Connexion Non-Anticipée de Composants en ScL : Une Voie pour l'Evolution des Logiciels

Luc Fabresse, Christophe Dony, Marianne Huchard

Connexion non-anticipée de composants en SCL : une voie pour l'évolution des logiciels

Luc Fabresse — Christophe Dony — Marianne Huchard

LIRMM – Université Montpellier II
161, rue Ada
F-34392 Montpellier Cedex 5

{fabresse,dony,huchard}@lirmm.fr

RÉSUMÉ. *La connexion non-anticipée de composants logiciels est un des points clés en génie logiciel car elle permet une meilleure réutilisation du code existant. Le travail présenté dans cet article se fonde sur l'idée qu'un tel mécanisme de connexion est aussi utile pour l'évolution des logiciels. C'est ainsi que nous présentons* SCL*, un langage à composants simple et dynamiquement typé intégrant un mécanisme de connexion de composants basé sur les ports et les propriétés des composants. Nous illustrons à travers des exemples comment ce mécanisme peut être utilisé pour faire évoluer une architecture logicielle.*

ABSTRACT. *Unanticipated connection of independently developed black-box components is a promising track in software engineering. The work introduced in this paper is based on the idea that the unanticipated connexion mechanism is also useful in software evolution. In this paper, we present* SCL*, a dynamically typed component-oriented language that provides an unanticipated black-box component connection mechanism that relies on ports and properties of components. Examples illustrate how this mechanism can be used to support the software architecture evolution.*

MOTS-CLÉS : *langage à composants, connexion non-anticipée, propriété, évolution d'architectures logicielles*

KEYWORDS: *component-oriented language, unanticipated connection, property, software architecture evolution*

1re soumission à *Atelier évolution*, le 17 février 2006.

1. Introduction

Le développement par assemblage de composants logiciels [HEI 01] est l'une des voies les plus prometteuses dans les recherches en génie logiciel. Ce mode de développement s'appuie sur la notion de composant logiciel : *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties"* [SZY 02]. Les composants sont des briques logicielles permettant de mieux réutiliser et faciliter la construction d'applications. De nombreux modèles et langages [GRO 02, MON 99, SEC 00, MCD 01, ALD 02, BRU 04, MAR 05] ont été imaginés et proposent de nouveaux concepts (port, interface, service, connecteur, composite) et mécanismes afin de mettre en œuvre ce nouveau paradigme.

Le travail présenté dans cet article considère les composants comme des boîtes noires développées indépendamment les unes des autres et se fonde sur l'idée qu'un mécanisme de connexion non-anticipée de tels composants permet l'évolution des logiciels. De notre point de vue, l'architecture d'un logiciel est constituée de composants et de connexions et faire évoluer un logiciel consiste à modifier son architecture c'est-à-dire à ajouter, retirer ou remplacer des composants et des connexions. Du fait de la capacité des composants à expliciter leurs dépendances externes, l'impact d'une évolution sur une architecture à base de composants est mieux contrôlable que dans le paradigme objet.

Dans cet article, nous présentons SCL (Simple Component Language) un langage à composants simple dynamiquement typé qui explore les notions de propriétés et de séparation stricte des codes métier et de connexion, favorisant ainsi la facilité d'écriture et la connexion non-anticipée de composants. Le mécanisme de connexion que nous proposons est unifié en ce qu'il permet de réaliser des connexions de composants du type requis/fourni comme dans la plupart des modèles actuels mais aussi des connexions basées sur les notifications de changement d'état des propriétés des composants [FAB 04].

La première partie présente de façon générale le langage SCL. Les deuxième et troisième parties présentent respectivement les connecteurs et les propriétés en SCL à travers l'exemple d'un client *chat*. La partie quatre montre comment le mécanisme de connexion de composants en SCL basé sur les connecteurs et les propriétés peut être utilisé pour faire évoluer une architecture logicielle. Finalement, nous concluons et présentons une suite possible à ce travail.

2. Survol des éléments de base de SCL

SCL est un langage à composants simple et dynamiquement typé qui repose sur un modèle de composants constitué des entités *composant*, *service*, *port*, *propriété* et *connecteur* et qui offre les principaux mécanismes d'*invocation de service* et de *connexion*.

