# Programmation d'applications concurrentes et réparties par objets, réflexion, interactions et agents

**Jean-Pierre Briot**

*Laboratoire d'Informatique de Paris 6*
*Paris 6 - CNRS, Pole IA*
*4 place Jussieu*
*75252 Paris Cedex 05*
*e-mail :* `Jean-Pierre.Briot@lip6.fr`

RÉSUMÉ. *Cet article résume rapidement les tendances actuelles d'utilisation du concept d'objet pour la programmation d'applications concurrentes et réparties et introduit les concepts complémentaires de réflexion, interaction et agent.*

ABSTRACT. *This paper briefly summarizes actual trends in using the concept of object for programming concurrent and distributed applications and introduces additional important concepts of reflection, interaction and agent.*

MOTS-CLÉS : *Programmation, concurrence, répartition, objet, réflexion, interaction, composant, agent.*

KEY WORDS: *Programming, concurrency, distribution, object, reflection, interaction, component, agent.*

## 1. Introduction

L'objectif de ce court article est de faire le point sur quelques concepts importants en matière de programmation d'applications concurrentes et réparties coopératives. La programmation par objets s'est d'ores et déjà imposée comme la norme en matière de programmation d'applications. Cependant l'objet s'est révélé au cours des années 80, alors que la majorité des applications informatiques restaient séquentielles et centralisées. On assiste depuis à une révolution technologique et surtout culturelle avec la transition inéluctable vers une informatique décentralisée et coopérative. Le concept d'objet reste à notre avis un excellent vecteur de base pour ces nouveaux enjeux, mais nous verrons qu'il n'est pas suffisant et des concepts complémentaires peuvent lui être adjoints : le concept de réflexion, le concept d'interaction et enfin le concept d'agent.

## 2. Objets

Les intérêts du concept d'objet (et de transmission de messages) pour la programmation concurrente et répartie proviennent à notre avis du fait que ces concepts sont à la fois assez forts pour assurer structuration, modularité et encapsulation, et assez souples (« mous ») pour recouvrir également granularités variées, hétérogénéité et interactions variées. Ce n'est d'ailleurs ainsi pas un hasard si la quasi-totalité des architectures de systèmes d'exploitation répartis développées actuellement se fondent sur la notion d'objet.

La programmation par objets repose sur quelques concepts simples et unificateurs. Un programme se décompose en un ensemble de modules autonomes, appelés *objets*, qui interagissent au travers d'un protocole de communication unifié, la *transmission de message*. Les objets correspondent à différentes entités des domaine et problème considérés. Un objet se présente comme une *capsule* contenant à la fois des données et des opérations opérant sur celles-ci. Le principe de transmission de message introduit une séparation entre la signature des opérations qui pourront être invoquées de l'extérieur (appelée *interface*) et la représentation interne de l'objet. Ceci est appelé le principe d'*encapsulation* et il favorise la séparation entre spécification des services et leur mise en œuvre. Il permet également abstraction et généricité.

Ces principes fondateurs de la programmation par objets permettent à la fois décomposition, modularité et protection. De manière à favoriser l'abstraction et la factorisation d'objets semblables, la plupart des langages et systèmes à objets reposent sur la notion de *classe*, comme abstraction et factorisation d'objets. De plus, des mécanismes de spécialisation, et en premier lieu l'*héritage*, permettent de réutiliser des propriétés déjà définies par certaines classes d'objets pour les étendre ou les modifier partiellement. Ces mécanismes augmentent encore la généricité et la réutilisation des programmes et ont permis de valider les intérêts de la programmation par objets au niveau du génie logiciel. Enfin, la programmation par objets intègre de manière naturelle une gestion dynamique des ressources, puisque l'on peut créer de nouveaux objets au fur et à mesure des besoins.

## 3. Objets pour la programmation concurrente et répartie

De par sa modularité un objet apparaît de manière naturelle comme une unité potentielle de concurrence et plus encore de répartition. La transmission de message assure en effet non seulement l'indépendance entre les services offerts par un objet et sa représentation interne, mais également l'indépendance vis-à-vis de son *emplacement* sur tel ou tel site (processeur, mémoire, ou machine). Enfin l'autonomie des objets (comme capsule de données et d'opérations associées) facilite la prise en compte de migration ou de duplication éventuelles.

