RenPar'17 / SympA'2006 / CFSE'5 / JC'2006
Canet en Roussillon, 4 au 6 octobre 2006

Usage des langages de script pour des composants adaptables.

Didier Donsez

Laboratoire LSR (UMR CNRS 5523) , Equipe ADELE
220 rue de la Chimie, Domaine Universitaire, BP 53
38041 Grenoble, Cedex 9, France
Didier.Donsez@imag.fr

Résumé
La programmation orientée composant est une méthodologie du génie logiciel permettant ainsi un développement modulaire et une maintenance plus aisée des logiciels complexes. Les langages de scripts offrent depuis longtemps la capacité de faire évoluer le code d'un programme au cours de l'exécution. Cet article s'intéresse à l'impact de l'usage des langages de script dans la programmation orientée composants. Il revisite les modèles de composants Fractal d'ObjectWeb et SCR de l'OSGi Alliance et décrit les deux implémentations de modèles de composants scriptables.

Mots-clés : Programmation orientée composant, adaptation dynamique, programmation incrémentale, langages dynamiques, langages de script.

---

1. Introduction

La programmation orientée composant [1,2] est une méthodologie du génie logiciel qui se focalise sur la réutilisation et l'intégration de composants existants pour le développement de logiciels. Plusieurs modèles de composants sont opérationnels. Parmi ceux-ci, certains sont destinés à des domaines d'application spécifiques, tels que EJB [3], CCM [4], CCA [5], SCR [6] tandis que d'autres comme Fractal [7] ou *feu*-Avalon [8] ont vocation à être généralistes. Pour ces modèles, le cycle de vie de l'application suit les phases strictement successives de développement, assemblage, configuration, déploiement et exécution. Ces phrases sont généralement sous la responsabilité de plusieurs acteurs.
La maintenance de l'application à composants comporte généralement des opérations de reconfiguration et d'adaptation de composants. La reconfiguration consiste à changer les valeurs des propriétés des composants ou à changer l'architecture de l'application en redéfinissant les connections entre les composants. L'adaptation consiste en général à remplacer un composant par un autre qui peut éventuellement fournir et requérir des contrats différents du composant remplacé. Généralement, l'adaptation requière de déconnecter le composant de l'application, de transférer son état courant vers un nouveau composant et puis de connecter ce dernier aux autres composants de l'application. Généralement, l'opération d'adaptation requiert généralement l'arrêt de l'application et oblige à reprendre le cycle de vie à partir de la phase d'assemblage. L'adaptation dynamique [9,10] consiste à introduire des modifications dans l'application au cours de son exécution sans nécessairement arrêter totalement l'application. Un des points délicats de l'adaptation dynamique est le transfert de l'état d'un composant à un autre quand l'état est constitué d'objets difficilement transférable tel que qu'un *socket* ou bien la pile d'exécution d'une *thread*.
L'adaptation est un événement de plus en plus fréquent dans le cycle de vie d'applications pour lesquels leurs concepteurs recherchent des propriétés comme la sensibilité au changement de l'environnement (*context-awareness*) [11] ou bien les propriétés autonomiques (*self-**) [12]. Ces dernières requièrent le plus souvent le placement dynamique de sondes/capteurs sur le logiciel au cours du fonctionnement ou bien encore la modification dynamique des règles de gestion autonomiques. Ceux-ci peuvent être programmées et conditionnées sous la forme de composants comme c'est la cas dans les canevas Lewys et Jade. L'opération d'adaptation doit répondre à plusieurs contraintes comme la sécurité, la fiabilité, la complétude et la reprise sur erreur (*recovery*) possibilité de retour en arrière ou de compenser dans le cas où l'adaptation ne peut se terminer avec succès.

