Fractal, Kilim, JAC : une expérience comparative

Frédéric Loiret, Lionel Seinturier, Eric Gressier-Soudan

Fractal, Kilim, JAC : une expérience comparative

Frédéric Loiret[1,2] **– Lionel Seinturier**[1,3] **– Eric Gressier-Soudan**[2]

[1] *Université Paris 6, Lab. LIP6, 4 place Jussieu, 75252 Paris cedex 05*
[2] *CNAM, Lab. CEDRIC, 292 rue Saint-Martin, 75141 Paris cedex 03*
[3] *INRIA Futurs, Project Jacquard, USTL-LIFL, Bât. M3, 59655 Villeneuve d'Ascq*

floiret@noos.fr, Lionel.Seinturier@lip6.fr, gressier@cnam.fr

RÉSUMÉ. *Cet article compare trois techniques pour la séparation des caractéristiques fonctionnelles et non-fonctionnelles des applications : Fractal, Kilim, et JAC. Les trois sont des frameworks du consortium ObjectWeb. Les deux premières sont dédiées à l'assemblage et la configuration de composants Java, tandis que JAC est un framework de programmation orientée aspect en Java. A travers l'étude d'un service distribué de partage de données selon l'algorithme de cohérence de K. Li & P. Hudak, nous montrons les similitudes et les différences entre les trois frameworks. Au delà de cet exemple, nous tentons de généraliser et proposons différents critères pour comparer les trois approches.*

ABSTRACT. *This paper compares three techniques for separating business and technical concerns in an application: Fractal, Kilim, and JAC. There are all three frameworks of the ObjectWeb consortium. The first two are dedicated to the assembly and configuration of Java components, whereas JAC is a Java framework for aspect-oriented programming. Through the case study of a distributed shared data management service based on the K. Li & P. Hudak's consistency protocol, we show the similarities and the differences between the three frameworks. We generalize this example, and we propose several criteria to compare the three approaches.*

MOTS-CLÉS : *Composants, programmation par aspects, séparation des préoccupations.*

KEYWORDS: *Components, aspect-oriented programming, separation of concerns.*

1. Introduction

La réutilisabilité et la composabilité ont toujours été des objectifs importants de l'ingénierie du logiciel. En leur temps, les approches procédurale et orientée objet ont chacune apporté leurs solutions. Actuellement, la tendance pour les applications complexes telles que les systèmes et les applications réparties, promeut un développement à base de composants. Au delà des différences de définitions entre les modèles, l'objectif reste le même : augmenter la réutilisabilité et permettre la création de nouvelles entités en assemblant des entités existantes. Dans le cadre de la répartition, l'adaptation des applications à des contextes d'exécution, des systèmes, et des réseaux variés est un enjeu important. L'identification des parties dépendantes de celles indépendantes de l'environnement doit permettre d'atteindre cet objectif. Il s'agit donc de séparer le code métier (i.e. la fonctionnalité qu'il remplit) du composant, du code technique (lié à son environnement d'exécution). Ce principe fait l'objet d'un domaine de recherche à part entière : la séparation des préoccupations [PAR 72][DIJ 76] (en anglais *separation of concerns*).

Cet article s'intéresse à trois *frameworks* actuels visant à atteindre cet objectif pour l'ingénierie des applications et des systèmes répartis : Fractal [BRU 02b], Kilim [HOR 02] et JAC [PAW 01]. Les trois sont des *frameworks* Java du consortium ObjectWeb[1]. Fractal s'intéresse à l'assemblage de composants, Kilim à leur configuration, et JAC à la programmation orientée aspect. Cet article compare ces trois approches au travers d'un service distribué de partage de données selon le protocole de gestion de cohérence de K. Li & P. Hudak [LI 89]. L'application a été implantée successivement avec ces trois *frameworks*, ce qui nous permet de proposer des critères de comparaisons. Bien que de taille limitée, nous pensons que cette application est suffisamment représentative des services *middleware* : elle comprend plusieurs sous-services qui interagissent fortement entre eux. Elle constitue donc, à nos yeux, une bonne base pour étudier la façon dont les trois frameworks mettent en œuvre la séparation des préoccupations.

Les trois *frameworks* comparés (Fractal, Kilim et JAC) ont été retenus pour leur côté novateur. Nous avons volontairement laissé de côté des approches certes plus matures, telles que les modèles EJB [Sun98], CCM [OMG] ou .Net [WIG 03]. Les solutions qu'elles apportent pour la séparation des préoccupations sont à notre avis trop rigides : les services techniques sont codés en dur dans la plate-forme ce qui limite l'étendue des préoccupations séparables. Bien que récemment le serveur d'applications EJB JBoss [jBo] a introduit avec son *framework* d'interception (JBoss-AOP) la possibilité d'insérer des appels à des services techniques quelconques, leur domaine d'utilisation nous semble trop restreint aux applications de gestion tournées vers l'Internet (même si des extensions de leurs middlewares sous-jacents existent pour d'autres domaines, comme par exemple RT-CORBA pour le temps réel). Nous nous intéressons aux outils généralistes pour la construction de systèmes et d'applications répartis.