## 4. Abstractions pour la programmation concurrente et répartie

La programmation par objets, telle qu'elle est, malgré ses intérêts n'apporte pas par elle-même toutes les réponses aux enjeux de la programmation concurrente et répartie pour exprimer des aspects tels que : gestion des ressources, contrôle de la concurrence, et résistance aux pannes. Différents types d'abstractions et de mécanismes, tels que : tâche, synchronisation, transaction, communication de groupe, ont été proposés, pour les aborder. Une question qui se pose alors est la manière dont on va associer le concept d'objet à ces abstractions [BRI 96]. Nous pouvons distinguer trois approches principales.

La première approche, dite *applicative*, applique tel quel le concept d'objet à la conception d'applications concurrentes et réparties. Différentes abstractions de concurrence et de répartition : tâche, sémaphore, moniteur, transaction, communication de groupe, fichiers, séquenceur, etc., sont des classes d'objets regroupées en bibliothèques. L'héritage permet d'organiser des hiérarchies de spécialisation.

## 5. Vers plus de transparence : réflexion

Cependant de telles abstractions restent assez lourdes à utiliser pour l'utilisateur final qui programme son application car il doit manipuler simultanément dans une même application les objets modélisant le domaine d'application ainsi que les objets de contrôle de la concurrence et de la répartition (exemple : tâches, verrous, RPC et transactions). Les choix d'association (à quel objet(s) de l'application doit-on associer une tâche ou une synchronisation ?) ne sont de plus pas triviaux.

Une deuxième approche, qualifiée d'*intégrée*, systématise une association (exemple : à chaque objet est implicitement associé une tâche dans les langages d'acteurs, à chaque objet est implicitement associé un verrou dans le langage Java). Une telle intégration simplifie les choix de conception mais au risque de choix un peu réducteurs et de possibles surcoûts.

Une troisième approche, qualifiée de *réflexive*, consiste à opérer de telles associations de manière sélective, tout en restant transparente. Pour ce faire, certaines caractéristiques du calcul sont *réifiées*, c'est-à-dire représentées par des objets (on parle alors de *méta-objets*), et deviennent alors explicites et ainsi modifiables. Ainsi par exemple ce qu'on peut appeler le cycle de calcul d'un objet est rendu explicite (réifié) par un certain nombre de méta-objets : réception de messages, sélection de messages, ressources du calcul et envoi de messages [MCA 95]. C'est par telle ou telle spécialisation de ces méta-objets que l'on va pouvoir spécialiser de manière fine et locale à un ou plusieurs objets la concurrence et la répartition des calculs. Par exemple, une spécialisation des méta-objets d'envoi et de réception de messages peut implanter une communication distante de type RPC. Une spécialisation du méta-objet de sélection de messages peut exprimer une certaine politique de synchronisation et enfin une spécialisation du méta-objet de ressources de calcul peut associer une activité (tâche) dédiée à l'objet.

Notez que l'approche réflexive ne se substitue pas aux deux approches précédentes : approche applicative pour dégager les abstractions fondamentales et approche intégrée au niveau de l'utilisateur, mais elle les relie, ce qui permet à la fois la flexibilité des abstractions et une grande transparence pour l'utilisateur [BRI 96].

## 6. Réifier les interactions

La transmission de messages entre objets induit des modèles d'interaction de type client serveur. (Notez cependant qu'un objet peut alternativement jouer le rôle de client ou de serveur suivant qu'il envoie ou reçoit un message). Les modèles de communication asynchrones offrent une première tentative de désynchronisation entre le client et le serveur. Les modèles de communication par diffusion, de type multicast, permettent de passer d'une communication de type point à point à une communication vers un groupe d'objets. Enfin la programmation par événements, de type « publish/subscribe », systématise la dynamicité de la gestion des membres du groupe et ainsi la facilité de reconnexion d'un objet émetteur à un ensemble d'objets receveurs. La notion de *connecteur*, venant des architectures logicielles [SHA 96], systématise la réification de la notion de référence et donc de communication entre objets. On peut alors reconnecter les objets (on parle alors plutôt de *composants*) entre eux sans avoir à les reprogrammer et on peut ainsi également exprimer de manière explicite différents modes de communication (synchrone, asynchrone, RPC...).

Cependant la connexion, et donc l'assemblage des composants, reste fondamentalement statique, c'est-à-dire intervenant avant la phase d'exécution. De plus les connecteurs restent encore relativement élémentaires et font intervenir un nombre limité d'objets.