Les langages de script [13] offrent depuis longtemps la capacité de faire évoluer le programme d'application en cours de l'exécution. Bien que la plupart d'entre eux sont généralistes (*general-purpose*), ceux-ci se retrouvent le plus souvent dans des niches particulières de développement telles que la programmation de scripts pour serveurs Web (PERL, Python) , la programmation de pages HTML dynamique pour (JavaScript/EcmaScript), la programmation d'interfaces graphiques (Tcl), pour l'invocation dynamique de services distants (IDLScript, E4X), pour des présentations dynamiques et interactives (ActionScript), pour la définition de règles pour les gestionnaires autonomiques (DRules), pour la programmation de *process flows* et de *work flows* (BPEL). Ceux-ci peuvent être également embarqués dans des langages déclaratifs tels que HTML AJAX ou Apache ANT via des canevas d'intégration tels que BSF et du JSR223 qui est désormais intégré à Java 6.0 Mustang.
La concision et la simplicité d'écriture des programmes développés rendent ces langages de script [14,15] attractifs au yeux des développeurs débutants et des experts métiers non-informaticiens au prix généralement d'une maintenance plus difficile des codes développés. La propriété qui nous intéresse dans cet article est que plusieurs de ces langages sont dynamiques et qu'ils autorisent une programmation incrémentale au cours de l'exécution du programme développé [16]. Le développeur développe et exécute son programme simultanément sans passer par le cycle traditionnel itératif développement-déploiement-exécution. Dans le contexte de l'adaptation dynamique, la programmation incrémentale apportée par les langages de script présente un intérêt majeur : l'état de l'application en cours d'exécution est relativement résistant aux changements et aux évolutions du code (fonctionnel ou non-fonctionnel) de l'application. Elle soulage ainsi le concepteur de l'utilisation de mécanismes souvent *ad hoc* pour le transfert d'état et pour le redémarrage des composants dont le comportement (code fonctionnel) a été modifié. De plus, elle ne remet pas en question les liaisons établies entre les composants.
Dans cet article, nous soutenons que la programmation incrémentale offerte par les langages de script est intéressante à plusieurs égards pour l'adaptation dynamique des composants. Elle est particulièrement intéressante pour le prototypage rapide de maquettes, le développement de programmes à usage unique qui ne sont exécutés que peu de fois par leurs auteurs (*workflow*...), la réalisation de programmes incomplets (*unfinished computing*) et enfin dans le développement des programmes utilisant un environnement fluctuant, éphémère et non connu à l'avance. Grâce aux langages de scripts, les composants concernés dans l'application peuvent ainsi évoluer fonctionnellement au fur et à mesure des évolutions du contexte sans requérir de redéploiement. En contrepartie, la programmation orientée composant offre aux langages de script l'opportunité d'être plus largement utilisés (en partie) dans le développement de très gros logiciels. Jusqu'à présent, la maintenance coûteuse des scripts limitait leur usage à grande échelle.
Pour cela, nous nous sommes intéressés à regarder l'impact de l'usage de la programmation incrémentale dans les modèles de composants. Nous avons revisité deux modèles de composants reconnus par rapport aux apports de celle-ci. Ces deux modèles sont celui de Fractal [7] défini par le consortium ObjectWeb et celui du SCR [6] spécifié par l'OSGi Alliance.
Cet article est structuré de la façon suivante. La section 2 présente notre proposition pour des extensions sur un modèle général de composants utilisant des langages de scripts. La section 3 présente FractScript, une implémentation partielle du modèle de composants Fractal pour le développement de composants primitifs avec des langages de script. La section 4 présente SCRScript, une extension du modèle de composants orientés services SCR pour le support de langages de script pour la programmation de services. La section 5 positionne nos travaux. La section 6 conclut avec quelques perspectives.

2. Proposition

Les modèles de composants opérationnels sont, pour la plupart, basés sur quelques dénominateurs communs qui sont des patrons de conception de la fabrique et des principes comme l'inversion de contrôle ainsi que la notion de type. Le type du composant est souvent lié aux facettes et réceptacles de ce dernier. La fabrique construit les instances d'un type de composant. Le typage garantit l'assemblage correct des composants entre eux et permet aussi de substituer des implémentations de composants par d'autres. Les contrôleurs permettent à la machine d'exécution du modèle (*runtime*) de contrôler ces instances. La notion de contrat [17] de services complète en partie de la notion de type pour réaliser un courtage des composants.

Notre proposition s'intéresse à revisiter ces notions dans le contexte des langages de script pour permettre le support plus générique de l'adaptation externe et l'adaptation interne lors de l'exécution. L'adaptation externe autorise l'évolution et le changement des contrats requis (réceptacles) et fournis (facettes) tandis que l'adaptation interne autorise l'évolution et le changement du comportement des méthodes ainsi que l'ajout et le retrait de variables d'état sans impact sur la valeur courante des variables existantes.