1. `www.objectweb.org`

L'article est organisé comme suit. La section 2 présente brièvement les trois *frameworks*. L'application servant d'étude de cas et son analyse en terme de fonctionnalités font l'objet de la section 3. La section 4 aborde alors sa mise en œuvre avec les trois approches. La comparaison, partie centrale de l'article, est traitée en section 5. Finalement, et de façon traditionnelle, la section 6 conclut cet article et présente nos perspectives de poursuite.

2. Présentation des approches

Les trois *frameworks* retenus dans cette étude permettent d'assembler, configurer et composer des entités logicielles développées de façon indépendante. Ils constistuent en cela des solutions pour la prise en charge de la séparation des préoccupations. Cette section présente les trois *frameworks* de manière indépendante.

2.1. *Fractal*

Fractal [BRU 02b] est un modèle de composants développé par FT R&D et l'INRIA. Contrairement à d'autres modèles comme les EJB ou CCM dont les composants sont plutôt de grain moyen et destinés aux applications de gestion tournée vers l'Internet, la granularité des composants Fractal est quelconque. Leurs caractéristiques font qu'ils s'adaptent aussi bien à des composants de bas niveaux (par exemple un *pool* d'objets) que de haut niveau (par exemple une IHM complète). Le but de Fractal est de développer et de gérer des systèmes complexes comme les systèmes distribués. Fractal est composé de deux modèles : un modèle abstrait et son implantation.

Le modèle abstrait est indépendant de tout langage de programmation. Il définit un modèle de composants récursifs dans lequel les composants peuvent être assemblés pour former des composants de plus gros grain. Ces derniers ne sont en rien différents des premiers et peuvent à leur tour être assemblés. Par analogie avec la biologie, un composant Fractal est une cellule avec un plasme entouré par une membrane. Le plasme peut contenir d'autres cellules. Une membrane contrôle et gère son plasme. En théorie, un composant peut appartenir à deux plasmes différents. Les conséquences et la prise en charge concrète d'une telle situation restent néanmoins à étudier. Chaque membrane définit un contexte de nommage et possède des interfaces internes et externes. Les cellules interagissent à l'aide de signaux échangés par les interfaces.

Plusieurs niveaux de détails sont pris en compte dans Fractal et amènent des composants aux fonctionnalités de plus en plus riches. Au niveau le plus bas, un composant est une entité d'exécution (i.e. un objet Java). Ce niveau est explicité afin de faciliter l'intégration de l'existant. Au niveau suivant, les composants fournissent des fonctionnalités d'introspection, permettant une découverte de leurs interfaces. Le niveau suivant est un niveau de configuration qui permet de contrôler et de modifier le contenu (i.e. les sous-composants) d'un composant et leurs liaisons. Plusieurs types de contrôleurs sont disponibles : de contenu, de liaison, d'attributs, intercepteurs. Les

niveaux suivants correspondent aux fonctionnalités d'instanciation et de typage des composants. Fractal fournit un langage de description d'architecture (ADL) dont la syntaxe XML permet de décrire des assemblages de composants. Julia [BRU 02a] est l'implantation de Fractal en Java de FT R&D.

2.2. *Kilim*

Kilim [HOR 02] est un projet porté par la société Kelua. Il s'agit d'un framework de configuration d'assemblages de classes Java.

Kilim en est à sa deuxième version. Une séparation claire est faite entre la vue composant et la vue objet avec des règles de projection explicites entre elles. Trois points importants caractérisent la vue composant : le modèle de composant qui est similaire à celui de Fractal, et deux niveaux de description. Le premier, statique et textuel, basé sur XML correspond à une description de l'assemblage de composants appelée *template*. Le second concerne la description du système au niveau exécutif.

Les trois concepts de base de Kilim sont ceux de *port*, de *provider* et de *transformer*. Un *port* est une variable utilisée pour récupérer la référence de l'interface d'un objet. Un *provider* est une abstraction d'un mécanisme de niveau langage permettant d'obtenir des références d'objets (par exemple, constructeur ou méthode de type *getter*). De façon symétrique, un *transformer* permet de modifier l'état d'un objet (par exemple, méthode de type *setter*).

Finalement, la notion de *template* permet, à l'aide d'un ADL XML, de fournir la méta-information nécessaire à la description, l'instanciation, la composition et la configuration des composants.

