E{Java, CaesarJ, Scala} : un exercice d'intégration de la programmation par objets, par aspects et par évènements

Jacques Noyé

E{Java, CaesarJ, Scala} : un exercice d'intégration de la programmation par objets, par aspects et par évènements ⋆

Jacques Noyé

ASCOLA - École des Mines de Nantes/INRIA, LINA
Jacques.Noye@mines-nantes.fr

Résumé

La programmation par évènements et la programmation par aspects sont des paradigmes de programmation qui s'avèrent compléter utilement la programmation par objets dans une très large gamme d'applications. Leur utilisation concomitante, bien que possible dans un langage comme Java, est toutefois malaisée. Malgré leur très grande proximité, les solutions proposées présentent de nombreuses faiblesses et des irrégularités qui sont des sources notables de perplexité et de complexité. Considérons ces paradigmes deux par deux.

Programmation par objets et programmation par évènements En Java, la programmation par événements fait essentiellement appel au schéma de conception « observateur » ou à ses variantes. Cette solution pose divers problèmes, en particulier l'ajout de code d'infrastructure qui vient cacher la logique de base du programme. Pour pallier à ces inconvénients, C# permet de définir un évènement sous la forme d'un champ d'un objet et d'y associer dynamiquement des gestionnaires. Ces évènements se comportent toutefois étrangement vis-à-vis de l'héritage.

Programmation par évènements et programmation par aspects Il est tout à fait naturel de considérer, les *points de jonction* de la programmation par aspects comme des évènements générés par l'exécution du *programme de base* [1], évènements que l'on dira *implicites* pour les distinguer des évènements *explicites*, considérés ci-dessus, dont le déclenchement est explicitement programmé. On peut alors voir l'*action* exécutée par un aspect comme un gestionnaire d'évènement. Mais quelle place doit accorder cette vision évènementielle aux *points de coupe*, ces prédicats chargés de la sélection des points de jonction ?

Programmation par objets et programmation par aspects Pour AspectJ, le standard de programmation par aspects en Java, les aspects sont des classes sans en être. Ils sont instanciés mais implicitement, possèdent des membres, points de coupe inclus, mais ces derniers sont statiques, et les possibilités d'héritage sont très limitées. Finalement, la possibilité offerte aux aspects d'intervenir sans restriction sur l'ensemble du programme de base casse le raisonnement modulaire propre à la programmation par objets.

Notre proposition pour résoudre ces problèmes est de centrer l'intégration des trois paradigmes autour des principes de la programmation par objets et de réaligner autour de ces principes l'ensemble des concepts rencontrés :

- il n'y a plus d'aspects, seulement des classes dont certaines, que l'on pourrait qualifier d'aspectuelles, jouent un rôle habituellement dévolu aux aspects ;
- les notions de point de jonction, de point de coupe et d'action disparaissent au profit des notions d'occurrence d'évènement, d'évènement et de gestionnaire d'évènement ;
- de la même manière qu'un point de coupe en programmation par aspects sélectionne des points de jonctions et peut, en tant que prédicat, être défini comme une composition de prédicats, un évènement sélectionne des occurrences d'évènements et peut être défini *déclarativement* par composition d'autres évènements, ultimement des évènements implicites ou explicites ;

⋆. Ce travail a été effectué en collaboration avec Angel Núñez, Jurgen Van Ham, Vaidas Gasiūnas et Mira Mezini. Il a été partiellement financé par le projet AMPLE : Aspect-Oriented, Model-Driven, Product Line Engineering (STREP IST-033710).

– évènements et gestionnaires sont des membres d'instance. Par exemple, les règles habituelles liées à l'héritage (redéfinition et utilisation de `super`) s'appliquent.

Cette combinaison de caractéristiques fournit un modèle régulier et très flexible de programmation qui permet de mélanger les paradigmes existants, tout en en corrigeant certaines faiblesses, mais donne aussi accès à des hybridations intéressantes comme la possibilité de définir des évènements déclaratifs portant sur des évènements explicites. Ce modèle a été implémenté avec quelques variations dans EJava [2], ECaesarJ [3,4,2] et EScala [5], qui étendent respectivement Java, CaesarJ et Scala.

Suivant les possibilités de quantification fournies pour définir les évènements implicites, le modèle, dans sa généralité, ne garantit pas la préservation d'un raisonnement modulaire en présence de classes aspectuelles [1]. C'est toutefois le cas dès que les évènements observés par les classes aspectuelles sont définis au niveau des classes observées. Dans tous les cas, la définition des évènements en tant que membres d'instance permet de contrôler la portée des évènements qui ne peuvent être observés que par l'intermédiaire des relations de leurs instances englobantes. Au contraire, les propositions connexes, citons [6] et [7], mettent en avant le *type* des évènements (ou des points de jonction suivant la terminologie employée). Ceux-ci ont alors une portée globale et réduire leur observation à un ensemble restreint de sources demande de faire appel à des filtres, potentiellement couteux, ou de revenir à la programmation fine de sujets et d'observateurs. La possibilité d'utiliser du sous-typage est toutefois attractive. Une analyse fine de ces différentes approches et de leur adaptation à différents types d'applications reste à faire. Il est très probable que cette analyse montre qu'aucune approche n'est véritablement meilleure que les autres sur l'ensemble des applications et on peut dès à présent se poser la question de les combiner. Pour finir, signalons que ces travaux ont démarré avec l'objectif de prendre en compte la concurrence suite à des réflexions à un niveau conceptuel [8,9]. Cette dimension a été laissée de côté dans un premier temps mais est de nouveau d'actualité et bénéficie du riche environnement fourni par Scala.

Références

1. Douence, R., Motelet, O., Südholt, M. : A formal definition of crosscuts. In : Meta-Level Architectures and Separation of Crosscutting Concerns, Third International Conference (Reflection 2001), Springer (2001)
2. Núñez, A. : A Programming Model Integrating Classes, Events and Aspects. PhD thesis, École des Mines de Nantes and Université de Nantes (2011)
3. Núñez, A., Noyé, J., Gasiūnas, V. : Declarative definition of contexts with polymorphic events. In : Proc. of the International Workshop on Context-Oriented Programming at ECOOP'09 (COP '09), ACM (2009)
4. Núñez, A., Noyé, J., Gasiūnas, V., Mezini, M. : Product Line Implementation with ECaesarJ. In : Building Software Product Lines : The AMPLE Approach. Cambridge University Press (2011)
5. Gasiūnas, V., Satabin, L., Mezini, M., Núñez, A., Noyé, J. : EScala : Modular event-driven object interactions in Scala. In : Proc. of the 10th International Conference on Aspect-Oriented Software Development (AOSD 2011), ACM (2011)
6. Rajan, H., Leavens, G.T. : Ptolemy : A language with quantified, typed events. In : Proc. of the 22nd European Conference on Object-Oriented Programming (ECOOP 2008), Springer (2008)
7. Inostroza, M., Tanter, E., Bodden, E. : Join point interfaces for modular reasoning in aspect-oriented programs. In : Proc. of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE '11), ACM (2011)
8. Douence, R., Le Botlan, D., Noyé, J., Südholt, M. : Concurrent aspects. In : Proc. of the 5th International Conference on Generative Programming and Component Engineering (GPCE '06), ACM (2006)
9. Núñez, A., Noyé, J. : An event-based coordination model for context-aware applications. In : 10th International Conf. on Coordination Models and Languages (COORDINATION 2008), Springer (2008)

1. EScala est plus strict et préserve un raisonnement modulaire. EJava et ECaesarJ laissent le choix au programmeur.