Un composant est constitué d'un état interne et de services. Les services représentent le comportement des composants qui à l'instar des méthodes d'un objet, possèdent un nom, des paramètres et, éventuellement, une valeur de retour. Les services invocables sur un composant sont dit entrants (ingoing). Les services invoqués par un composant mais qu'il ne définit pas sont dits sortants (outgoing). Les services requis par un composant sont généralement des services sortants qui doivent être pourvus lors de sa connexion pour qu'il soit opérationnel.

Les ports sont les points d'interaction d'un composant et constituent le support des mécanismes de connexion et d'invocation de service. Un port entrant d'un composant permet à d'autres composants d'invoquer un ensemble restreint et fixé des services qu'il définit. Un port sortant d'un composant lui permet d'invoquer des services externes qui seront pourvus lors de sa connexion. Une invocation de service est syntaxiquement similaire à un envoi de message dans un langage à objets. Le receveur d'une invocation de service est un port et le sélecteur un nom de service. Si le port receveur est un port entrant, c'est le service portant le nom du sélecteur défini sur le composant auquel appartient ce port qui est exécuté. Si le port receveur est un port sortant, le service exécuté dépend des connexions réalisées. La figure 1 montre un extrait du code SCL de description d'un composant CHATCLIENT en utilisant une syntaxe Smalltalk puisque le prototype actuel de SCL est implémenté en Squeak [ING 97].

```
SCLComponentBuilder create: #ChatClient
    properties: 'chatText pseudo'
    outPorts: 'Networking'
    inPorts: 'Chatting'.

ChatClient>>init
    (self port: #Chatting)
        addServiceSelector: #join: ;
        addServiceSelector: #leave ;
        addServiceSelector: #sendMessage:.

ChatClient>>join: serverAdrs
    "implementation details ... "
    (self port: #Networking) connectTo: serverAdrs.
    "..."
```

Figure 1. *Déclaration en* SCL *d'un composant* CHATCLIENT

Le composant CHATCLIENT est constitué du port `Chatting` permettant l'invocation des services `join:`, `leave` et `sendMessage:` et du port `Networking` sur lequel est invoqué le service sortant et requis `connectTo:`. Le service `connectTo:` est requis car indispensable au fonctionnement du composant CHATCLIENT et sera effectivement connu lors de la connexion de ce composant.

3. Connecteurs en SCL

Le mécanisme de connexion de composants permet de satisfaire les services requis par un composant en utilisant les services fournis par d'autres composants. C'est généralement l'architecte logiciel qui choisit des composants existants, les adapte et réalise les connexions de composants afin de réaliser une application. La connexion de composants boîtes noires pose des problèmes d'adaptation qui sont bien souvent résolus par la création d'un adapteur [GAM 95] commme dans ComponentJ [SEC 00], CCM [GRO 02] ou Fractal [BRU 04]. En SCL, ce sont les connecteurs, entités représentant les connexions entre les composants qui permettent de traiter ces problèmes d'adaptation comme en ArchJava [ALD 03] ou Sofa [PLA 05]. Un connecteur permet, en SCL, d'établir des connexions entre des composants via leurs ports. Dans une connexion, les ports sortants sont appelés ports sources (ports à partir desquels les invocations de services arrivent) et les ports cibles sont des ports entrants permettant d'invoquer des services. La figure 2 montre un exemple de connexion volontairement simplifié puisqu'aucune adaptation n'est nécessaire.

```
chat := ChatClient new.
netManager := NetworkManager new.

SclBinaryTransmitterConnector new
    source: ( chat port: #Networking )
    target: ( netManager port: #Networking );
    connect.
```

Figure 2. *Connexion de deux composants en* SCL

Dans cet exemple, la connexion est réalisée par un connecteur spécial SCLBINARYTRANSMITTERCONNECTOR qui transmet toutes les invocations de services provenant du port source vers le port cible et retourne le résultat. Cette connexion permet de rendre le composant CHATCLIENT opérationnel puisque tous ses services requis sont fournis par le composant NETWORKMANAGER.

4. Propriétés en SCL

Le concept de propriété est une caractéristique novatrice de SCL qui permet de réaliser des connexions de composants basées sur des notifications de leur changement d'état. Le concept de propriété est issu du modèle Java Bean [HAM 97] et représente l'état externe et accessible d'un composant. Nous avons montré l'intérêt des propriétés des composants [FAB 04] et plus particulièrement mis en évidence des méta-propriétés de ces propriétés comme *nac* (notify after change) ou *nbc* (notify before change). Les services de notification d'une propriété sont des services sortants mais non requis par le composant et sont regroupés au sein d'un port nommé port de notification de la propriété. Le programmeur d'un composant déclare une propriété très simplement comme illustré sur la figure 1. Un exemple de connexion utilisant les

propriétés est illustré sur la figure 3 qui montre l'architecture logicielle de notre client de *chat*.