2.1. Contrôleurs

L'adaptation externe et l'adaptation interne sont pilotées au moyen de quatre nouveaux contrôleurs exposés par le conteneur des composants du modèle cible. Ces contrôleurs peuvent venir en remplacement ou en supplément des contrôleurs natifs des modèles cibles. Ces contrôleurs[1] sont le FacetController, le ReceptacleController, ScriptController et le PropertyController. L'action des contrôleurs est réalisée au niveau de l'instance de composant.

Le contrôleur de facette FacetController permet l'ajout et le retrait des facettes sur une instance du composant. Les propriétés d'une facette sont utilisées quand le modèle supporte le courtage. La méthode setFacet() permet également de faire évoluer une facette existante par l'ajout et le retrait d'interfaces dans la signature d'une facette. Le *handler* de la méthode setFacet() spécifie un objet chargé de traiter de l'appel de la méthode quand la méthode n'est pas implémentée par le script ou lorsqu'une exception est levée par celle-ci. Dans l'exemple donné en annexe, l'objet HttpServletHandler retourne silencieusement pour la méthode Servlet.init() qui n'est pas nécessairement implémentée par le script et retourne une réponse avec un code de statut "501 Not Implemented" si la méthode service() n'est pas présente. Lorsqu'il n'y a pas de *handler*, les exceptions levées par le moteur de script sont propagées.

```
interface FacetController {
        String[] getFacetNames();
        void setFacet (String name, String[] interfaces,
                       Map properties, Handler handler);
        String[] getFacetInterfaces(String name);
        Map getFacetProperties(String name);
        void removeFacet(String name);
}
```

Le contrôleur de réceptacles ReceptacleController permet l'ajout et le retrait de réceptacles à l'instance du composant. A l'instar de la facette, la méthode setReceptacle permet l'ajout et le retrait d'interfaces sur un réceptacle. Le filtre d'un réceptacle est utilisé pour la recherche de facettes quand le modèle supporte le courtage. Le filtre peut être une expression booléenne ou entière sur les propriétés des facettes. L'expression booléenne élimine les facettes ne vérifiant pas l'expression tandis que l'expression entière réalise un tri entre plusieurs facettes candidates. Remarquons que certains modèles de composants (ServiceBinder [18], SCR [6], FROGi [19]) indiquent le caractère obligatoire ou optionnel de la liaison d'un réceptacle vers une facette de composant. Le caractère obligatoire assujettit le cycle de vie du composant qui change d'état (actif ou passif) à la présence ou à l'absence de liaison. Ce caractère peut être spécifié par le biais de contrôleur définissant les transitions en fonction des dépendances personnalisables du composant avec son environnement [20].

```
interface ReceptacleController {
        String[] getReceptacleNames ();
        void setReceptacle (String name, String[] interfaces,
                            String filter, boolean multiplicity);
        String[] getReceptacleInterface (String name);
        String getReceptacleFilter (String name);
        void removeReceptacle (String name);
}
```

Le contrôleur de script ScriptController permet d'évaluer incrémentalement un script programmé dans un langage de script quelconque pourvu que son interpréteur soit disponible dans l'environnement

[1] Par commodité, ces contrôleurs sont exprimés en Java dans l'exemple mais ils peuvent être aussi décrits sous la forme d'interfaces C#, d'interfaces CORBA IDL ou de types de port (*porttypes*) W3C WSDL.

d'exécution. Ces scripts ou ces fragments de script contiennent généralement la définition des fonctions correspondantes aux opérations des interfaces de facettes. Le script peut contenir également des déclarations de variables ce qui permet d'ajouter des variables d'état. La méthode evaluateFc() peut être invoquée successivement plusieurs fois pour compléter ou pour remplacer les définitions des opérations. Ce mode de fonctionnement est assimilé à la boucle interactive de l'interpréteur de script. La méthode reset() supprime la définition des opérations, les variables et leurs états.

```
interface ScriptController {
        void setLanguage(String language) throws UnsupportedLanguageException;
        void evaluateFc(String text) throws SyntaxErrorException
        String getScriptText();
        void reset();
        void reset(String language) throws UnsupportedLanguageException;
}
```

Le contrôleur de variable permet l'ajout et le retrait des propriétés dans l'instance de composant. Ces propriétés sont manipulées sous la forme de variables dans le script. Ce contrôleur consulte et positionne aussi la valeur d'une propriété. Le changement de valeur d'une propriété peut provoquer des erreurs de conversion de type si le langage de structure ne supporte pas le type de la nouvelle valeur ou si le langage de script ne supporte le polymorphisme des variables.

```
interface PropertyController {
        String getPropertyNames();
        Object getProperty(String name);
        void addProperty(String name, Object value)
                throws ImpedanceMismatchException, ClassCastException;
        void removeProperty(String name);
}
```

