# De l'Orienté Objet à l'Orienté Tâches – Des modèles embarqués pour l'intégration et le traçage d'un nouveau type de composants

## *From Object-Oriented to Tasks-Oriented – Embedded models for component integration and tracing*

**Arnaud LEWANDOWSKI (1), Grégory BOURGUIN (1), Jean-Claude TARBY (2)**

(1) LIL, Université du Littoral Côte d'Opale, France
Arnaud.Lewandowski@lil.univ-littoral.fr, Gregory.Bourguin@lil.univ-littoral.fr

(2) LIFL, Université des Sciences et Technologies de Lille, France
Jean-Claude.Tarby@univ-lille1.fr

**Résumé**. Les utilisateurs des nouvelles technologies étant de plus en plus exigeants, la problématique de la malléabilité des systèmes informatiques et de leur adéquation aux nouveaux usages n'a jamais été aussi prégnante. C'est notamment une des raisons de l'utilisation massive des approches à base de composants, favorisant l'adaptation et la réutilisation. Toutefois, ce succès reste terni par certaines lacunes majeures, relatives notamment à l'accessibilité des mécanismes d'intégration et d'adaptation des composants d'une part, et au test et à l'évaluation de l'utilisation des composants dans leur contexte d'usage d'autre part. Pour pallier ces difficultés, dues en partie à un problème d'ordre sémantique, nous proposons dans cet article une solution qui tend à marier les modèles de tâches, du domaine de l'IHM, et les modèles de composants existants. Cette solution s'inscrit dans un véritable cadre pluridisciplinaire dédié au principe de co-évolution et correspond, dans un premier temps, à une nouvelle démarche de conception descendante — l'approche Orientée Tâches (OT) — déclinée en 5 étapes et supportée par STOrM, un outil dédié à la création et à la manipulation de Composants Orientés Tâches, les COTs.
**Mots-clés**. Composant, Tâche, Intégration, Trace.

**Abstract.** It has been demonstrated for many years that system tailorability, as defined by Mørch, is a strong issue for supporting the users emerging needs. In order to create such tailorable environments, a widely used solution consists in component-based approaches. However these technologies still face some drawbacks, especially regarding the abstraction level of their integration means, and their evaluation in their context of use. In this paper, and in order to palliate these lacks, we propose a new solution that tends to merge tasks models, from the HCI research field, and existing component models. This solution consists in a new top-down design approach — the Task Oriented approach — supported by STOrM, a tool dedicated to the creation and manipulation of Task Oriented Components.
**Keywords**. Component, Task, Integration, Trace.

## 1 Introduction

L'informatique est aujourd'hui plus que jamais un domaine pluridisciplinaire. En environ deux décennies, et en particulier dans le domaine des Interactions Homme-Machine (IHM), il est frappant de constater à quel point les concepteurs informatiques ont fait évoluer leur science au travers de collaborations étroites avec les experts de domaines différents. Ainsi, les psychologues, les ergonomes et les informaticiens se sont alliés pour inventer de nouveaux modèles de conception facilitant le dialogue entre ces différents intervenants. Parmi ces modèles, nous pouvons citer les modèles de tâches, de plus en plus utilisés par les informaticiens (Clerckx *et al.*, 2006, Luyten *et al.*, 2005, Paris *et al.*, 2005, Reichart *et al.*, 2004). De nouvelles approches comme la conception participative (Schuler et Namioka, 1993, O'Neill et Johnson, 2004) ont vu le jour, mettant encore plus en avant ce besoin de pluridisciplinarité en plaçant chaque acteur lié au système, y compris l'utilisateur final, comme un expert de son domaine et complètement intégré au processus de conception du logiciel. On peut également citer, à un niveau encore plus générique, les collaborations avec les spécialistes en Sciences Humaines et Sociales (SHS), grâce par exemple à la Théorie de l'Activité (TA) (Bedny et Meister, 1997), qui ont permis de mieux comprendre les fondements des activités humaines que tout système informatique tente de supporter, soulignant par exemple la propriété fondamentale de l'émergence des besoins utilisateurs envers leur système, et mettant en exergue la nécessité de repenser la conception des systèmes informatiques pour qu'ils ne répondent plus simplement à un ensemble de besoins définis *a priori*, mais pour qu'ils puissent être co-construits (Bardram, 1998) par leurs propres utilisateurs et *in situ*.

Depuis plusieurs années, nous avons nous-mêmes participé à cette évolution pluridisciplinaire de la conception des systèmes, nous fondant sur les apports de la TA, et proposant de supporter la co-évolution au sein des Systèmes Interactifs (SI) (Bourguin *et al.*, 2001, Bourguin et Derycke, 2005). La co-évolution, qui s'intéresse à la relation dialectique entre les utilisateurs et leurs systèmes, est définie par Bourguin et Derycke comme « traduisant tant les ajustements continus, négociés et socialement situés des pratiques de la part des individus, que les ajustements apportés au comportement du système interactif [supportant et influençant leurs activités] ». En accord avec ce principe, il est nécessaire de fournir aux utilisateurs les moyens permettant de (re)définir dynamiquement et de façon coopérative leurs supports informatiques. Ces idées ont été mises en œuvre au sein de la plateforme CooLDA et une de ses instances, le projet CooLDev (Lewandowski, 2006). Ces travaux nous ont amenés à mettre en relation directe certains résultats provenant de la TA avec des techniques génériques de l'Informatique.

Il est toutefois important de souligner que nous ne sommes pas les seuls chercheurs qui, depuis plusieurs années, tentent d'apporter des réponses au caractère émergeant des besoins des utilisateurs. Du fait qu'ils sont pour nous une grande source d'inspiration, nous souhaitons en particulier citer les travaux de Mørch, qui a traduit cette caractéristique par la notion de « malléabilité » (« end-user tailoring » en anglais), employée pour « décrire l'activité consistant à adapter les applications informatiques génériques en fonction des usages et des besoins des utilisateurs »[1] (Mørch, 1997). Dans son étude, Mørch identifie trois niveaux pour réaliser cette malléabilité : le *paramétrage* (*customization* en anglais), l'*intégration*, et l'*extension* de composants :

[1] « End-user tailoring is the term I use […] to describe the activity of adapting generic computer applications to local work practices and user needs »

- Le *paramétrage* consiste à éditer les préférences de l'application, ou des composants qui la constituent, par la sélection d'une ou plusieurs valeurs dans un ensemble prédéfini d'options.
- L'*intégration* va plus loin en permettant de modifier l'application par l'ajout, la suppression ou la réorganisation de fonctionnalités réalisées sous la forme de composants. Ces composants intégrables peuvent aussi bien être de bas niveau (de l'ordre des commandes) que de haut niveau (de l'ordre des applications). Cette intégration peut notamment être réalisée au travers de scripts ou de macros, permettant de piloter différents composants afin de les faire fonctionner « ensemble ».
- L'*extension* désigne la modification de l'implémentation de l'application, ou de ses composants, et correspond à la redéfinition même du code fonctionnel dans le langage d'implémentation et au niveau d'abstraction de ses constituants de plus bas niveau.

Comme le souligne Mørch, l'extension de composants correspond à la fois au plus grand niveau de malléabilité et au mécanisme le moins accessible aux utilisateurs. C'est pourquoi, du fait que nous souhaitons mieux supporter la co-évolution au sein des systèmes, les travaux présentés dans ce papier s'intéressent aux deux premiers niveaux proposés par Mørch, c'est-à-dire, le paramétrage et l'intégration de composants.

D'une manière plus générale, et même si les fondements théoriques et les propositions diffèrent, il est fort intéressant de constater que nombre des travaux qui s'intéressent au support des besoins émergents sont basés, comme les nôtres, sur des approches à base de composants. Les composants logiciels semblent donc apporter une réelle solution aux nouveaux problèmes liés à la conception. Il s'avère même que ces composants, autrefois considérés comme de simples briques à assembler par l'informaticien, peuvent, voire devraient, être parfois destinés directement aux utilisateurs finaux. Les avancées technologiques ont en effet largement contribué à rendre les utilisateurs de plus en plus exigeants vis-à-vis des systèmes informatiques, et la démocratisation de l'Internet a entraîné la création d'une pléthore d'outils destinés à supporter les activités humaines. Or, si autrefois l'utilisateur n'avait d'autre choix que de s'adapter à un système rigide et sans concurrence, il va aujourd'hui vouloir tenter d'adapter son système, recherchant de nouvelles fonctionnalités qu'il voudrait lui-même intégrer à son environnement de travail, ceci en accord avec le principe de co-évolution. Comme nous l'avons souligné plus tôt, la communauté scientifique a déjà bien identifié le fait que l'intégration dynamique de nouvelles fonctionnalités dans un système est un problème auquel répondent bien les approches composants. Il est vrai que les solutions recherchées par les utilisateurs sont en général d'une granularité plus forte[2] que les composants manipulés par les développeurs. En effet, si le développeur manipule classiquement des entités comme des boutons ou des zones de texte, il est plus probable que l'utilisateur souhaite intégrer un outil tel qu'un client de discussion en ligne ou « tchat ». Cependant, même s'il est d'un haut niveau d'abstraction, un tel outil se présente aussi sous la forme d'un composant logiciel qu'il faudra intégrer dynamiquement et harmonieusement au sein du système informatique de l'utilisateur. De plus, certains composants utilisés par les informaticiens comme les Services Web (Ferris et Farrel, 2003) s'avèrent aussi

[2] Par granularité plus forte, nous entendons "avec des grains plus gros" et non pas "avec plus de grains".

aujourd'hui être de granularité bien plus forte et au fonctionnement bien plus complexe qu'un widget classique.

Même si les composants logiciels sont susceptibles d'apporter une réponse au besoin de malléabilité, plusieurs problèmes restent à résoudre. En effet, les enjeux à satisfaire sont au minimum de concevoir des composants à la fois intégrables (de manière fine dans l'environnement cible), et supportant correctement la tâche pour laquelle ils ont été construits (et qui correspond à la fonction recherchée par son utilisateur, qu'il soit concepteur ou non). Néanmoins, une étude de l'état de l'art nous a permis de noter plusieurs manques dans les moyens liés aux composants actuels en vue de la satisfaction de ces enjeux. Le but de cet article est donc de souligner ces problèmes que nous avons identifiés et d'y proposer une solution. Cette proposition correspond à une première étape dans nos travaux pour le support de la co-évolution. Elle est dédiée à la spécification d'un nouveau mariage entre les modèles de tâches, issus du domaine des IHM et qui sont aujourd'hui peu utilisés et pas ou très peu outillés dans le développement de logiciels, et les modèles de composants existants. C'est pourquoi nous abordons dans un premier temps une nouvelle démarche descendante de conception. Enfin, nous montrerons que cette proposition est pluridisciplinaire à plusieurs titres, puisqu'elle associe des résultats issus de l'ergonomie et de l'ingénierie du logiciel, mais devrait aussi apporter de nouveaux outils à la pluridisciplinarité des démarches actuelles de conception.

La partie 2 soulignera les problèmes liés aux approches composants actuelles. La partie 3 présentera le principe global de notre proposition sous la forme d'un modèle étendu de composants logiciels : des composants non plus simplement Orientés Objets (OO), mais aussi Orientés Tâche (OT). La partie 4 s'intéressera aux moyens de conception descendante de Composants OT ou COTs, avec la présentation de STOrM, un outil que nous avons créé sous la forme d'un plug-in Eclipse et qui permet d'assister ce processus de création. La partie 5 donnera des exemples de mise en œuvre de composants OT. Enfin, les parties 6 et 7 présenteront respectivement les perspectives ouvertes suite à ces travaux ainsi que nos conclusions.

## 2 Les problèmes liés aux approches composants actuelles

La programmation orientée composants a sans doute été largement motivée par le succès rencontré par les composants standardisés couramment utilisés dans les domaines d'ingénierie plus classiques tels l'électronique (résistances, transistors, programmateurs, etc.) ou encore la mécanique (vis, boulons, compteurs, etc.). Depuis longtemps, ces disciplines utilisent des composants plus ou moins simples pour construire des systèmes plus complexes. De cette notion découle la définition d'un composant informatique, dont les JavaBeans (Sun Microsystems, 1997), les composants distribués de type Corba (Wang *et al.*, 2001), les EJB (Enterprise JavaBeans) (Blevins, 2001), ou encore les Services Web (Ferris et Farrel, 2003) sont les représentants les plus connus. Un composant tel que nous l'entendons peut alors être défini comme « […] une unité de composition avec des interfaces spécifiées de manière contractuelle et des dépendances contextuelles explicites. Un composant logiciel peut être déployé de manière indépendante et il est sujet à composition par des tiers »[3] (Szyperski et Pfister, 1997).