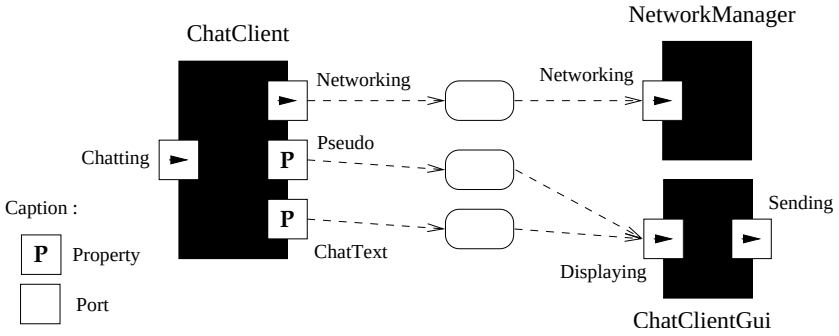

Figure 3. *Architecture logicielle d'un client* chat *en* SCL

```
client := ChatClient new.
clientGui := ChatClientGui new.

SclBinaryNACConnector new
    source: ( client notifyPortOf: #ChatText )
    target: ( clientGui port: #Displaying )
    glue: [ :source :target :message |
        target displayText: ( message arguments at: 2 ).
        ] ;
    connect.
```

Figure 4. *Connexion de deux composants utilisant des notifications de propriétés*

Le composant CHATCLIENTGUI doit rafraîchir la fenêtre de *chat* à chaque fois qu'un nouveau message est reçu par le composant CHATCLIENT c'est-à-dire que sa propriété `ChatText` est modifiée. Cette fonctionnalité est réalisée par la connexion décrite sur la figure 4 par l'utilisation d'un connecteur SCLBINARYNACCONNECTOR qui ne réceptionne que les notifications nac (notify after change) de son port source qui est ici le port de notification de la propriété `ChatText`.

5. Évolution d'architectures logicielles en SCL

L'évolution d'un logiciel se traduit par une modification de son architecture c'est-à-dire l'ajout ou le retrait de composants ou de connexions. Avec ce mécanisme de connexion, des fonctionnalités peuvent facilement être ajoutées à notre application. Par exemple, dans l'architecture représentée sur la figure 3, le port `Sending` du composant CHATCLIENTGUI doit être connecté afin que l'utilisateur puisse envoyer des messages. Ce port peut être connecté au port `Chatting` du composant CHATCLIENT directement ou par l'intermédiaire d'un composant LOGGER qui sauvegarderait les

messages avant de les transmettre au composant CHATCLIENT. Un mécanisme de connexion est donc approprié pour l'évolution des logiciels et celui de SCL permet de résoudre les problèmes d'adaptation tels que *names mismatch* (par exemple le composant CHATCLIENTGUI requiert un service `connectToServer`: alors que le composant CHATCLIENT fournit le service `join`:) dans le connecteur et sans nécessité de créer un adapteur comme dans la plupart des modèles existants.

La notion de propriété vient compléter notre mécanisme de connexion de composants et apporte de nouvelles possibilités en terme d'évolution des architectures. Par exemple, ajouter la fonctionnalité d'envoi automatique du titre de la chanson actuellement lue par un lecteur de musique à travers notre client *chat* revient à ajouter une connexion entre le port de notification de la propriété `currentTitle` du composant MUSIPLAYER et le port `Chatting` du composant CHATCLIENT comme illustré par la figure 5. L'ajout de fonctionnalités à notre application ne modifie donc pas les composants existants et a un impact localisé sur l'architecture existante.