2.2. Types

Le type d'un composant est généralement associé à sa liste des facettes et sa liste de réceptacles. Dans les composants orientés services, il est même limité à la liste des facettes. Dans notre contexte, les deux contrôleurs FacetController et ReceptableController permettent tous les deux de faire évoluer (ajouter ou retirer) les listes de facettes et de réceptacles d'une instance de composant. Le type d'une instance de composant n'a donc plus un caractère immuable comme le considèrent les implémentations de nombreux modèles de composants. Le type d'un composant est généralement considéré immuable tout au long de sa vie. Une des conséquences est l'introduction possible d'erreurs dans l'architecture lors de l'usage de ces opérations. L'architecture doit être vérifiée dynamiquement après chaque opération.

2.3. Prototypes

Généralement, les modèles de composants s'appuient sur les principes d'instanciation des langages de classes : les instances d'un composant sont créées par une fabrique associée au type et à l'implémentation du composant. Les instances d'un composant ont toutes le même type et le même comportement fonctionnel tout au long de leurs vies. Dans notre contexte de composants évoluant du point de vue du type et du comportement, la fabrique traditionnelle ne peut pas tenir ce rôle. Elle est remplacée par une usine à prototype. Dans notre contexte, un prototype est un patron de composants ayant un type initial (facettes et réceptacles), un comportement initial (i.e. un script exécuté à l'instanciation) et une liste initiale de contrôleurs. Le type et le comportement de chaque instance peuvent mutés indépendamment par la suite par l'action des contrôleurs sur l'instance. Les contrôleurs peuvent être également ajoutés et retirés tout au long de la vie de l'instance du composant. Des questions ouvertes que nous n'adressons pas pour l'instant, sont l'évolutivité du prototype et l'instanciation par clonage d'instances. L'évolutivité du prototype permettrait de faire évoluer simultanément toutes les instances de composant ayant été créées à partir d'un même prototype de composant. L'instanciation par clonage d'instances permettrait de dupliquer l'état interne et le comportement d'une instance déjà créée et éventuellement établir les mêmes liaisons sur ses facettes et ses réceptacles quand la multiplicité le permet.

3. FractScript

Fractal [7] définit un cadriciel de construction d'applications à partir d'un modèle de composants supportant la création de compositions hiérarchiques. Le cadre de conception Fractal fournit une API permettant, entre autres, de définir des types de composants, de fabriquer des instances à partir de ces types, de configurer ces instances puis les connecter entre elles. Un composant Fractal fournit et requiert des interfaces fonctionnelles. Celles-ci sont relatives à la logique applicative et sont des interfaces serveurs (i.e. facettes) et des interfaces clientes (i.e. réceptacle). Le composant fournit également un ensemble d'interfaces de contrôle appelées contrôleurs. La liste des interfaces de contrôle inclut, entre autres, une interface dédiée à la gestion du cycle de vie (LifecycleController), à la liaison des interfaces clientes et serveurs entre instances de composants (BindingController) et au contrôle du contenu (ContentController) dans le cas où le composant est composite. L'interface dédiée à la configuration des attributs (AttributeController) est muni d'accesseurs et/ou de modifieurs dans la signature est setXX/getXX ou XX est le nom de l'attribut. Plusieurs instances d'un composant Fractal peuvent être créées à partir d'une fabrique associée à un type de composant. La spécification Fractal a fait l'objet de plusieurs implémentations alternatives pour Java et d'autres langages de classes. On peut citer parmi celles-ci, Julia, AOKell, Fractal/Proactive, Fractalk, Think, FractNet ...

Un des objectifs premiers de FractScript est de permettre l'utilisation de composants primitifs développés avec des langages de script dans des implémentations Java de Fractal comme Julia et AOKell. FractScript complète la membrane de base de composants scriptables pour les quatres contrôleurs FacetController, le ReceptacleController, ScriptController et le PropertyController décrits dans la section précédente. Ces contrôleurs ne sont pas néanmoins orthogonaux des contrôleurs existants (LifecycleController, BindingController, AttributeController) et interagissent de concert avec eux. Le script de chaque composant contient une variable référençant le contexte FractScript du composant en donnant accès aux 4 contrôleurs. Un exemple commenté d'utilisation de ces contrôleurs est donné en annexe.