2.3. *JAC*

Issue de recherches menées depuis 1997 par G. Kiczales et son équipe du Xerox PARC, la programmation orientée aspect [KIC 97] (AOP pour Aspect-Oriented Programming) trouve son origine dans les travaux sur la réflexivité, les protocoles à méta-objets [KIC 91] et les systèmes ouverts. Il s'agit d'appliquer le principe « diviser pour régner » de [PAR 72] et [DIJ 76] aux langages de programmation. Le compilateur AspectJ [KIC 01] est l'aboutissement de ces travaux et l'un des environnements de programmation orientée aspect les plus utilisés.

Java Aspect Components (JAC) [PAW 01] est un *framework* de programmation orientée aspect développé depuis 1998 auquel ont participé la société AOPSYS et les laboratoires LIP6, CNAM-CEDRIC et LIFL. Contrairement à AspectJ qui étend le langage Java et fournit un compilateur, JAC est un environnement dynamique dans lequel les aspects sont des entités présentes au moment de l'exécution et qui sont écrites en Java pur. Cette caractéristique rend JAC particulièrement adapté à une programmation en environnement ouvert et réparti : les caractéristiques non-fonctionnelles

des applications, développées sous forme d'aspects, peuvent être ajoutées ou retirées dynamiquement afin d'adapter l'application à un contexte particulier ou de l'étendre avec de nouveaux services techniques.

La programmation orientée aspect part du principe que l'entrelacement de code métier et de code technique au sein d'une application, nuit à son développement, son évolutivité et sa maintenance. Plus la partie technique est complexe (voir par exemple les applications CORBA), plus le problème se pose de façon cruciale. De plus, le code technique lié à une même préoccupation (par exemple, la sécurité, la persistance ou les traces) est souvent dispersé à plusieurs endroits de l'application. L'AOP propose une approche pour séparer ces deux types de code et rassembler dans une même entité logicielle (aspect) le code d'une préoccupation. Un aspect avec JAC (ou avec AspectJ) définit pour cela une coupe et du code d'aspect. Une coupe désigne l'ensemble des localisations du code métier (on parle de points de jonction) sur lesquels le code de l'aspect doit être ajouté (on parle de tissage effectué par un tisseur d'aspects, *aspect weaver* en anglais). JAC utilise des expressions régulières pour définir les coupes et des encapsuleurs (*wrappers* en anglais) pour le code des aspects. Les encapsuleurs fournissent du code à exécuter avant et/ou après les points de jonction. Lorsque plusieurs encapsuleurs sont présents sur un même point de jonction, on parle de chaîne *wrappante*.

D'un point de vue technique, JAC se présente comme un conteneur d'aspects et de composants métiers. Les deux sont chargés à la demande en fonction des besoins des applications (contrairement aux serveurs d'applications comme les EJB où les services techniques sont codés en dur dans le conteneur). JAC est distribué avec un ensemble d'aspects couvrant les besoins usuels de la programmation répartie (communications distantes CORBA et RMI, persistance, transactions, définition d'IHM Swing et HTML, authentification, gestion de sessions, ...), un atelier de génie logiciel permettant de concevoir avec une notation UML une application et ses aspects, et un support d'exécution.

3. Etude de cas : service de gestion de données réparties partagées cohérentes

3.1. *Choix de l'algorithme*

Historiquement cette étude sur les approches à composants pour les services systèmes répond à des besoins d'ingénierie [THI 97] qui ont émergés lors du prototypage de protocoles de cohérence au-dessus du micro-noyau ChorusOS avec différents outils système : *mapper* puis *proxy-mapper*. L'objectif principal de ce projet sur ChorusOS était de fournir un service de gestion de données cohérentes générique capable d'être utilisé pour tester différentes formes de cohérences. Le résultat a été obtenu en plusieurs itérations [GRE 02]. Les différents développeurs ont utilisé toutes les possibilités de la programmation orientée objet (C++). Malgré tous ces efforts, le service obtenu n'était pas encore assez modulaire et adaptable, nécessitant des retouches significatives à chaque changement de cohérence ou de primitives système. Il nous a

semblé intéressant d'explorer, avec Fractal, Kilim et JAC, de nouvelles façons de réaliser un tel service. Bien que simple, ce service est représentatif des problèmes de systèmes répartis : il comprend plusieurs services en interaction forte les uns avec les autres. A plus long terme, nous souhaiterions utiliser ce type de service implanté selon une approche composants pour gérer des données d'un jeu multi-joueurs et celles d'applications de calculs distribués.

