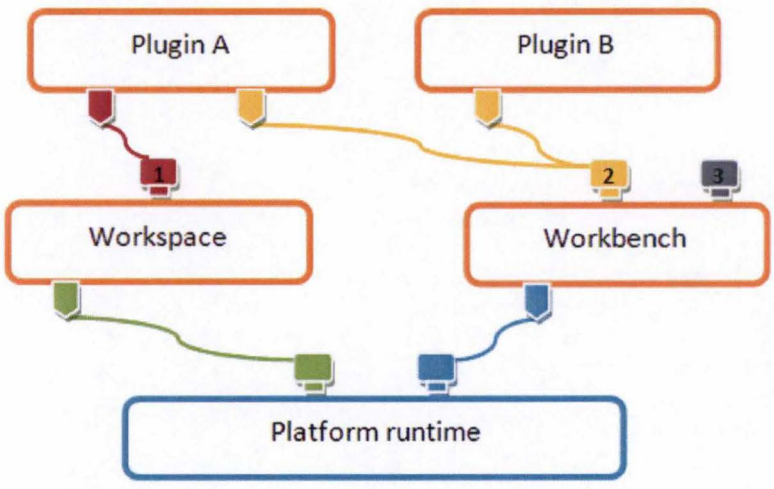

FIGURE 2.15 – L'empilement des plugins

Chaque rectangle symbolise un plugin, mis-à-part le Platform runtime. Chaque plot ( ) symbolise un point d'extension, auquel un ou plusieurs autres plugins peuvent venir brancher une extension ( ). Un point d'extension peut avoir une extension (plot 1), plusieurs extensions (plot 2), ou aucune (plot 3). Chaque extension est quant à elle toujours reliée à un et un seul point d'extension.

L'activation des plugins est, à la base, démarrée par le Platform runtime. Un plugin ne pourra donc être exécuté dans une instance d'Eclipse que s'il est relié, directement ou non, à un des plugins de base via une relation d'extension.

Deux des plugins de base sont présentés sur le schéma : le workbench et le workspace.

- Le workbench expose des points d'extension qui permettent à d'autres plugins :
  - d'étendre les menus, les barres d'outils ;
  - de créer de nouvelles vues. Une vue est une zone du workbench dans laquelle un plugin peut incorporer sa propre interface, par exemple une console d'erreurs, un éditeur de texte, un navigateur de fichiers ;
  - de s'enregistrer auprès de certains événements [14] pour en être notifiés quand ils se produisent.
- Le workspace expose des points d'extension qui permettent à des plugins d'interagir avec des ressources, comme les fichiers.

2.5.3 Plugins Eclipse et bundles OSGi

Dans la littérature, la documentation et le jargon habituel, on parle toujours de plugins Eclipse. Or depuis la version 3.0, les plugins Eclipse sont en fait des **bundles OSGi**.

OSGi [41] est la spécification d'une plateforme de services Java. Eclipse fournit l'une des nombreuses implémentations disponibles de cette spécification, et est devenue l'implémentation de référence pour la dernière version (OSGi Release 4). OSGi est un framework destiné aux systèmes qui doivent être opérationnels pour de longues durées et nécessitant des mises-à-jour dynamiques. La plate-forme OSGi est une plate-forme d'exécution d'applications Java colocalisées au sein de la même machine virtuelle Java [3].

Un bundle OSGi contient l'implémentation d'un certain nombre de services, qu'il expose. Il est soumis à un cycle de vie, et peut être installé, activé, désactivé et désinstallé dynamiquement. Il peut faire appel aux services exposés par les autres bundles.

Les plugins Eclipse sont des bundles OSGi, qui utilisent le registre des plugins. En Eclipse, les termes plugin et bundle représentent la même chose, et peuvent être employés indifféremment.

Structure d'un plugin

Un plugin se présente sous la forme d'une archive Java (JAR), copiée dans le répertoire /plugins d'Eclipse. Un fichier manifest.mf est inclus dans ce JAR, et inclus une description du plugin et de ses relations avec d'autres plugins. On retrouve également le fichier plugin.xml, qui contient la description des extensions et points d'extensions. La figure 2.16 montre le contenu et la structure du plugin workbench, dans l'archive Java `org.eclipse.ui\_3.1.0.jar` [15, 26]. Le framework OSGi s'occupe

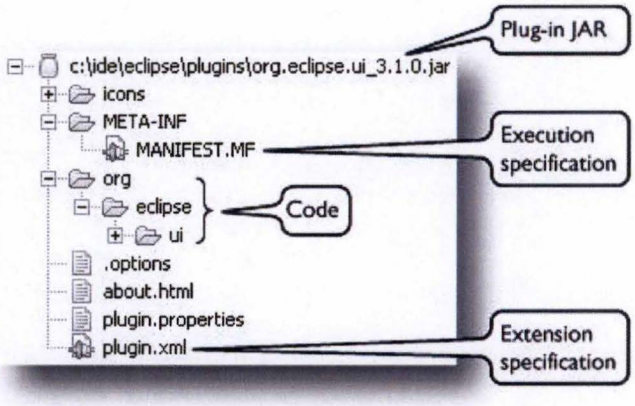

FIGURE 2.16 – Structure d'un plugin

du chargement des bundles en les isolant les uns des autres suivant certaines règles de sécurité. Le classpath d'un bundle est construit dynamiquement, en fonction des dépendances décrites dans le fichier manifest.

Dynamique des plugins

Les plugins Eclipse peuvent être dynamic-aware et/ou dynamic-enabled [8, 26].

- Un plugin dynamic-aware [5] supporte d'être en relation avec d'autres plugins dynamiques. C'est le cas du plugin workbench : il supporte l'ajout et la suppression de plugins lui fournissant des extensions (points de menu, éditeurs, etc).

  Un plugin dynamic-aware doit libérer toutes les références qu'il détient vers des classes définies dans d'autres plugins, quand ces autres plugins sont supprimés du système. En particulier, si un plugin définit des points d'extension qui chargent des classes d'autres plugins, il faut supprimer ces références quand ces plugins sont supprimés dynamiquement. Un plugin dynamic-aware doit enregistrer un listener auprès du registre des plugins. Ce listener est averti par le registre quand des extensions sont ajoutées ou supprimées du système.

  La figure 2.17 illustre un plugin A dynamic-aware : lorsque le plugin B est supprimé du système, le plugin A supprime toutes les références qu'il détient vers les classes du plugin B.

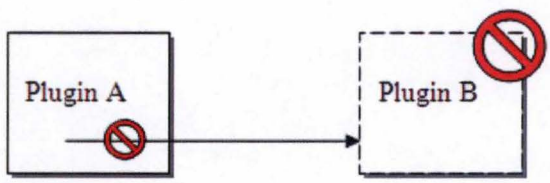

FIGURE 2.17 – Un plugin dynamic-aware

Il est généralement plus important d'être dynamic-aware pour les plugins à la base d'une chaîne de dépendances comme le workspace ou le workbench.

- Un plugin dynamic-enabled [6] supporte d'être dynamiquement ajouté ou supprimé. Pour être dynamic-enabled, un plugin doit simplement s'assurer qu'il libère bien toutes les ressources au moment où il se désinstalle.

  La figure 2.18 illustre un plugin A dynamic-enabled : lorsqu'il est supprimé du système, il libère toutes les références qu'il détient sur les classes d'autres plugins.

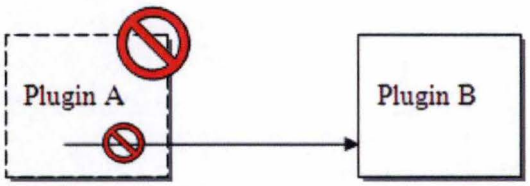

FIGURE 2.18 – Un plugin dynamic-enabled

  Il est généralement plus important pour les plugins près du sommet d'une chaîne de dépendances d'être dynamic-enabled.

Même si un plugin est fully-dynamic (c'est-à-dire aware et enabled), il n'est pas toujours possible de le supprimer proprement, si un plugin avec lequel il interagit n'est pas dynamic-aware. Tant qu'un composant détient une référence sur une classe définie dans un plugin, ce plugin ne peut pas être complètement supprimé de la mémoire.

Un plugin peut toujours être ajouté ou supprimé dynamiquement. Des erreurs ou des effets de bord risquent cependant de se produire si ce plugin ou ceux vers lesquels il pointe ne sont pas dynamiques.

2.5.4 Les services OSGi

Les services sont des éléments clés de l'architecture OSGi. Un bundle peut s'enregistrer comme fournisseur d'un certain service. Les autres bundles peuvent découvrir et employer ce service. La figure 2.19 présente le mécanisme des services OSGi [1] :

A titre de comparaison, la figure 2.20 présente le mécanisme des extensions Eclipse :

Les deux mécanismes servent à faciliter et gérer des interactions entre des composants. Le mécanisme des services permet à un composant qui implémente une certaine interface de s'enregistrer et de s'exposer publiquement. Les composants intéressés utilisent ce service si nécessaire.

De son côté, le mécanisme des extensions est un moyen pour un plugin d'être complété par les fonctionnalités d'un autre plugin. Il en résulte un collaboration "privée" entre le plugin hôte et le plugin extension. Une extension n'est destinée qu'à contribuer à un point d'extension précis, et non à être publique et utilisable par tout point d'extension intéressé. Les services, au contraire, exposent leur fonctionnalités publiquement à tout autre bundle intéressé. De plus, les extensions sont déclaratives et peuvent concerner autre chose que du code, contrairement aux services.

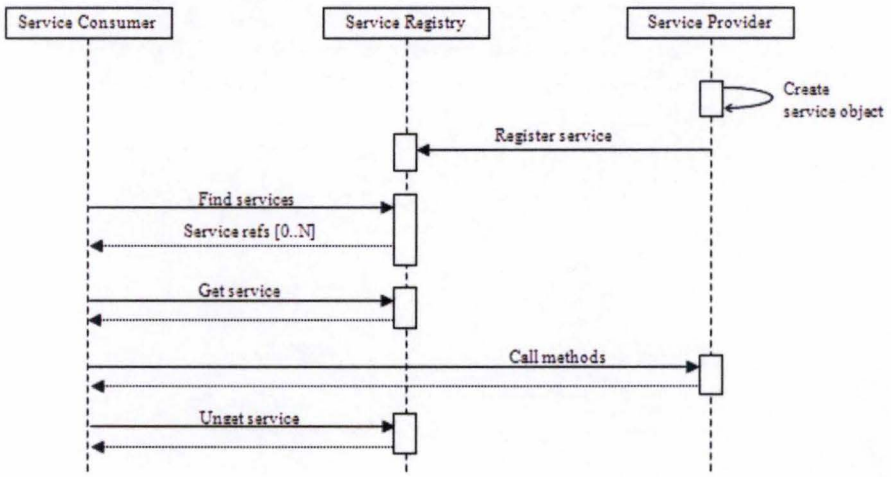

FIGURE 2.19 – Mécanisme des services OSGi

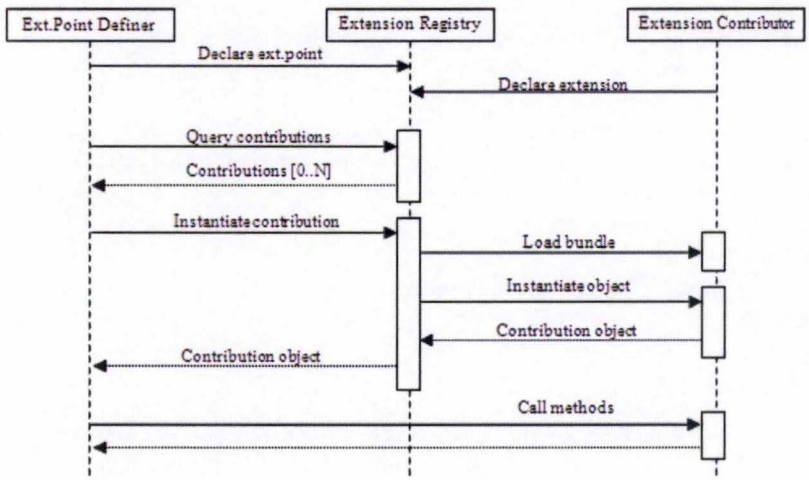

FIGURE 2.20 – Mécanisme des extensions Eclipse

Eclipse fait peu usage des services OSGi. Les services ne sont pas adéquats pour développer des applications de type RCP (Rich Client Platform). Par contre ils peuvent se justifier dans d'autres contextes, comme un process batch ou un serveur.

Les services OSGi standards

La spécification OSGi prévoit un certain nombre de services standards. Eclipse en implémente toute une série, dont le service `HttpService`. La documentation [3] donne un descriptif de ce service, repris ci-dessous et annoté en italique.

> "Le service `HttpService` permet à des bundles de publier une application Web [...] constituée de servlets et de JSP respectant la spécification J2EE mais aussi des documents statiques comme des documents HTML, des icônes, des fichier JAR d'applet ou de midlet *(applet pour appareils mobiles, comme des téléphones portables)*. Pour cela, le bundle qui implémente

le serveur HTTP chargé de recevoir les requêtes (*Service Provider*, voir figure 2.19 page 40), enregistre un service `HttpService`. Un bundle publiant une application Web (*Service Consumer*) se lie à ce service (via le *Service Registry*) et enregistre chacune des servlets en invoquant la méthode `registerServlet()` du service `HttpService`. Les ressources statiques qui sont conditionnées dans le fichier JAR du bundle, sont enregistrées via un objet `HttpContext` avec la méthode `registerRessource()`. [...] Une application Web peut contenir des scripts JSP (Java Server Page) qui seront compilés à la volée par la servlet embarquée JspServlet comme dans un serveur J2EE classique. [...] Le service `HttpService` peut également servir à publier des services Web. Plusieurs implémentations de services Web sont basées sur le reconditionnement en bundle d'implémentations comme AXIS (*une implémentation du protocole SOAP*) ou la très légère kSOAP (*une implémentation du protocole SOAP pour appareils mobiles*)."

