Intégrer l'Architecture d'une Application avec son Implémentation grâce à ArchJava

Michel NOLARD

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique
Rue Grandgagnage, 21ă
5000 - Namur (Belgium)
Tél. 32 81 72 49 66
Fax. 32 81 72 49 67
michel.nolard@outmax.org

Résumé Les ADL[1] (*Architecture Description Language*) sont trop peu intégrés avec les langages d'implémentation des applications elles-mêmes. ArchJava est une solution à ce problème. Il permet de décrire une architecture d'application de façon simple, fiable, transparente et intégrée à Java. Otez-vous donc de la tête l'idée qu'il vous faudra apprendre un langage de plus pour y parvenir ! ArchJava garantit l'intégrité des contraintes architecturales de l'application dés la compilation : la structuration en composants, les modèles de connexion inter-connexion, la gestion des accès aux données partagées.

1 Introduction

Lorsque le programmeur développe une application distribuée, il se retrouve confronté à des problèmes d'expressivité du langage de programmation utilisé. Le code source de l'application[2] ne lui permet pas d'exprimer de façon lisible et facilement maintenable les contraintes architecturales.

En effet, si le programmeur veut exprimer des contraintes telles que la découpe en composants, les modalités de communication entre ceux-ci, ou les contraintes d'intégrité sur les accès aux données partagées entre eux, il est très vite amené à faire un choix *entre* exprimer clairement ces contraintes dans un autre langage que le langage du code source de l'application *et* complexifier le cycle de développement de l'application[3] en ne les exprimant que par une forme procédurale éparpillée dans le code source, le rendant illisible et peu maintenable.

Pour exprimer ces contraintes sans polluer le code source de l'application, il peut aussi utiliser un langage spécialisé de type ADL. Cependant, cela nécessite la maîtrise d'un autre langage en plus du langage utilisé pour l'implémentation

[1] Langage spécialisé permettant d'exprimer les contraintes architecturales d'une application : les inter-relations entre composants, leurs modes de communications, les contraintes sur cette architecture... e.g Rapide

[2] Le code qui est effectivement compilé pour former le code exécutable.

[3] Ici, j'envisage le cycle de développement réduit : implémentation et maintenance.

de l'application proprement dite. De plus, cela complexifie la maintenance de l'application car, en cas de modification, il faut aller apporter des modifications dans plusieurs fichiers pour une même opération.

Chacune de ces deux solutions possède ses inconvénients propres, mais au final, le programmeur se retrouve quand même avec une lourdeur dans le développement et la maintenance du code source qui n'est ni acceptable pour une application distribuée de taille sérieuse, ni même pour une application plus simple.

Ce qu'il serait plus agréable d'avoir, c'est un moyen élégant d'intégrer l'expression des contraintes directement dans le code source sans avoir tous les problèmes pré-cités. ArchJava est une amélioration apportée au langage Java qui propose, par le biais de l'ajout d'un ensemble de mots-clés, une solution à tous ces problèmes dans le cadre du développement d'une application distribuée programmée en Java. ArchJava est aussi un compilateur pour le langage ArchJava permettant de générer du bytecode à destination de la JVM (*Java Virtual Machine*).

Alors que certains ADL mettent un pied dans l'implémentation, ArchJava saute à pieds joints dans le design de l'architecture, tout en permettant de traiter l'architecture de façon dynamique : modification temps-réel des connexions entre les composants, ... L'apport d'ArchJava se fait sur trois aspects inextricablement liés du design architectural de l'application :

1. la structuration en composants, avec ArchFJ[4] (Sect. 2)
2. la communication entre les composants, grâce à l'abstraction des connecteurs de composants [5] (Sect. 3)
3. l'accès aux données partagées entre les composants, grâce aux annotations [3] (Sect. 4)

Je me propose de vous présenter, tout au long de cet article, comment ArchJava permet de gérer ces différents aspects ainsi que son intégration avec Java.

2 La Structuration en Composants

L'approche suivie par ArchJava consiste à intégrer l'expression des contraitnes d'architecture dans Java lui-même. L'architecture peut ainsi être mise à jour au fur et à mesure de l'évolution du code source. Il en est ainsi notamment pour le principal aspect de la création d'une architecture : la structuration en composants.