[3] « 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 » (Szyperski et Pfister, 1997).

Le fait qu'un composant ait pour vocation d'être utilisé par des tiers pose de manière implicite la question de son intégration dans l'environnement où il va être utilisé, intégration qui ne sera en général pas réalisée par son concepteur, mais potentiellement par ses utilisateurs. Il est à noter que l'approche que nous décrivons dans cet article focalise exclusivement sur les problèmes d'*intégration* au sens de Mørch, et vise à terme les utilisateurs. Dans la suite de cet article, sauf mention contraire, le terme *intégration* se rapporte donc au deuxième niveau de malléabilité décrit dans notre introduction[4].

Conformément à cette définition, et pour atteindre une telle malléabilité, il est aussi nécessaire que le composant soit en mesure d'offrir *a priori* un support correspondant aux usages envisagés par celui qui l'intègre à son environnement. Comme nous l'avons souligné ci-avant, les espoirs envers les technologies liées aux composants sont aussi grands que les problèmes qui restent à résoudre pour que ces technologies répondent pleinement aux attentes des concepteurs et des utilisateurs. Dans nos travaux, issus principalement de recherches tentant de créer une plate-forme capable de supporter le principe de co-évolution (Bourguin et Derycke, 2005), nous nous sommes donc principalement intéressés à ces deux problèmes posés par les approches composants actuelles : le problème de l'intégration des composants et le problème de leur test. Même s'ils peuvent à première vue sembler différents, nous allons montrer que ces deux problèmes sont en réalité reliés à un certain manque de sémantique au sein des technologies composants existantes.

### 2.1 Le problème de l'intégration des composants

De manière à illustrer le type d'intégration qui nous intéresse plus particulièrement dans cet article, et les manques quant aux moyens existants pour le problème général de l'intégration de composants, nous allons citer les travaux que nous réalisons dans le cadre du projet CooLDev, un environnement global et intégré de Coopération dédié au Développement de Logiciels.

#### *L'exemple du projet CooLDev*

De manière simplifiée, on peut définir CooLDev comme un environnement distribué offrant, par l'articulation d'outils contribuant à leurs réalisations, un support aux activités de ses utilisateurs. Il existe déjà de nombreux environnements de ce genre, dédiés à divers types d'activités. La plupart d'entre eux sont caractérisés par un serveur centralisé fournissant l'accès à l'application au travers d'un serveur Web, et de clients sous forme de navigateurs Web pouvant s'y connecter. On peut par exemple citer les environnements tels que SourceForge (Augustin *et al.*, 2002) aussi dédié au développement coopératif de logiciels, ou encore, dans un tout autre domaine, les environnements intégrés dédiés à l'enseignement à distance tels que Moodle (Dougiamas, 2001). Notre objectif dans cet article n'est pas d'énoncer l'ensemble des propriétés et concepts qui différencient CooLDev des plateformes existantes. Le lecteur pourra trouver de plus amples informations à ce sujet dans (Lewandowski, 2006). Nous soulignerons simplement pour l'instant le fait que, dans le cas de CooLDev, les clients ne sont pas des navigateurs Web, mais des extensions

[4] Néanmoins, rappelons que des paramètres peuvent être envoyés au composant au travers des méthodes qu'il propose pour son intégration, ce qui correspond à un certain paramétrage. On peut aussi considérer qu'un environnement intégrateur est lui-même étendu par l'ajout de composants (ou leur réorganisation). Ce mécanisme relève encore du niveau *intégration*. L'extension, au sens de Mørch, et qui correspond à la redéfinition même du code fonctionnel du composant, est ici hors de propos.

de la plateforme Eclipse, environnement largement utilisé par les développeurs du monde entier.

La particularité de ces plates-formes est de fournir une sorte de portail permettant aux utilisateurs d'accéder à un ensemble d'outils qui ont été intégrés à l'environnement. Ainsi, au travers de SourceForge ou de CooLDev, l'utilisateur peut avoir directement accès à un outil de forum, de tchat, de gestion de configuration tel CVS, de bug-tracking, etc., ces outils étant assemblés de manière à supporter l'activité de développement de logiciels. On peut considérer que chacun de ces outils est un composant supportant une (sous) activité et qui a été intégré et contextualisé pour supporter l'activité globale spécifique de développement de logiciels. Par exemple, un outil de tchat n'est *a priori* pas dédié à un domaine d'application particulier, mais, intégré à CooLDev, il est contextualisé en étant automatiquement connecté au canal de discussion lié au projet et à la communauté de l'utilisateur. Sans l'environnement global, chaque utilisateur devrait fournir pour chacun de ses outils les paramètres de connexion (identifiant, mot de passe, adresse du serveur, etc.) lui permettant de travailler avec ses collaborateurs pour chaque projet spécifique. Il devrait ainsi se connecter successivement dans le tchat, le forum, l'outil de gestion de configuration, etc. Grâce à CooLDev, il n'a plus qu'à s'identifier une fois et c'est la plate-forme qui pilote et configure chaque outil intégré de manière à fournir automatiquement aux utilisateurs un environnement de travail cohérent. De plus, l'intégration permet de gérer plus qu'une simple configuration de lancement. En effet, elle peut aussi gérer une certaine synergie entre les composants qui supportent une même activité. L'utilisateur n'est alors plus forcé de gérer lui-même et mentalement tous les liens entre ses divers outils : une partie peut être déléguée à l'environnement. Par exemple dans CooLDev, on peut imaginer que le fait d'effectuer un *« commit »* dans l'outil de gestion de configuration CVS déclenche un message automatique dans le tchat pour prévenir la communauté du changement, et modifie dynamiquement les droits des utilisateurs envers d'autres outils/composants, par exemple les droits d'écriture dans un espace partagé, de manière à laisser la main aux testeurs pour qu'ils évaluent et annotent une nouvelle version du logiciel en cours de développement.

Enfin, il est important de rappeler que la malléabilité des plateformes est une propriété fortement recherchée dans de nombreux travaux (Vicente, 2000, Mørch *et al.*, 2004, Won *et al.*, 2005), y compris dans les nôtres qui s'intéressent à supporter une certaine co-évolution entre l'environnement et ses utilisateurs. La malléabilité implique que les composants formant un support d'activité particulier ne se connaissent pas *a priori* : en d'autres termes, le tchat et l'outil de gestion de configuration n'ont certainement pas été créés par les mêmes concepteurs et ne savent pas qu'ils seront un jour utilisés dans une telle synergie. De même, CooLDev ne peut anticiper les besoins futurs de ses utilisateurs et connaître à l'avance quels composants y seront éventuellement intégrés, ces composants n'existant simplement peut-être pas encore aujourd'hui. L'enjeu principal est donc de fournir les moyens pour que ces futurs composants soient à la fois assez ouverts et bien conçus pour pouvoir être intégrés aisément avec la finesse nécessaire au support des mécanismes que nous venons d'évoquer.

### *Les moyens existants pour l'intégration*

Le problème de l'intégration de composants logiciels, même si ces derniers ne sont pas directement destinés à des plateformes applicatives intégratives comme CooLDev, est un domaine de recherche complexe. De nombreuses solutions techniques tentent d'ailleurs d'y apporter des réponses. Par exemple, les composants JavaBeans, Corba, EJB, et Services Web cités plus haut, sont conçus en vue de leur

intégration future. Y sont parfois associés des langages de composition (van der Aalst, 2003) qui permettent d'intégrer de manière fine ces composants ou services informatiques au sein d'applications. Néanmoins, ces différentes méthodes d'intégration fonctionnent suivant le même principe : il est possible de découvrir dynamiquement les objets sur l'Internet, de les instancier, de découvrir par introspection (Maes, 1987) leurs méthodes publiques ainsi que leurs éventuels canaux d'événements, et finalement de les utiliser. Si ces mécanismes sont fort intéressants pour parvenir à une intégration fine et même dynamique, ils apportent principalement une solution à la dimension technique du problème et il est à noter que ces solutions technologiques sont exclusivement destinées à des développeurs expérimentés, notamment en raison de leur complexité, de leur coût de mise en œuvre et de la spécificité des techniques employées (Heineman et Councill, 2001). De plus, il faut souligner que les moyens techniques évoqués ci-avant ne répondent pas à eux seuls à l'ensemble du problème.

Intégrer dynamiquement un composant de manière fine suppose que l'on puisse l'utiliser, mais aussi que l'on puisse comprendre *comment* l'utiliser. L'exemple de l'intégration du tchat dans CooLDev dont nous avons parlé précédemment est un exemple typique de cette problématique. CooLDev étant en partie basé sur Eclipse, le tchat correspond à un composant de type plug-in pour Eclipse, présentant entre autres les propriétés des JavaBeans. Un utilisateur de CooLDev voulant concevoir un support d'activité particulier intégrant le tchat va donc pouvoir le télécharger (par exemple sous forme d'archive « *.jar* ») et découvrir ses moyens d'intégration — grâce au mécanisme d'introspection — qui permettent de l'instancier dynamiquement, de découvrir ses méthodes et de les invoquer. Si le tchat a été conçu comme un composant ouvert, celui-ci présentera certainement un ensemble de méthodes permettant de le piloter telles que *sendMessage*, *connect*, *disconnect*, ou encore *changeUserInfo*. Même si ces noms de méthodes peuvent paraître assez explicites, il est souvent difficile de savoir qu'en faire. Pour pallier ce premier problème de sémantique, les technologies Orientées Objets (OO) fournissent quelques supports pour leur compréhension. On peut par exemple citer le WSDL (Booth et Liu, 2006), langage de description des Services Web, ou encore la Javadoc, une documentation des composants JavaBeans générée à partir de balises insérées dans le code desdits composants sous forme de commentaires. Cette documentation se présentera généralement sous une forme telle que celle présentée sur la Figure 1.

Néanmoins et même si elle est utile, ce genre de documentation décrit principalement le fonctionnement des méthodes, mais ne décrit pas ou peu la façon de les utiliser. Tout développeur a d'ailleurs déjà été confronté à des questions liées à ce problème, questions dont la principale est peut-être : « dans quel ordre ces méthodes doivent-elles être appelées par l'application intégrative pour que le composant fonctionne correctement ? » Par exemple, dans le cas de notre tchat, après avoir créé une instance du composant, un appel direct à la méthode *sendMessage* générera une erreur à l'exécution du fait que l'on aura omis de faire préalablement un appel à la méthode *connect* qui inscrit effectivement un utilisateur dans un canal du serveur sur lequel on peut envoyer des messages. Il est vrai que dans l'exemple du tchat, on peut imaginer qu'un intégrateur découvrant les méthodes *connect* et *sendMessage* comprendra certainement qu'il est nécessaire de procéder à une authentification avant d'envoyer des messages. Ceci s'explique par le fait que ce schéma d'utilisation correspond à un protocole aujourd'hui bien répandu dans les esprits, ce qui explique aussi pourquoi nous avons choisi cet exemple dans