Chapitre 3

Synthèse et éléments clés des technologies et architectures

Ce chapitre compare les différentes architectures entre elles.

Au vu de l'hétérogénéité des architectures, des critères de comparaison transversaux sont tout d'abord établis.

Les différentes architectures sont ensuite passées en revue au regard de ces critères.

Enfin, un tableau récapitulatif permet d'effectuer un parallèle entre celles-ci.

3.1 Critères de comparaison

Vu que les architectures sont fort hétérogènes, des critères de comparaison transversaux doivent être déterminés.

Ces critères vont contribuer à évaluer trois aspects des architectures :

- L'extensibilité : l'architecture peut-elle être étendue et par quel biais, avec quelle souplesse et quel type d'interaction ?
- Les performances : l'architecture est-elle performante, au regard de la technologie employée ?
- Le réemploi : l'architecture est-elle réutilisable pour créer un nouveau métaCASE ?

Indépendance

Mesure de l'indépendance du plugin par rapport à l'application. Un plugin devant implémenter certaines interfaces de l'API de l'application, ou être copié dans un répertoire précis pour pouvoir être utilisable, a une indépendance faible.

Dynamique

Aptitude à installer et désinstaller un plugin sans nécessiter un redémarrage de l'application. L'installation d'un plugin comprend son chargement en mémoire et sa mise à disposition dans l'application.

Intégration dans l'interface

Aptitude de l'application a intégrer des plugins dans son interface, et de manière transparente pour l'utilisateur.

Interactions entre plugins

Aptitude des plugins à communiquer entre eux, à se compléter, à offrir des services les uns aux autres.

Interactions avec l'application

L'interaction peut être bidirectionnelle ou unidirectionnelle :
- bidirectionnelle : le plugin et l'application peuvent tous deux prendre l'initiative de la communication.
- unidirectionnelle : l'initiative de la communication est prise exclusivement soit par le plugin, soit par l'application.

Extensibilité

Aptitude de l'application à être étendue par de nouvelles fonctionnalités, intégrées dans l'interface ou non.

Interopérabilité

Aptitude de l'application à interagir avec des plugins écrits en différents langages, pour différentes plateformes.

Accès distant

Aptitude de l'application à interagir avec des plugins présents sur d'autres machines, ainsi que d'être accédée par des applications distantes.

Performances

Les performances évaluent, suivant le cas :

- l'aptitude à ne charger un plugin que quand il est utilisé. Ce point est toujours mis en avant par les technologies qui fonctionnent de cette façon, argumentant que le chargement systématique de tous les plugins utilise inutilement de la mémoire (certains plugins ne seront jamais employés) et ralentit le démarrage de l'application. Cependant, le chargement systématique des plugins au démarrage peut, une fois l'application démarrée, accélérer l'accès aux plugins puisqu'ils sont déjà tous en mémoire. Cet aspect n'est jamais mis en avant, et passe pour négligeable à la lecture de la documentation.
- l'efficacité du protocole de communication utilisé (par exemple HTTP).

Pérennité

La pérennité évalue la durée de vie de la technologie employée. Une grande communauté d'utilisateurs, l'emploi de standards ou le support de grandes entreprises sont des indicateurs d'une durée de vie plus longue que l'emploi de technologies propriétaires ou peu connues.

Réemploi

Le réemploi de l'architecture est évalué en fonction de critères comme la présence de documentation, un code disponible ou non en open source, une architecture propriétaire ou standard, etc.

Facilité de la mise en oeuvre

Réemploi et mise en oeuvre ne vont pas toujours de pair : la présence d'une documentation exhaustive mènerait à la conclusion d'un réemploi élevé, mais ne présage pas de la facilité de mise en oeuvre.

3.2 Evaluation des métaCASE

L'ensemble des critères est évalué pour chaque métaCASE. Cette évaluation se base sur l'état de l'art, et passe par l'attribution d'un niveau (*faible, moyen, élevé*) et d'une justification pour chaque critère. Quand le critère ne peut pas être évalué ou n'est pas d'application, il est suivi de la mention *N/A* (non applicable), et quand il n'est pas du tout supporté, il est suivi de la mention *Non.*

3.2.1 Protégé

Indépendance : Faible

Les plugins doivent implémenter des interfaces spécifiques, doivent utiliser l'API s'ils veulent interagir avec les bases de connaissances, et doivent être copiés dans un répertoire de Protégé. Un point positif est que l'API est utilisable indépendamment du client Protégé.

Dynamique : Faible

L'ajout d'un nouveau plugin nécessite un redémarrage de Protégé.

Intégration dans l'interface : Elevé

Protégé permet d'intégrer complètement dans son interface des applications en interactions avec ses bases de connaissances. L'utilisateur ne voit pas la différence entre les parties natives de Protégé et les plugins.

Interactions entre plugins : Moyen

Un plugin peut dépendre d'un autre plugin, et faire usage de services offerts via ses méthodes publiques. Il s'agit plus d'un partage de classes utiles que d'une vraie interaction.

Interactions avec l'application : Moyen

Les application intégrées dans Protégé peuvent accéder aux bases de connaissances, mais ne peuvent pas capturer d'événements. La communication est unidirectionnelle, de l'application vers Protégé.

Extensibilité : Elevé

Protégé permet d'intégrer des applications entières dans son interfaces, de fournir de nouveaux formats de sauvegarde, de personnaliser la saisie et l'affichage des données.

Interopérabilité : Moyen à Elevé

Les plugins et les applications externes qui font usage de l'API doivent obligatoirement être écrits en Java. Java est un cependant un langage très courant, et de plus multiplateformes.

Accès distant : Moyen à Faible

Les plugins sont tous locaux, et doivent être copiés dans un répertoire précis de Protégé. L'API de Protégé permet à des applications externes d'accéder les bases de connaissances, mais l'API (fichier JAR) doit être accédé localement par l'application. Cependant, L'API permet apparemment d'accéder des bases de connaissances distantes.

Performances : Moyen

Les plugins sont, au moins pour le type TabWidget, chargés en mémoire dès le démarrage de Protégé.

Pérennité : Elevé

Protégé est écrit en Java, est soutenu par une grande communauté (100.000 membres) et est utilisé par des centaines de groupes de recherche.

Réemploi : Faible

Protégé est écrit comme une collection de plugins, son architecture est donc nativement orientée vers l'extension. De plus Protégé est open source, on peut donc plonger dans le code de son architecture pour le réutiliser ou s'en inspirer. Cependant son architecture de plugins est propriétaire (elle n'utilise pas une technologie particulière, à part Java) et n'est pas documentée.

Facilité de la mise en oeuvre : Faible

3.2.2 GME et COM

Indépendance : Moyen

Les composants ne dialoguent entre eux que par interfaces COM. Local ou distant, le composant COM est implémenté de la même façon. Cependant, pour être utilisables, les composants doivent être enregistrés auprès de GME via un petit utilitaire.

Dynamique : Elevé

L'instanciation d'un composant se produit au moment de son utilisation.

Intégration dans l'interface : Elevé

Les décorateurs permettent de personnaliser le dessin des modèles créés dans GME. L'interface des add-ons, interpréteurs et plugin sont bien intégrés.

Interactions entre plugins : Elevé

Se fait via leurs interfaces COM, qui permettent potentiellement une interaction élevée.

Interactions avec l'application : Elevé

L'interaction est bidirectionnelle : Les composants peuvent appeler les services exposés par les interfaces COM de GME, et peuvent s'enregistrer auprès de GMeta et GModel pour être notifiés d'événements par GME.

Extensibilité : Elevé

GME permet la création de composants qui réagissent à des événements, qui interagissent avec les modèles, et qui s'intègrent dans son interface.

Interopérabilité : Moyen

Les composants peuvent être écrits dans tout langage qui supporte COM. Cependant, la portabilité de COM sur d'autres plateformes que Windows n'est pas garantie. De plus, si un composant est écrit en Java il doit obligatoirement tourner sur la Virtual Machine de Microsoft, et donc sous Windows.

Accès distant : Elevé

COM permet un accès distant transparent.

Performances : Elevé

COM n'accède et ne charge les composants que si nécessaire.

Pérennité : Elevé

COM est une technologie mature et l'une des architectures de base de Microsoft Windows.

Réemploi : Moyen

COM peut facilement être réutilisé et est documenté par Microsoft. L'architecture d'événements est quant à elle complètement propriétaire, de même que l'intégration des composants dans l'interface. Limité à COM, le réemploi se limite plus à une architecture de communication entre composants qu'une architecture de plugins.

Facilité de la mise en oeuvre : Elevé

3.2.3 MetaEdit+ et les services web

Indépendance : Elevé

Basée sur les services web, l'API peut être accédée par n'importe quelle application, locale ou distante. Pour accéder l'API, il faut cependant que le serveur de MetaEdit+ soit actif.

Dynamique : N/A

MetaEdit+ ne permet pas l'intégration de plugins. Ce critère est donc non applicable.

Intégration dans l'interface : N/A

Idem.

Interactions entre plugins : N/A

Idem.

Interactions avec l'application : Moyen

Les application distantes peuvent interroger l'API de MetaEdit+. MetaEdit+ ne transmet de lui-même aucune information et n'émet aucun événement.

Extensibilité : Non

MetaEdit+ donne un accès à ses fonctionnalités, mais ne permet pas de les étendre.

Interopérabilité : Elevé

Les applications qui accèdent l'API peuvent être écrite en n'importe quel langage de programmation, et tourner sur n'importe quelle plateforme.

Accès distant : Elevé

Les services web opèrent au-dessus du protocole HTTP.

Performances : Moyen

L'usage du protocole HTTP donne des temps de réponse moins bons que ceux d'autres architectures d'accès distant comme CORBA ou COM.

Pérennité : Elevé

Les services web sont une technologie en expansion.

Réemploi : Elevé

L'architecture n'est pas propriétaire et la technologie des services web est légère et facile.

Facilité de la mise en oeuvre : Elevé

3.2.4 Meta-Environment et Toolbus

Indépendance : Moyen

Les composants doivent être pourvus d'un adaptateur afin de pouvoir communiquer avec le Toolbus. Ceux-ci peuvent cependant être écrits dans n'importe quel langage, et peuvent être localisés sur une machine distante

Dynamique : Elevé

Les composants peuvent être démarré indépendamment du Toolbus, et décider par eux-même de s'y connecter et de s'en déconnecter pour continuer leur exécution de leur côté. L'initiative peut également être prise par le Toolbus, qui peut démarrer et terminer l'exécution du composant.

Intégration dans l'interface : Non

Les composants peuvent fournir une nouvelle interface, mais pas s'intégrer dans celle d'un composant existant.

Interactions entre plugins : Non

L'interaction entre composants est gérée par le Toolbus. Les composants n'interagissent jamais directement entre eux.

Interactions avec l'application : Elevé

Un composant peut recevoir des instructions du Toolbus, peut envoyer des informations en réponse à une demande du Toolbus, ou en envoyer de sa propre initiative.

Extensibilité : Moyen

De nouveaux process et de nouveaux composants peuvent être ajoutés. Il n'y a cependant pas moyen de modifier, étendre des fonctionnalités existantes sans toucher au code des process ou des composants.

Interopérabilité : Moyen à Elevé

Les composants qui se connectent au Toolbus peuvent être écrits en n'importe quel langage de programmation, pourvu que l'adaptateur pour s'interfacer avec le Toolbus existe pour ce langage. L'aspect multiplateforme du Toolbus n'est pas documenté.

Accès distant : Elevé

La mise en oeuvre d'applications distribuées est une des caractéristiques principales du Toolbus. La technologie employée n'est pas détaillée.

Performances : N/A

Les performances du Toolbus n'ont pas pu être déduites de la documentation.

Pérennité : N/A

Le Toolbus est une architecture pour laquelle il n'y a pas encore eu beaucoup de mises en pratique. Son concept est innovant et n'a pas d'équivalent sur marché.

Réemploi : Faible

Le Toolbus est une architecture particulière et propriétaire, qui implique la maîtrise du langage Tscript et de l'interfaçage entre les composants et le Toolbus. La coordination entre composants est inhabituelle, puisque tout est concentré dans les process du Toolbus. Le Toolbus paraît peu approprié pour le développement de clients riches.

Facilité de la mise en oeuvre : Faible

3.2.5 Eclipse et OSGi

Indépendance : Moyen à Elevé

Les détails d'interconnexion entre deux plugins sont extraits dans un fichier de configuration, ce qui laisse le code du plugin *presque* indépendant. Les plugins doivent en effet présenter un objet callback, qui implémente une interface du plugin a étendre.

Dynamique : Elevé

Les plugins peuvent être mis à jour et démarrés dynamiquement, sans redémarrage d'Eclipse.

Intégration dans l'interface : Elevé

Les points d'extension du workbench permettent à tout plugin de rajouter des points de menu, des toolbars, des interfaces utilisateurs élaborées.

Interactions entre plugins : Elevé

Le mécanisme des extensions et points d'extensions permet à un plugin d'étendre un autre plugin, par du code ou de la documentation. Le mécanisme des services OSGi permet à un bundle d'utiliser les services exposés par d'autres bundles.

Interactions avec l'application : Elevé

Les plugins qui constituent le noyau d'Eclipse peuvent tous être étendus par d'autres plugins.

Extensibilité : Elevé

Extensions et points d'extensions permettent une évolution et une extension quasi illimitée de l'application.

Interopérabilité : Elevé

Eclipse est écrit en Java, et est multiplateformes.

Accès distant : Moyen à Elevé

OSGi fournit des services standards, dont le service HttpService fournissant un serveur web, et permettant la mise en place de services web.

Performances : Elevé

Un plugin n'est chargé en mémoire que s'il est utilisé.

Pérennité : Elevé

Eclipse est écrit en Java, bénéficie d'une grande communauté, d'une grande popularité, et est à la base de plusieurs applications dont des applications commerciales. OSGi est une architecture mature, employée par des entreprises telles que NOKIA ou BMW.