L'algorithme de cohérence est celui défini par K. Li & P. Hudak [LI 89]. Son principe est simple : il fonctionne suivant un protocole à invalidation sur écriture. Le dernier écrivain est le propriétaire de la copie de référence, il délivre des copies à chaque lecteur. Dans le système, il n'y a qu'une seule copie en écriture, celle du propriétaire, ou *n* copies en lecture. L'invalidation consiste à envoyer un message à tous les lecteurs en cours quand un nouvel écrivain récupère la copie de référence du précédent propriétaire. Le prédicat « 1 écrivain ou exclusivement n lecteurs » est assuré par un jeu de verrous d'exclusion mutuelle dans les tables d'accès aux données entre tous les acteurs effectuant un accès simultané à la même page. Il existe plusieurs variantes de l'algorithme. La version avec gestionnaire centralisé repose sur un annuaire des propriétaires qui permet de localiser directement le détenteur de la copie de référence d'une donnée. Celui-ci joue le rôle de séquenceur et ordonne totalement toutes les demandes d'accès. D'autres versions plus réparties jouent, soit sur la localisation du propriétaire en utilisant une heuristique de localisation par propriétaire probable, soit sur une gestion répartie de la liste des lecteurs (*copyset distribué*) en cours qui implique une gestion distribuée des invalidations. L'algorithme offre une cohérence séquentielle, voire atomique en fonction de l'implantation choisie. Un ordre total sur les accès est toujours mis en œuvre à travers la localisation et la gestion de verrous sur les différents sites impliqués.

Après un premier travail sur la version centralisée optimisée, les résultats présentés ci-dessous sont relatifs à la version répartie, souvent baptisée « avec propriétaire probable et *copyset* distribué ».

3.2. *Les fonctionnalités séparées*

À partir des spécifications de l'algorithme, nous avons réalisé une analyse dont le but est de faire apparaître les différents blocs fonctionnels intervenant dans le service de données réparties partagées cohérentes. Comme toute analyse, elle est subjective : d'autres solutions pourraient être proposées. Néanmoins, cette analyse ne constitue pas l'objectif de ce travail qui reste la comparaison de Fractal, Kilim et JAC. Partant de là, la même analyse a été mise en œuvre dans chacune des trois approches.

À partir des spécifications de l'algorithme, nous avons identifié et isolé sept fonctionnalités. Cette analyse résulte de plusieurs itérations sur l'implantation du service. Notamment, la mise en œuvre de la version centralisée a fait apparaître le découpage actuel qui a été consolidé lors du passage à la version distribuée.

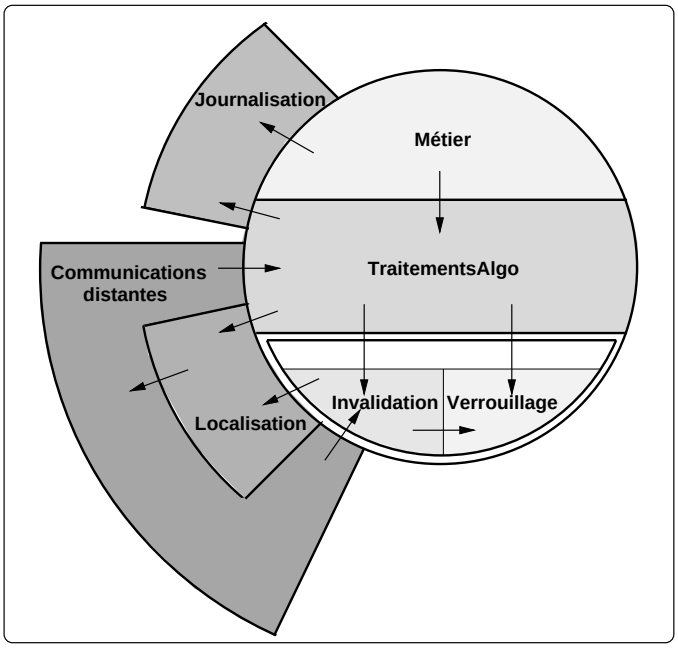

Figure 1. *Schématisation des fonctionnalités séparées.*

Nous avons considéré les deux opérations de lecture et d'écriture (1) correspondent à la logique fonctionnelle (`métier`) de notre application. Le choix nous semble justifié par le fait que l'objectif premier du service reste d'accéder en lecture et en écriture à des données partagées et réparties.

Les autres fonctionnalités sont alors considérées comme relevant du plan non-fonctionnel : au sein de l'algorithme, la cohérence des données est assurée par la mise en œuvre du mécanisme d'`invalidation` (2) et par la gestion du `verrouillage` (3) des accès aux données. Le service de `localisation` (4) de la copie de référence est nécessaire lors des défauts. Rappelons qu'à chaque défaut, il y a toujours une localisation de copie, mais lors d'une lecture, on cherche la première copie disponible. Certains traitements algorithmiques nécessaires à la `mise en œuvre de l'algorithme` (5) ont été isolés. Ces traitements, indépendants des fonctionnalités d'invalidation, de verrouillage et de localisation, concernent par exemple, la gestion de l'ordre total. La gestion de la répartition de notre application a été modularisée au sein d'une fonctionnalité de `communications distantes` (6), dont les traitements sont délégués à la couche middleware Java RMI. Enfin, nous avons ajouté une fonctionnalité de `journalisation` (7). Cette fonction n'existe pas dans la spécification d'origine mais elle est intéressante pour superviser l'exécution du service. De plus, c'est une fonctionnalité que l'on retrouve dans quasiment tous les services *middleware*.