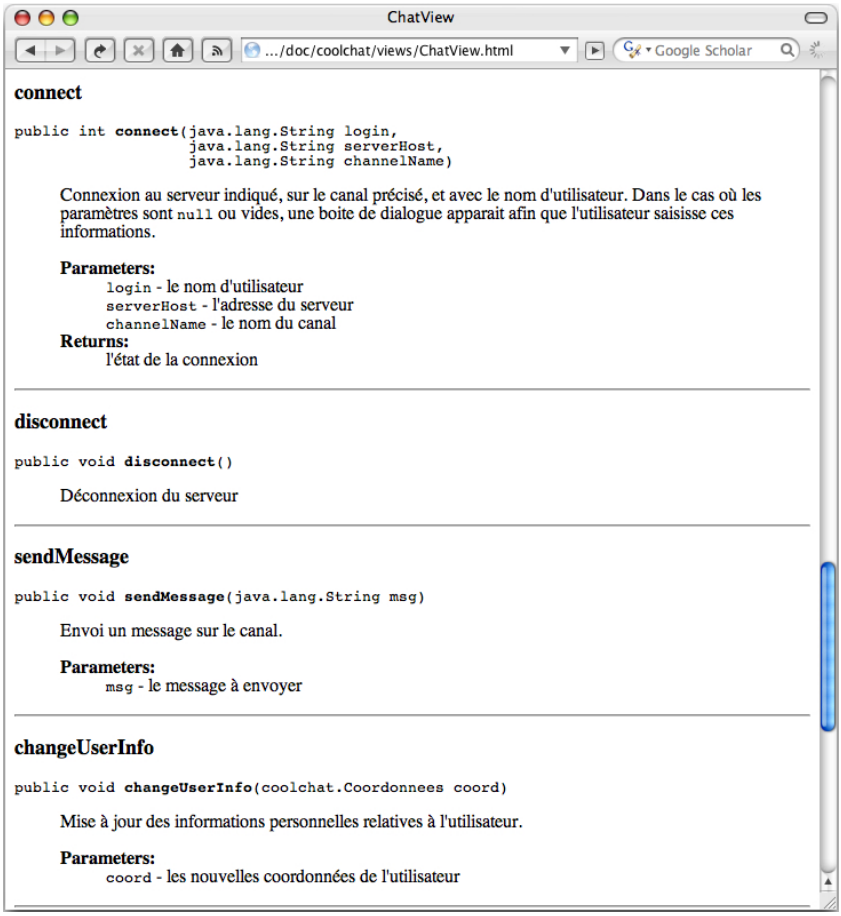



**Figure 1.** *Extrait de la Javadoc associée au composant de tchat intégrable.*

cet article. Cependant, la compréhension est moins évidente si l'on considère la méthode *changeUserInfo*. En effet, on peut dans un cas imaginer que si les données décrivant l'utilisateur, comme son pseudo ou son icône, sont stockées sur son poste, il est possible d'appeler *changeUserInfo* sans connexion préalable. Mais on peut aussi imaginer que si ces données sont stockées de manière centralisée sur le serveur, il est nécessaire d'invoquer au préalable la méthode *connect*… Du point de vue de l'utilisateur du composant, l'ambiguïté ne peut difficilement être levée sans procéder à des tests fastidieux, ou sans devoir explorer l'implémentation même du composant, ce qui n'est à la fois pas souhaitable dans une approche composant, mais aussi souvent impossible, les sources n'étant en général pas disponibles. Par extrapolation, sur des exemples plus complexes et moins stéréotypés que le tchat, possédant des noms de méthodes ayant du sens pour leur concepteur mais pas forcément pour ceux qui vont les intégrer, il apparaît que les problèmes liés à la compréhension pour une bonne intégration se font plus prégnants. Une solution largement mise en œuvre aujourd'hui sur l'Internet, et qui ne nous semble pas réellement satisfaisante, consiste à rechercher des exemples d'utilisation du composant, en épluchant le code d'une autre application qui l'utilise, ou en

procédant à une initiation à sa mise en œuvre sous forme de tutoriaux, dans le meilleur des cas...

Pour toutes ces raisons, seuls des informaticiens très motivés et passionnés sont en général capables d'intégrer au mieux la plupart des composants émergeants sur l'Internet car, procédant par l'étude de codes existants, ils doivent presque totalement reconstruire mentalement la mécanique de fonctionnement de l'outil qu'ils tentent d'intégrer. Cette problématique limite la réutilisation à des utilisateurs experts, ce qui nous semble constituer une forte limite aux technologies composants par rapport aux attentes qu'elles ont générées en terme de support à la création d'environnements malléables.

Cette analyse nous amène à penser que les difficultés rencontrées dans l'intégration de composants proviennent en grande partie d'un manque de sémantique dans les moyens proposés quant à leur documentation. Ce manque a d'ailleurs déjà été remarqué par la communauté de chercheurs et plusieurs travaux sont en cours (Kiniry, 2003, Medjahed *et al.*, 2003), visant à pallier ce problème. Cependant, et avec les modèles actuels, nous avons déjà souligné que même les informaticiens ont des difficultés à intégrer les composants pour outiller leurs applications. De ce fait, on peut remarquer et comprendre que les nouvelles propositions restent généralement à destination de développeurs expérimentés. Or, dans le cadre de nos travaux sur la co-évolution, nous avons montré qu'il serait fort bénéfique de faciliter cette intégration fine et dynamique de composant, y compris pour et par des utilisateurs moins expérimentés.

De ce point de vue, il existe des moyens qui permettent de procéder à une intégration plus facile : c'est par exemple le cas des plug-ins tels que ceux pour Photoshop (destinés à des utilisateurs qui ne sont pas *a priori* des informaticiens), pour Firefox (permettant de personnaliser simplement ce navigateur Web bien connu), ou encore pour Eclipse (OTI, 2003). Ce type de mécanisme est aussi qualifié de « hard integration » et, comme l'a démontré Mørch (1997), si cette intégration est plus accessible aux utilisateurs, elle l'est au détriment de sa souplesse d'intégration ; la spécification de la façon dont un composant sera intégré est en effet souvent directement codée dans le composant, et il est difficile de (re)définir finement ce « schéma » d'intégration autrement qu'en en modifiant le code. L'intégration fine et dynamique que nous recherchons, alors qualifiée de « soft integration », est donc toujours un problème prégnant. Cependant, un élément semble important pour guider nos travaux dans ce sens : ces utilisateurs, informaticiens ou non, ne sont pas forcément des spécialistes de la technologie mise en œuvre, mais sont guidés par la tâche qu'ils veulent accomplir.

### 2.2 Le problème du test des composants

Une autre dimension fort importante dans le fait qu'un composant soit conçu pour être utilisé par des tiers est qu'il puisse *a priori* offrir un support en adéquation avec les usages envisagés par celui qui l'intègre à son environnement. Lorsqu'un composant a été conçu et implémenté, au-delà du problème que nous venons d'évoquer quant aux moyens permettant son intégration, deux problèmes majeurs restent encore à résoudre. Le premier concerne l'évaluation du composant en lui-même et le second concerne l'évaluation de l'utilisation du composant vis-à-vis de son intégration avec d'autres composants. Même si ces deux questions sont très différentes quant à leurs contenus et leurs finalités, les techniques mises en œuvre pour y apporter des réponses restent identiques (Hilbert et Redmiles, 2000).

Déterminer comment un composant est perçu, et donc utilisé, est une question fondamentale pour permettre aux équipes de développement de confirmer ou d'infirmer leurs hypothèses de conception (type et niveau du public, attentes de

celui-ci, etc.). Cette question est fondamentale également pour des aspects d'évolution du composant dans son contexte d'utilisation. En effet, même si un composant répond parfaitement aux attentes de ses utilisateurs, ces derniers vont évoluer et donc se détacher petit à petit de l'usage prévu initialement pour le composant ; celui-ci devra être modifié (c'est-à-dire avoir une capacité d'adaptation), ou se modifier par lui-même (capacité d'adaptabilité), pour coller à nouveau aux attentes des utilisateurs. Cette question est fondamentale enfin pour des aspects de connexion du composant avec son contexte d'utilisation. Un composant est rarement utilisé tout seul, sans aucun lien avec d'autres composants. Même s'il n'évolue pas, les autres composants évoluent de leur côté et il est important de faire en sorte que ces évolutions connexes ne le remettent pas en cause.

Identifier l'usage d'un composant dans son contexte réel d'utilisation permet de savoir s'il répond aux attentes des utilisateurs, de savoir s'il est utile et utilisable, et enfin d'estimer les besoins nécessaires pour son évolution à court, moyen et long terme. Or, quelle que soit la méthodologie employée pour concevoir un composant, la façon dont celui-ci sera perçu et utilisé ne peut jamais être connue à l'avance. Pour répondre à cette question, deux grandes catégories de solutions existent, et peuvent être combinées pour de meilleurs résultats. La première consiste à interviewer les utilisateurs pour leur demander comment et pourquoi ils utilisent le composant, comment ils le perçoivent, etc. Cette solution a l'avantage de fournir beaucoup plus facilement et rapidement des informations qui seraient difficiles, voire impossibles, à obtenir à partir de traces produites par le composant. Ces traces, dont l'analyse constitue la seconde solution, peuvent être utilisées à des fins diverses telles que l'amélioration de la robustesse aux pannes, l'évaluation de l'utilisabilité, de la capacité d'intégration, etc. Elles peuvent donc être utilisées en complément des interviews, mais nécessitent cependant beaucoup de travail pour en extraire de l'information pertinente. Comme nous le verrons plus loin (cf. partie 5.2), notre travail concerne uniquement cette seconde catégorie de solution.

Les traces utilisées habituellement pour l'évaluation des composants peuvent être de natures aussi diverses que des captures vidéo par caméra (pour l'écran, mais aussi pour l'utilisateur en action, etc.), des captures sonores par micro, des captures écran par des outils dédiés, des recueils de données par des logiciels espions interceptant toutes les interactions de l'utilisateur (frappe au clavier, déplacement de la souris, etc.), des fichiers « logs » produits automatiquement par les systèmes d'exploitation ou des serveurs, etc. Notre travail se rapproche de cette dernière catégorie puisque nous produisons des traces informatiques, mais d'un niveau d'abstraction plus élevé que les fichiers « logs » des serveurs.

Concernant cette production de traces, trois solutions sont possibles : (1) insérer manuellement dans le code source original des méthodes permettant de produire les traces, (2) utiliser les fichiers « logs » des serveurs, ou (3) séparer le code produisant les traces du code du composant et les faire fonctionner simultanément. Comme nous voulons rester maîtres des traces produites, la solution des fichiers « logs » est à bannir. Étant donné que notre postulat de base est de ne pas modifier le code initial des composants, nous avons opté pour la solution (3), grâce aux apports de la programmation orientée aspects (POA) (Filman *et al.*, 2005).

L'utilisation de la POA pour la production de traces (Tarby, 2006) a un énorme avantage sur toutes les autres techniques utilisées. En effet, la POA repose sur le principe de la séparation des préoccupations (« Separation of Concerns » en anglais, SoC). Avec un codage en POA, le code d'un composant est divisé en blocs indépendants (classes, packages…) ayant chacun des rôles bien distincts. Par exemple, tout ce qui a trait à la sécurité peut être mis à part, et venir se greffer si

nécessaire sur le reste du code. Ce principe de SoC allège énormément l'écriture du code, mais permet surtout de « greffer » du code suivant les besoins. C'est ainsi que, dans notre travail, le code associé à la production des traces est totalement extérieur au code métier initial. Si l'application doit être tracée, ce code de traçage sera alors greffé automatiquement au code métier ; si l'application ne doit pas être tracée, on exécute uniquement le code métier. Dans les deux cas, le code métier n'a pas besoin d'être modifié par les développeurs, contrairement aux autres techniques habituelles qui consistent à y ajouter explicitement des ordres de trace.

Concernant les traces par la POA, nous utilisons AspectJ (Kiczales *et al.*, 2001) et nous avons développé plusieurs applications prototypes d'injection de mécanismes de traces dans du code métier. La dernière application créée est un plug-in Eclipse (cf. Figure 2) qui permet de choisir directement, parmi les méthodes publiques du code du composant, celles pour lesquelles des traces devront être générées, en indiquant également le format que ces traces devront utiliser. Ces choix étant faits, le plug-in génère automatiquement le code des aspects associés à cette production de traces. Si l'application doit être tracée, il suffit d'exécuter le composant avec le code des aspects. Si l'application ne doit pas être tracée, il suffit de supprimer le paquetage contenant les aspects pour que l'application fonctionne normalement, sans générer de trace. Le but de cet article n'est pas d'exposer en détail cette approche, mais le lecteur pourra trouver de plus amples descriptions spécifiques à cette facette de nos travaux dans (Tarby, 2006).