Réemploi : Elevé

Eclipse peut être utilisé tel quel pour créer une nouvelle application de type client riche. Une documentation fournie existe pour la mise en oeuvre des plugins.

Facilité de la mise en oeuvre : Moyen à Elevé

3.3 Tableau comparatif des différentes architectures

Légende
↑ Elevé
↗ Moyen à Elevé
→ Moyen
↘ Moyen à Faible
↓ Faible
✖ Non
N/A Non applicable

	Protégé	GME & COM	MétaEdit+ & services web	Méta-Environment & Toolbus	Eclipse & OSGi
Indépendance	↓	→	↑	→	↗
Dynamique	↓	↑	N/A	↑	↑
Intégration dans l'interface	↑	↑	N/A	✖	↑
Interactions entre plugins	→	↑	N/A	✖	↑
Interactions avec l'application	→	↑	→	↑	↑
Extensibilité	↑	↑	✖	→	↑
Interopérabilité	↗	→	↑	↗	↑
Accès distant	↘	↑	↑	↑	↗
Performances	→	↑	→	N/A	↑
Pérennité	↑	↑	↑	N/A	↑
Réemploi	↓	→	↑	↓	↑
Mise en oeuvre	↓	↑	↑	↓	↗

Ce tableau met en lumière les points suivants :

- Certaines fonctionnalités sont mal supportées par beaucoup de métaCASE, dont l'ajout et la suppression dynamiques de plugins, ainsi que les interactions entre plugins.
- Peu de fonctionnalités sont communes à tous les métaCASE. Les critères d'interopérabilité et d'interaction avec l'application sont les seuls à être supportés par tous.
- Les technologies sur lesquelles s'appuient les architectures orientent les fonctionnalités qu'elles peuvent mettre à disposition. Ainsi, MetaEdit+ basé sur le standard des services web ne permet aucune intégration de plugins, tandis que Protégé et GME basés sur des architectures propriétaires en permettent l'intégration complète.
- Eclipse est la seule architecture couvrant l'ensemble des fonctionnalités.

Troisième partie

Identification des besoins d'extensibilité et mise en oeuvre

Chapitre 4

Scénarios

Ce chapitre présente différents scénarios d'extension d'un métaCASE. Ces scénarios sont établis à partir de l'état de l'art et d'idées originales, et sont répertoriés afin de mieux appréhender les besoins d'extensibilité des métaCASE. Ils sont présentés en terme de fonctionnalités requises, faisant abstraction des technologies à employer, excepté les deux derniers scénarios mettant en oeuvre les services web.

A ce stade, l'état de l'art permet déjà d'identifier un premier panel de fonctionnalités variées, réparties en trois groupes :

- Le chargement de plugins locaux.
- Le chargement de plugins distants.
- L'accès d'une application distante au métaCASE.

Les scénarios vont permettre d'étoffer ce panel de fonctionnalités.

4.1 Plugins inclus dans le métaCASE, avec interface utilisateur

Ces scénarios illustrent des plugins intégrés dans l'interface du métaCASE, et lui rajoutant des fonctionnalités plus ou moins complexes. Ces plugins sont de petites applications pourvues d'une interface propre, mais intégrée dans celle du métaCASE. Un exemple type est le TabWidget de Protégé (voir page 15). Suivant leur complexité, ces plugins peuvent également agir sur les modèles ou réagir à des événements du métaCASE.

4.1.1 Scénario 1 : Plugins sans interaction directe avec le métaCASE

Ces plugins permettent de fournir des outils annexes, vus par l'utilisateur comme faisant partie intégrante du métaCASE. Lorsqu'il fait usage d'un de ces plugins, l'utilisateur interagit en fait avec l'interface du plugin et non celle du métaCASE. Ces plugins sont autonomes et n'interagissent pas avec le métaCASE, mise à part leur intégration visuelle. Par exemple :

- Une calculatrice.
- Un éditeur de texte, pour prendre quelques notes.
- Un browser, pour afficher des news utiles ou consulter Internet.

Fonctionnalités requises

- L'enregistrement du plugin : le métaCASE doit prendre connaissance des plugins à intégrer. Dans Protégé et Eclipse, ces plugins sont copiés dans un répertoire particulier. Dans GME, ils sont enregistrés via un point de menu dédié.
- L'intégration du plugin : le métaCASE doit pouvoir intégrer dans son interface utilisateur l'interface du plugin. Pour cela, le plugin doit répondre à certaines exigences du métaCASE. Dans Protégé, les plugins doivent implémenter des interfaces Java spécifiques. Par exemple, les plugins à ajouter parmi les tabs de Protégé doivent implémenter l'interface TabWidget. Dans Eclipse, les plugins doivent répondre aux exigences des points d'extensions qu'ils étendent. Le point d'extension se charge d'ajouter un accès à l'interface du plugin, par exemple via un point de menu.

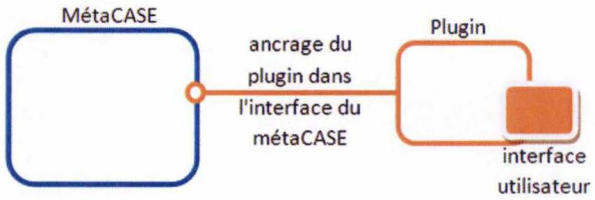

FIGURE 4.1 – Plugin sans interaction directe avec le métaCASE

La figure 4.1 illustre un plugin s'incorporant dans l'interface du métaCASE :

- L'ancrage reprend les notions d'enregistrement et d'intégration.
- Le plugin présente sa propre interface utilisateur.

4.1.2 Scénario 2 : Plugins accédant aux fonctionnalités du métaCASE

Ces plugins permettent de lire les modèles du métaCASE et de les modifier. Par exemple :

- Contrôler la validité d'un modèle, par exemple la cohérence des contraintes entre elles.
- Exporter les modèles dans des formats particuliers.
- Interpréter le contenu d'un modèle pour générer de la documentation ou du code source.

- Visualiser les modèles d'une manière différente de celles proposées par le métaCASE, par exemple donner un aperçu de plus haut ou de plus bas niveau, avec plus ou moins de détails.
- Modifier le contenu d'un modèle, par exemple en corrigeant les fautes d'orthographe.

Fonctionnalités requises

- Pour permettre l'accès aux modèles, le métaCASE doit fournir une API. L'API consiste en un set de fonctions appelables par les plugins et donnant accès aux modèles et aux fonctionnalités du métaCASE. L'idéal est d'avoir une API permettant aux plugins de réaliser au minimum tout ce que l'utilisateur pourrait réaliser via l'interface du métaCASE.

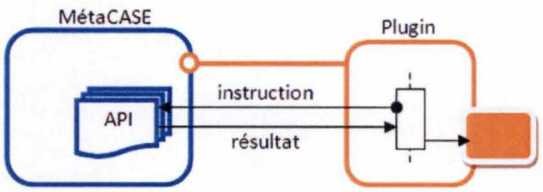

FIGURE 4.2 – Plugin accédant le métaCASE

La figure 4.2 illustre un plugin accédant au métaCASE :

1. Le plugin envoie une instruction au métaCASE via son API.
2. Le plugin reçoit le résultat et l'affiche dans son interface le cas échéant.

Remarque

Dans ce scénario, l'initiative de la communication est toujours prise par le plugin, et non par le métaCASE. Le plugin ne sait pas intercepter d'événements se produisant sur les modèles, et ne sait donc pas y réagir directement. Son action est liée soit à une commande externe (venant par exemple de l'utilisateur), soit à une logique interne (il agit par exemple toutes les 5 minutes).

4.1.3 Scénario 3 : Plugins appelés par le métaCASE

Ces plugins sont à l'écoute de l'activité du métaCASE, et permettent de réagir aux changements d'états d'un modèle. Par exemple :

- Contrôler la syntaxe d'une contrainte dès qu'elle est introduite (voir les contraintes OCL de GME, page 22).

Fonctionnalités requises

- Pour permettre aux plugins d'être à l'écoute du métaCASE, celui-ci doit exposer des événements auxquels les plugins peuvent s'abonner. Lorsqu'un événement survient, le métaCASE le communique à tous les plugins abonnés. La souscription à un événement peut se faire via l'API, ou dans le cas d'Eclipse via l'enregistrement d'une extension.

La figure 4.3 illustre un plugin à l'écoute d'événements émis par le métaCASE, et en particulier le scénario du vérificateur de syntaxe OCL de GME :

1. Le plugin s'enregistre auprès de l'événement onChange de la contrainte OCL.
2. Le métaCASE notifie au plugin les changements opérés sur la contrainte par l'utilisateur.
3. Le plugin vérifie la syntaxe de la contrainte modifiée, et le cas échéant affiche les erreurs dans son interface.

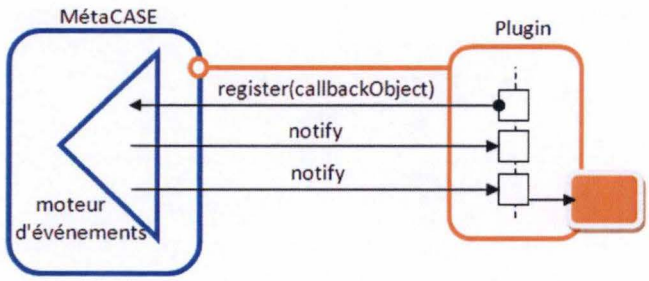

FIGURE 4.3 – Plugin à l'écoute des événements du métaCASE

Remarque

Si les événements doivent mener à un affichage utilisateur, celui-ci se fait dans l'interface du plugin. Le vérificateur de contraintes OCL ne touche pas à l'interface où la contrainte a été introduite, ni à la contrainte elle-même.

4.1.4 Scénario 4 : Communication bilatérale entre plugin et métaCASE

Ce scénario combine les scénarios 2 et 3 : le métaCASE notifie des événements aux plugins, et les plugins agissent sur le métaCASE. Par exemple :

- La mise en page automatique du modèle pendant son élaboration par l'utilisateur : le plugin aligne correctement les objets graphiques, met les mots clés en majuscule, permet des raccourcis d'écriture (le plugin remplace par exemple "\int" par "Integer"). La configuration de la mise en page se fait dans l'interface utilisateur du plugin.

Fonctionnalités requises

- Le métaCASE doit exposer des événements diversifiés, déclenchés lors de modifications de texte, de déplacements d'objets graphiques, etc.
- L'API du métaCASE doit être étoffée et permettre la mise en forme de texte, le repositionnement d'objets graphiques, etc.
- L'API du métaCASE doit permettre d'agir sur des modèles chargés en mémoire de travail : les modèles doivent pouvoir être modifiés en direct, et le résultat visible aussitôt dans l'interface utilisateur.

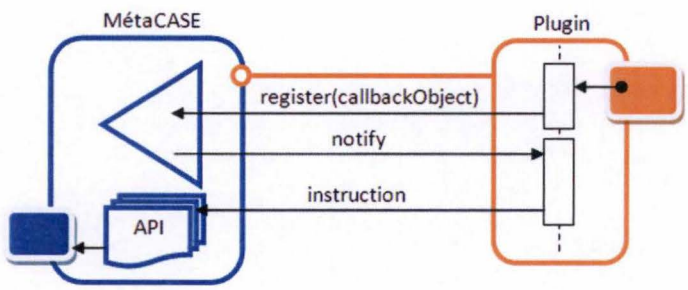

FIGURE 4.4 – Communication bilatérale entre plugin et métaCASE

La figure 4.4 illustre une communication bilatérale entre un plugin et le métaCASE, et en particulier le scénario de mise en page automatique d'un modèle :

1. L'utilisateur indique ses préférences de mise en page dans l'interface du plugin.
2. Le plugin s'enregistre auprès des événements ad hoc du métaCASE (par exemple tous les événements onTextChange du modèle).
3. Le métaCASE notifie au plugin les changements opérés sur les modèles par l'utilisateur.
4. Le plugin interprète ces changements et renvoie des instructions de mise en page par l'intermédiaire de l'API du métaCASE.

4.2 Plugins inclus dans le métaCASE, sans interface utilisateur

Ces plugins sont intégrés dans le métaCASE mais n'ont pas d'interface propre. Ils réalisent des traitements d'arrière-plan, ou agissent dans l'interface existante du métaCASE.

4.2.1 Scénario 5 : Le plugin vérificateur de syntaxe OCL

Le métaCASE prévoit un éditeur dans lequel l'utilisateur peut introduire des contraintes OCL. Un plugin vérifie la syntaxe des contraintes pendant leur introduction, et surligne les éléments de syntaxe incorrects. Ce plugin agit dans l'interface même du métaCASE, à l'inverse du vérificateur de syntaxe OCL du scénario 3, qui dispose de sa propre interface.

Fonctionnalités requises

- L'API du métaCASE doit permettre de formater du texte. Le plugin reçoit un pointeur vers le texte à vérifier, auquel il ajoute un surlignement supplémentaire en utilisant les fonctions de mise en forme de l'API.

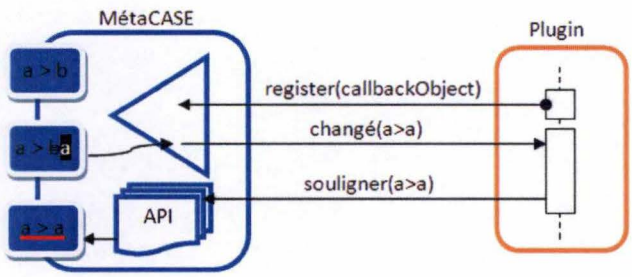

FIGURE 4.5 – Plugin vérificateur de syntaxe OCL

La figure 4.5 illustre le scénario :

1. Le plugin s'enregistre auprès de l'événement onChange de l'éditeur de contraintes OCL.
2. Le métaCASE notifie au plugin les changements opérés sur les contraintes par l'utilisateur.
3. Le plugin vérifie la syntaxe des contraintes modifiées, et demande au métaCASE de surligner les contraintes erronées.