L'implémentation actuelle est limitée aux langages de script dont les *service providers* ont être définis pour le canevas de JSR 223 *"Scripting for the Java Platform"*. Ce canevas qui fait désormais parti de Mustang (Java Platform 6.0), offre une interface unifiée aux moteurs de script et l'implémentation de référence fournit les *service providers* pour JavaScript, Groovy (JSR 241) et PHP. Les interpréteurs de script (appelés aussi moteurs) peuvent écrit en Java (Rhino pour JavaScript) ou accessible via une interface JNI (Mozilla JS DLL pour JavaScript, Php.dll pour PHP). La membrane du composant FractScript est actuellement "tisée" manuellement sur la base d'un *proxy* dynamique. Elle implémente plusieurs politiques de synchronisation alternatives entre les contrôleurs. L'usine de prototypes est actuellement implémentée. Chaque instance de composant scripté peut aussi être créée à partir d'un prototype vierge (n'ayant que le contrôleur FacetController). L'implémentation courante gère les contrôleurs comme des facettes par souci de simplicité et d'unification. Nous envisageons d'utiliser AOKell pour tisser automatiquement la membrane afin d'assembler les composants FractScript avec d'autres composants natif AOKell. Chaque instance de composant FractScript possède son propre objet Type qui peut évoluer au gré des manipulation des contrôleurs FacetController et ReceptacleController.

4. SCRScript

La spécification OSGi [3] définit un canevas de déploiement et d'exécution dynamique de services. Initialement destiné aux passerelles domotiques et immotiques, cette spécification est utilisée également pour le développement d'applications à *plugin* telles qu'Eclipse et par les noyaux de plusieurs serveurs applicatifs (Geronimo, JOnAS). La programmation d'applications avec OSGi suit les principes de la programmation orientée service dynamique. La programmation orientée service [21] construit l'application à partir d'entités logicielles faiblement couplées entre elles et éventuellement opérées par des organisations tierces. Ces entités, appelés services, offrent des fonctionnalités décrites contractuellement. Le contrat de services OSGi est syntaxique (interface Java) et qualitatif (propriétés de courtage) sans négociation. Le contrat permet de sélectionner (ou courter) les services pertinents pour l'application. Le courtage et la liaison sont réalisés juste au moment de l'utilisation (*late binding*) et tout service respectant le contrat peut être substitué par un autre. Dans OSGi, l'application (qui peut elle-même fournir des services) référence directement (c.a.d. sans *proxy*) les objets implémentant les services et invoque les méthodes sur les objets sans surcoût. Les services sont considérés comme autonomes car l'organisation

initiatrice de l'application n'a pas de prise sur l'administration de ceux-ci. De plus, le cycle de vie (activité) des services n'est pas nécessairement synchronisé avec celui de l'application. Dans ces conditions, la programmation orientée service dynamique suppose d'une part, que de nouveaux services peuvent apparaître alors que l'exécution de l'application est entamée et d'autre part que d'autres services utilisés peuvent disparaître temporairement ou définitivement avant la terminaison de l'application.
La prise en charge par le développeur du dynamisme de services est une tâche fastidieuse qui est trop fréquemment la cause d'erreurs empêchant un fonctionnement non-stop de la plateforme OSGi et des applications qui se partagent celle-ci[2]. Conscient de ce fait, le groupe d'experts (CPEG) chargé de la rédaction de la version 4 de la spécification a introduit un modèle de composants orientés services appelé *Service Component Runtime* (SCR). Ce modèle à composants qui s'inspire très largement de ServiceBinder [18], un modèle à composants qui adresse la gestion dynamique des liaisons vers les services requis (i.e réceptacles) et la gestion du cycle de vie en fonction de la présence ou l'absence des services requis obligatoires. Dans ce modèle, le service fourni et les services requis sont immuables. L'implémentation du composant est fournie par une classe Java. Ce modèle de composant fait actuellement l'objet du JSR 291 et donne le point de départ à d'autres modèles de composants dynamiques comme iPOJO [22], ECR (Enterprise Component Runtime) [23]...
SCRScript est une implémentation du Service Component Runtime qui supporte à la fois le développement des composants orientés services dynamique dans le langage Java (définis dans la spécification) ou dans des langages de script. SCRScript crée une instance de composant à partir de la description XML correspondant à un prototype de composant. L'instance enregistre un service auprès du plateforme OSGi. Ce service comporte à la fois les interfaces fonctionnelles fournies par le composant et les interfaces de quatre contrôleurs. Par la suite, ces interfaces peuvent être ajoutées ou retirées du service par manipulation de contrôleur FacetController. Les contrôleurs sont accessibles par le composant (code fonctionnel) au travers de son contexte. La membrane du composant est réalisée au moyen d'un proxy dynamique Java. Elle partage une partie de son code avec l'implémentation de FractScript. Les modifications du service requis et des contrôleurs sont notifiées aux autres composants de l'application via l'annuaire de services de la plateforme. L'ajout d'une interface oblige à désenregistrer le *proxy* courant pour obliger les usagers du service à relâcher sa référence, régénérer un nouveau *proxy* dynamique comportant les interfaces supplémentaires puis ré-enregistrer le service. Le retrait d'interface ne nécessite pas de re-géneration du *proxy*. La modification de la propriété objectClass du service associé au filtrage des appels de méthode est suffisante. Comme FractScript, l'implémentation actuelle de SCRScript utilise le canevas du JSR 223. La prochaine implémentation supportera la construction de plusieurs instances par l'usine de prototype. Les trois politiques d'instanciation envisagées sont celle conforme à la *pseudo-fabrique* de services org.osgi.framework.ServiceFactory, celle conforme au SCR et enfin une politique originale introduisant la notion de session similaire à celle des composants J2EE et CCM.