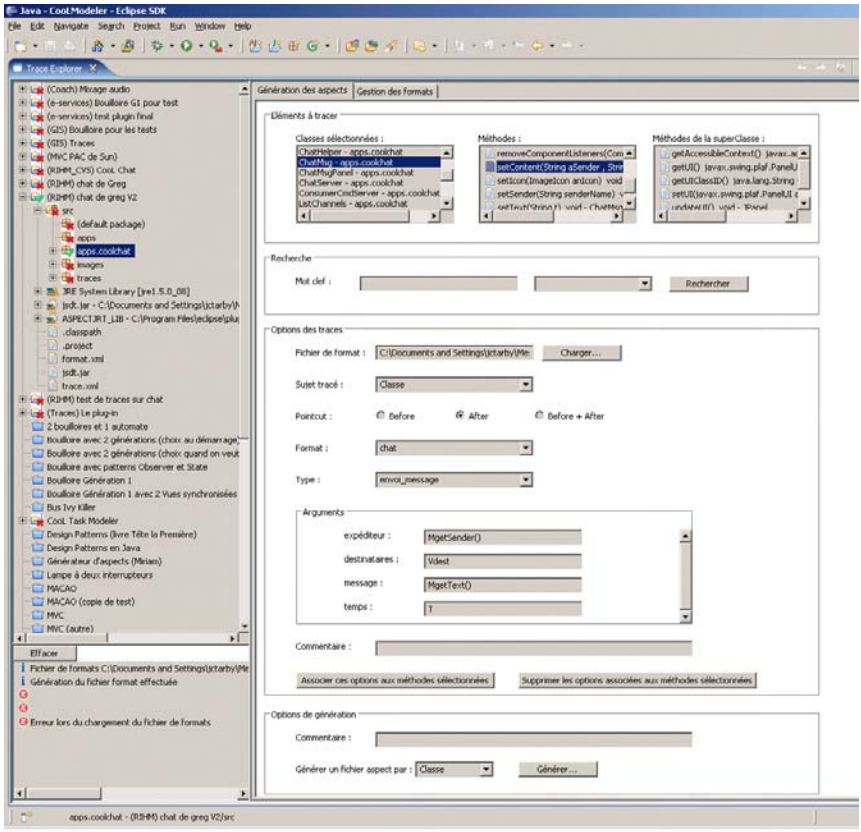



**Figure 2.** *Plug-in de génération des aspects de trace*

Les traces produites par les aspects sont actuellement écrites en XML, mais d'autres formats d'écriture sont tout à fait possibles. Une fois produites, les traces doivent être analysées pour livrer les secrets attenants à l'utilisation du composant concerné, et même si l'analyse des traces que nous faisons pour le moment reste assez simple, elle permet toutefois de produire des vues d'un niveau sémantique plus élevé que de simples « logs ».

Cependant, le problème est que les traces sont produites à partir des méthodes du composant et que, sans connaître la sémantique du composant, il est très difficile de savoir quelles méthodes doivent être tracées, et quelles traces peuvent ou doivent être générées. Par exemple, dans le cas de l'envoi de message dans un tchat, faut-il tracer la méthode *sendMessage()* ou *sendText()* ? De manière à répondre à cette question, il est nécessaire de fournir un effort très important pour déterminer tout d'abord quelles méthodes tracer, puis quelles données récupérer dans les traces, et enfin pour extraire un niveau d'abstraction suffisant à partir de ces traces brutes (Settouti *et al.*, 2006). Une fois encore, ces difficultés se révèlent en partie induites par un manque de sémantique au sein des composants. Le traçage de méthodes par la POA offre un moyen technique très intéressant pour procéder aux tests nécessaires, mais le niveau d'abstraction proposé par les méthodes actuelles et leur documentation diverge des besoins des acteurs que sont par exemple les ergonomes. Ce que veut tracer l'ergonome n'est en réalité pas des appels de méthodes au sein d'un composant, mais un ensemble de tâches effectuées par l'utilisateur et supportées par ces méthodes.

## 3 Vers une solution au manque de sémantique : les COTs (Composants Orientés Tâche)

### 3.1 Un point de vue particulier sur les composants

Du fait de nos travaux depuis de nombreuses années dans le domaine des IHM (Interactions Homme-Machine) et plus particulièrement du TCAO (Travail Coopératif Assisté par Ordinateur), nous avons été amenés à considérer les composants/outils du point de vue particulier de la tâche utilisateur. De ce point de vue, chaque composant peut être considéré comme un support à une *tâche utilisateur générique* plus ou moins complexe. Ainsi, on peut dire qu'un outil comme Firefox supporte la tâche générique d'exploration du Web, cette tâche étant elle-même complexe puisque composée de (sous-)tâches comme la gestion de signets ou encore la recherche de chaînes de caractères dans une page Web. Une instance de cette tâche générique correspond par exemple à l'activité de collecte d'information pour la rédaction d'un article dans la revue RIHM. De la même manière, la tâche générique supportée par l'outil de tchat est la discussion synchrone à distance qui supportera par exemple une activité particulière de débat dans la conception d'un logiciel au sein de CooLDev. Il nous apparaît alors que contextualiser un composant signifie inscrire sa tâche générique dans le contexte d'une tâche plus globale : la tâche supportée par l'environnement intégrateur. Cette contextualisation se traduira alors par la création de liens entre la tâche intégrative et celle du composant intégré. De la même manière, définir ce qui doit être tracé sur un composant correspond à désigner les (sous-) tâches supportées par le composant et qui doivent être espionnées. D'une manière plus générale, tester un composant correspond à vérifier l'adéquation entre la tâche générique pour laquelle il a été créé et celle réalisée par les utilisateurs.

### 3.2 Le modèle de tâches en tant que chaînon manquant

L'utilisation des tâches prend une place de plus en plus importante dans le Génie Logiciel, principalement dans le domaine des IHM. Nombreux sont les travaux dans ce domaine qui portent sur les moyens visant à exprimer les tâches des utilisateurs. Cette approche orientée tâches est généralement utilisée dans les phases amont ou aval d'un processus de conception (Clerckx *et al.*, 2004, Delotte *et al.*, 2004, Lu *et al.*, 2003, Luyten *et al.*, 2003). Toutefois, il est intéressant de constater que si ces méthodes proposent de commencer la conception d'un composant par une modélisation de la tâche, cette démarche s'efface progressivement pour finalement être absorbée par une démarche de conception Orientée Objet (OO) inspirée par l'ingénierie informatique. Cette approche de conception classique tend à transformer les modèles de tâches en modèles d'objets, d'où découle implicitement la structuration en classes du composant (cf. Figure 3, partie supérieure). Le modèle de tâches de base se retrouve noyé, inscrit de manière implicite dans la complexité du code produit. En effet, les approches orientées tâches ne sont pas ou peu utilisées *pendant* le cycle de conception et de développement, c'est-à-dire après le recueil des besoins et leur analyse.

Par ailleurs, les outils de conception actuels, principalement ceux reconnus en Génie Logiciel — qu'il s'agisse d'Environnements de Développement Intégrés (EDI) dédiés à supporter les phases de programmation/test, ou des Ateliers de Génie Logiciel (AGL) comme Rational Rose, tendant à supporter l'intégralité du processus de production — n'intègrent absolument pas les approches orientées tâches. UML lui-même, n'intègre pas du tout la notion de tâche telle que nous la connaissons en IHM. Par exemple, la différence de notion entre tâche interactive, tâche système et tâche manuelle n'existe pas dans UML. Pourtant, nombreux sont les travaux qui ont tenté de remédier à ce manque (Bruins, 1998, Nunes *et al.*, 2000, Pinheiro da Silva, 2002, Ruault, 2002, Scogings and Phillips, 2004), profitant de similitudes entre certains concepts d'UML et des approches orientées tâches pour créer des « traducteurs ». Malheureusement, on ne peut que constater l'absence de résultats concrets de ces travaux. Même si certains d'entre eux ont voulu faire évoluer UML en demandant à y incorporer les notions associées aux tâches, la version actuelle d'UML souffre toujours cruellement du manque de ces notions de tâches, utilisateurs, rôles, buts, etc.

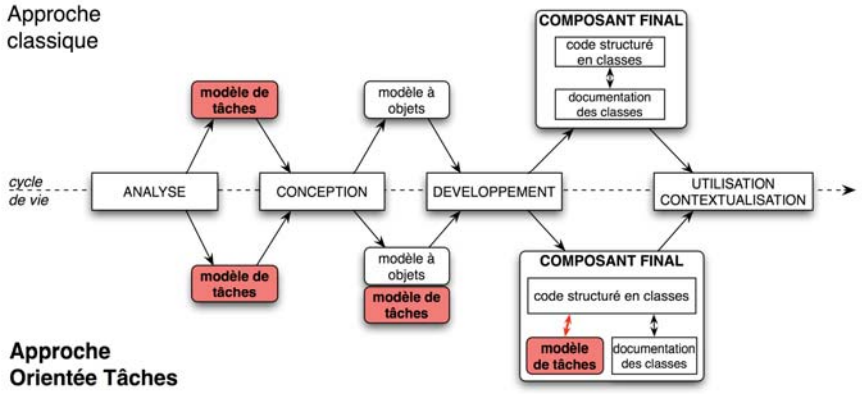



**Figure 3.** *Schématisation de l'approche de conception classique, et de l'approche Orientée Tâches qui tend à préserver le modèle de tâches tout au long du processus de développement.*

On constate donc que le modèle de tâches dont on tire bénéfice dans les phases amont du processus se dilue progressivement dans la conception et l'implémentation, et n'est plus accessible de manière explicite au sein du composant livré. Même si le code produit reflète les tâches initialement identifiées, il est très difficile d'en déduire le modèle qui en est à l'origine. Ceci rejoint le fait évoqué précédemment concernant la nécessité pour les intégrateurs d'entrer dans le code du composant pour en extraire la logique de fonctionnement, et surtout sa logique d'utilisation (Richard, 1983). En d'autres termes, l'utilisateur du composant doit presque complètement reconstruire mentalement le modèle de tâches sous-jacent pour le rendre à nouveau explicite, que ce soit en vue de son intégration, ou en vue de son test par la génération et l'analyse de traces telles que nous les avons décrites.

C'est pourquoi, de manière à faciliter la contextualisation et le traçage des composants logiciels, tous deux fortement liés à leurs tâches sous-jacentes, nous proposons de mieux utiliser le modèle de tâches des composants, sorte de chaînon manquant qui disparaît généralement entre la phase de conception et le code produit. Nous avons choisi le terme de Composants Orientés Tâches ou COT pour désigner ce nouveau type de composants logiciels. Comme le décrit la Figure 3 (partie basse), le principe d'un COT est qu'il intègre le modèle de tâches qui en décrit la logique d'utilisation, ceci en plus de sa documentation classique Orientée Objet. Dans cette démarche, certaines parties du code fonctionnel (représenté par les méthodes du COT) sont reliées aux tâches du modèle qui en sont à l'origine.

L'intérêt de disposer de tels composants, embarquant leur modèle de tâches et les liens qui les unissent à leur code, est multiple. D'une part, la contextualisation d'un COT pourra s'effectuer grâce et au travers de son modèle de tâches. Comme nous l'avons souligné dans la partie 2.1, les méthodes existantes pour l'intégration fine et dynamique de composants révèlent un problème de sémantique et de niveau d'abstraction. Le fait de disposer du modèle de tâches du composant permet de combler — au moins en partie — ces lacunes. En effet, en conservant des liens explicites entre ses méthodes et son modèle de tâches, un composant OT informe sur ses *« points d'entrée »*, c'est-à-dire les (sous-)tâches directement accessibles depuis un environnement intégrateur. Pour contextualiser un composant OT, il « suffira » donc de spécifier les (sous-)tâches particulières du composant que l'environnement global déclenchera (*via* des appels aux méthodes fonctionnelles auxquelles ces (sous) tâches font référence). Ainsi, l'intégration ne se fait plus par introspection sur une liste de méthodes : elle est rendue accessible au travers du modèle de tâches, qui relève le niveau d'abstraction en l'éloignant du code et y ajoute la sémantique propre au modèle de tâches. De la même manière, en conjuguant le traçage par aspects et les COTs, l'ergonome peut spécifier et générer des aspects produisant des traces d'usages, non plus en découvrant et sélectionnant les méthodes du composant, mais directement en sélectionnant les tâches à tracer.

Enfin, il est à noter que les modèles de tâches — lorsqu'ils sont utilisés — servent déjà, en tant qu'objet partagé, à une meilleure communication entre les différents acteurs (y compris les futurs utilisateurs) du processus complexe de conception. L'approche proposée par les COTs devrait donc aussi pouvoir servir de support à une meilleure collaboration entre les différents acteurs de ce processus.

## 4 La création de Composants Orientés Tâches

### 4.1 Les démarches possibles

Ainsi que nous l'avons évoqué, le bénéfice potentiel de l'utilisation des modèles de tâches dans l'approche COT est présent à tous les stades du cycle de vie

du composant logiciel, depuis sa conception, puis son développement, et jusqu'à son intégration et son évaluation. Pour obtenir des COTs, embarquant leur propre modèle de tâches, plusieurs démarches sont envisageables. D'une part, une approche qu'on peut qualifier d'ascendante, dans le cas où on dispose d'un composant « classique », qu'on a par exemple téléchargé sur l'Internet, et qui n'a pas été développé dans cette optique OT. On peut alors tout à fait imaginer qu'un spécialiste décide de reconstruire *a posteriori* le modèle de tâches de ce composant, et de le lier avec le code de celui-ci en spécifiant leurs liens. Le nouveau composant disponible contiendra alors son modèle de tâches, et il sera plus compréhensible. Cette approche, techniquement réalisable, reste très difficile à mettre en œuvre concrètement. En effet, d'une part, reconstruire le modèle de tâches du composant n'est pas une affaire triviale. D'autre part, il s'agit de trouver les points d'ancrage dans le code qui permettront de faire le lien avec le modèle de tâches ainsi reconstruit. Or cette dernière étape est très difficile dans la plupart des cas étant donné que le modèle objet noie réellement le modèle de tâches. Nous espérons donc un jour pouvoir mettre en œuvre cette démarche ascendante tendant à réutiliser et « COTiser » des composants existants, mais pour ce faire, il nous faut d'abord affiner et valider les liens que nous voulons tisser entre l'approche OO classique et l'approche OT développée dans cet article.