ArchJava permet de définir le type d'objet architectural *composant* (**component**). Un *composant* peut être simple ou être lui-même un agrégat de un ou plusieurs autres *composants*, que l'on appelle *sous-composants*. Un composant est un objet qui communique avec d'autres composants par le biais de *ports*. Les *ports* sont les canaux de communication privilégiés entre composants : aucun

[4] *Architectural Formal Java* [1]. C'est le langage au coeur du système ArchJava, la représentation interne, sur laquelle toutes les preuves formelles ont été établies.

message ne peut passer entre deux composants s'il n'est issu d'un port et destiné à autre port. Un composant doit donc posséder des ports pour être inséré dans une architecture, à moins qu'il ne soit le composant principal, celui qui représente l'entièreté de l'application à lui tout seul. Celui-ci est un composant, puisque composé de composants, mais il ne possède aucun ports.

La description d'une architecture s'exprime en énonçant les communications entre les composants, et donc par les connexions entre ports. Les ports sont des objets internes aux composants, fournissant des *types de messages* entrants et sortants. Ces *types de messages* sont matériaisés par les méthodes des ports. Ce sont des méthodes Java conventionnelles qui permettent donc de passer des objets Java ordinaires.

Les ports peuvent être constitués indifféremment de types de messages entrants et/ou sortants. Un port ne contient aucune donnée membre, il n'est donc constitué *que* de méthodes. Il ne saurait en être autrement, puisqu'il n'est qu'un point d'ouverture d'un composant ; le composant, lui, possède des données membres lui permettant de mener ses tâches à bien, comme pour tout objet.

Passons donc à un exemple pour mieux saisir ces nouveaux concepts.

2.1 Exemple : le Compilateur Simplifié

Prenons l'exemple d'un compilateur simplifié (Fig. 1), et observons comment il est possible de le structurer en composants à l'aide d'ArchJava. Le `Compiler` est le composant principal de l'application. Il résulte de l'assemblage de plusieurs composants suivant le modèle du pipe-line. Tout d'abord, le `Scanner` découpe le flux de caractères entrant en lexèmes. Le flux de mots qu'il produit constitue le flux d'entrée du `Parser` qui reconstitue l'arbre syntaxique correspondant. Enfin, le `CodeGen` traite cette structure et fournit du code compilé, ce code étant le résultat du traitement, par le `Compiler`, du flux de caractères entrant.

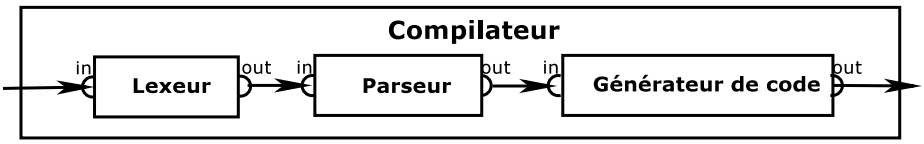

Fig. 1. Architecture du compilateur simplifié.

Nous avons donc bien à faire à un pipe-line où l'entrée d'un composant est la sortie du précédent. Chaque composant n'étant connecté qu'à un seul autre en entrée (nommé `in`) et à un seul autre en sortie (nommé `out`). Nous allons rester simple pour ce premier exemple, et considérer que tous ces composants tourneront sur la même machine. La communications entre eux se fera donc par simple appel de méthodes Java.