5. Travaux connexes

L'idée d'utiliser les langages de script dans l'approche à composant n'est pas nouvelle. Nierstrasz et al [24] proposent un modèle de composants dont la construction des liaisons (appelé "*glue*") entre les ports d'entrée et de sortie des composants se réalisent par scripting. Les scripts étaient eux même des composants qui ce fait sont hierarchiques. Cependant, les auteurs ne semblent pas traiter l'instanciation dynamique des composants.
FScript [25] est un langage de script dédié à la spécification et à la reconfiguration consistante d'assemblage de composants Fractal. SAFRAN (*Self-Adaptive FRActal compoNents*) l'utilise comme langage support pour réaliser l'aspect auto-adaptatif de composants Fractal. Les scripts peuvent être modifiés dynamiquement depuis une console. Cependant, FScript reste un langage dédié à l'adaptation de composants.
L'implémentation Apache Tuscany [26] du modèle SCA (*Service Component Architecture*) [27] a annoncé supporter l'écriture de composants en JavaScript. Cependant, ce modèle du SCA reste statique du point de vue du typage des composants (services fournis et services requis) et dans son comportement.
Enfin, la notion de prototype que nous reprenons pour l'instanciation des composants a été introduite

[2] Ceci est actuellement la principale cause d'erreurs lors de la mise à jour dynamique des *plugins* Eclipse.

par les langages à objets dits "prototypiques" [28,29,30] comme Self [31], JavaScript et d'autres qui sont pour beaucoup des langages dynamiques.
L'idée de l'adaptation interne des instances d'un langage n'est pas récente non plus. Nous ne développerons pas les travaux relatifs qui pour la plupart, requièrent une modification de la machine virtuelle. Une liste des travaux est décrite dans [32].

6. Conclusion et Perspectives

Dans cet article, nous nous sommes intéressés à regarder l'impact de l'usage de langages de script sur les concepts de contrôleurs, de type et d'instanciation attachés aux modèles de composants. Nous avons revisités deux modèles de composants reconnus par rapport aux extensions proposées : celui de Fractal défini par le consortium ObjectWeb et celui du SCR spécifié par l'OSGi Alliance. Ceci a donné lieu à deux implémentations FractScript [33] et SCRScript [34].
Nous pensons que l'usage de langages de script peut se généraliser dans la réalisation de gros logiciels maintenables quand ils sont mis en œuvre au travers de modèles de composants. Ce constat part du fait qu'il y a actuellement un fort engouement pour des langages comme Java et PHP dans l'industrie du Web et que les concepteurs des plateformes Java 7.0 et .NET 3.0 ouvrent maintenant largement la porte à ces langages en allant jusqu'à introduire de nouvelles instructions dans leurs machines virtuelles respectives pour permettre l'exécution performante des programmes scriptés [35]. Plus généralement, le domaine de recherche sur les langages dynamiques est en résurgence [36] comme en temoignent les très récentes manifestations sur ce thème [37,38].
Une perspective à court de ce travail sera d'évaluer l'utilisabilité de composants scriptés dans le contexte applicatif : celui des gestionnaires autonomiques et celui des architectures de médiation dans des domaines tels que le *Machine-to-Machine* (M2M). Dans ces deux domaines, les composants ciblés sont principalement orientés vers le filtrage d'événements et la reconnaissance de motif dans des historiques d'événement. Des langages comme PERL et Python pourraient être des très bons candidats pour la mise au point de tels composants.
Une autre perspective à ce travail est l'usage de composants scriptés pour le support d'architectures dynamiques. L'idée est de pourvoir décrire les différents changements possibles et les opérations à mener aux moyens de langages dynamiques. L'intérêt sera de pouvoir modifier les règles régissant l'architecture courante d'une application sans être obliger de redémarrer celle-ci.