Du fait de ces problèmes concrets, l'approche que nous avons choisie pour valider notre démarche consiste, dans un premier temps, à mettre en œuvre une approche descendante. Nous proposons de fournir une démarche et des moyens de conception permettant de construire et de développer des COTs. Ce processus contient plusieurs étapes dont les cinq plus marquantes sont les suivantes :

- Idéalement, comme nous l'avons précisé dans la partie 3.2, le développement est précédé par la création d'un modèle de tâches décrivant notamment le comportement attendu du composant. C'est ce modèle de tâches qui sera le point de départ de la création d'un COT. Un arbre de tâches est donc construit par un ergonome, éventuellement en collaboration avec d'autres acteurs du processus de développement comme les informaticiens, ou les futurs utilisateurs. Les tâches peuvent être commentées de manière à ce que le modèle soit bien compris par tous et propose au final une représentation partagée du COT. En particulier, l'ergonome anticipant sur ses futurs travaux de tests du composant pourra détailler les tâches qui sont susceptibles d'être tracées, en spécifiant par exemple ce qui caractérise de son point de vue le début et/ou la fin d'une tâche, ce qu'il aimerait récupérer dans la trace, qui a démarré la tâche (en cas d'outil partagé), etc.
- Le modèle de tâches est alors étendu par l'informaticien en spécifiant les méthodes clés qui seront implémentées — ou qui feront appel à des méthodes existantes — en lien direct avec les tâches et commentaires mis en avant dans le modèle créé précédemment. Comme nous le verrons plus loin, certaines de ces méthodes seront créées pour être utiles du point de vue des moyens fournis par le composant pour sa future éventuelle intégration dans des environnements tiers, d'autres le seront pour répondre au besoin de traçage des tâches du composant. En accord avec le principe d'externalisation des moyens de génération des traces d'usage énoncés plus haut, ces méthodes ne seront pas elles-mêmes génératrices de traces, mais sont pensées pour être tracées dans l'approche de type Programmation Orientée Aspects exposée dans la partie 2.2.
- Des squelettes d'implémentation et de documentation du composant peuvent alors être directement déduits (générés de manière automatique) du modèle de tâches étendu par les acteurs dans les étapes décrites ci-avant.

- Les squelettes sont implémentés (dans une approche classique OO), de manière à respecter les spécifications du modèle de tâches étendu.
- Le composant est livré avec sa documentation habituelle dans une approche OO (comme les fichiers constituant la Javadoc, pour un composant de type JavaBean), mais aussi avec son modèle hybride liant tâche utilisateur et code du composant, formant la documentation d'un COT. Ce modèle se présente sous la forme d'un fichier XML décrivant l'arbre de tâches du composant ainsi que ses liens avec son code fonctionnel. Pour exploiter cette nouvelle documentation, nous proposerons, comme nous le verrons plus loin, divers outils d'introspection et de manipulation des COTs.

### 4.2 Un modèle aux multiples facettes

Dans la partie 3.2, nous avons évoqué dans quelle mesure les COTs, s'appuyant sur des modèles hybrides à la fois OO et OT, peuvent répondre au manque de sémantique commun aux problèmes de l'intégration et du traçage. Nous détaillerons plus amplement ces résultats dans la partie 5. Cependant, il est important de noter dès à présent que l'intégration et le traçage correspondent à des activités différentes, ce qui nous a amené à identifier deux types de méthodes qui apparaissent dans le modèle d'un COT : les méthodes d'intégration et les méthodes de traçage. Cette scission provient du fait que les besoins et la finalité de ces deux activités diffèrent.

L'activité de contrôle d'un composant dans le cadre de son intégration nécessite un point d'accès permettant à un environnement tiers de communiquer avec le composant : une instance de COT créée fournit une référence avec laquelle l'application intégrative interagit. C'est cet objet « façade », ou *wrapper*, qui propose un ensemble de méthodes permettant à une autre application informatique de piloter le composant en remplaçant ou « simulant » l'utilisateur. Par exemple, les méthodes du *wrapper* peuvent permettre de *shunter* ou de *piloter* certaines tâches utilisateur définies dans le modèle du COT : dans notre exemple de tchat, un appel à *connect* remplace le fait que l'utilisateur doive remplir une boite de dialogue contenant ses paramètres de connexion ; c'est CooLDev qui effectue pour son utilisateur la connexion du tchat au bon serveur, ceci en utilisant les informations qu'il possède à son sujet et sur la tâche globale de développement logiciel à laquelle participe le tchat. Ces méthodes particulières sont généralement pensées dans le but exclusif de l'intégration et du pilotage externe de l'application. Leur implémentation appelle d'autres méthodes internes au composant. Les méthodes du *wrapper* ne sont appelées que lorsque le composant est intégré à un environnement tiers. Elles ne correspondent pas directement à l'activité du composant, mais sont exclusivement mises en œuvre dans l'interaction entre le composant et son environnement intégrateur. Cette technique est celle utilisée classiquement par les développeurs et proposée par les diverses technologies composants standardisées déjà citées plus haut. Par exemple, dans les services Web, les fichiers WSDL (Web Service Description Language) présentent les méthodes qui peuvent être appelées sur la référence d'objet « façade » fournie par le serveur.

De ce fait, nous avons appliqué cette même technique éprouvée pour spécifier le squelette d'implémentation d'un COT du point de vue de ses méthodes d'intégration. La solution technique consiste à générer le *wrapper* à partir du modèle de tâches étendu, sous forme d'une classe supplémentaire qui servira d'intermédiaire entre le modèle de tâches et le code du composant. Ce principe est décrit sur la Figure 4. La partie gauche représente le modèle de tâches, élaboré par l'ergonome, et sur la base duquel l'informaticien va concevoir l'architecture du composant. Cette réflexion va le conduire à étendre le modèle de tâches en y ajoutant les méthodes

évoquées précédemment, en vue d'une potentielle intégration. C'est à partir de ces méthodes que le *wrapper* (au centre de la figure) est généré. Liant code et modèle de tâches, le *wrapper* est une classe contenant l'ensemble de ces méthodes. Le développeur réalise ensuite l'implémentation des méthodes de ce *wrapper* en y inscrivant les « bons » appels aux méthodes du code (partie droite) de façon à réaliser effectivement la tâche qui leur correspond[5]. Cette classe pourra ensuite être utilisée pour la contextualisation du composant dans l'environnement global.

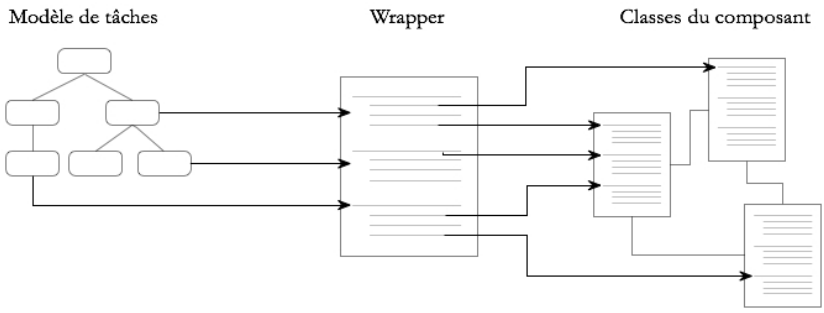



**Figure 4.** *Principe de spécification des liens entre le modèle de tâches et le code d'un composant au travers d'un wrapper en vue de sa future intégration.*

L'activité de traçage, quant à elle, ne s'intéresse pas directement aux interactions entre le composant et l'environnement qui l'intègre, mais s'intéresse principalement aux interactions entre le composant et son utilisateur final. Cette activité ne peut donc être observée au travers du *wrapper*. Les méthodes correspondant aux tâches que l'on désire tracer sont alors cette fois les méthodes internes du composant. Le modèle de tâches est donc dans ce cas utilisé pour définir et générer un autre squelette d'implémentation qui souligne des mécanismes directement impliqués dans la vie du composant, qu'il soit piloté par une application intégrative ou non.

Voilà pourquoi, dans le modèle de tâches étendu d'un COT, on pourra trouver différentes facettes, un point de vue « intégration » et un point de vue « traçage ». Chaque tâche du modèle d'un COT peut éventuellement présenter à la fois des méthodes d'intégration qui soulignent la possibilité de contrôler le COT dans le cadre de son intégration, ainsi que des méthodes de traçage permettant de tracer sa réalisation. Nous allons maintenant donner un exemple de création d'un COT au travers de la présentation de STOrM, l'outil que nous avons créé pour supporter la démarche que nous venons de présenter.

### 4.3 STOrM : un outil de conception de COT

***Un modeleur de tâches étendues***

Le processus de développement d'un COT étant intimement lié à la définition de son modèle de tâches étendu, nous avons développé un modeleur supportant cette activité, l'outil STOrM (Simple Task Oriented Modeler), dont une instance est représentée dans la Figure 5.

[5] Dans le cas d'une approche ascendante, tendant à « COTiser » un composant existant, cette étape se traduira sans doute par le fait d'associer aux méthodes du wrapper les méthodes existantes propres au composant et qui assurent déjà son fonctionnement.

Cet outil, fondé sur GEF (Grafical Editing Framework), a été implémenté en tant que plug-in Eclipse pour plusieurs raisons qu'il nous semble intéressant de souligner. Tout d'abord, associé à divers plug-ins comme le JDT (Java Development Tools), Eclipse est reconnu comme un très bon Environnement de Développement Intégré, largement utilisé par les développeurs. Le fait de fournir un plug-in Eclipse permettant de créer les modèles de tâches d'applications informatiques nous apparaît comme un premier pas vers l'intégration effective des modèles de tâches au sein des outils et donc du processus de développement de logiciels. D'autre part, nous avons déjà souligné que le développement d'un COT résulte d'une collaboration étroite entre ergonome, développeur, etc. Or, l'environnement CooLDev a pour but de mieux supporter la collaboration au sein des équipes de développement et est lui-même bâti sur Eclipse. Notre objectif est donc que STOrM soit utilisé au sein des activités coopératives de développement supportées par CooLDev. La convergence de ces éléments démontre que nous sommes animés par une vision globale, qui dépasse le cadre de cet article, et dans laquelle les outils de développement et d'utilisation des COTs sont destinés à mieux supporter la collaboration et l'interdisciplinarité par et pour une nouvelle conception logicielle.

Les concepts qui ont été mis en œuvre pour créer et représenter les modèles de tâches sont issus de CTTE (Mori *et al.*, 2002) et K-Made (Baron *et al.*, 2006). Ces deux outils, comme d'autres tels que Tamot (Lu *et al.*, 2003) ou Euterpe (van Welie *et al.*, 1998), existent déjà et il est légitime de se demander pourquoi nous avons choisi de construire un nouveau modeleur. Une partie de la réponse a été énoncée ci-avant : les outils existants ne répondent pas aux besoins de notre plateforme globale. De plus, nous n'avions besoin que de certaines des fonctionnalités qu'ils offrent, tout en ayant besoin d'en ajouter d'autres spécifiques à notre démarche. En effet, ces outils de modélisation de tâches ont des buts qui sont éloignés des nôtres. Généralement, ils sont utilisés pour réaliser des modélisations à gros grain, ou pour faire de la simulation des modèles de tâches, et éventuellement pour générer du code (souvent celui de l'interface homme-machine).

Pour notre modeleur, nous avions besoin d'une part d'un outil de modélisation simple, c'est-à-dire ne nécessitant pas toutes les fonctionnalités habituelles des modeleurs de tâches, et d'autre part nous voulions générer les squelettes du COT et non du code pour l'interface homme-machine. Enfin, la dernière raison concerne le formalisme mis en oeuvre. Nous avons opté pour une fusion de CTTE (Paterno, 2004) et de K-Made, en reprenant la vue arborescente de CTTE, les icônes de K-Made, et en combinant les opérateurs temporels de CTTE et K-Made. Ces choix sont issus de plusieurs années de pratique des outils de modélisation avec des publics très divers tels que des informaticiens (étudiants, enseignants, développeurs), de simples utilisateurs de l'informatique, et enfin des ergonomes. Puisque nous voulions un outil permettant de créer librement des modèles de tâches — avec des tâches « isolées », c'est-à-dire sans lien de parenté, et éventuellement sans type, ce qui n'est pas possible dans certains des outils évoqués ci-avant — et que nous voulions ajouter aux modèles de tâches habituels des extensions liées à notre travail, nous avons choisi de nous inspirer d'un certain nombre de points forts des outils existants pour créer le nôtre.