4.2.2 Scénario 6 : L'ajout de nouveaux formats de sauvegarde

Ces plugins ajoutent des formats de sauvegarde supplémentaires dans les options de sauvegarde des modèles, par exemple :

- Un format d'enregistrement compatible avec un autre métaCASE.
- Un format texte lisible, qui décrit les objets du modèle et leurs relations.

Fonctionnalités requises

- Le métaCASE doit permettre l'ajout de nouvelles entrées dans la liste des formats de sauvegarde disponibles à l'utilisateur, et l'appel à des fonctions de callback lorsque ces formats sont sélectionnés.

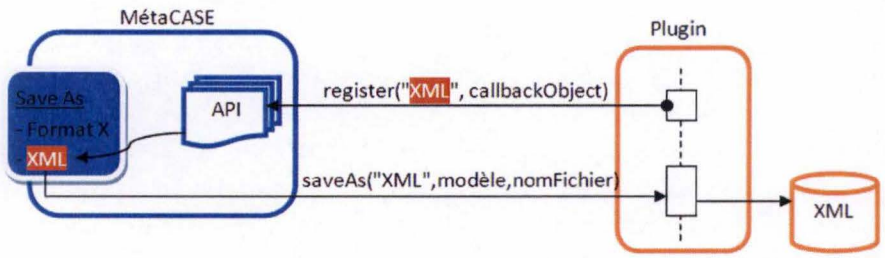

FIGURE 4.6 – Ajout d'un nouveau format de sauvegarde

La figure 4.6 illustre l'ajout d'un nouveau format de sauvegarde :

1. Le plugin fait appel à l'API pour ajouter le format de sauvegarde XML aux formats disponibles. Il renseigne également la fonction de callback à appeler lorsque l'utilisateur choisi ce format pour sauver ses modèles.
2. L'utilisateur décide de sauver en XML et sélectionne l'option parmi les formats de sauvegarde. Le métaCASE appelle la fonction de callback, en lui transmettant le modèle à sauvegarder.
3. Le plugin convertit le modèle en XML et l'enregistre sous le nom de fichier donné par l'utilisateur.

Remarque

Ce scénario revient, dans un contexte plus général, à permettre la personnalisation de l'interface propre du métaCASE. On peut également imaginer l'ajout d'un point de menu, ou l'ajout de rubriques dans l'aide du métaCASE. Ce dernier exemple (voir Eclipse, page 36) illustre un plugin qui n'est pas un programme, mais un contenu statique : il consiste en un document à inclure.

4.2.3 Scénario 7 : Les plugins graphiques

Ces plugins permettent de personnaliser la représentation graphique des modèles du métaCASE (voir les décorateurs graphiques de GME, page 23). Lorsqu'un objet du modèle doit être rendu graphiquement, celui-ci fait appel au plugin graphique associé, qui se charge de produire le dessin adapté. Le plugin renvoie le dessin au métaCASE qui l'affiche dans son interface utilisateur.

Fonctionnalités requises

- Le métaCASE doit permettre l'association d'un plugin graphique aux objets graphiques des modèles. Cette association se fait manuellement par l'utilisateur : il choisit pour chaque objet graphique le plugin qui donne le rendu approprié.

- Les objets graphiques doivent prévoir un rendu par défaut au cas où ils n'ont pas de plugin associé.
- Si un plugin est associé à un objet graphique, ce plugin est appelé chaque fois que l'objet doit être redessiné, en lieu et place du rendu par défaut de l'objet.
- Le plugin graphique doit implémenter des interfaces spécifiques et fournir les fonctions attendues par le métaCASE, par exemple une fonction de dessin `draw()` qui renvoie une représentation graphique dans le format voulu. La fonction `draw()` devrait être paramétrable pour donner des rendus différents suivant les propriétés des objets.
- Le plugin produit uniquement un dessin, et ne se charge pas de son positionnement dans l'interface du métaCASE.

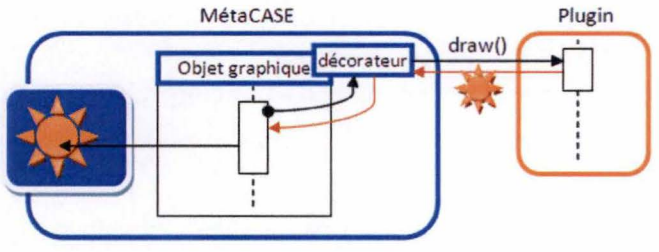

FIGURE 4.7 – Plugin graphique

La figure 4.7 illustre le scénario d'un plugin réalisant le rendu graphique d'un objet :

1. Un plugin graphique est associé à l'objet graphique d'un modèle (attribut décorateur).
2. Au moment où l'objet graphique doit être dessiné, le métaCASE considère le plugin associé, et appelle sa fonction `draw()`.
3. Le plugin renvoie le dessin associé à l'objet.
4. Le métaCASE affiche le dessin dans son interface.

4.3 Plugins inclus dans le métaCASE, avec chargement distant

4.3.1 Scénario 8 : Chargement distant

Ces plugins sont localisés sur une machine distante. Leur chargement implique l'utilisation de technologies d'accès distant.

L'exécution de ces plugins peut soit se produire dans le même processus que celui du métaCASE, soit sur la machine distante :

- Si le plugin s'exécute dans le même processus que le métaCASE, l'accès distant consiste simplement à rapatrier le code source du plugin et démarrer son exécution. Ceci est par exemple utile pour des plugins dont le code source doit fréquemment être mis à jour.

- Si le plugin s'exécute sur la machine distante, l'accès distant consiste à appeler les fonctions du plugin distant et à récupérer le résultat de leurs traitements. Ceci est par exemple utile pour des plugins dont le code source doit rester inaccessible.

4.4 Services distants appelés par le métaCASE

Dans ces scénarios, le métaCASE utilise des services exposés par des applications distantes. L'initiative de la communication est toujours prise par le métaCASE.

Fonctionnalités requises

- Le métaCASE doit pouvoir accéder le service distant avec la technologie et le protocole supportés par celui-ci.
- Le métaCASE doit transmettre ses modèles dans le format attendu par le service distant, et doit pouvoir interpréter le résultat.
- Le métaCASE devrait journaliser les événements qui se passent lors de sa communication avec le service distant, pour informer l'utilisateur (demande envoyée, attente de résultat, communication interrompue, service indisponible, etc).

4.4.1 Scénario 9 : Appel à un service distant synchrone

Le métaCASE fait appel à un service distant qui renvoie sa réponse directement. Par exemple :
- Appel à un service de validation de modèle.
- Appel aux fonctionnalités d'un autre métaCASE.
- Consultation d'une librairie de métamodèles.

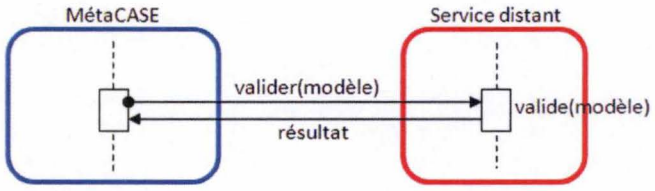

FIGURE 4.8 – Appel à un service distant synchrone

La figure 4.8 illustre un appel à un service distant synchrone :

1. Le métaCASE appelle le service distant et lui transmet le modèle à valider. Il reste en attente de la réponse du service distant.
2. Le service distant valide le modèle, et renvoie le résultat au métaCASE.
3. Le métaCASE récupère le résultat et procède à son traitement.

4.4.2 Scénario 10 : Appel à un service distant asynchrone

Le métaCASE fait appel à un service distant qui ne renvoie pas de réponse immédiate. Le métaCASE doit interroger le service distant pour récupérer le résultat. Ce scénario se justifie pour des traitements qui prennent beaucoup de temps ou qui ne nécessitent pas une réponse immédiate, par exemple :
- Appel à un service de génération de code à partir d'un modèle.

Fonctionnalités requises

- Le métaCASE doit pouvoir effectuer une demande de réponse différée, associée à sa demande initiale. Pour cela, le métaCASE peut par exemple disposer d'un journal des tâches à effectuer, dans lequel il répertorie les demandes de réponse à effectuer. La demande de réponse peut être initiée par un événement extérieur (par exemple l'utilisateur) ou par une logique interne (par exemple l'attente d'un certain délai entre la demande et la réponse).

- Le métaCASE doit pouvoir réitérer sa demande de réponse, si le service distant n'a pas encore produit le résultat.

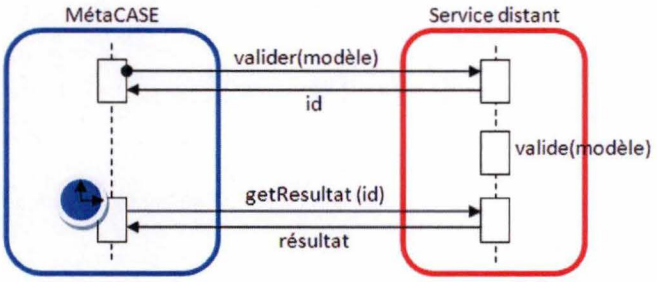

FIGURE 4.9 – Appel à un service distant asynchrone

La figure 4.9 illustre un appel à un service distant asynchrone :

1. Le métaCASE appelle le service distant et lui transmet le modèle à valider. Il reste en attente de la réponse du service distant.
2. Le service distant réceptionne le modèle, y associe un identifiant qu'il renvoie au métaCASE.
3. Le métaCASE associe cet identifiant à sa demande de validation, et termine la communication avec le service distant.
4. Après un certain temps, le métaCASE recontacte le service distant, et lui demande le résultat de la validation en lui transmettant l'identifiant associé. Il reste en attente de la réponse du service distant.
5. Le service distant retourne le résultat au métaCASE.

4.4.3 Scénario 11 : Appel à un service distant de type événementiel

Le métaCASE propage certains de ses événements vers le service distant. Celui-ci réagit ou non aux événements, et le cas échéant transmet le résultat de son traitement au métaCASE, par l'intermédiaire d'un objet de callback enregistré préalablement par celui-ci.

Fonctionnalités requises

- Le métaCASE doit être capable de propager ses événements vers le service distant.
- Le métaCASE doit disposer d'un objet de callback répondant aux exigences du service distant (l'objet doit sans doute implémenter une interface particulière).
- Le métaCASE doit pouvoir enregistrer son objet de callback auprès du service distant.
- Le métaCASE doit transmettre ses événements vers le service distant après l'enregistrement de son objet de callback.
- Le métaCASE doit pouvoir filtrer les événements qu'il transmet au service distant.

La figure 4.10 illustre un dialogue avec un service distant de type événementiel :

1. Le métaCASE enregistre son objet de callback auprès du service distant.
2. Le métaCASE propage les événements qui se produisent sur son modèle vers le service distant.
3. Le service distant valide le modèle : si le résultat est bon, il ne renvoie aucun feedback ; si le résultat est mauvais, le service distant appelle l'objet de callback du métaCASE et lui transmet les erreurs de validation.

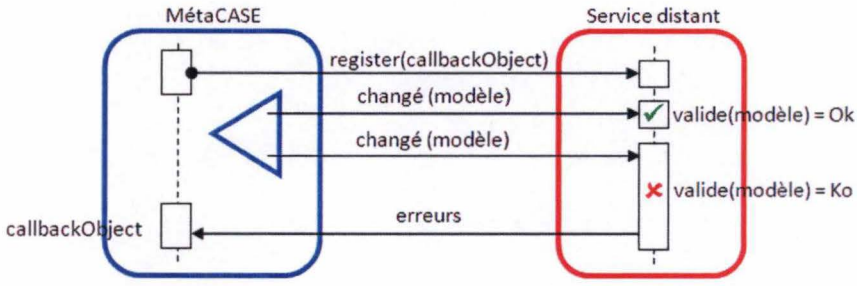

FIGURE 4.10 – Appel à un service distant de type événementiel

Remarque

Deux façons de transmettre les changements d'un modèle vers le service distant peuvent être imaginés :

- Le métaCASE transmet le modèle au service distant au début du scénario, puis transmet uniquement les deltas. Ce scénario implique une interprétation des deltas par le service distant, et leur intégration au scénario de base avant de pouvoir interpréter le modèle complet.
- Le métaCASE transmet le modèle complet à chaque fois qu'un événement se produit. Ce scénario augmente la quantité de données échangées et les délais de communication.

4.4.4 Scénario 12 : Appel à un service distant avec plugin intermédiaire

Les trois scénarios précédents présupposent que le métaCASE dispose des objets de callback nécessaires, supporte les protocoles des services distants, etc. L'ajout d'un plugin intermédiaire entre le métaCASE et le service distant permet de gagner en flexibilité : si le service distant est modifié ou remplacé, il suffit d'adapter le plugin à ses nouvelles exigences. De même, s'il faut se connecter vers un nouveau service distant, il suffit d'ajouter un nouveau plugin supportant le protocole et le type d'interaction imposé par le service (synchrone, asynchrone, avec objets de callback, etc).

La figure 4.11 illustre ce scénario dans le cas d'un service distant de type événementiel :

1. Le plugin enregistre son objet de callback auprès du service distant.
2. Le plugin s'enregistre auprès de certains événements du métaCASE.
3. Le métaCASE notifie le plugin de ces événements, qui les propage vers le service distant.
4. Le service distant, si besoin, appelle l'objet de callback du plugin et lui transmet le résultat de son traitement.
5. Le plugin propage le résultat vers le métaCASE par l'intermédiaire de son API, ou réalise un traitement interne.

4.5 Applications distantes accédant aux fonctionnalités du métaCASE

Le métaCASE agit ici comme fournisseur de services, accessibles par des applications distantes.