7. Bibliographie

1. Szyperski, C., "Component software : beyond object-oriented programming," ACM Press/Addison-Wesley Publishing Co., 1998.
2. Nierstrasz O., Gibbs S.J., Tsichritzis D., "Component-Oriented Software Development", 160-165, CACM, Volume 35, Number 9, September 1992
3. Common Component Architecture (CCA) Forum, http ://www.cca-forum.org/
4. OMG, "CORBA Component Model Specification, Version 4.0", http ://www.omg.org/docs/formal/06-04-01.pdf
5. Sun Microsystems, "Enterprise JavaBeans Specification Version 2.0",August 2001, http ://java.sun.com/ products/ejb/docs.html
6. OSGi Alliance, "OSGi Service Platform Specification (4d Release)", Octobre 2005, http ://www.osgi.org
7. Bruneton E., Coupaye T., Leclercq M., Quema V., Stefani J.-B.. "An open component model and its support in Java". 7th International Symposium on Component-Based Software Engineering (CBSE-7), LNCS 3054, pp 7-22, May 2004.
8. Apache Software Foundation, "The Avalon Framework", http ://jakarta.apache.org/avalon
9. Ketfi M., Belkhatir N., Cunin P.Y., "Automatic Adaptation of Component-based Software : Issues and Experiences", PDPTA'02, Juin 2002, Las Vegas, Nevada, USA
10. Buckley J., Mens T., Zenger M., Rashid A., Kniesel G., "Towards a Taxonomy of Software Change", Journal of Software Maintenance and Evolution : Research and Practice, John Wiley & Sons, 2003.
11. Dey A., "Providing Architectural Support for Building Context-Aware Applications"'. PhD thesis, College of Computing, Georgia Institute of Technology, 2000.