La vision des fonctionnalités séparées est schématisée sur la figure 1. Il est important de noter que ces fonctionnalités ne sont pas orthogonales les unes par rapport aux autres. Elles répondent à des besoins différents, mais sont en interaction les unes avec les autres pour aboutir au service final. Sur la figure 1, deux fonctionnalités partageant une même frontière sont en interaction l'une avec l'autre. Les flêches illustrent le sens des dépendances entre ces fonctionnalités.

Après cette étape d'identification des préoccupations, il est possible de modéliser de manière générique l'application sous forme de diagramme de classes UML (voir [LOI 03] pour plus de détails). La transposition consiste à associer à chaque fonctionnalité une classe. Les dépendances entre classes sont modélisées par des liens d'agrégation. Chaque classe implante un ensemble de méthodes lié à la sémantique de la fonctionnalité qui lui est associée. De manière à capturer l'ensemble de la sémantique de l'application, on modélise alors les diagrammes d'interaction liés au déclenchement de chaque opération source (par exemple une lecture).

4. Implantation de l'étude de cas

A partir des fonctionnalités isolées et d'une modélisation générique, il est intéressant d'étudier une transposition de notre vision de l'application vers un paradigme de programmation donné. Dans le cadre d'un modèle de composants techniques, on associe à un composant un objet Java usuel qui implante une fonctionnalité bien précise. Dans notre cas, nous avons cherché à encapsuler les fonctionnalités `invalidation` et `verrouillage` au sein d'un composant de plus haut niveau `cohérence`. Il suffit ensuite de définir l'assemblage de tous nos composants selon les règles de dépendances que nous avons identifiées. Avec JAC, la composition des aspects se modélise par une notation UML enrichies des concepts de l'AOP [PAW 02].

4.1. *avec Fractal*

A chaque interface Fractal est associée une interface Java dans laquelle on définit les signatures des méthodes (qui définissent donc un service). Chaque fonctionnalité est implantée au sein d'une classe Java dont l'instance sera encapsulée dans un composant primitif Fractal. Chaque classe implante les interfaces que le composant fourni et définit les références sur les interfaces qu'il requiert (cette classe implante alors une interface Fractal spécifique qui définit trois méthodes pour gérer le processus de liaison entre composants). L'ADL de Fractal décrit l'assemblage des composants (leurs types et l'ensemble des types d'interfaces qu'ils requièrent et/ou fournissent). On invoque alors les méthodes de l'API de Julia en chargeant le *template* associé à notre composant racine.

4.2. *avec Kilim*

L'implantation des différentes classes se fait en Java pur. En effet, le modèle de composants Kilim n'intervient pas directement au niveau du code, mais au niveau de la méta-représentation de l'application décrite dans les *templates* associés aux composants : à ce niveau de description, on utilise alors les abstractions du modèle de composant Kilim permettant d'instancier et de lier les instances de l'application. A l'exécution, on utilise les méthodes de l'API de Kilim pour charger le *template* racine de l'application.

4.3. *avec JAC*

On définit les composants d'aspects qui implantent des méthodes permettant de configurer le tissage des aspects associés autour de points de jonction (définis par des expressions régulières). On implante les méthodes *wrappantes* qui correspondent aux traitements associés à nos aspects non-fonctionnels puis les objets de base. Du fil d'exécution associé à chaque opération métier, on détermine l'ordre d'exécution des méthodes *wrappantes* correspondant à la sémantique de l'application. Celui-ci est relativement complexe dans notre cas : il a fallu gérer certaines difficultés telles le fait de faire passer de « l'information non-fonctionnelle » entre aspects ou encore de ne pas exécuter certaines méthodes *wrappantes* en fonction du contexte d'exécution. Enfin, on détermine les points de jonction dans des fichiers de configuration.

5. Comparaison des trois mises en œuvre

Dans cette partie, nous cherchons à comparer les différentes approches selon des critères qui nous ont paru pertinents.

5.1. *en terme de modèle de programmation*

Les critères retenus en terme de modèle de programmation ont été tirés de [MAR 02].

1) Les points d'entrée du composant au sein du modèle Fractal sont définis par les interfaces de contrôle (formulées par les spécifications) et les interfaces fonctionnelles (construites par le développeur). Dans Kilim, les points d'entrée sont les *ports*. Au sein de JAC, nous pouvons considérer que les points de jonction constituent les points d'entrée entre les objets de base et les composants d'aspects. La définition de l'encapsulation pour les trois modèles peut être déduite de cette définition de points d'entrée.