2.2 Le Code Source de l'Exemple

Voici comment il est possible de traduire cette architecture en ArchJava. Evidemment, pour des raisons de clareté de l'exposé, je n'ai pas fourni l'entièreté du code, puisque la création d'un compilateur n'en est pas l'objet. Remarquez l'intégration parfaite de ArchJava dans la structure du langage Java. Les commentaires sont situés à la suite.

```
public component class Compiler {
        private final Scanner scanner = new Scanner();
        private final Parser parser = new Parser();
        private final CodeGen codegen = new CodeGen();

        connect scanner.out, parser.in;
        connect parser.out, codegen.in;

        ...
}

public component class Scanner {
        public port in
        { ... }

        public port out {
                provides Token nextToken();
        }

        ...
}

public component class Parser {
        public port in {
                requires Token nextToken();
        }

        public port out {
                provides AST parse();
        }

        AST parse() {
                Token tok = in.nextToken();
                return parseExpr(tok);
        }

        AST parseExpr(Token tok)
        { ... }

        ...
}
```

```
public component class CodeGen {
    public port in {
        requires AST parse();
    }

    public port out{
        provides AsmCode generateCode();
    }

    AsmCode generateCode() {
        ...
    }

    ...
}
```

2.3 Analyse du Code Source de l'Exemple

L. 1 : Le mot-clé **component** indique que nous créons une classe de type architectural *composant*. `Compiler` est le composant principal de l'application : le *compilateur*.

L. 2–4 : `Compiler` est constitué de plusieurs composants dont il suffit d'instancier les classes.

L. 6–7 : L'architecture *pipe-line* est définie : la sortie du `Scanner` est connectée à l'entrée du `Parser` et la sortie du `Parser` est connectée à l'entrée du `CodeGen`.

L. 12–21 : Le composant `Scanner` est défini. Le composant `Parser`, qui est plus détaillé sert de référence pour les commentaires.

L. 23–41 : Définition du composant `Parser`.

L. 24 et 28 : Le mot-clé **port** indique que la définition suivante est celle d'un *port de connexion* entre composants.

L. 25 et 29 : Un *port de connexion* peut fournir et/ou nécessiter certains types de messages. Comme nous travaillons avec Java, les types de messages entre composants ne sont autres que des méthodes Java ordinaires. Les mots-clés ArchJava **provides** et **requires** sont respectivement utilisés afin de différencier la fourniture et la nécessité d'un type de message.

L. 32–40 : Les méthodes de la classe `Parser` sont implémentées de façon habituelle. Les méthodes appartenant à des ports sont définies et implémentées comme les autres méthodes[5].

[5] La méta-classe Java `Method` peut donc être utilisée pour les gérer et les rechercher dynamiquement.

L. 33 : Le port d'entrée `in` est accessible comme toute classe interne (`inner class`) Java conventionnelle. Les méthodes sont les siennes, pas celles du composant agrégé.

L. 43–57 Le composant `CodeGen` est implémenté de la même façon que les autres composants.

3 Les Communications Entre Composants

Dans l'exemple de la section précédente (voir P. 3), nous avons vu que nous désirions que les composants communiquent suivant le *pattern* du pipe-line. Or, une architecture qui n'est pas fiable n'étant pas utilisée, nous devons assurer au programmeur que le compilateur ArchJava ne lui permettra pas d'outrepasser ses droits en accédant sauvagement à des objets auxquels, architecturalement parlant, il ne doit pas avoir accès. Nous désirons donc une garantie stipulant que les composants ne fonctionneront qu'exclusivement suivant le modèle du *pipe-line*.

ArchJava garantit que les communications entre composants respecteront les connexions établies à l'aide de la déclaration **connect** et qu'aucune autre connexion ne sera acceptée. La vérification de cette propriété est réalisée à la compilation, évitant l'apparition impromptue de messages d'erreur lors de l'exécution. En effet, à quoi bon détecter un défaut d'implémentation si l'application est déjà déployée !

L'architecture permet les appels entre composants connectés et entre un parent et ses sous-composants *immédiats*. Elle interdit les appels externes à un sous-composant et les appels entre composants non-connectés. Elle interdit aussi les appels violant la structure hiérarchique de l'architecture (e.g `Compiler1.Scanner` appelant `Compiler2.Parser`).