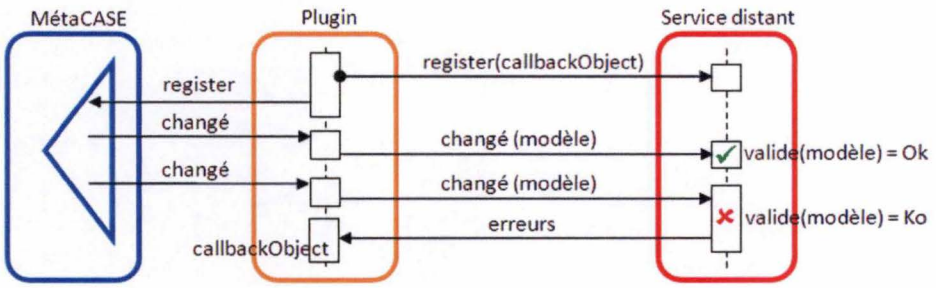

FIGURE 4.11 – Appel à un service distant avec plugin intermédiaire

4.5.1 Scénario 13 : Accès au métaCASE via son API

Ce scénario simple illustre l'accès d'une application distante au métaCASE, par l'intermédiaire de son API. Par exemple :

- Animation des modèles du métaCASE. Ce scénario est évoqué dans MetaEdit+ (page 2.3.3), où une application externe, générée à partir d'un modèle du métaCASE, anime ce même modèle durant son exécution.

Fonctionnalités requises

- L'API doit être accessible par une technologie d'accès distant.
- L'API doit permettre d'agir sur les modèles chargés en mémoire de travail.
- L'API doit permettre d'animer les objets graphiques des modèles (accéder leurs propriétés de mise en forme)

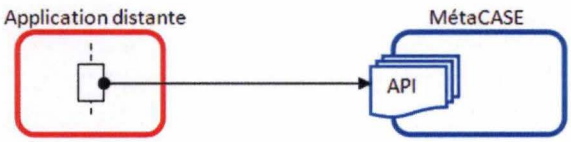

FIGURE 4.12 – Accès au métaCASE via son API

La figure 4.12 illustre un accès distant sur l'API du métaCASE.

4.5.2 Scénario 14 : Accès au métaCASE via une API locale

Dans ce scénario, l'API du métaCASE est localisée sur la machine de l'application distante, et accessible localement par celle-ci. L'application distante utilise l'API pour accéder les données du métaCASE, sans que celui-ci ne soit démarré. Comme suggéré dans Protégé (voir page 18), l'API se charge de faire un accès distant vers la base de données du métaCASE. Elle permet d'agir sur les modèles, de les lire et de les modifier. On peut également imaginer un scénario avec une base de données accédée localement par l'API.

Ce scénario permet à une application d'accéder directement les modèles, l'API jouant le rôle d'interface entre l'application et le format propriétaire des modèles du métaCASE.

Fonctionnalités requises

- L'API doit pouvoir être exportée hors du métaCASE.
- L'API doit permettre un accès distant vers la base de données du métaCASE.

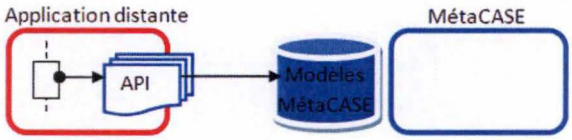

FIGURE 4.13 – Accès au métaCASE via une API locale

La figure 4.13 illustre un accès local à l'API du métaCASE :

1. L'API du métaCASE est accédée localement par l'application distante.
2. L'application distante utilise les fonctions de l'API pour accéder la base de données du métaCASE.

4.5.3 Scénario 15 : Accès par services web, avec serveur dans le métaCASE

Ce scénario envisage l'interrogation du métaCASE par l'intermédiaire des services web. Les services web pourraient être la technologie d'accès distant du scénario 13, et serviraient la communication entre l'application distante et l'API du métaCASE. Ils sont repris ici dans un scénario à part entière, illustrant des accès au métaCASE dans le contexte Internet. Par exemple :

- Un moteur de recherches Internet souhaite accéder un moteur d'inférences de Protégé pour orienter et affiner ses recherches.
- Un internaute accède les modèles via son browser : il se connecte sur un site web qui interroge le métaCASE par l'intermédiaire des services web. Le métaCASE renvoie le modèle demandé, le site web l'interprète et produit une visualisation HTML du modèle à destination de l'internaute. On peut également imaginer l'accès à certaines fonctions de mise à jour des modèles via le browser.
- Même scénario que le précédent mais en lecture seule : le métaCASE renvoie son modèle sous forme d'une image, directement visualisable dans un browser et sans besoin d'un site web intermédiaire pour l'interpréter.

Fonctionnalités requises

- Le métaCASE doit disposer d'un serveur web à l'écoute des requêtes HTTP émises par les applications distantes. Le serveur web est soit démarré automatiquement par le métaCASE, soit manuellement par l'utilisateur auquel cas une interface de commande doit être prévue.
- Le serveur web du métaCASE peut soit accéder directement les modèles, soit le faire via l'API.
- Le descriptif des services web (fichier WSDL) exposés par le métaCASE doit être connu de l'application distante : il est soit diffusé manuellement, soit enregistré dans un annuaire UDDI (voir le chapitre sur les services web, page 25).

La figure 4.14 illustre l'accès d'une application distante au métaCASE par l'intermédiaire des services web :

1. L'application distante, après avoir pris connaissance du descriptif du service web exposé par le métaCASE, fait une requête vers le serveur web du métaCASE et se met en attente de la réponse.
2. Le serveur web du métaCASE reçoit la requête, la traite, et renvoie la réponse vers l'application distante (suivant le protocole des services web).

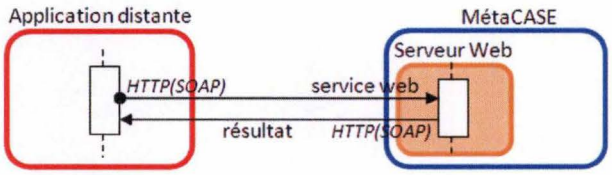

FIGURE 4.14 – Serveur de services web dans le métaCASE

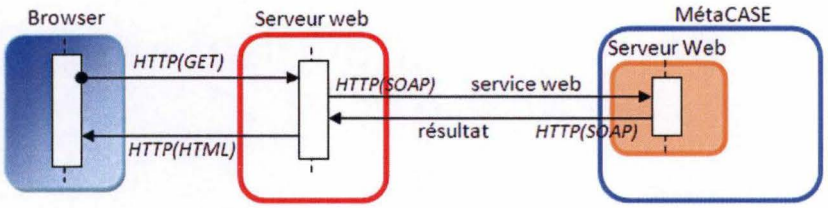

FIGURE 4.15 – Accès d'un browser au métaCASE

La figure 4.15 illustre l'accès d'un browser au métaCASE par l'intermédiaire des services web :

1. Le browser contacte le serveur du site web ad-hoc.
2. Le serveur du site web effectue une requête vers les services web du métaCASE.
3. Le métaCASE répond, et le serveur du site web interprète la réponse.
4. Le serveur du site web renvoie une réponse au browser, au format HTML.

4.5.4 Scénario 16 : Accès par services web, avec serveur hors du métaCASE

Ce scénario est une variante du précédent, le serveur web n'étant plus inclus dans le métaCASE. Celui-ci traduit les requêtes reçues par l'intermédiaire des services web en requêtes vers l'API du métaCASE, et inversement pour renvoyer la réponse vers l'application distante.

Un métaCASE existant, n'ayant pas inclus le mécanisme des services web dès sa conception, pourrait être complété grâce à une configuration comme celle-ci. Ce scénario peut être considéré comme l'adjonction d'un plugin "serveur web" au métaCASE.

La figure 4.16 illustre ce scénario.

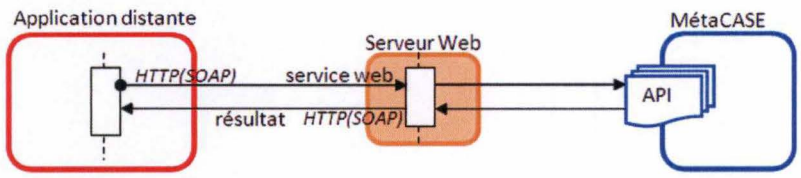

FIGURE 4.16 – Serveur de services web hors du métaCASE

Chapitre 5

Mise en oeuvre des scénarios

En parallèle de l'établissement des scénarios, les fonctionnalités nécessaires à leur mise en oeuvre ont été identifiées. Ainsi, de nouvelles fonctionnalités non présentes dans l'état de l'art se sont dégagées, dont l'accès du métaCASE à des services distants. De même, certaines fonctionnalités présentes dans l'état de l'art ont été mieux cernées, comme le besoin de pouvoir agir sur les modèles tant en mémoire de stockage qu'en mémoire de travail.

Ce chapitre commence, après avoir discuté de l'intérêt des différents scénarios, par établir une liste de ces fonctionnalités.

Ces fonctionnalités sont ensuite confrontées à l'état de l'art, afin d'identifier les architectures les mieux adaptées au besoin d'extensibilité.

Des propositions de mise en oeuvre sont ensuite avancées pour compléter les architectures existantes.

5.1 Intérêt des scénarios

Une partie de ces scénarios découle de l'état de l'art. Outre le fait que leurs applications pratiques sont intéressantes, ces scénarios sont ceux de métaCASE matures et largement utilisés, ce qui appuie leur intérêt. Ces scénarios sont basés sur les fonctionnalités suivantes :

- L'intégration du plugin dans l'interface utilisateur.
- La présence d'une API permettant d'agir sur les modèles en mémoire de travail et en mémoire de stockage.
- La présence d'un moteur d'événements auxquels les plugins peuvent s'abonner.
- L'accès au métaCASE par une application distante.
- L'accès aux modèles du métaCASE par une API utilisée à distance.
- L'ajout de formats de sauvegarde.
- L'association de plugins graphiques faisant le rendu des objets des modèles.

Les autres scénarios proviennent d'idées originales. Ils sont basés sur des fonctionnalités analogues à celles des scénarios énumérés ci-dessus (API, moteur d'événements), ainsi que sur les fonctionnalités suivantes :

- L'accès du métaCASE à un service distant.
- L'accès du métaCASE à un service distant via un plugin intermédiaire.
- L'accès au métaCASE par une application distante, avec technologie d'accès distant hors du métaCASE (scénario 16).

Ces fonctionnalités d'accès distant sont considérées comme intéressantes, au vu des scénarios qu'elles permettent : rendre les modèles du métaCASE accessibles à d'autres applications, interconnecter le métaCASE avec un autre métaCASE et profiter de ses fonctionnalités, connecter le métaCASE à une librairie de métamodèles, etc.

Tous les scénarios présentés sont considérés comme candidats au métaCASE idéal. Toutes les fonctionnalités nécessaires pour les mettre en oeuvre vont donc être étudiées.

5.2 Identification des besoins fonctionnels

Cette section inventorie les fonctionnalités nécessaires à la mise en oeuvre des scénarios. Ces fonctionnalités ont déjà été énumérées lors de la présentation des scénarios, et sont regroupées ici de manière unifiée.

5.2.1 Intégration dans l'interface

- Enregistrement du plugin (le métaCASE prend connaissance du plugin).
- API :
  - Le plugin signale qu'il souhaite être intégré dans l'interface du métaCASE.
    Fonctionnalité résultante :
    - Prévoir des emplacements libres dans l'interface de base du métaCASE.
  - Ajout de fonctionnalités dans l'interface du métaCASE :
    - Ajout de formats de sauvegarde.
    - Ajout de points de menu.
    - Ajout de contenu (voir remarque du scénario 6, page 60).
- Renseigner un plugin graphique pour les objets graphiques (par l'utilisateur, via l'interface du métaCASE).

5.2.2 Actions sur les modèles

- API :
  - Fonctions donnant accès aux modèles :
    - Dans la mémoire de stockage.
    - Dans la mémoire de travail.
  - Manipulations des modèles :
    - Accès en lecture et en écriture.
    - Accès aux propriétés graphiques des modèles :
      - Mise en forme de texte.
      - Déplacement d'objets graphiques.

5.2.3 Etre à l'écoute du métaCASE

- Moteur d'événements auxquels les plugins peuvent s'abonner :
  - Abonnement du plugin.
  - Propagation d'événements se produisant sur les modèles ou dans le métaCASE. Par exemple :
    - Texte Changé.
    - Objet Déplacé.
    - Objet Sélectionné (par exemple lors de la sélection d'un nouveau format de sauvegarde).

5.2.4 Chargement de plugins distants

5.2.5 Accès à un service distant

- Technologie d'accès distant incluse dans le métaCASE.
- Propagation des événements vers le service distant, suivant le format attendu.
- Différer la demande de validation et la demande de réponse.
  - Gestion d'une communication asynchrone.
- Accès à un service distant via un plugin intermédiaire :
  - Enregistrement du plugin.
  - Moteur d'événements (dialogue du métaCASE vers le plugin).
  - API (dialogue du plugin vers le métaCASE).

5.2.6 Accès par une application distante

- API :
  - Accessible par une application distante.
  - Utilisable localement par l'application distante, et accédant à la mémoire de stockage distante du métaCASE, sans démarrer celui-ci.

5.3 Tableau de couverture des besoins fonctionnels par l'état de l'art

	Protégé	GME & COM	MetaEdit+ & services web	Meta-Environment & Toolbus	Eclipse & OSGi
Scénarios 1 à 9 : plugins intégrés dans le métaCASE					
Enregistrement et intégration dans l'interface	✓	✓	✗	✗	✓
API : ajouts dans l'interface du métaCASE (points de menus, etc)	✓	✗	✗	✗	✓
API : ajout de contenu	✗	✗	✗	✗	✓
API : accès à la mémoire de stockage (lecture/écriture)	✓	✓	✓	?	⚑
API : accès à la mémoire de travail (lecture/écriture)	?	✓	✓	?	⚑
API : mise en forme de texte	?	?	?	?	⚑
API : déplacement d'objets graphiques	?	?	✓	?	⚑
Référence vers un plugin graphique	✗	✓	✗	⚑	⚑
Evénements	✓	✓	✗	✓	⚑
Chargement de plugins distants	⚑	✓	✗	✓	⚑
Scénarios 10 à 13 : accès à un service distant					
Technologie d'accès distant	✗	✓	✗	✓	⚑
Evénements distants	✗	✓	✗	✓	⚑
Communication distante asynchrone	✗	✗	✗	✓	⚑
Via plugin intermédiaire API + événements	⚑	⚑	✗	⚑	⚑
Scénarios 14 à 17 : accès distant au métaCASE					
API accessible à distance	⚑	✓	✓	✓	⚑
API "locale" accédant la mémoire de stockage du métaCASE	✓	✗	✗	✗	⚑

Légende

✓ Supporté en natif

⚑ Mise en oeuvre possible

? Non documenté

✗ Non supporté

Ce tableau met en évidence qu'aucune des architectures ne couvre nativement tout le panel de fonctionnalités. Cependant, il fait ressortir que certaines architectures peuvent être agencées afin de supporter des fonctionnalités supplémentaires.