2) La notion d'identité (la capacité à définir de manière non ambigüe une entité logicielle) d'un composant dans Fractal se fait par l'intermédiaire d'un type de composant au niveau du modèle et d'une interface `ComponentIdentity` au niveau de

l'exécution qui permet alors de récupérer toutes les références d'interfaces externes du composant. Dans le modèle proposé par Kilim, on trouve respectivement la notion d'instance de *template* et la classe `RtComponent` à l'exécution. Dans JAC, l'identité d'une entité logicielle n'est marquée que par la distinction entre un objet de base et un composant d'aspect.

3) La composabilité au sein de Fractal est structurelle, elle est assurée par des liaisons entre interfaces clientes et serveurs, la notion de composant composite assure la récursivité. Une vérification sémantique de l'assemblage est rendue possible au moment de l'analyse de l'ADL puisque le typage des interfaces y est défini. Au sein de Kilim, l'assemblage est également structurel, il s'effectue par l'intermédiaire des opérateurs `plug` et `bind`, la composition entre *templates* assure la récursivité. Les composants Kilim ne sont pas typés, la sémantique de l'assemblage est donc à la charge du développeur. Dans JAC, l'assemblage entre objets de base et composants d'aspect est effectué par le processus de tissage, le long des coupes transversales définissant le déploiement des aspects. Le modèle est récursif dans le sens où l'on peut tisser un nombre illimité d'aspects autour d'une même méthode de base. La sémantique de la composition est comportementale puisque celle-ci se manifeste à l'exécution (par exemple, lors d'un appel de méthode). La vérification de la sémantique d'assemblage est à la charge du développeur qui doit s'assurer du bon ordre d'exécution des *wrappers*.

5.2. *en terme d'impact sur la séparation des préoccupations*

Il est évident que la vision de la séparation des préoccupations détermine la manière d'implanter notre application, qu'elle soit développée avec un modèle de composants techniques ou avec un modèle de programmation orientée aspects. C'est de cette vision qu'est déterminé une modélisation sous forme de diagrammes UML, que sont explicités les dépendances entre fonctionnalités et le fil d'exécution correspondant à la sémantique de l'application.

5.3. *en terme de dispersion de code*

Dans le cadre des modèles Fractal et Kilim, la construction de l'application s'effectue par assemblage de composants. Des liaisons entre composants peuvent être créées et détruites. Néanmoins, l'utilisation d'un composant et donc d'une fonctionnalité, se fait via des interfaces et des appels de méthode. Le code des composants est bien sûr modulaire, mais son utilisation reste dispersée et donc transverse aux différents autres composants.

Au sein de la plate-forme JAC, l'assemblage des entités se configure de manière indépendante du code de l'application et s'effectue alors dynamiquement au moment du processus de tissage. Il n'y a donc aucune référence codée en dur entre objets de

base et composants d'aspects.

5.4. *en terme de vision de l'application et de son évolutivité*

La vision d'une application assemblée avec Fractal est claire puisque le développeur est contraint par des règles d'écriture définies dans les spécifications (définitions d'interfaces et leurs implantations, implantations des mécanismes de liaisons). Les liaisons de type client/serveur permettent de montrer le sens de la dépendance entre les composants. Avec Kilim, la vision de l'application est plus ambigüe car le modèle n'impose aucune règle d'écriture du code, l'opérateur `plug` n'est pas asymétrique, le « sens » de la liaison ne reflète pas obligatoirement le sens de la dépendance entre deux composants et au niveau de l'ADL, les mécanismes de liaisons sont manipulés au même titre que les mécanismes d'instanciation. Cependant, l'utilisation de Kilim n'impose aucune contrainte au niveau du code de l'application, contrairement à Fractal. Avec JAC, la vision de l'application peut s'avérer complexe, surtout dans le cadre de notre prototype où de nombreux aspects non-fonctionnels sont tissés et demeurent inter-dépendants.

De manière à tester les capacités d'évolution des plates-formes, nous avons étudié la mise en œuvre d'une autre alternative concernant la gestion d'une cohérence causale au sein de l'algorithme initialement proposé. Cette modification a un impact sur les structures de données utilisées, sur des fractions de code concernant différentes fonctionnalités et bien évidemment sur le fil d'exécution lié à la sémantique de l'application. Nous nous sommes alors rendus compte que de nombreux changements d'implantation étaient alors nécessaires concernant les plates-formes de composants techniques, essentiellemment à cause de modifications de signatures de certaines méthodes des interfaces des composants permettant l'assemblage de l'application. Dans le cas de JAC, la transposition est plus souple : pour cet exemple précisemment, les dépendances entre aspects non-fonctionnels demeurent inchangées, les modifications de code sont localisées et influencent de manière moins marquée le reste de l'application.

5.5. *en terme de lignes de code*

La quantité de lignes de code qui a été nécessaire à l'implantation de l'algorithme est représentée sur la figure 2.