Grâce à ce système, le programmeur peut se concentrer tranquillement sur les communications locales et risque moins d'oublier certains cas de figures issus d'un trop grand nombre de possibilités à gérer. C'est une application de la loi de Demeter[6]. Elle consiste en un principe de programmation orienté objet énoncé par le groupe de recherche Demeter [7] qui implique des bénéfices non négligeables pour le programmeur : réduction de l'espace des problèmes, diminution de la

[6] Une foule de liens associés à l'analyse des implications de cette loi dans le cadre de la programmation orientée objet est disponible sur [6] :

> The Law of Demeter was originally formulated as a style rule for designing object-oriented systems. "Only talk to your immediate friends" is the motto. The style rule was discovered at Northeastern University in the fall of 1987 by Ian Holland.
>
> A more general formulation of the Law of Demeter is : Each unit should have only limited knowledge about other units : only units "closely" related to the current unit. Or : Each unit should only talk to its friends ; Don't talk to strangers.

[7] Demeter est un outil CASE. Voir http ://www.ccs.neu.edu/research/demeter/

complexité de chaque unité de code, gain d'efficacité, réduction du temps de déboggage, accroissement de la maintenabilité de l'application, ...

3.1 Classes de Connexion

Jusqu'ici, nous n'avons fait que voir comment ArchJava permettait de renforcer l'intégrité des communications par spécification du flux de contrôle. Cependant, rien n'a encore été proposé jusqu'ici pour pouvoir les matérialiser de façon propre et respectueuse de ces contraintes.

En effet, lorsque le programmeur désire découpler l'implémentation des composants de la façon dont ils communiquent, pour plus de clareté dans le programmes, il cherche en fait à transformer les connexions entre les composants en objets à part entière, des objets dont le seul but serait de gérer la communication en tant que telle. Pas facile, me direz-vous ? Et bien non, très simple en réalité, pour peu que l'on utilise l'*abstraction des connecteurs* que fournit ArchJava.

Pour ce faire, ArchJava fournit au développeur la classe `Connector` dont il pourra dériver tous les connecteurs nécessaires à son application. Un tel connecteur permet de gérer la sérialisation des données pour permettre de transformer le simple appel de méthode Java en une véritable requête Corba ou RMI, de la crypter par RSA, de la transférer à plusieurs composants à la fois (mode *broadcast*), ... Ce principe permet de rendre le composant indépendant de la localisation de son interlocuteur, ainsi que de la façon de le contacter.

Evidemment, dans le cas où le développeur définit son propre connecteur, il ne faut plus que ArchJava réalise une connexion de méthodes comme il le fait par défaut lors de l'usage du mot-clé **connect**. Pour cela, il suffit de préciser que cette connexion correspond à un modèle (en anglais, *pattern*) à l'aide du mot-clé **pattern**, qui indique à ArchJava que le développeur lui fournit une classe qui permettra ce type de connexion, cette classe héritant de `Connector`.

Reprenons l'exemple de notre compilateur simplifié. Imaginons que nous désirions obtenir un fichier objet à partir de notre code source. Nous devons alors définir un composant `Assembler` qui reprenne le code généré par le compilateur et l'assemble. Pour corser la chose, nous voudrions que le composant `CodeGen` se trouve sur une autre machine que celle où se trouve `Assembler`. Pour qu'ils continuent à communiquer, nous voudrions qu'ils utilisent un socket TCP/IP.

La figure 2 nous montre notre projet de façon plus claire. Voici le code associé à cet exemple. Les commentaires sont situés à la suite.

```
public component class CodeGen {
        public port in {
                requires AST parse();
        }

        public port interface client {
                provides client (CodeGen generator, InetAddress address);
                provides AsmCode generateCode() throws IOException;
        }
```