En conséquence certains chercheurs commencent à vouloir exprimer et donc réifier des modèles d'*interactions* arbitrairement complexes entre objets. L'objectif est de pouvoir exprimer ces modèles d'interactions de manière générique et externe aux différents objets, ainsi de manière indépendante de leurs comportements propres. Une telle réification de modèles d'interaction et leur expression, en général par des protocoles, est déjà connue dans des cas précis, par exemple les protocoles transactionnels. Mais l'idée est de systématiser une telle approche à divers modèles d'interaction et de calcul coopératif (synchronisation, coordination, réplication, etc.). Un objectif supplémentaire est de pouvoir appliquer ces modèles d'interaction dynamiquement à un ensemble d'objets de manière à adapter dynamiquement les interactions en fonction de la situation [STU 94]. Un exemple est d'augmenter dynamiquement la fiabilité d'un serveur par réplication active, puis par rétraction, au cours des calculs en fonction de l'évolution de la criticalité de ce serveur.

## 7. Vers les agents

Le concept d'*agent* pousse encore plus loin cette dynamicité. Il est bien sûr actuellement très difficile de donner une définition consensuelle de ce qu'est un agent, en particulier dans un court article [BRI 00]. Mais deux caractéristiques importantes qui le distinguent des objets sont à notre avis la capacité d'initiative (souvent appelée *pro-activité*) en fonction de sa mission et des circonstances rencontrées [GAS 98] et le concept d'*organisation* qui régit les modes d'interaction entre agents. On dépasse donc largement le schéma du client serveur et les schémas d'interaction entre agents peuvent devenir arbitrairement complexes. Les agents se situent donc pour nous à un niveau de conception plus élevé que les objets. Mais les techniques de programmation à base d'objets, réflexion, composants et interactions, que nous avons résumées, sont tout à fait appropriées pour les implémenter [GUE 99].

## 8. Conclusion

Comme on le voit, la programmation d'applications concurrentes et réparties se dirige vers toujours plus de flexibilité et de dynamicité. Une difficulté est de ce fait de trouver un équilibre entre la complexité accrue des architectures et la facilité d'utilisation pour l'utilisateur, tout en assurant une efficacité suffisante. Un autre enjeu est de pouvoir obtenir certaines garanties (compatibilité entre objets et protocoles, propriétés de sûreté et de vivacité, etc.) sur les programmes malgré leur dynamicité.

Remerciements

La première partie de cet article se base sur un article écrit en collaboration avec Rachid Guerraoui [BRI 96]. Nous remercions également Michel Dao pour son aide pour finaliser cet article.

## 9. Bibliographie

[BRI 96] BRIOT J.-P., GUERRAOUI R., « Objets pour la programmation parallèle et répartie : intérêts, évolutions et tendances », *Technique et Science Informatiques (TSI)*, vol. 15, n° 6, 1996, p. 765–800.

[BRI 00] BRIOT J.-P., DEMAZEAU Y., Eds., *Les Multi-Agents*, IC2, Hermès, Printemps 2000, à paraître.

[GAS 98] GASSER L., BRIOT J.-P., « (Jean-Pierre Briot interviews Les Gasser on) Agents and Concurrent Objects », *IEEE Concurrency*, vol. 6, n° 4, 1998, p. 74–81.

[GUE 99] GUESSOUM Z., BRIOT J.-P., « From Active Objects to Autonomous Agents », *IEEE Concurrency*, vol. 7, n° 3, 1999, p. 68–76.

[MCA 95] MCAFFER J., « Meta-Level Programming with CodA », *European Conference on Object Oriented Programming (ECOOP'95)*, n° 952 LNCS, Springer-Verlag, août 1995, p. 190–214.

[SHA 96] SHAW M., GARLAN D., *Software Architecture - Perspectives on an emerging discipline*, Prentice Hall, 1996.

[STU 94] STURMAN D., AGHA G., « A Protocol Description Language for Customizing Failure Semantics », *13th Symposium on Reliable Distributed Systems*, IEEE Computer Society Press, octobre 1994, p. 148–157.

***Jean-Pierre Briot*** *est directeur de recherche au CNRS, membre du Laboratoire d'Informatique de Paris 6 (LIP6). Il y est responsable du thème de recherche "Objets et Agents pour les Systèmes d'Information et de Simulation (OASIS)", au carrefour du génie logiciel, de l'intelligence artificielle et de l'informatique répartie.*

Traduction du titre pour le sommaire : *Programming concurrent and distributed applications with objects, reflection, interactions and agents*