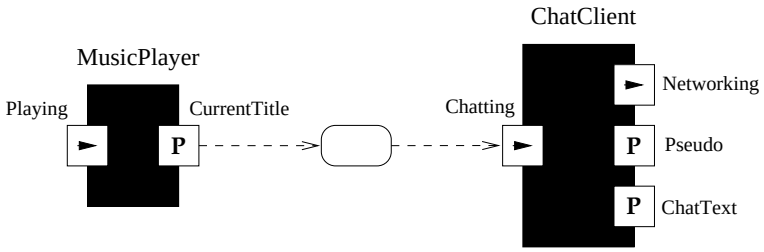

Figure 5. *Évolution de l'application client* chat *grâce aux propriétés*

6. Conclusion et perspectives

Dans cet article, nous avons présenté SCL, un prototype de langage à composants simple et dynamiquement typé. Ce langage intègre un mécanisme de connexion non-anticipée de composants indépendamment développés. Ce mécanisme repose sur les concepts de services, ports, propriétés et connecteurs. En plus des connexions classiques du type requis/fourni, ce mécanisme permet de réaliser des connexions de composants basées sur leurs changements d'états externe grâce aux propriétés. A travers quelques exemples simplifiés, nous avons montré que ce mécanisme de connexion est très bien adapté pour faire évoluer des architectures logicielles. De plus, il semble que l'impact d'une évolution pourra mieux être maîtrisé et contrôlé dans ce contexte.

Ce travail s'inscrit dans une démarche plus globale qui vise à définir un langage à composants minimal qui n'intègrerait que les mécanismes fondamentaux et novateurs du développement par composants afin de mieux les comprendre et les identifier. La connexion non-anticipée de composants est l'un de ces mécanismes qui nous semble prometteur notamment pour l'évolution des architectures logicielles. La poursuite de ce travail nécessite d'étudier des exemples plus complets afin de mettre en place une méthodologie pour l'évolution des architectures logicielles.

7. Bibliographie

[ALD 02] ALDRICH J., CHAMBERS C., NOTKIN D., « ArchJava : connecting software architecture to implementation. », *ICSE*, ACM, 2002, p. 187-197.

[ALD 03] ALDRICH J., SAZAWAL V., CHAMBERS C., NOTKIN D., « Language Support for Connector Abstractions. », CARDELLI L., Ed., *ECOOP*, vol. 2743 de *Lecture Notes in Computer Science*, Springer, 2003, p. 74-102.

[BRU 04] BRUNETON E., COUPAYE T., LECLERCQ M., QUÉMA V., STEFANI J.-B., « An Open Component Model and Its Support in Java. », CRNKOVIC I., STAFFORD J. A., SCHMIDT H. W., WALLNAU K. C., Eds., *CBSE*, vol. 3054 de *Lecture Notes in Computer Science*, Springer, 2004, p. 7-22.

[FAB 04] FABRESSE L., DONY C., HUCHARD M., PICHON O., « Vers des composants logiciels interfaçables », *Actes de ALCAA (Agents Logiciels, Coopération, Apprentissage, Activité humaine)*, Montpellier, 2004, p. 33-48.

[GAM 95] GAMMA E., HELM R., JOHNSON R., VISSIDES J., *Design Pattern : Elements of Reusable Object Oriented Software*, Addison-Wesley, 1995.

[GRO 02] OBJECT MANAGEMENT GROUP, « Manual of Corba Component Model V3.0 », 2002, *http ://www.omg.org/technology/documents/formal/components.htm*.

[HAM 97] HAMILTON G., « JavaBeans », Api specification, juillet 1997, Sun Microsystems, Version 1.01.

[HEI 01] HEINEMAN G. T., COUNCILL W. T., Eds., *Component-based software engineering : putting the pieces together*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[ING 97] INGALLS D., KAEHLER T., MALONEY J., WALLACE S., KAY A., « Back to the future : the story of Squeak, a practical Smalltalk written in itself », *OOPSLA '97 : Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York, NY, USA, 1997, ACM Press, p. 318–326.

[MAR 05] MARVIE R., « Picolo : A Simple Python Framework for Introducing Component Principles », *Euro Python Conference 2005*, Göteborg, Sweden, june 2005.

[MCD 01] MCDIRMID S., FLATT M., HSIEH W. C., « Jiazzi : New-Age Components for Old-Fashioned Java. », *OOPSLA*, 2001, p. 211-222.

[MON 99] MONSON-HAEFEL R., *Enterprise JavaBeans*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1999.

[PLA 05] PLASIL F., « Enhancing component specification by behavior description : the SOFA experience », *WISICT '05 : Proceedings of the 4th international symposium on Information and communication technologies*, Trinity College Dublin, 2005, p. 185–190.

[SEC 00] SECO J. C., CAIRES L., « A Basic Model of Typed Components », *Lecture Notes in Computer Science*, vol. 1850, 2000, p. 108–129.

[SZY 02] SZYPERSKI C., *Component Software : Beyond Object-Oriented Programming (2nd Edition)*, Addison-Wesley, 2002.