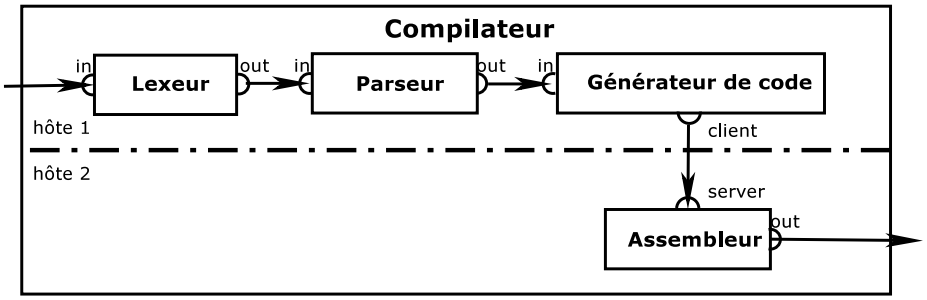

Fig. 2. Vue d'ensemble de notre projet – Un assembleur et un générateur de code sur des hôtes différents, communicants par TCP/IP.

```
10
        ...
    }

    public component class Assembler {
15      public port interface server {
            requires AsmCode generateCode() throws IOException;
        }

        public port out {
20          provides ObjectFile assemble();
        }
    }

    public component class Compiler {
25      private final Scanner scanner = new Scanner();
        private final Parser parser = new Parser();
        private final CodeGen codegen = new CodeGen();
        private final Assembler asm = new Assembler();

30      connect pattern Scanner.out, Parser.in;
        connect pattern Parser.out, CodeGen.in;
        connect pattern CodeGen.client, Assembler.server;

        public Compiler() {
35          TCPConnector.registerObject(asm, ASSEMBLER_PORT, "server");

            connect (scanner.out, parser.in);
            connect (parser.out, codegen.in);
            connect (codegen.client, asm.server);
40      }

        connect pattern CodeGen.client, Assembler.server with TCPConnector{
            client (CodeGen generator, InetAddress address) {
```

```
                connect (generator.client, asm.server) with
45                  new TCPConnector (address, ASSEMBLER_PORT,
                        "server");
            }
        };
}
```

Pour des raisons de clareté de l'exposé, je ne fournis pas le code source[8] de TCPConnector. Celui-ci, héritant de Connector, transforme les appels aux méthodes en données sérialisées qu'il transmet ensuite aux composants destinataires. Il fait pour cela grand usage des méta-classes Java `Method` et `Parameter`.

L. 6–8 Pour commencer notre amélioration, nous modifions la définition du composant `CodeGen`, en lui ajoutant une *interface de port client* (**port interface**). Le fait que ce soit une *interface de port* plutôt qu'un simple *port* provient de la nature *non unique* de l'instanciation du port client. En effet, on peut très bien imaginer qu'un *client* se connecte à plusieurs *server*. L'utilité de cette clause n'est pas transcendantale dans cet exemple, elle n'est là qu'à titre indicatif, afin de révéler la puissance d'ArchJava. Cependant, dans une application où plusieurs `Assembler` seraient disponibles, il faudrait instancier ce port pour chaque connexion avec un `Assembler.server`, d'où l'utilisation du **interface**.

L. 7 Le port client possède un constructeur dont le seul but est de gérer l'initialisation de la connexion TCP/IP (enregistrement de l'adresse, ...). Le type de message `generateCode( )` du port `client` de `CodeGen` se voit adjoindre la clause **throws** `IOException`, puisque des erreurs de connexion peuvent désormais survenir. La méthode elle-même n'est cependant pas modifiée, car c'est `TCPConnector` qui génère cette exception, faisant passer la méthode en question pour responsable. Cela permet de gérer de façon propre la capture des exceptions, sans avoir à mettre en place un gestionnaire d'erreur dans `TCPConnector` lui-même. C'est l'une des particularités des `Connector`, due à leur faculté d'encapsulation et d'abstraction complète des connexions inter-composants. Ce sont là les seules modifications apportées au composant `CodeGen`.

L. 15–17 L'interface `client` du composant `CodeGen` est la correspondante de l'interface `server` du composant `Assembler`.

L. 20 Le composant `Assembler` a pour *simple* tâche de fournir la méthode `assemble( )` par le biais de son port `out`. Il n'a pas de raison particulière d'être au courant du type d'interconnexion effectivement réalisée entre lui et `CodeGen` ou tout autre composant.

Le composant principal `Compiler` a été modifié un peu plus, mais seulement dans le but de modifier les connexions inter-composants, rien de plus.

L. 28 Le nouveau composant membre `asm` est déclaré de type `Assembler`.

[8] Pour les petits curieux, [?] propose un code commenté de TCPConnector appliqué à un système de stockage de poèmes.