L'architecture d'Eclipse apparaît comme la plus ouverte et la plus adaptable aux fonctionnalités supplémentaires. Moyennant un agencement, cette architecture peut toutes les implémenter.

5.4 Complétion de l'état de l'art pour couvrir les besoins fonctionnels

En partant du tableau comparatif de la section 3.3 page 52 et du tableau de couverture de la section 5.3 page 71, trois architectures sont candidates à être complétées : Protégé, GME et Eclipse. Elles présentent toutes les trois l'intégration de plugin dans l'interface, une interaction avec l'application, une pérennité élevée, une architecture d'événements, la possibilité de charger des plugins distants, etc. MetaEdit+ et le Toolbus sont beaucoup plus limités sur ces critères.

Des propositions de complétion de l'état de l'art sont avancées pour Eclipse et Protégé.

5.4.1 Remarques préliminaires

A propos des fonctionnalités non supportées

Ces fonctionnalités sont reprises dans le tableau avec, en regard du métaCASE concerné, une croix rouge (✘). Ces fonctionnalités ne pourront pas être mise en oeuvre dans le métaCASE sans une refonte de son code de base. Elles auraient dû être prises en compte lors de la conception du métaCASE.

A propos des fonctionnalités non documentées

Ces fonctionnalités sont reprises dans le tableau avec, en regard du métaCASE concerné, un point d'interrogation (?). Si ces fonctionnalités s'avèrent ne pas être présentes dans le métaCASE, elles ne pourront pas être mise en oeuvre sans une refonte de son code de base.

Quatre de ces fonctionnalités concernent l'API : l'accès à la mémoire de stockage, l'accès à la mémoire de travail, la mise en forme de texte et le déplacement d'objets graphiques. Pour rendre ces fonctionnalités disponibles, il faut modifier l'API.

Deux de ces fonctionnalités concernent l'accès distant : la disposition d'une technologie d'accès distant et les événements distants. De même que pour l'API, le code du métaCASE devrait être modifié. Cependant, l'emploi d'un plugin intermédiaire, comme suggéré dans le scénario 12, permet de les rendre disponibles et d'avoir un métaCASE ouvert à l'accès distant. Elles restent cependant non disponibles en natif.

A propos des scénarios d'accès distant

Ces scénarios d'accès distants, tels que présentés, impliquent que le métaCASE implémente une technologie d'accès distant particulière, le limitant aux possibilités de cette technologie (portabilité, temps de réponse, etc). L'implémentation d'une technologie d'accès distant particulière pourrait se justifier si cette technologie était éminemment populaire dans le monde des métaCASE. Au vu de l'état de l'art, ce n'est pas le cas : chaque métaCASE implémente une technologie différente.

Le choix d'une technologie limite le métaCASE aux services supportant cette technologie. De même, des services utilisant une même technologie n'ont pas nécessairement un mode de fonctionnement identique : dialogue synchrone, asynchrone, format des données échangées, etc. Il est peu probable d'arriver à implémenter une solution interne au métaCASE qui soit suffisamment flexible pour mettre en place un accès facile à de nouveaux services, et supportant un nombre suffisamment diversifié de format de données, etc.

Une solution flexible et facile à mettre en place est l'emploi de plugins intermédiaires, comme suggéré dans le scénario 12. Si un nouveau besoin non couvert par le métaCASE se présente, ou si une technologie évolue significativement, il suffit de procéder à une mise à jour ou à un ajout de plugin. De plus le choix d'une technologie d'accès distant n'est plus lié à la phase de conception du métaCASE, et peut être envisagé plus tard. Ceci pour autant que le métaCASE prévoie les ressources nécessaires pour l'ajout d'un tel plugin : une API et un moteur d'événements.

Dans le même esprit, la mise en place d'un serveur web permettant à une application externe d'accéder au métaCASE devrait se faire en dehors du métaCASE, comme suggéré dans le scénario 16. Un serveur web externe peut être ajouté après la phase de conception du métaCASE, et ses mises à jour sont plus faciles.

Un serveur web interne présente cependant deux caractéristiques qui pourraient jouer en sa faveur :

- Il est intégré dans l'interface du métaCASE.
- Il peut faire un accès direct aux fonctionnalités du métaCASE sans passer par l'API.

Le premier point n'est pas déterminant : un serveur externe pourrait également présenter une interface intégrée dans le métaCASE, tels les plugins des scénarios 1 à 4.
Le second point pourrait permettre au serveur d'agir plus finement sur le métaCASE, peut-être avec plus de rapidité. L'API est cependant sensée fournir les fonctionnalités nécessaires à une application externe, et à priori une fonctionnalité non prévue dans l'API n'a pas à être disponible via un autre biais. De même, l'API fournit déjà toute une série de services, comme la connexion à une base de données ou la gestion des erreurs, qu'il serait absurde de réécrire.

Pour ces raisons, la position du mémoire est de recommander l'usage de plugins intermédiaires pour les accès distants, entrants et sortants.

5.4.2 Protégé

a) Chargement de plugins distants

Protégé ne permet que de charger des plugins locaux, répondant à trois critères [49] :
- Ils doivent implémenter une interface particulière.
- Ils doivent être référencés dans le fichier manifest de Protégé.
- Ils doivent se trouver dans le répertoire /plugin de Protégé.

Proposition de mise en oeuvre

Un moyen de charger un plugin distant dans cette configuration est d'avoir un plugin local servant d'intermédiaire. La figure 5.1 illustre ce mécanisme avec un plugin de type TabWidget :
- Le plugin local implémente l'interface `AbstractTabWidget`, est référencé dans le fichier manifest et se trouve dans le répertoire /plugin.
- Le plugin distant implémente également les fonctions requises par l'interface `AbstractTabWidget` (en l'occurrence la fonction `initialize()`).

- Lorsque Protégé charge le plugin local, il appelle sa méthode d'initialisation. Le plugin local charge le plugin distant, et lui délègue le traitement en appelant sa méthode d'initialisation.
- Toutes les autres méthodes du plugin local, héritées de l'interface `AbstractTabWidget`, font appel aux méthodes correspondantes du plugin distant.

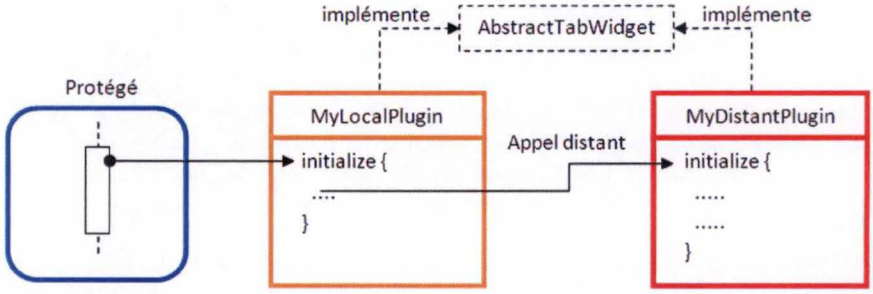

FIGURE 5.1 – Chargement de plugins distants dans Protégé

Le plugin local agit comme un wrapper : il emballe le plugin distant dans une couche compatible avec l'architecture de Protégé. Ce mécanisme n'est cependant pas une mise en oeuvre réelle d'un chargement de plugin distant : pour Protégé, le plugin reste un plugin local.

b) Accès à un service distant via un plugin intermédiaire

La mise en oeuvre est identique au chargement de plugins distants. Le plugin se charge de mettre la technologie d'accès distant en oeuvre, et de faire l'intermédiaire avec Protégé.

c) API accessible à distance

L'API de Protégé est accessible localement. Pour l'accéder à distance, il faut implémenter une couche intermédiaire qui se charge d'exposer ses fonctions au monde extérieur.

L'API de Protégé est utilisable en dehors de Protégé [46] (techniquement, il s'agit d'un fichier JAR autonome). Cette API pourrait être adjointe à un serveur exposant des services web, chaque service faisant le relais entre le monde extérieur et une fonction de l'API.

5.4.3 Eclipse et OSGi

Comme présenté dans le tableau de la page 71, Eclipse ne présente aucune limitation majeure à l'ensemble des fonctionnalités. Cela est dû aux caractéristiques suivantes :

- Eclipse est une architecture orientée plugins.
- Eclipse a été conçu pour servir de base à des applications extensibles de type client riche.
- Certaines limitations sont plus liées au métaCASE lui-même qu'à l'architecture de plugins, comme la mise en forme de texte. Eclipse étant une architecture vierge prête à accueillir le code d'un nouveau métaCASE, elle n'est pas encore orientée dans une direction fermant certaines portes fonctionnelles.

a) API : accès à la mémoire de stockage

L'API doit permettre à un plugin de lire et modifier les modèles stockés sur disque :

- Le plugin doit pouvoir récupérer une liste de modèles.

- Le plugin doit pouvoir choisir un modèle particulier.
- Les objets du modèle et leurs attributs doivent pouvoir être accédés ainsi que leurs propriétés de mise en forme.
- Le plugin doit pouvoir modifier les objets et les modèles, ce qui comprend leur création, leur mise à jour et leur suppression.

Proposition de mise en oeuvre

- Soit l'API est un ensemble de classes compactées sous forme d'une archive java (JAR), accessible par tous les plugins qui la référencient dans leur fichier manifest.
- Soit l'API est accessible via un point d'extension. Lors de l'activation du plugin, le point d'extension lui envoie une référence vers l'API par l'intermédiaire d'une méthode d'initialisation précisée dans le contrat, comme illustré dans la figure 5.2.

```
MyPlugin implements InterfaceAPI {
   Initialisation (API theAPI) {
      Database database = theAPI.getDatabase ();
      List modeles = database.getListeModeles();
   }
}
```

FIGURE 5.2 – Accès à l'API via un point d'extension

b) API : accès à la mémoire de travail

La mise en oeuvre est identique à l'accès à la mémoire de stockage.

c) API : mise en forme de texte et déplacement d'objets graphique

La mise en oeuvre est identique à l'accès à la mémoire de stockage.

d) Evénements

Le métaCASE doit prévoir des points d'extension pour permettre aux plugins de s'enregistrer aux événements se produisant sur les modèles et dans le métaCASE.

Proposition de mise en oeuvre

Le métaCASE présente un point d'extension "ModelEvents" auquel les plugins peuvent brancher des extensions. Le contrat du point d'extension exige que les plugins extensions spécifient :
- Les événements auxquels ils désirent s'abonner.
- Un objet de callback à appeler lors de la survenance des événements, qui implémente l'interface Evénement. L'interface Evénement oblige l'objet de callback à implémenter la méthode événementDéclenché, qui reçoit en paramètre le type d'événement et une référence vers l'entité qui l'a émis.

Les objets des modèles doivent produire des événements. Ces objets sont soit les objets natifs servant à la métamodélisation, et sont implémentés pour produire de tels événements, soit les objets issus de la métamodélisation et servant à la modélisation, et sont générés de telle façon à produire de tels événements. Dans les deux cas, il faut prévoir la fonctionnalité des événements au moment de la conception du métaCASE. Dans notre exemple, l'objet représente une personne avec deux attributs : un nom et un matricule. Il est représenté

graphiquement par un rond avec nom et matricule affichés dedans. Les événements sont par exemple :
- TexteChangé sur l'attribut nom
- TexteChangé sur l'attribut matricule
- ObjetDéplacé et ObjetSupprimé sur l'objet personne

Le métaCASE est abonné à tous les événements de tous les objets, et les transfère à chaque plugin extension, en appelant la fonction événementDéclenché de leur objet de callback. Cette architecture est illustrée dans la figure 5.3.

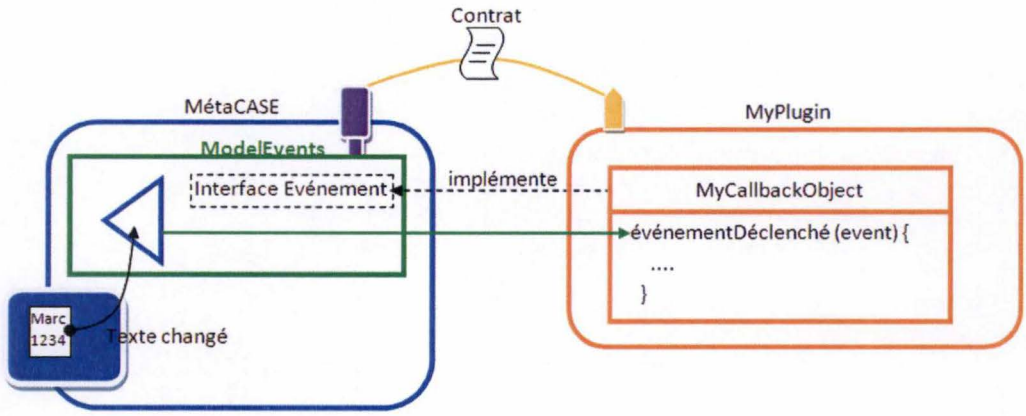

FIGURE 5.3 – Propagation d'événements du métaCASE vers un plugin

e) Chargement de plugins distants

Eclipse présente le même type de limitation que Protégé pour le chargement des plugins : ceux-ci doivent être copiés dans un répertoire précis. Une première solution, identique à celle de Protégé, consiste à avoir un plugin intermédiaire qui se charge de l'accès distant à un autre plugin.
Une deuxième solution possible avec Eclipse consiste à implémenter dans le métaCASE la possibilité d'accès distant. Cette solution est présentée en deux parties, la première pour les plugins tournant dans le même processus que le métaCASE, la seconde pour les plugins tournant sur une machine distante.