C'est ainsi que nous avons intégré par exemple l'opérateur temporel « inconnu » et la tâche « inconnue » de K-Made, gardé les opérateurs proposés par CTT, etc., pour assurer une grande liberté de conception. Par ailleurs, il est prévu que les opérateurs temporels puissent être affichés en français, en anglais, ou en mode graphique. Enfin, force est de constater qu'il n'existe toujours pas aujourd'hui un consensus autour des modèles de tâches qui permettrait d'avoir un outil universel.

Même si les concepts sous-jacents à ces modèles sont en majorité les mêmes, beaucoup de détails les éloignent encore trop. Par exemple, le détail d'une tâche est très différent dans CTTE, dans Tamot et dans K-Made, et certains termes (« abstrait » par exemple) n'ont pas du tout la même définition dans ces mêmes outils.

***Un exemple de tchat en tant que COT***

L'exemple que nous allons utiliser pour illustrer l'usage de STOrM dans la création de COTs est encore une fois celui du tchat. Comme indiqué dans la partie 4.1, STOrM est utilisé par l'ergonome pour créer le modèle de tâches du futur COT — celui du tchat est partiellement représenté sur la Figure 5. Sur cette base, le développeur va pouvoir augmenter le modèle de tâches en y inscrivant les premières pierres qui vont constituer le fondement du code fonctionnel du composant. La première extension (première dans notre exemple, l'ordre n'étant pas important dans la démarche) se situe au niveau de la facette d'intégration du composant. Le développeur doit, très tôt dans le processus de développement, envisager le fait que le composant développé sera potentiellement utilisé dans le cadre d'un environnement global intégrateur comme CooLDev. Il s'agit donc de *penser* la conception du composant en termes d'intégration et de réutilisation. C'est une étape essentielle pour toutes les technologies composants puisqu'elle définit les moyens futurs permettant l'intégration plus ou moins fine ou contrôlée du composant. Nous avons déjà évoqué dans la partie 2.1 que les composants ou outils qui n'ont pas été développés dans ce sens sont très difficilement intégrables dans un environnement tiers, notamment parce qu'ils ne proposent pas de méthode accessible permettant d'adapter leur comportement au contexte d'usage. Il est donc nécessaire de prendre en compte cette dimension dès la conception.

Comme nous l'avons déjà souligné, les méthodes d'intégration vont permettre à l'environnement intégrateur du COT de piloter ce dernier au travers des tâches qu'il supporte. C'est donc sur la base du modèle de tâches élaboré par l'ergonome que le développeur peut s'interroger sur les tâches candidates à ce mécanisme. Dans une approche collaborative entre ergonome et développeur, il se peut aussi que certaines tâches soient ajoutées dans le modèle en étant mises en avant du fait qu'elles servent à une meilleure intégration du composant. STOrM propose de supporter cette phase d'extension en fournissant au développeur la possibilité d'enrichir le modèle de tâches en y indiquant la signature des méthodes publiques d'intégration qui rendront accessibles certaines tâches pour un environnement tiers. Ces méthodes sont pointées par ❶ dans l'exemple présenté sur la Figure 5.

C'est ainsi que dans le tchat, et pour permettre la mise en œuvre de synergies telles que nous les avons évoquées dans la partie 2.1, le développeur a par exemple jugé pertinent de rendre accessible (et paramétrable) la tâche de connexion, permettant ainsi à un environnement tiers réalisant l'intégration de configurer automatiquement le composant en se substituant à l'utilisateur dans la saisie des informations de connexion. Ceci est caractérisé par la présence de la méthode *connect*. Dans le cas où les paramètres sont « remplis » par l'environnement intégrateur, la tâche *Se connecter au tchat* est directement réalisée. Si les paramètres sont nuls, une boîte de dialogue sera proposée à l'utilisateur. Enfin, par cette extension, le développeur, associé à l'ergonome, indique que la méthode *connect* doit être la première méthode appelée suite à l'instanciation du tchat dans son contexte cible.

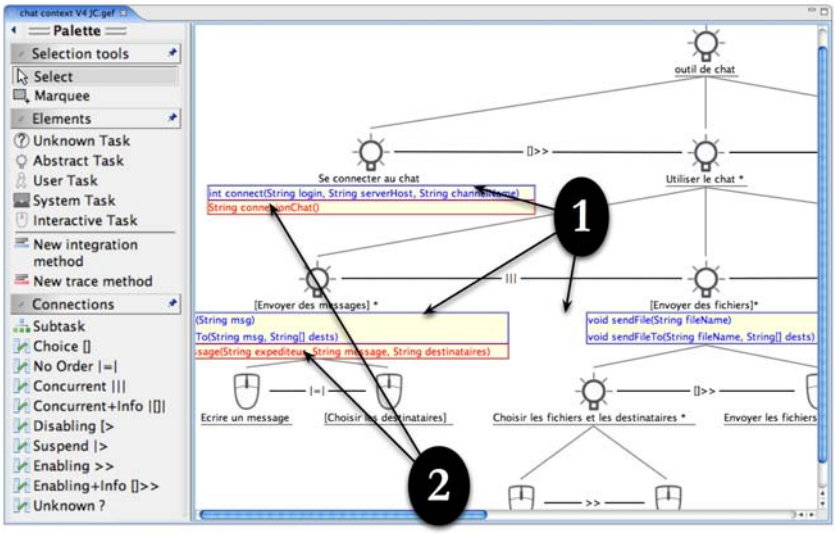



**Figure 5.** *Le modèle de tâches du tchat, étendu avec les méthodes susceptibles d'être utilisées pour la future contextualisation du composant (1), ainsi qu'avec les méthodes internes du composant qui serviront pour son traçage (2).*

La deuxième contribution du développeur se situe au niveau du traçage du composant OT. Toujours grâce à STOrM, l'ergonome a pu ajouter des commentaires textuels liés aux tâches du modèle. Ces commentaires ne sont pas représentés dans la Figure 5 par souci de lisibilité. Ils sont aujourd'hui affichés dans une fenêtre connexe mise à jour en fonction de la tâche sélectionnée. Si l'ergonome l'a jugé utile, ses commentaires expliquent ce qui lui semble pertinent d'éventuellement tracer par rapport à telle ou telle tâche. Ainsi, dans notre exemple, pour la tâche *Se connecter au tchat*, l'ergonome a précisé « j'aimerais savoir quel profil s'est connecté, et à quelle heure »[6] ; pour la tâche *Envoyer des messages*, celui-ci a ajouté « j'aimerais savoir de façon globale qui a envoyé un message, quel est le message et à qui il est destiné ». Ces informations sont utiles au développeur et le guident dans le processus de conception en lui précisant les fonctionnalités attendues pour que l'architecture du composant reflète bien le modèle de tâches.

Notre outil lui permet d'indiquer sur le modèle de tâches des méthodes clés qui interviennent ainsi dans la réalisation de chaque tâche potentiellement traçable. Ces méthodes sont signalées par ❷ sur la Figure 5. Rappelons que contrairement aux méthodes évoquées précédemment et destinées à la génération du *wrapper* d'intégration, il s'agit dans cette étape d'identifier les méthodes du code fonctionnel interne du composant. Ainsi, dans le modèle de tâches du tchat, le développeur a identifié la méthode *envoiMessage* comme la partie du code fonctionnel du COT réalisant la tâche *Envoyer des messages*. Cela signifie que cette méthode interne participe à la réalisation de la tâche *Envoyer des messages*. De cette manière, le futur traçage du COT par l'ergonome sera facilité : le modèle de tâches désigne directement les méthodes impliquées par l'envoi de message ; ainsi, le code du composant n'a plus à être exploré et étudié pour identifier les méthodes correspondant à cette tâche et devant être tracées.

[6] De telles traces sont bien évidemment obtenues après signature d'un accord avec les participants, même si seuls des pseudos sont utilisés (au lieu des noms réels).

***Un générateur de squelettes d'implémentation à partir du modèle de tâches***

Une fois ce modèle enrichi par le développeur, STOrM offre la possibilité de générer les squelettes d'implémentation correspondants. En particulier, nous générons à la demande une classe *Wrapper* (en Java) qui sera dédiée à l'intégration future du COT et qui offrira les *« points d'entrée »* proposés par le composant (cf. partie 4.2). De la même manière, le squelette d'implémentation du cœur du composant et correspondant aux méthodes traçables du COT peut être généré. Cette génération automatique est réalisée en utilisant le JDT (Java Development Tools), un ensemble de plug-ins pour Eclipse transformant ce dernier en Environnement de Développement Intégré dédié à la programmation Java. Cet ensemble de plug-ins propose divers outils permettant de générer et de manipuler du code Java. C'est en s'appuyant sur ces derniers que nous avons développé ces fonctionnalités, qui permettent par exemple d'obtenir automatiquement, pour le modèle de tâches décrit auparavant, la classe *ChatIntegrationWrapper* représentée sur la Figure 6. Le squelette de cette classe est généré à partir des méthodes d'intégration que nous venons de décrire et qui viennent étendre le modèle de tâches. Un embryon de Javadoc associé à chacune de ces méthodes est également généré, et reprend certaines informations comme la tâche associée à chaque méthode, ainsi que la description qu'a faite l'ergonome de cette tâche. Ce squelette constitue un élément de base que le développeur n'aura « qu'à » compléter en implémentant le corps de chaque méthode — ou, dans une démarche ascendante non traitée ici, en y connectant un composant fonctionnel existant en réutilisant certaines de ses méthodes — et en complétant la Javadoc associée selon son point de vue.

ChatIntegrationWrapper.java

```
package chatclient;

public class ChatIntegrationWrapper {

    /**
     * Un appel à cette méthode permet de réaliser la tâche "Se connecter au chat"
     *
     * Commentaire: Avant de chater, vous devez vous connecter avec un login valide, c-a-d non utilisé.
     *              Cette phase est tout à fait classique, et ne demande pas à être détaillée. Si l'utilisateur se
     *
     * @param login         le nom d'utilisateur
     * @param serverHost    l'adresse du serveur
     * @param channelName   le nom du canal
     *
     * @return l'état de la connexion
     */
    public static int connect(String login,String serverHost,String channelName){
    }

    /**
     * Un appel à cette méthode permet de réaliser la tâche "Se déconnecter"
     *
     * Commentaire: Pour se déconnecter, il suffit de le demander. La déconnexion est immédiate, aucune confirmatio
     *
     */
    public static void disconnect(){
    }

    /**
     * Un appel à cette méthode permet de réaliser la tâche "[Envoyer des messages] *"
     *
     * Commentaire: L'envoi de messages est optionnel, et à volonté.
     *
     * @param msg le message à envoyer
     */
    public static void sendMsg(String msg){
    }
```

**Figure 6.** *La classe Wrapper générée à partir du modèle de tâches enrichi (cf. Figure 5), destinée à être utilisée par l'environnement global intégrateur pour piloter le composant.*

Par ailleurs, il est intéressant de noter que le modèle de tâches et les squelettes générés restent liés et synchronisés. En d'autres termes, toute modification de la signature d'une méthode effectuée depuis le modèle de tâches est répercutée dans le code de la classe. De la même manière, toute modification de la signature d'une

méthode dans l'éditeur de code est répercutée au niveau de sa représentation dans le modèle de tâches. Cette fonctionnalité supplémentaire — développée en s'appuyant notamment sur le JDT — offre ainsi au développeur un nouveau point de vue, que l'on peut également qualifier de point de vue « Orienté Tâches » sur le composant en cours de développement et qui, nous l'espérons, constitue un outil supplémentaire au bénéfice de cet acteur.

## 5 L'utilisation des COTs

Même s'il s'agit d'un de nos objectifs à plus ou moins court terme, les moyens permettant d'exploiter les COTs au sein de STOrM, comme l'assemblage de composants par manipulation directe et graphique des modèles de tâches ou encore l'analyse de traces totalement intégrée au sein du modeleur, n'ont pas encore tous été implémentés. Nous développerons plus loin dans la partie 6 les perspectives de ces extensions qui permettront d'automatiser et d'intégrer plus avant les nouvelles fonctionnalités liées au modèle particulier des COTs, aussi bien du point de vue de leur intégration que de leur traçage. Néanmoins, nous pouvons dès aujourd'hui montrer dans quelle mesure cette nouvelle approche fournit déjà des moyens très intéressants.