Nous avons implanté un prototype en Java « pur », d'abord de manière monolithique (sans SoC[2]) puis en prenant en compte notre vision des fonctionnalités séparées (avec SoC). Dans le deuxième cas, le code engendré est bien sûr plus élevé puisque les classes sont fragmentées. Nous avons donc besoin de code supplémentaire pour les « reconstituer ». Il en est de même pour les trois plates-formes étudiées : en utilisant

2. séparation des préoccupations (en anglais *separation of concerns*).

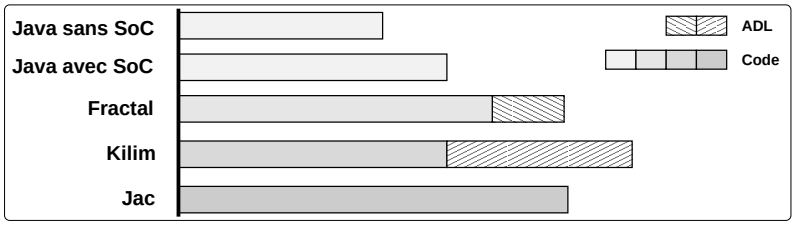

Figure 2. *Quantité de lignes de code en fonction de la plate-forme d'accueil.*

les interfaces Fractal, les méthodes *setter/getter* avec Kilim et les composants d'aspect avec JAC.

Le code de la version Java (avec SoC) est strictement identique à celui utilisé avec Kilim : dans le premier cas, on utilise une méthode `main` pour assembler les instances (par les constructeurs et les méthodes *setter*), alors que dans le cas de Kilim, l'assemblage est décrit par l'ADL.

Le code engendré par le prototype Fractal et JAC est plus élevé que celui de Kilim : en effet, au sein de ces deux premières plates-formes, le modèle de composants intervient au niveau du langage (gestion du mécanisme de liaison entre interfaces avec Fractal, gestion des *wrappers* avec JAC).
La description de l'application avec l'ADL de Kilim est plus coûteuse que celle de Fractal : on manipule beaucoup de mécanismes avec Kilim pour obtenir une méta-description complète de l'application, avec Fractal, on ne décrit que l'assemblage des composants alors que les mécanismes mis en œuvre interviennent au niveau des sources.

6. Conclusion & perspectives

Cet article présente, au travers d'une étude de cas, une comparaison entre trois *frameworks* Java : Fractal, Kilim et JAC. Cet article est un résumé d'une étude plus complète dont les détails peuvent être trouvés dans [LOI 03]. Même si ces trois *frameworks* ont des objectifs techniques différents, ils adressent le problème de la séparation des préoccupations et visent à permettre la construction de logiciels complexes. Fractal aborde le problème via l'assemblage de composants, Kilim via leurs configuration, et JAC via la programmation par aspects.

L'étude de cas traitée est un service distribué de partage de données selon l'algorithme de cohérence de K. Li & P. Hudak (voir section 3.1). Bien que de taille modeste, cette application est suffisamment significative pour mettre en avant six aspects non-fonctionnels à composer avec le code métier (voir section 3.2). Historiquement cette étude répond à des besoins d'ingénierie [THI 97] qui ont émergés lors du prototypage de services de cohérence au-dessus du micro-noyau ChorusOS dont l'objectif principal de ce projet était de fournir un service générique. Le service obtenu n'était pas

encore assez modulaire et adaptable. L'utilisation d'approches à composants permet de cantonner les évolutions nécessaires à certains modules sans risquer de compromettre l'ensemble du service.

L'implantation de cette étude de cas avec les trois *frameworks* (voir section 4) nous a permis d'aboutir aux observations de la section 5 et aux conclusions suivantes.

La composition est structurelle avec Fractal et Kilim (ADL XML), tandis qu'elle relève plus du domaine comportemental avec JAC (processus de tissage). La séparation des préoccupations étant intimement liée à la conception de l'application, il n'est pas surprenant de constater qu'aucune des trois approches mentionnées ne l'influence. Le code est moins dispersé avec JAC, alors qu'avec Fractal et Kilim, les appels aux services techniques restent dispersés dans l'application. L'évolutivité de l'application est donc moins complexe avec JAC. La vision de l'application est claire avec Fractal alors que la richesse des concepts de Kilim nuit plus à son appréhension. Néanmoins, contrairement à Fractal, Kilim n'impose aucune contrainte au niveau du code de l'application. Quant à JAC, la complexité du tissage engendrée par l'approche AOP n'est pas à négliger. En terme de code, le source Java Kilim reste « pur » mais la partie ADL XML est plus longue que celle de Fractal. Les codes de Fractal et JAC sont comparables. Finalement, en terme de performance, seul Kilim n'introduit pas d'indirections dans le code ce qui fournit les meilleures performances. Les indirections de JAC sont nettement plus coûteuses que celles de Fractal. En guise de remarque finale, on peut constater qu'aucune des trois approches ne l'emporte nettement, même si la clarté de Fractal, l'évolutivité de JAC et les performances de Kilim peuvent être mises en avant.