1) Plugins distants tournant dans le même processus que le métaCASE

Cette alternative est possible grâce aux chargeurs de classes Java [54]. Toute classe Java est chargée en mémoire par l'intermédiaire d'un chargeur de classes. La librairie de Java en propose plusieurs, qui permettent de charger des classes situées soit dans le classpath de l'application, soit dans le file system de la même machine, soit sur une autre machine du réseau en spécifiant une URL. Ils rapatrient le code de la classe identiquement à une ressource (fichier, image, etc), et le chargent ensuite en mémoire. Un chargeur de classes peut également être créé de toute pièce pour remplir un besoin particulier.

Le chargement d'une extension traditionnelle est déléguée au platform runtime. Une recherche dans l'API d'Eclipse et dans la documentation n'a pas permis de constater si le platform runtime permet ou non le chargement d'une classe distante.

Proposition de mise en oeuvre

En supposant que la fonctionnalité ne soit pas disponible, ce qui est peu probable, une solution facile peut de toute façon être mise en oeuvre. Si on reprend l'exemple illustré dans la figure 2.14 du chapitre Eclipse, le contrat d'un point d'extension spécifie l'emplacement de l'objet de callback du plugin. Le contrat devrait permettre à une extension de préciser l'emplacement de cet objet (ou plus précisément de sa classe) sur le réseau, par exemple par l'intermédiaire d'une url et d'un numéro de port. Le plugin hôte (c'est-à-dire le métaCASE) doit alors utiliser un chargeur de classes spécifique pour charger ces classes distantes.

2) Plugins tournant sur une machine distante

Java dispose d'une librairie permettant d'accéder des objets Java chargés sur une machine distante : la librairie RMI (Remote Method Invocation [55]). La figure 5.4 illustre le fonctionnement de RMI en l'appliquant aux métaCASE.

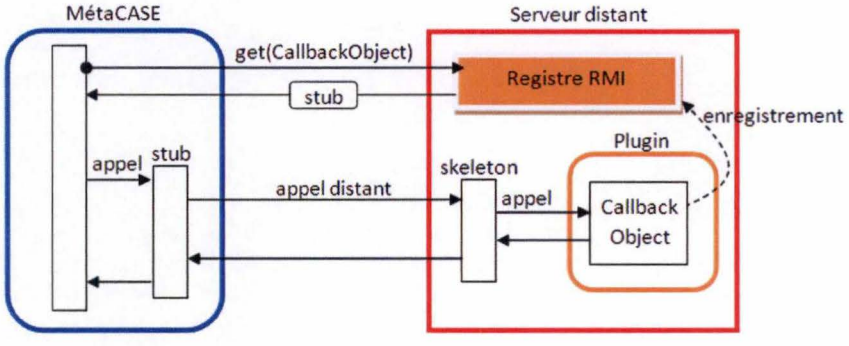

FIGURE 5.4 – Appel à un plugin distant avec RMI

- L'objet de callback du plugin distant s'enregistre dans un registre RMI. Ce registre répertorie les objets distants, et peut être interrogé pour obtenir les références vers ces objets.
- Le métaCASE interroge le registre RMI pour obtenir une référence vers l'objet de callback. En retour, le métaCASE reçoit un objet proxy, appelé *stub*. Vu de l'extérieur, ce stub est identique à l'objet de callback car il implémente les mêmes interfaces. Le métaCASE communique directement avec le stub, comme si c'était l'objet de callback lui-même. En interne, le stub se charge de lui relayer les appels du métaCASE.
- Le stub ne dialogue en réalité pas directement avec l'objet de callback, mais avec un objet intermédiaire appelé *skeleton*. Pour les échanges sur le réseau, le stub transforme l'appel de la méthode distante en une suite d'octets, ensuite retraduite par le skeleton en un appel local.

Proposition de mise en oeuvre

- Le métaCASE doit connaître l'adresse du registre RMI et le nom de l'objet de callback distant : ceux-ci seront obtenus via le contrat du point d'extension.
- Le métaCASE fait appel au registre RMI et reçoit le stub correspondant à l'objet de callback.
- Le stub implémentant les mêmes interfaces que l'objet de callback, il présente les fonctionnalités attendues par le métaCASE qui peut dès lors l'utiliser.

Il suffit donc d'avoir d'une part un contrat qui permet à l'extension de spécifier l'adresse du registre et le nom de l'objet distant à utiliser, et d'autre part d'implémenter l'appel au registre lui-même. Un chargeur de classes RMI pourrait être élaboré pour le métaCASE, se chargeant du dialogue avec le registre RMI et la réception du stub. Tous les autres aspects (démarrage du registre RMI, enregistrement de l'objet distant dans le registre, démarrage de la machine virtuelle distante, création des stub et skeleton, etc) ne sont pas à charge du métaCASE.

D'autres architectures que RMI peuvent être employées, comme par exemple CORBA (Common Object Request Broker Architecture [40]) qui fonctionne de façon analogue (stub, skeleton, registre). Un avantage sérieux de CORBA est de permettre à une application Java d'accéder à des objets distants non-Java, contrairement à RMI qui est "tout Java". RMI est cependant plus facile à mettre en oeuvre.

Une fois la classe chargée, que ce soit via un chargeur de classes ou via RMI, le plugin hôte ne voit aucune différence avec une classe locale.

f) Evénements distants

Les événements du métaCASE doivent être propagés en employant la technologie d'accès distant définie au point précédent. Deux cas peuvent se présenter :

1. La technologie d'accès distant est suffisante.
   Dans le cas d'une technologie comme RMI, la propagation d'un événement vers un objet distant se fait identiquement à la propagation vers un objet local : il suffit de propager les événements vers le stub. Du point de vue du métaCASE, le stub est l'objet distant lui-même, et les événements sont transmis tels quels.
2. La technologie d'accès distant est insuffisante.
   Dans le cas d'une technologie comme les services web, la propagation des événements du métaCASE nécessite une transformation vers le format des services web. Un adaptateur doit être prévu pour établir une correspondance entre chaque événement et le service web ad hoc. Dans le cas de RMI, cet adaptateur est inhérent à la technologie : il s'agit du stub.

g) API "locale" accédant la mémoire de stockage du métaCASE

Comme pour Protégé, l'API peut être compactée dans une archive Java. Cette archive peut être utilisée comme librairie auprès d'une autre application. Celle-ci renseigne l'archive dans son classpath, et peut dès lors accéder ses fonctionnalités. Pour que l'API puisse être ainsi utilisée hors du métaCASE, il faut qu'elle n'ait aucune dépendance vers d'autres librairies du métaCASE. Si c'était le cas, il faudrait exporter ces librairies également.

Quatrième partie

Conclusions

Limites et recherches futures

Ce mémoire a étudié les besoins d'extensibilité des métaCASE. Néanmoins, le sujet n'a pas été épuisé, comme l'indiquent les commentaires suivants :

- L'état de l'art pourrait être complété afin d'élargir le panel de fonctionnalités supportées par les métaCASE. Le nombre de métaCASE étudiés dans ce mémoire a dû être limité à cinq, vu l'ampleur de chaque étude individuelle.

- Le panel de scénarios n'a pas été établi avec l'aide d'utilisateurs de métaCASE, et passe peut-être sous silence des scénarios utiles et innovants. L'apport de l'état de l'art tempère toutefois ce risque, mais il serait intéressant d'avoir le retour d'utilisateurs expérimentés.

- Ce mémoire envisage presque exclusivement les besoins d'extensibilité du point de vue de l'utilisateur, et non du point de vue du concepteur. Cependant, des fonctionnalités faisant partie du noyau du métaCASE pourraient également être externalisées dans des plugins, afin de faciliter la maintenance et la mise à jour du logiciel lui-même. L'architecture interne des métaCASE n'a pas été étudiée et, par conséquent, sa modularité non plus.

- Plusieurs points concernant les plugins n'ont pas été soulevés. Ces points sont repris dans la liste suivante, non exhaustive :
  - La gestion des versions.
  - La gestion des transactions.
  - La gestion des licences.
  - La mise à jour automatique.
  - Le packaging et la distribution.

  Certains de ces points, dont la gestion des transactions, auraient pu faire l'objet d'un des scénarios du chapitre 4. Cependant, l'étude de la totalité des aspects d'extensibilité dépasse le cadre de ce seul mémoire.

- Les technologies proposées pour la mise en oeuvre des scénarios au chapitre 5 n'ont pas été explorées en profondeur. La problématique du chargement des ressources ou du chargement de packages par les chargeurs de classes n'est, par exemple, pas évoquée. Ces technologies devraient être étudiées plus en détail afin d'évaluer leur adéquation aux métaCASE.

Conclusion

L'état de l'art, établi au chapitre 2, a tout d'abord fait apparaître des architectures fort différentes les unes des autres, du point de vue des technologies employées et des fonctionnalités disponibles.

Afin de pouvoir comparer ces architectures hétérogènes, des critères de comparaison transversaux ont été déterminés au chapitre 3, et un tableau comparatif a été établi. Celui-ci a mis en lumière les points suivants :

- Certaines fonctionnalités sont mal supportées par beaucoup de métaCASE, dont l'ajout et la suppression dynamiques de plugins, ainsi que les interactions entre plugins.
- Peu de fonctionnalités sont communes à tous les métaCASE. Les critères d'interopérabilité et d'interaction avec l'application sont les seuls à être supportés par tous.
- Les technologies sur lesquelles s'appuient les architectures orientent les fonctionnalités qu'elles peuvent mettre à disposition. Ainsi, MetaEdit+ basé sur le standard des services web ne permet aucune intégration de plugins, tandis que Protégé et GME basés sur des architectures propriétaires en permettent l'intégration complète.
- Eclipse est la seule architecture couvrant l'ensemble des fonctionnalités.

Des critères de réemploi ont également été soulevés afin de voir si ces architectures peuvent être facilement réutilisées pour construire de nouveaux métaCASE. A ce stade de l'étude, GME, MetaEdit+ et Eclipse ont paru les plus appropriés.

La diversité des architectures de l'état de l'art a permis d'identifier un premier panel de fonctionnalités variées, réparties en trois groupes :

- Le chargement de plugins locaux.
- Le chargement de plugins distants.
- L'accès d'une application distante au métaCASE.

Afin de compléter ces fonctionnalités, plusieurs scénarios d'extensibilité ont été imaginés au chapitre 4.

En parallèle de l'établissement de ces scénarios, les fonctionnalités nécessaires à leur mise en oeuvre ont été identifiées. Ainsi, de nouvelles fonctionnalités non présentes dans l'état de l'art se sont dégagées, dont l'accès du métaCASE à des services distants. De même, certaines fonctionnalités présentes dans l'état de l'art ont été mieux cernées, comme le besoin de pouvoir agir sur les modèles tant en mémoire de stockage qu'en mémoire de travail.

La mise en commun de l'état de l'art et des scénarios a produit le panel de fonctionnalités final, repris au chapitre 5. Un tableau, mettant en regard ces fonctionnalités avec les architectures des métaCASE, a tout d'abord montré qu'aucune d'elles ne couvre totalement ce panel. Cependant, et ce qui est plus intéressant, il a mis en évidence que certaines architectures peuvent être agencées afin de supporter des fonctionnalités supplémentaires. L'architecture d'Eclipse apparaît comme la plus ouverte et la plus

adaptable aux fonctionnalités supplémentaires. Moyennant un agencement, cette architecture peut toutes les implémenter.

Des propositions d'implémentation ont ensuite été données, afin de montrer comment les architectures existantes peuvent être adaptées pour supporter plus de fonctionnalités. Le mémoire recommande enfin d'implémenter les fonctionnalités d'accès distant non pas dans le noyau de l'architecture, mais dans des plugins externes aux métaCASE.

En conclusion, ce mémoire a identifié les besoins d'extensibilité des métaCASE, en combinant l'étude de fonctionnalités existantes avec celle de fonctionnalités souhaitées, et a évalué leur possibilité de mise en oeuvre dans des architectures existantes.