### 5.1 L'intégration de COTs

Dans la partie 2.1, nous avons évoqué en détail les problèmes induits par le manque de sémantique au sein des composants du point de vue de leur intégration. Dans le cas d'un COT, les méthodes d'intégration sont implémentées au sein du *wrapper* dont le squelette a été généré par STOrM et dont l'implémentation a été réalisée par les développeurs en s'appuyant sur le modèle de tâches. Au travers du *wrapper*, un environnement tiers peut instancier le COT. La technique d'introspection sur le *wrapper* permet de récupérer la liste des méthodes en vue de son intégration. Cependant, si l'introspection « classique » des technologies OO est toujours possible, le COT offre un nouveau point de vue sur ses méthodes d'intégration, le modèle de tâches étendu étant livré avec le composant. Ainsi, l'ouverture du COT dans notre modeleur permet un nouveau type d'introspection permettant de découvrir les méthodes d'intégration du composant, non plus au travers d'une simple liste, mais cette fois-ci au travers de la tâche supportée par le composant.

Ce nouveau point de vue sur les composants permet de lever les problèmes de sémantique que nous avions exposés. Pour reprendre l'exemple du tchat, dont le modèle de tâches étendu (avec filtrage sur la facette intégration) est représenté dans la Figure 7, il est maintenant possible d'étudier aisément le fonctionnement du composant et de visualiser rapidement ses fonctionnalités qui ont été jugées comme clés par ses concepteurs (ergonomes, développeurs, etc.). Chaque possibilité offerte par le composant en terme d'intégration est représentée par une (sous-)tâche étendue, contextualisée dans le modèle de tâches global du composant. Les méthodes d'intégration sont liées à ces tâches. De ce fait, l'ambiguïté (cf. partie 2.1) à propos de la méthode *changeUserInfo* est levée puisque le modèle de tâches, avec la sémantique des transitions entre tâches, indique que la tâche *Se connecter au tchat* doit être réalisée avant de faire un appel à cette méthode. L'intégrateur n'a pas besoin de savoir où sont stockées les informations, le modèle de tâches est clair. Enfin, pour aller plus loin dans l'affichage des différentes facettes du COT, STOrM permet d'afficher simultanément le modèle de tâches du composant ainsi que la Javadoc de chaque méthode spécifiée, offrant ainsi des informations à un niveau d'abstraction

plus bas, comme des renseignements spécifiques à propos des paramètres, etc., mais cette fois-ci contextualisées au niveau d'abstraction de la tâche.

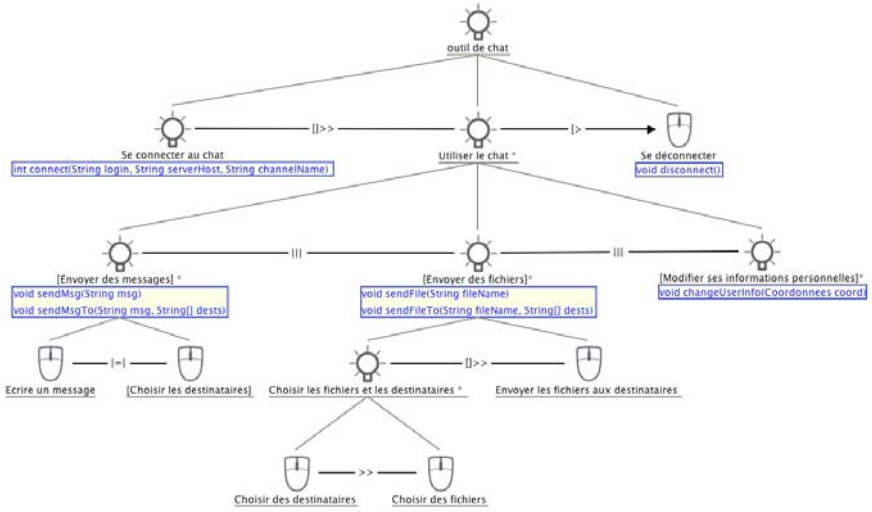



**Figure 7.** *Introspection sur le COT de tchat en vue de son intégration : Affichage des méthodes d'intégration contextualisées dans le modèle de tâches.*

## 5.2 Le traçage des COTs

***Tracer grâce aux tâches***

Nous avons vu précédemment (cf. partie 2.2) que tracer un composant sans connaître sa sémantique est un travail qui demande beaucoup d'efforts quant à la compréhension de son code. Les COTs, quant à eux, affichent leur sémantique en donnant du sens à leurs méthodes par l'intermédiaire de leurs modèles de tâches. Cette caractéristique facilite leur traçage. En effet, si on veut tracer un COT, grâce à son modèle de tâches étendu, on sait quelles méthodes correspondent aux tâches (cf. partie 4.3). Ainsi, pour le tchat, la tâche *Envoyer des messages* est réalisée, dans le code interne du COT, par la méthode *void envoiMessage(String expediteur, String message, String[][7] destinataires)*. Dans un premier temps et pour l'instant, si l'ergonome décide de tracer cette tâche, il l'indique à l'informaticien qui, en regardant le modèle de tâches étendu saura qu'il devra tracer la méthode *envoiMessage*. Ce dernier indiquera à l'ergonome qu'il peut récupérer directement les paramètres *expéditeur*, *message* et *destinataires*. Le modèle de tâches étendu aura donc permis :

- à l'ergonome de spécifier les tâches du composant, ainsi que les traces qu'il aimerait éventuellement obtenir par la suite ;
- au développeur de traduire les tâches en code objet ;
- à l'ergonome, en collaboration avec le développeur, d'indiquer les traces qu'il désire recueillir à un moment donné, indications qui sont traduites en aspects de trace par l'informaticien. Ce dernier point est important car il démontre que les multiples facettes des COTs aident à la collaboration entre les acteurs du processus de conception.

Avec les COTs, l'ergonome peut donc choisir plus facilement les informations qu'il désire récupérer, sans se préoccuper de l'implémentation réelle du composant.

[7] String[] signifie qu'on stocke les noms des destinataires dans un tableau.

En désignant une tâche, il désigne indirectement à l'informaticien la (ou les) méthode(s) associée(s) à cette tâche, et l'informaticien pourra indiquer les traces immédiates possibles grâce au déploiement d'aspects — en l'occurrence les paramètres de la (ou des) méthode(s), ainsi que certaines informations de type « système » toujours disponibles (comme la présence de réseau ou non, etc.) ou accessibles par l'intermédiaire de méthodes publiques du composant.

***Fichier de formats de trace déduit du modèle de tâches***

Pour créer les traces, il est nécessaire de préciser ce que l'on veut récupérer comme information, et sous quelle forme ces informations doivent être stockées (Tarby, 2006). Les aspects se chargent ensuite d'intercepter les informations requises pour les traces, et de les stocker en fonction de formats qui leur sont associés. Ce formatage repose sur un fichier de formats écrit en XML (cf. Figure 8). Un fichier de formats correspond à une activité que l'on veut tracer, cette activité pouvant regrouper bien évidemment plusieurs composants, par exemple un composant de tchat, un composant de dessin et un composant de FTP.

Ici encore, l'approche orientée tâches des COTs simplifie grandement l'écriture de ce fichier de formats. En effet, même s'il est toujours possible d'écrire très facilement son propre fichier de formats, à la main ou grâce à notre plug-in de création des aspects de trace (cf. 2.2), la majorité de ce fichier peut être déduite du modèle de tâches étendu, c'est-à-dire du modèle de tâches commenté et lié aux méthodes d'implémentation. Sur la Figure 8, dans la partie pouvant être générée automatiquement, *nom de la tâche* désigne la tâche devant être tracée (nom extrait du modèle de tâches), et la liste des attributs <*attr*> correspond aux paramètres de la méthode servant au traçage de la tâche (cf. Figure 5, ❷). Notons que si la tâche possède plusieurs méthodes servant au traçage, on aura alors plusieurs blocs <*type*> dans le fichier de formats.

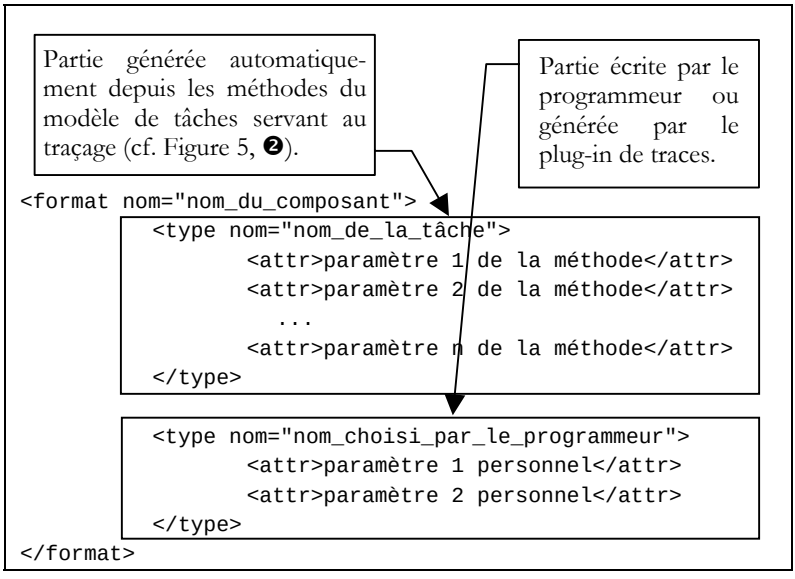



**Figure 8.** *Exemple de structure de fichier de format de traces*

La Figure 9 contient un extrait d'un fichier de formats de traces (suivant la syntaxe décrite ci-dessus), dont une partie concerne le traçage du composant de

tchat (<*format nom="tchat"*>). Les traces générées pour ce composant (<*type nom="…"*>), avec pour chacune d'elles d'éventuels paramètres (<*attr*>…</*attr*>), seront envoyés dans le fichier XML de traces final. Le nom et le nombre des paramètres est différent d'une trace à l'autre ; ceci est laissé à la discrétion du programmeur, suivant les requêtes de la personne qui désire tracer le composant.

```
<formats>
   <format nom="tchat">
       <comment>Traçage du tchat</comment>
       <type nom="envoi_message_daté">
          <comment>Envois de messages</comment>
          <attr>expéditeur</attr>
          <attr>destinataires</attr>
          <attr>message</attr>
          <attr>heure</attr>
       </type>
       <type nom="masque_affiche_groupe">
          <comment>Afficher/Masquer les connectés</comment>
          <attr>commande</attr>
          <attr>heure</attr>
       </type>
       ...
```

**Figure 9.** *Exemple de fichier de formats de traces pour le tchat*

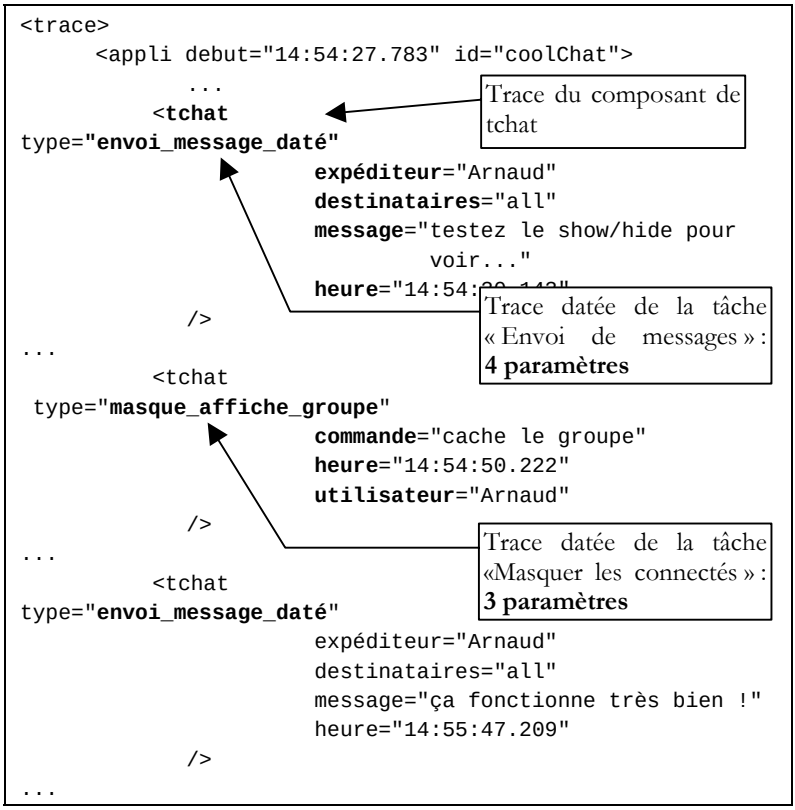
***Aspects de trace et génération de traces en XML***

Les traces sont générées en XML grâce aux aspects, créés par l'intermédiaire du plug-in évoqué précédemment, et en utilisant un fichier de formats de trace défini au préalable, tel que celui vu ci-avant. La technique employée repose sur un enchaînement d'aspects. Notre plug-in génère en effet deux ensembles de fichiers. Le premier ensemble contient les aspects associés aux traces que l'on veut récupérer ; le contenu de ces aspects est donc spécifique aux traces à produire. Chaque aspect appartenant à cet ensemble précise la méthode à tracer, le moment où la trace est produite (principalement au début ou à la fin de la méthode), et les paramètres à récupérer ; au final, après tissage des aspects, ceci revient à injecter dans l'application une seule ligne de code par trace à produire. Le second ensemble est un paquetage invariant, composé d'un ensemble de fichiers dont le contenu est immuable, quelles que soient les traces à générer. Le rôle de ce second ensemble est, pour chaque trace, d'intercepter l'exécution de la ligne de code insérée par le premier ensemble, et de traiter cette interception en récupérant les différents paramètres qu'elle retourne. Ces paramètres permettent ensuite de générer le code XML de la trace.