12. Kephart J.O., Chess D.M., "The Vision of Autonomic Computing", IEEE Computer, Janvier 2003.
13. Barron D.W., "The World of Scripting Languages", John Wiley & Sons, 2000, ISBN 0-471-99886-9.
14. Ousterhout, J.K., "Scripting : higher level programming for the 21st Century", Computer, Mar 1998, Volume : 31, Issue : 3, pp 23-30
15. Prechelt L., "An Empirical Comparison of Seven Programming Languages", IEEE Computer 33 (10) : 23-29 (2000)
16. Savikko V., "Generative and Incremental Approach to Scripting Support Implementation", Software Engineering Research and Practice, 2003, pp105-111
17. Beugnard A., Jézéquel J.M., Plouzeau N., Watkins D., "Making components contract aware", IEEE Computer, 32(7), 1999, pp 38-45.
18. Cervantes H., Hall R., "A Framework for Constructing Adaptive Component-based Applications : Concepts and Experiences", 7th Symposium on Component-Based Software Engineering (CBSE), Mai 2004, Edinburgh, Ecosse.
19. Désertot M., Cervantes H., Donsez D., "FROGi : Fractal components deployment over OSGi", 5th International Symposium on Software Composition (SC 2006) , LNCS 4089, 25-26 March 2006, Vienna, Austria.
20. Offermans M., "Auto Management of Service Dependencies", article en ligne, 2005, http ://www.osgi.org/news _events/documents/AutoManageServiceDependencies_byMOffermans.pdf
21. Bieber, G., Carpenter, J., "Introduction to Service-Oriented Programming", OpenWings whitepaper, Septembre 2001, http ://www.openwings.org/download/specs/ServiceOrientedIntroduction.pdf
22. Escoffier C., "iPOJO", http ://www-adele.imag.fr/users/Clement.Escoffier/dev/ipojo
23. Eclipse, "Enterprise Component Framework", http ://www.eclipse.org/proposals/ecp/
24. Nierstrasz O.M., Tsichritzis D., de Mey V., Stadelmann M., "Objects + Scripts= Applications", Esprit 1991 Conference, Dordrecht, NL, 1991, pp. 534-552
25. David P.C., "Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation", Thèse de Doctorat de l'Université de Nantes, Juillet 2005.
26. Apache Software Foundation, "Tuscany", http ://incubator.apache.org/tuscany
27. BEA Systems, IBM, IONA, Oracle, SAP AG, Siebel Systems, Sybase, "Service Componen Architecture Specifications", http ://www-128.ibm.com/developerworks/library/specification/ws-sca/
28. Mulet Ph., "Réflexion et langages à prototypes", Thèse de Doctorat de l'Université de Nantes, 1995.
29. Lieberman H., "Using prototypical objects to implement shared behavior in object-oriented systems", OOPLSA'86, p.214-223, September 29-October 02, 1986, Portland, Oregon
30. Taivalsaari A., "Classes versus prototypes : Some philosophical and historical observations". Journal of Object-Oriented Programming (JOOP), 10(7) :44-50, 1997.
31. Ungar D., Smith R.B., "Self : The Power of Simplicity", OOPSLA'87 Conference Proceedings, pp. 227-241, Orlando, FL, October, 1987.
32. Bergel A., Ducasse S., "Scoped and Dynamic Aspects with Classboxes", RSTI - L'Objet (programmation par aspects), Volume 11, Number 3, pp. 53-68, 2005
33. FractScript, http ://www-adele.imag.fr/users/Didier.Donsez/dev/fractscript/
34. SCRScript, http ://www-adele.imag.fr/users/Didier.Donsez/dev/osgi/scrscript
35. Hamilton G., Reinhold M., Bracha G., "Evolving the Java Language", TS-7955, JavaOne 2005, San Francisco , Mai 2005.
36. Nierstrasz O., Bergel A., Denker M., Ducasse S., Gaelli M., Wuyts R., "On the Revival of Dynamic Languages", 4th International Symposium on Software Composition (SC05), , LNCS 3628, pp. 1-13, 2005.
37. Workshop on Revival of Dynamic Languages, ECOOP 2006, July 2006, Nantes, France, http ://prog.vub.ac.be/ wdmeuter/RDL06/
38. Dynamic Language Symposium, OOPLSA 2006, October 2006, Portland, Oregon, http ://www.dcl.hpi.uni-potsdam.de/dls2006/openconf.php

8. Annexe : Exemple de composant FractScript évoluant dynamiquement

```
HttpDispatcher hd = ...
...

// le composant est instancié par l'usine à prototype
Component comp=Prototype.newInstance(
    new String[]{
            ScriptController.class.getName(),
            FacetController.class.getName(),
            BindingController.class.getName(),
            LifeCycleController.class.getName(),
        },false
    );
...

// la facette s est ajoutée après l'instanciation
((FacetController)comp).setFacet(
        "s", new String[]{ HttpServlet.class.getName()},
        null, new HttpServletHandler());
...

// le comportement  est ajoutée après l'instanciation
((ScriptController)comp).setLanguage("JavaScript");
((ScriptController)comp).evaluateFc(
        "function service(req,res){"+
        "  var out=res.getOutputStream();"+
        "  out.println(\"<html><body>Hello World</body></html>\");"+
        "}"
        , false);

// la facette s est liée au receptacle d'un autre composant
((BindingController)hd).bindFc(
        "servlet",((Component)comp).getFcInterface("s"));
((LifeCycleController)comp).startFc();
((LifeCycleController)hd).startFc();
...

// la variable counter et la fonction incr() sont ajoutées
((LifeCycleController)comp).stopFc();
((ScriptController)comp).evaluateFc(
        "var counter=0;"+
        "function incr(){ return ++counter; }"
        , false);

// la fonction service(req,res) de la facette s est remplacée
((ScriptController)comp).evaluateFc(
        "function service(req,res){"+
        "  var name=req.getParameter(\"name\");"+
        "  var out=res.getOutputStream();"+
        "  out.println(\"<html><body>Hello \"+name+\""+
        "              \" (\"+incr()+\")</body></html>\");"+
        "}"
        , false);
((LifeCycleController)comp).startFc();
...
```