L. 30–32 Viennent ensuite les déclarations des connexions inter-composants. On remarque de suite la présence d'un nouveau mot-clé **pattern**. On peut voir aussi que les modèles de connexion sont établis entre des classes plutôt qu'entre des instances. C'est la différence entre un modèle de connexion (**connect parttern**) et une connexion simple (**connect**). Ce concept de `modèle` de connexion est celui sous-jacent à l'abstraction des connexions dont ArchJava est capable. Les lignes en questions avertissent le compilateur ArchJava que la connexion entre des composants de ce type se fera suivant une règle définie par l'utilisateur (*user-defined*) par le biais d'une classe dérivée de `Connector`.

L. 34–40 Puisque nous avons annoncé cette possibilité de connexion entre des composants, nous devons maintenant dire à ArchJava comment les mettre en place et entre quels composants celles-ci vont être effectivement instaurées.

L. 35 Ainsi, dans le constructeur de `Compiler`, le port `server` du composant `asm` est enregistré dans la liste des ports de composants pouvant recevoir des requêtes par TCP/IP. La classe de connecteurs `TCPConnector` gère cette liste afin de prévenir toute tentative de connexion frauduleuse ou simplement erronée.

L. 37–39 Ces lignes indiquent qu'il faut qu'ArchJava procède à des connexions dynamiques (par opposition aux connexions statiques qui sont définies à l'extérieur de toute méthodes et qui ne demandent pas de parenthèses).

L. 37–38 Une connexion *conventionnelle* est établie entre les composants `Scanner` et `Parser`, ainsi qu'entre `Parser` et `CodeGen`, comme auparavant. Cette fois, cependant, la connexion est clairement posée entre des *instances* de composants.

L. 39 Cette ligne établit une connexion entre les composants `CodeGen` et `Assembler`. Cette connexion établie dynamiquement n'est pas conventionnelle : il faut donc spécifier ses modalités d'établissement.

L. 41–48 La définition du *pattern de connexion* dicte à ArchJava les opérations à réaliser pour pouvoir connecter un port `client` d'un composant `CogeGen` avec un port `server` d'un composant `Assembler`.

L. 43–46 Il lui faut pour cela lancer l'initialisation de la connexion TCP/IP dans le cadre de la création d'une nouvelle interface de port client. Ces lignes stipulent très clairement à ArchJava que la connexion s'effectue *avec* (**with**) l'aide d'un `Connector` : un TCPConnector.

Les modifications apportées à notre programme ont été rapides modulo développement de `TCPConnector` mais, surtout, elles étaient simples[9] et auraient pu être réalisées indépendamment[10].

Pour prouver les avantages de ArchJava, et vérifier sa fiabilité, ses auteurs lui ont fait passer le baptême du feu en l'utilisant pour concevoir deux applications complexes :

[9] `Simple` n'a jamais impliqué *facile*.

[10] Possibilité de parallélisme entre plusieurs développeurs

Aphyds[2] Un outil de conception de circuits électroniques complexes

PlantCare[5] Un projet mené au Intel Research Seattle d'une application capable de s'autoconfigurer[11] et de réagir de façon robuste aux pannes[12] et aux changements environnementaux. Ce système a pour but de prendre soin de plantes dans un environnement d'habitation ou de bureau.

Dans les deux cas, ArchJava a permis de faire coller l'implémentation à la vision architecturale des développeurs, tout en leur facilitant les modifications et la maintenance du code source.

4 La Régulation des Accès aux Données Partagées

L'intérêt de spécifier complètement les modes d'accès aux données partagées par plusieurs composants vient d'un double but : garantir l'intégrité des communications afin de réduire la complexité des modifications apportées à l'application.

Un problème substiste, empêchant de garantir l'intégrité des communications inter-composants : il est possible que deux composants s'appellent sans que cela ne soit autorisé, en utilisant des fonctions de première classe (*first-class functions*) par exemple. Une *fonction de première classe* n'est rien d'autre qu'un object fonctionnel au sens opératoire du terme : la fonction *est* un objet. Les `Action` Java sont un bon exemple de fonctions de première classe.