Notons que les critères de comparaison qui nous ont permis d'aboutir à ces observations auraient pu être complétés. Des comparaisons en terme de cycle de vie des applications, de déploiement ou de maintenance auraient pu être abordées. Nous les reportons à une étude future.

En terme de perspectives, nous pouvons souligner que comme toute approche expérimentale, cette étude nécessiterait d'être complétée avec d'autres applications. En particulier, malgré la taille modeste de l'application, le nombre d'aspects est important. Il n'est pas acquis que pour d'autres applications, notamment comportant une partie métier plus significative, les conclusions seraient identiques. D'autre part, les avantages comparés de l'approche assemblage/configuration à la Fractal/Kilim, et de l'approche par aspects à la JAC, plaideraient pour une approche mixte dans laquelle les composants pourraient être assemblés, mais aussi se voir étendus par des services non-fonctionnels développés sous forme d'aspects.

7. Bibliographie

[BRU 02a] BRUNETON E., « Julia Tutorial », 2002, `http://fractal.objectweb.org`.

[BRU 02b] BRUNETON E., COUPAYE T., STEFANI J., « The Fractal Composition Framework », 2002, `http://fractal.objectweb.org`.

[DIJ 76] DIJKSTRA E., *A Discipline of Programming*, Prentice-Hall, 1976.

[GRE 02] GRESSIER-SOUDAN E., « Contribution aux Messageries Industrielles », Habilitation à Diriger des Recherches, LIFL, Décembre 2002.

[HOR 02] HORN F., DELPIANO F., « The Kilim Configuration Framework », 2002, http://kilim.objectweb.org.

[jBo] jBoss Group, « jBoss Application Server », www.jboss.org.

[KIC 91] KICZALES G., DES RIVIERES J., BOBROW D., *The Art of the Metaobject Protocol*, MIT Press, 1991.

[KIC 97] KICZALES G., LAMPING J., MENDHEKAR A., MAEDA C., LOPES C., LOINGTIER J., IRWIN J., « Aspect-Oriented Programming », *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, vol. 1241 de *Lecture Notes in Computer Science*, Springer, Juin 1997, p. 220-242.

[KIC 01] KICZALES G., HILSDALE E., HUGUNIN J., KERSTEN M., PALM J., GRISWOLD W., « An Overview of AspectJ », *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, vol. 2072 de *Lecture Notes in Computer Science*, Springer, Juin 2001, p. 327-353.

[LI 89] LI K., HUDAK P., « Memory Coherence in Shared Virtual Memory Systems », *ACM Transactions on Computer Systems (TCOS)*, vol. 7, n° 4, 1989, p. 321-359.

[LOI 03] LOIRET F., « Assemblage de composants et programmation par aspects », Rapport de DEA Systèmes Informatiques Répartis, Université Paris 6, Septembre 2003, http://imhodeb.tuxfamily.org/acpoa.

[MAR 02] MARVIE R., PELLEGRINI M.-C., « Modèles de composants, un état de l'art », *L'objet*, vol. 8, Hermès, 2002, p. 61-89.

[OMG] OMG, « CORBA Component Model », www.omg.org.

[PAR 72] PARNAS D. L., « On the criteria to be used in decomposing systems into modules », *Communications of the ACM*, vol. 15, n° 12, 1972, p. 1053–1058, ACM Press.

[PAW 01] PAWLAK R., SEINTURIER L., DUCHIEN L., FLORIN G., « JAC : A Flexible Solution for Aspect-Oriented Programming in Java », *Proceedings of Reflection'01*, vol. 2192 de *Lecture Notes in Computer Science*, Springer, Septembre 2001, p. 1-24, http://jac.objectweb.org.

[PAW 02] PAWLAK R., DUCHIEN L., FLORIN G., LEGOND-AUBRY F., SEINTURIER L., MARTELLI L., « An UML notation for aspect-oriented software design », 2002, http://jac.aopsys.com/papers/uml/uml.html.

[Sun98] Sun Microsystems, « Enterprise Java Beans White Paper », Décembre 1998, http://www.javasoft.com/products/ejb/.

[THI 97] THIEBOLD V., « Mémoire Répartie Partagée sur CHORUS : de l'implantation de la cohérence causale vers un mappeur générique », Mémoire d'ingénieur, CNAM, 1997.

[WIG 03] WIGLEY A., WHEELWRIGHT S., BURBIDGE R., MACLEOD R., SUTTON M., *Microsoft .NET Compact Framework (Core Reference)*, Microsoft Corporation, 2003.