Bibliographie

[1] Bartlett N., *A Comparison of Eclipse Extensions and OSGi Services*, http ://www.eclipsezone.com/articles/extensions-vs-services/, Mis à jour le 01/02/2007, Consulté le 23/11/2007

[2] CWI Centrum Wiskunde & Informatica, *Meta-Environment*, http ://www.meta-environment.org, Mise à jour : 2006, Consulté le 08/02/2009

[3] Donsez D., *La plate-forme dynamique de services OSGi*, http ://sardes.inrialpes.fr/ecole/livre/pub/Chapters/OSGI/osgi.html, Mis à jour le 08/01/2007, Consulté le 15/08/2009

[4] Eclipse Foundation, *Eclipse.org Home*, http ://www.eclipse.org, Mise à jour : 2008, Consulté le 19/03/2008

[5] Eclipse Foundation, *How do I make my plug-in dynamic aware ?*, http ://wiki.eclipse.org/FAQ_How_do_I_make_my_plug-in_dynamic_aware/, Mise à jour : 2004, Consulté le 10/07/2009

[6] Eclipse Foundation, *How do I make my plug-in dynamic enabled ?*, http ://wiki.eclipse.org/FAQ_How_do_I_make_my_plug-in_dynamic_enabled/, Mise à jour : 2004, Consulté le 10/07/2009

[7] Eclipse Foundation, *Notes on the Eclipse Plug-in Architecture*, http ://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html, Mis à jour le 03/07/2003, Consulté le 19/03/2008

[8] Eclipse Foundation, *What is a dynamic plug-in ?*, http ://wiki.eclipse.org/FAQ_What_is_a_dynamic_plug-in/, Mise à jour : 2004, Consulté le 10/07/2009

[9] Heering J., de Jong H.A., de Jonge M., Kuipers T., Klint P., Moonen L., Olivier P.A., Scheerder, van den Brand M.G.J., van Deursen A., Vinju J.J., Visser E., Visser J.,*The ASF+SDF Meta-Environment : a Component-Based Language Development Environment*, CWI Centrum Wiskunde & Informatica, Pays-Bas, 2001. Article disponible sous http ://citeseerx.ist.psu.edu/viewdoc/summary ?doi=10.1.1.64.5948

[10] Hogeboom M., Lin F., Esmahi L., Yang C., *Constructing Knowledge Bases for E-Learning Using Protégé 2000 and Web Services*, National Research Council of Canada, Mars 2005. Article disponible sous http ://iit-iti.nrc-cnrc.gc.ca/iit-publications-iti/docs/NRC-47441.pdf

[11] Hunt C., *Component Object Model (COM) Development on Mac OS X*, www.macdevcenter.com/pub/a/mac/2004/04/16/com_osx.html, Mis à jour le 16/04/2004, Consulté le 20/04/2009

[12] Isazadeh H., *Architectural analysis of METACASE*, Department of Computing and Information Science, Queen's University Kingston, Ontario, Canada, 1997. Article disponible sous http ://www.collectionscanada.gc.ca/obj/s4/f2/dsk2/ftp04/mq20654.pdf

[13] Isazadeh H., Lamb D.A., *CASE Environments and MetaCASE Tools*, Department of Computing and Information Science, Queen's University Kingston, Ontario K7L 3N6, 1997. Article disponible sous ftp ://ftp.qucis.queensu.ca/pub/reports/1997-403.pdf

[14] IBM, *Developing Eclipse plug-ins*,
http ://www.ibm.com/developerworks/opensource/library/os-ecplug/, Mis à jour le 01/06/2002, Consulté le 19/03/2008

[15] IBM, *Understanding how Eclipse plug-ins work with OSGi*,
http ://www.ibm.com/developerworks/opensource/library/os-ecl-osgi/, Mis à jour le 06/06/2006, Consulté le 19/03/2008

[16] IBM, *Working the Eclipse Platform*,
http ://www.ibm.com/developerworks/linux/library/os-plat/, Mis à jour le 01/11/2001, Consulté le 19/03/2008

[17] de Jong H., Klint P.,*ToolBus : The Next Generation*, CWI Centrum Wiskunde & Informatica, Pays-Bas, page 4, 2003. Article disponible sous
http ://homepages.cwi.nl/ paulk/publications/fmco02.pdf

[18] de Jong H., Klint P., Lankamp A., Olivier P.,*DocGuide to ToolBus Programming*, CWI Centrum Wiskunde & Informatica, Pays-Bas, pages 2-6 et 12-20, 15/07/2008. Article disponible sous
http ://www.meta-environment.org/doc/books//technology/toolbus-guide/toolbus-guide.pdf

[19] Jung J., *Meta-Modelling Support for a General Process Modelling Tool*, University Duisburg-Essen, Germany, 2005. Article disponible sous
http ://citeseerx.ist.psu.edu/viewdoc/download ?doi=10.1.1.97.1415&rep=rep1&type=pdf

[20] Klint P., *Home page of Paul Klint*, http ://homepages.cwi.nl/p̃aulk/, Consulté le 08/02/2009

[21] Klint P.,*The ToolBus, a service-oriented architecture for language-processing tools*, CWI Centrum Wiskunde & Informatica, Pays-Bas, pages 23 et 36-39, 2007. Article disponible sous
http ://www.win.tue.nl/ipa/archive/springdays2007/toolbusIPA2007.pdf

[22] Klint P., Vinju J.,*The Architecture of The Meta-Environment*, CWI Centrum Wiskunde & Informatica, pages 2, 5 et 14-15, Pays-Bas, 14/05/2008. Article disponible sous
http ://www.meta-environment.org/doc/books//meta-environment/architecture-meta-environment/architecture-meta-environment.pdf

[23] Klint P., Vinju J.,*The Extension Points of The Meta-Environment*, CWI Centrum Wiskunde & Informatica, Pays-Bas, 01/05/2008. Article disponible sous
http ://www.meta-environment.org/doc/books//meta-environment/extending-meta-environment/extending-meta-environment.pdf

[24] Lédeczi A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason C., Nordstrom G., Sprinkle J., Volgyesi P, *The Generic Modeling Environment*, Vanderbilt University, Institute for Software Integrated Systems. Article disponible sous
http ://www.isis.vanderbilt.edu/sites/default/files/GME2000Overview.pdf

[25] Lédeczi A., Maroti M., Völgyesi P., *The Generic Modeling Environment : Technical Report*, Vanderbilt University, Institute for Software Integrated Systems. Article disponible sous
http ://www.isis.vanderbilt.edu/sites/default/files/GMEReport.pdf

[26] Lemieux J-M., McAffer J., *Eclipse Rich Client Platform : Designing, Coding, and Packaging Java Applications*, Addison Wesley Professional, 2005. Ouvrage disponible sous
http ://book.javanb.com/eclipse-rich-client-platform-designing-coding-and-packaging-java-applications-oct-2005/toc.html

[27] MetaCase Company, *Getting Started with the MetaEdit+ API*,
http ://www.metacase.com/support/45/manuals/watchtut/we-5_1.html, Consulté le 06/01/2009

[28] MetaCase Company, *How is the code-generation linked to MetaEdit+ ?*,
http ://www.metacase.com/faq/search.asp ?Search=api#MWBHow_is_the_code-gene, Consulté le 06/01/2009

[29] MetaCase Company, *La technologie METACASE*, MetaCase Finland, 2004. Article disponible sous http ://www.metacase.com/papers/ABC_to_metaCASE_in_French.pdf

[30] MetaCase Company,*MetaEdit+ metaCASE tool : Technical Summary*, MetaCase Company. Article disponible sous http ://www.metacase.com/papers/MetaEditPlus.pdf

[31] MetaCase Company, *MetaCase - Domain-Specific Modeling with MetaEdit+*, http ://www.metacase.com, Mise à jour : 2009, Consulté le 06/01/2009

[32] MetaCase Company, *Success stories*, http ://www.metacase.com/cases/, Consulté le 25/08/2009

[33] Microsoft, *COM : Component Object Model Technologies*, http ://www.microsoft.com/com/default.mspx, Mise à jour : 2009, Consulté le 20/04/2009

[34] Microsoft MSDN Library, *The Component Object Model : A Technical Overview*, http ://msdn.microsoft.com/en-us/library/ms809980.aspx, Version 2009, Consulté le 20/04/2009

[35] Molnar Z., Balasubramanian D., Lédeczi A.,*An Introduction to the Generic Modeling Environment*, Vanderbilt University, Institute for Software Integrated Systems, 2005. Article disponible sous http ://www.dsmforum.org/events/MDD-TIF07/GME.2.pdf

[36] Musen M.A., *Building Ontologies with Protégé-2000*, Stanford Medical Informatics, Stanford University. Présentation disponible sous http ://www.fas.org/dh/conferences/MusenDigitalHuman.ppt

[37] Nebut M., Laboratoire d'Informatique Fondamentale de Lille *Aperçu du langage OCL*, http ://www2.lifl.fr/ñebut/ens/svl/coursOcl.html, Mis à jour le 24/02/2006, Consulté le 20/04/2009

[38] Noy N.F., Sintek M., Decker S., Crubézy M., Fergerson R.W., Musen M.A., *Creating Semantic Web Contents with Web Contents with Protégé-2000*, Stanford University, pages 62-65, Avril 2001. Article disponible sous http ://icc.mpei.ru/documents/00000829.pdf

[39] Noy N.F., Sintek M., Decker S., Crubézy M., Fergerson R.W., Musen M.A., *Protégé-2000 : A Flexible and Extensible Ontology-Editing Environment*, Stanford Medical Informatics, Stanford University. Présentation disponible sous http ://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-52/oas01-noy-presentation.pdf

[40] Object Management Group*CORBA Home Page*, http ://www.corba.org/, Mise à jour : 2009, Consulté le 20/08/2009

[41] OSGi Alliance, *OSGi Alliance*, http ://www.osgi.org, Mise à jour : 2009, Consulté le 12/06/2009

[42] Pritchard J., *COM and CORBA side by side*, Addison-Wesley, pages 18-23, 1999.

[43] Shankar R.D., Tu S.W., Musen M.A., *Use of Protégé-2000 to Encode Clinical Guidelines*, Stanford Medical Informatics, 2002. Article disponible sous http ://citeseerx.ist.psu.edu/viewdoc/summary ?doi=10.1.1.20.2993

[44] Stanford Center for Biomedical Informatics Research, *An introduction to developing plug-ins*, http ://protege.stanford.edu/doc/pdk/plugins/overview.html, Mise à jour : 2009, Consulté le 16/02/2009

[45] Stanford Center for Biomedical Informatics Research, *Class Project*, http ://protege.stanford.edu/protege/3.4/docs/api/core/edu/stanford/smi/protege/model/Project.html, Mise à jour : 2009, Consulté le 17/04/2009

[46] Stanford Center for Biomedical Informatics Research, *Core protégé & protégé-frames apis*, http ://protege.stanford.edu/doc/pdk/kb-api.html, Mise à jour : 2009, Consulté le 17/04/2009

[47] Stanford Center for Biomedical Informatics Research, *Createproject & export versus back-end plug-ins*, http ://protege.stanford.edu/doc/pdk/plugins/import_and_export_plugins.html , Mise à jour : 2009, Consulté le 17/04/2009

[48] Stanford Center for Biomedical Informatics Research, *How to write a project plug-in*, http ://protege.stanford.edu/doc/pdk/plugins/project_plugin.html, Mise à jour : 2009, Consulté le 17/04/2009

[49] Stanford Center for Biomedical Informatics Research, *How to write a tab widget plug-in*, http ://protege.stanford.edu/doc/pdk/plugins/tab_widget.html, Mise à jour : 2009, Consulté le 17/04/2009

[50] Stanford Center for Biomedical Informatics Research, *Instructions for declaring dependencies between Protege plug-ins*, http ://protegewiki.stanford.edu/index.php/PluginDependencies, Mise à jour : 2009, Consulté le 17/04/2009

[51] Stanford Center for Biomedical Informatics Research, *Protege 3.4.1 API*, http ://protege.stanford.edu/protege/3.4/docs/api/core/, Mise à jour : 2009, Consulté le 17/04/2009

[52] Stanford Center for Biomedical Informatics Research, *TabWidget example source code*, http ://protege.stanford.edu/doc/pdk/plugins/plugin-examples-src.zip, Mis à jour le 08/01/2007, Consulté le 17/04/2009

[53] Stanford Center for Biomedical Informatics Research, *Welcome to Protégé*, http ://protege.stanford.edu, Mise à jour : 2009, Consulté le 16/02/2009

[54] Sun Microsystems, *Class ClassLoader*, http ://www.j2ee.me/j2se/1.4.2/docs/api/java/lang/ClassLoader.html, Mise à jour : 2003, Consulté le 23/06/2009

[55] Sun Microsystems*Introduction à RMI*, http ://java.sun.com/javase/technologies/core/basic/rmi/index.jsp, Mise à jour : 2009, Consulté le 20/08/2009

[56] Travassos W., *Transformation Rules with MetaEdit+*, McGill University, Montreal, 2008. Article disponible sous http ://msdl.cs.mcgill.ca/people/hv/teaching/MSBDesign/projects/repository /Willer%20Travassos/projectReport.pdf

[57] Vanderbilt University, Institute for Software Integrated Systems, *Extensibility of GME*, http ://w3.isis.vanderbilt.edu/Projects/gme/extensibility.html, Consulté le 03/01/2009

[58] Vanderbilt University, Institute for Software Integrated Systems,*GME 5 User's Manual*, Vanderbilt University, Institute for Software Integrated Systems, pages 27, 70 et 94, 2005. Article disponible sous http ://www.isis.vanderbilt.edu/sites/default/files/GMEUMan.pdf

[59] Vanderbilt University, Institute for Software Integrated Systems, *GME Overview*, http ://www.isis.vanderbilt.edu/Projects/gme/, Mis à jour le 06/12/2008, Consulté le 03/01/2009

[60] Vanderbilt University, Institute for Software Integrated Systems, *Lesson 8 : Building Java Based Interpreters*, http ://w3.isis.vanderbilt.edu/Projects/gme/Tutorials/Lesson8.html, Consulté le 03/01/2009

[61] Vanderbilt University, Institute for Software Integrated Systems, *MetaGME*, http ://www.isis.vanderbilt.edu/Projects/gme/#tabs-2, Mis à jour le 06/12/2008, Consulté le 03/01/2009

[62] W3C Working Group, *Web Services Architecture*, http ://www.w3.org/TR/ws-arch/, Mis à jour le 11/02/2004, Consulté le 11/07/2009

[63] Wikipédia, *CASE tool*, http ://en.wikipedia.org/wiki/CASE_tool, Mis à jour le 30/12/2008, Consulté le 06/01/2009

[64] Wikipédia, *Component Object Model*, http ://en.wikipedia.org/wiki/Component_Object_Model, Mis à jour le 08/04/2009, Consulté le 20/04/2009

[65] Wikipédia, *MetaCASE tool*, http ://en.wikipedia.org/wiki/MetaCASE_tool, Mis à jour le 01/01/2009, Consulté le 06/01/2009

[66] Wikipédia *Object Constraint Language*, http ://en.wikipedia.org/wiki/Object_Constraint_Language, Mis à jour le 09/02/2009, Consulté le 20/04/2009

[67] Wikipédia, *Ontologie*, http ://fr.wikipedia.org/wiki/Ontologie_(informatique), Mis à jour le 10/04/2009, Consulté le 17/04/2009

[68] Wikipédia, *Service Web*, http ://fr.wikipedia.org/wiki/Service_Web, Mis à jour le 06/07/2009, Consulté le 11/07/2009