Il peut arriver aussi que la référence à un composant soit passée par le biais des connexions inter-composants, permettant ainsi à des composants non-autorisés d'appeler les méthodes du composant référencé. Pour mieux comprendre la problématique, illustrons la dans le cadre de notre compilateur simplifié (Fig. 3) : si on interdit à l'objet `Action` de posséder la moindre référence vers un **component**, il n'y a aucun moyen pour que `CodeGen` puisse appeler `Scanner` de façon détournée au travers de `Action`.

Le système de typage d'ArchJava interdit à toute classe n'étant ni le composant parent d'un composant quelconque, ni un composant qui lui est connecté légalement, de posséder une référence vers cet objet. Ce système permet même de partager l'objet `Action` tout en vérifiant qu'aucun flux de contrôle ne puisse se faire au travers de cet objet.

Pour complèter cette contrainte, il faut préciser que les composants ne sont pas des objets transmissibles par le biais des méthodes appartenant aux ports. On ne les retrouvera donc ni en tant que type de retour, ni en tant que type d'arguments.

---

[11] [7] nous en dit plus sur les moyens dont dispose le programmeur ArchJava pour gérer cet aspect du développement.

[12] ArchJava dispose de capacités d'assistance à la gestion des pannes [4]

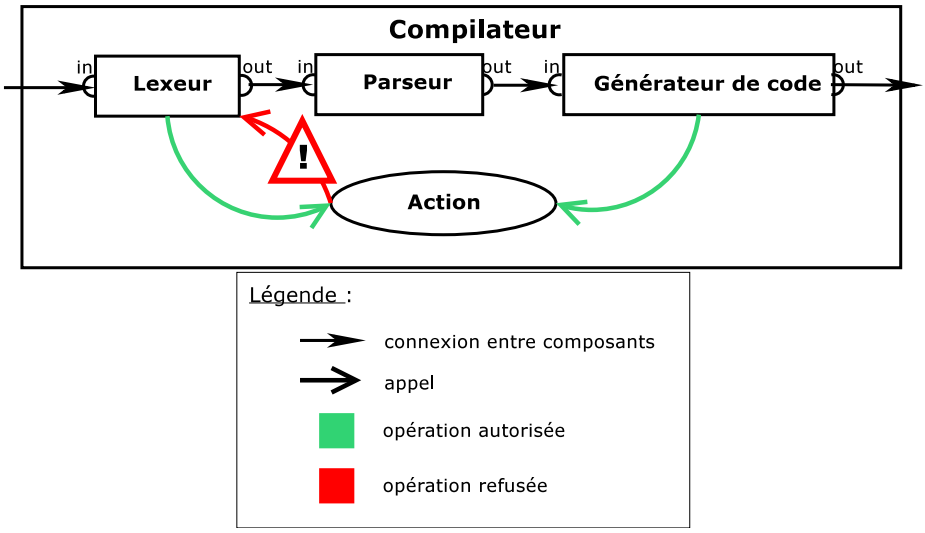

Fig. 3. Partage d'une fonction de première classe entre plusieurs composants non connectés.

4.1 Les Annotations en ArchJava

ArchJava supporte un concept nommé *annotations*[3] dont le but est de permettre de spécifier précisémment l'*unicité* et le type de *possession* pour toutes les instances d'objets.

Les annotations permettent de palier des problèmes de sécurité dans les applications. L'un d'entre eux est tellement sournois qu'il s'est présenté sous la forme d'une faille de sécurité dans le JDK 1.1.1[13]. Le problème est le suivant : un objet peut se retrouver à renvoyer à un appelant une référence vers ses données internes plutôt qu'une copie de celles-ci. Cela permet à l'objet appelant de les modifier comme bon lui semble (i.e sans contrôle aucun de la part de l'objet possesseur).

Si un objet se retrouve partagé entre plusieurs composants, il faut spécifier l'unicité de celui-ci (i.e des *références* vers celui-ci) ainsi que les contraintes de possession de cet objet. L'unicité est indiquée à l'aide du mot-clé **unique**. La possession est représentée par **owned**.

Prenons un petit exemple pour y voir plus clair :

```
public class MyClass {
    private owned Object[] signers;

    public Object[] getSigners() {
        return signers;
```

[13] JDK=*Java Development Kit*, Kit de développement pour Java, distribué par Sun Microsystems sur son site web : http ://www.java.sun.com.

```
    }
}
```

L. 1–2 Dans l'exemple ci-dessus, on a à faire à une classe `MyClass` dont chaque instance renferme une variable membre privée nommée `signers`.

L. 2 Logiquement aucune référence à `signers` ne doit jamais être renvoyée. Cette information le programmeur la donne au compilateur en utilisant le mot-clé **owned**.

L. 5 Il peut donc programmer tranquillement ensuite puisque, lorsqu'il commet un impair, le compilateur rugit avec une `erreur à la compilation`. La résolution des références (*references aliasing*) à un objet **owned** ne se fait qu'à l'intérieur de l'objet propriétaire.

Pour corriger notre code, nous pouvons utiliser une méthode réalisant la copie d'un tableau : `Object[] arraycopy(Object[])`. On désire indiquer, pour clore cette faille de sécurité, que l'objet qui recevra cette copie sera ne pourra s'en servir qu'une fois, la référence devant être détruite après usage. Une sorte de *"Ce message s'autodétruira dans 5 secondes"* pour Java. On dit que dans ce cas la référence est *non persistente*. Pour matérialiser cette contrainte, l'annotation **unique** est utilisée. Notre classe devient donc :

```
public class MyClass {
    private owned Object[] signers;

    public unique Object[] getSigners() {
5       return arraycopy(signers);
    }
}
```

Fig. 4. La ligne 2 annotée avec **owned** et la ligne 4 avec **unique**.

Nous avons donc un contrôle complet de l'accès à cette variable, aussi bien au sein des méthodes de `MyClass` qu'à l'extérieur, dans les classes appelantes.

5 Conclusion

ArchJava permet de structurer proprement et de façon fiable une application écrite en Java. En outre, il reste entièrement compatible avec les librairies standard Java. Le code que le développeur écrit correspond directement à sa vision de l'architecture de l'application, et tout composant de l'architecture donne lieu à l'implémentation d'un composant.

De plus, ArchJava est facile à apprendre pour quelqu'un connaissant déjà Java, puisqu'il est fondé sur celui-ci et qu'il ne lui ajoute que quelques mots-clés très simples.

Certaines limitations sont encore légèrement gênantes et devront être levées pour une plus grande acceptation de cet outil par le public. Par exemple, l'implémentation actuelle d'ArchJava ne permet pas qu'un **component** implémente d'une interface.

Depuis la structuration en composants jusqu'au traitement des accès aux données partagées entre ceux-ci, en passant par les types de connexions inter-composants, ArchJava permet de spécifier fiablement son application distribuée en facilitant sa maintenance et son évolution. Toutes les propriétés de cet outil ayant été démontrées formellement, tout programmeur peut se permettre de l'utiliser pour un réaliser des applications fiables en Java.

Références

1. John Aldrich, Craig Chambers, and David Notkin. « Architectural Reasoning in ArchJava ». *ECOOP 2002*, June 2002.
2. John Aldrich, Craig Chambers, and David Notkin. « ArchJava : Connecting Software Architecture to Implementation ». *ICSE 2002*, May 2002.
3. John Aldrich, Valentin Kostadinov, and Craig Chambers. « Alias Annotations for Program Understanding ». *OOPSLA 2002*, 2002.
4. John Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. « Architecture-Centric Programming for Adaptive Systems ». *Workshop on Self-Healing Systems.*
5. John Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. « Language Support for Connector Abstractions ». *the European Conference on Object-Oriented Programming (ECOOP '03)*, July 2003.
6. The Demeter Group. « Law of Demeter (General Formulation) ». http ://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/general-formulation.html.
7. Vibha Sazawal and John Aldrich. « Architecture-Centric Programming for Context-Aware Configuration ». *Workshop on Engineering Context-Aware Object-Oriented Systems and Environments.*