La Figure 10 montre un exemple de traces produites par l'exécution des tâches « Envoyer des messages » et « Afficher/Masquer les connectés ». Sur cet exemple, on constate que le composant de tchat a généré des traces en utilisant différents *types* de traces (en respectant les paramètres <*attr*> qui leur étaient associés dans le fichier de formats). Nous disposons donc d'une grande souplesse pour générer des traces.

Grâce à l'approche OT, nous voyons qu'il est possible de générer des fichiers de traces contenant des informations que l'on aura choisies à partir du modèle de tâches, et dans un format lui aussi issu du modèle de tâches. Etant donné la liberté

dont nous disposons quant aux contenus de tels fichiers, nous pouvons exploiter ces traces de façon beaucoup plus efficace qu'avec des fichiers « logs » classiques.




```
<trace>
      <appli debut="14:54:27.783" id="coolChat">
            ...
          <tchat
type="envoi_message_daté"
                    expéditeur="Arnaud"
                    destinataires="all"
                    message="testez le show/hide pour
                             voir..."
                    heure="14:54:
            />
...
          <tchat
 type="masque_affiche_groupe"
                    commande="cache le groupe"
                    heure="14:54:50.222"
                    utilisateur="Arnaud"
            />
...
          <tchat
type="envoi_message_daté"
                    expéditeur="Arnaud"
                    destinataires="all"
                    message="ça fonctionne très bien !"
                    heure="14:55:47.209"
            />
...
```

**Figure 10.** *Exemple de traces produites par le composant de tchat*

## 6 Perspectives

Les travaux que nous venons de présenter laissent entrevoir de nombreuses perspectives. Nous ne pourrons pas exposer dans cet article l'ensemble des idées qui nous motivent et qui ont émergé suite à ces premiers résultats. C'est pourquoi nous n'en présenterons ici que quelques-unes que nous envisageons de mettre en œuvre à court terme.

Ainsi que nous l'avons expliqué, nous nous sommes concentrés dans ces travaux sur le niveau *intégration* de la malléabilité telle que la définit Mørch (1997), et les problématiques qui s'y rapportent. Les COTs et les mécanismes associés permettent d'y apporter des éléments de réponse intéressants, mais soulèvent encore de nombreuses interrogations. Par exemple, nous avons montré comment le contrôle de l'intégration de COTs est rendue accessible par le pilotage et l'articulation de leurs tâches ; il serait intéressant d'étudier jusqu'où peut aller ce genre d'articulation, et s'il est envisageable de se rapprocher davantage du niveau le plus avancé de malléabilité, à savoir l'extension, qui correspond à la redéfinition même du code fonctionnel d'un composant. Par exemple, serait-il possible de

substituer une sous-tâche particulière d'un composant par une autre sous-tâche similaire, issue d'un autre composant ? Si ces considérations dépassent pour l'instant le cadre des COTs, elles n'en demeurent pas moins pertinentes et constituent une piste intéressante que nous envisageons d'approfondir.

Par ailleurs, dans le cadre de nos travaux pour la co-évolution, nous souhaitons grandement encore relever le niveau d'abstraction des moyens pour l'intégration dynamique. Nous avons spécifié dans la partie 3 que, de notre point de vue, intégrer un composant correspond à contextualiser sa tâche dans une tâche plus globale. Le modèle de tâches étendu étant directement lié au code du composant via le *wrapper* d'intégration, notre objectif est de permettre d'intégrer les composants, non plus en écrivant dans le code les appels de méthodes nécessaires, mais en tissant graphiquement des liens entre leurs modèles de tâches.

Du point de vue du traçage et étant donné qu'un COT connaît à la fois son modèle de tâches, sa Javadoc et son code Java, nous souhaitons que l'ergonome puisse générer lui-même directement les aspects de trace sans qu'il ait connaissance du code Java du composant. Pour cela, nous voulons fusionner les fonctionnalités de l'outil de création d'aspects de trace présenté dans la partie 2.2 avec celles de STOrM. Lorsque l'ergonome voudra tracer un COT, il pourra spécifier ses traces directement au travers du modèle de tâches du composant. La sélection d'une tâche du modèle désignera implicitement la (ou les) méthode(s) de traçage associée(s). L'outil proposera alors de choisir les méthodes et les paramètres à tracer, mais ceci à un niveau d'abstraction plus élevé que dans l'introspection classique car mettant notamment en œuvre les informations disponibles dans la Javadoc (cf. Figure 11). En choisissant et en ordonnant les éléments qui l'intéressent, l'ergonome créera implicitement le fichier de formats de traces, ainsi que les aspects associés.

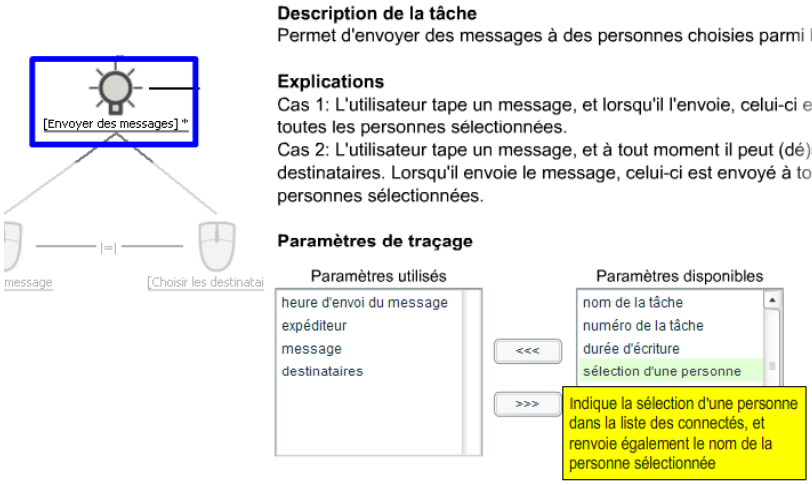



**Figure 11.** *Perspective d'IHM permettant la spécification du traçage d'une tâche*

Une autre perspective concerne l'analyse des traces générées qui est actuellement faite « à la main », ou au mieux assistée par des traitements XSLT *ad hoc*. Ainsi, à partir des traces vues dans la partie 5.2, et de leur lien avec le modèle de tâches du COT, nous pouvons construire des modèles des tâches effectives des utilisateurs (Figure 12) ; ces modèles peuvent ensuite être comparés aux modèles de tâches prescrites des applications testées (Abed et Ezzedine, 1998). Nous pouvons aussi transformer ces traces en scénarios pouvant être rejoués dans CTTE ou dans

K-Made par exemple. Nous envisageons donc de fournir des bibliothèques de traitement (XSLT, Java, etc.) automatisant les transformations qui permettent par exemple de passer des fichiers XML de traces à des modèles de tâches effectives ou à des scénarios pouvant être rejoués à volonté.

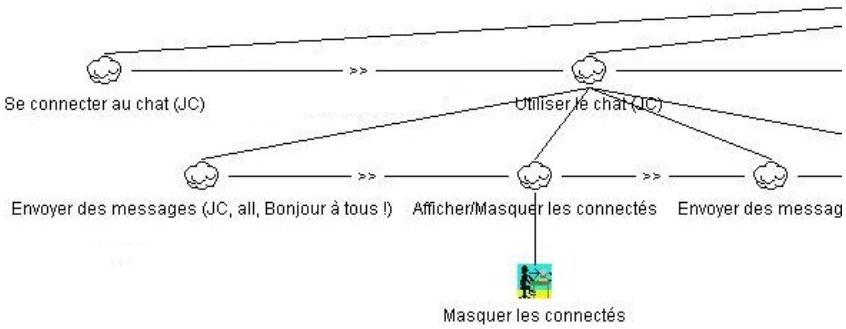



**Figure 12.** *Extrait de CTTE affichant le modèle de tâches effectives issu des traces du tchat*

En l'état, le modèle de tâches des COTs ne prend pas en compte la dimension coopérative de tâches intrinsèques à certains composants. Par exemple, la gestion des rôles au sein d'une activité coopérative est déléguée à l'environnement intégrateur qui doit utiliser les méthodes d'intégration des composants pour les piloter de manière à refléter le rôle de chaque utilisateur dans l'activité globale. Une nouvelle perspective est d'imaginer qu'un composant puisse lui-même intégrer des modèles de tâches dédiés à certains types de rôles. Il n'y aurait alors plus un seul modèle de tâches dans le COT, mais un modèle de tâches global (c'est-à-dire faisant apparaître de façon explicite les rôles, les tâches, et les liens qui les unissent, comme le fait déjà CTTE en mode coopératif) et plusieurs autres modèles reflétant différentes facettes d'utilisation du composant.

Enfin, étant donné que ces travaux ont été initiés et mis en œuvre dans le cadre du projet CooLDev, nos idées ont été appliquées à des composants du type JavaBeans. Néanmoins, nous avons plusieurs fois souligné que les concepts et techniques formant la base des COTs sont communs à la plupart des grandes technologies composants standardisées. Nous souhaitons donc vivement transposer ces travaux et mettre notre démarche et nos outils au service d'autres technologies. En particulier, les Services Web nous apparaissent comme de très bons candidats à une adaptation OT, ces composants servant souvent des tâches d'un assez haut niveau d'abstraction. La création des SWOTs (Services Web Orientés Tâche) devrait apporter, nous l'espérons, de nouveaux moyens facilitant à la fois leur test, leur découverte et leur intégration.

## 7 Conclusion

Comme l'ont démontré depuis plusieurs années les recherches pluridisciplinaires s'intéressant à la conception de logiciels, la malléabilité doit être une propriété fondamentale des nouveaux systèmes interactifs de manière à répondre au problème de l'émergence des besoins des utilisateurs. Cette malléabilité peut être supportée par les technologies à base de composants logiciels et le mécanisme d'intégration. Toutefois, nous avons démontré dans cet article que les modèles de composants existants souffrent encore aujourd'hui d'un problème d'ordre sémantique. Ce manque rend à la fois difficile l'intégration des composants,

et les activités liées à leurs tests. Pour pallier ce problème, nous avons proposé une nouvelle démarche de conception qualifiée d'Orientée Tâches (OT), tentant de marier les modèles de tâches aux modèles de composants existants. En effet, dans les approches classiques de conception, le modèle de tâches n'est que peu exploité. Dans l'approche OT, le modèle de tâches devient le pivot autour duquel les composants sont construits, testés et intégrés, tout en fédérant les compétences des différents acteurs du processus de développement (ergonomes, développeurs, utilisateurs, etc.). La démarche que nous avons présentée se découpe en cinq étapes principales et permet de construire des Composants Orientés Tâches (COTs).

L'intérêt d'un COT est qu'il embarque son modèle de tâches, étendu par deux ensembles de méthodes clés spécifiées en vue respectivement de son intégration dans un environnement tiers et de son traçage dans une approche de type POA. Ces méthodes sont utilisées pour générer le squelette d'un *wrapper* d'intégration ainsi que le squelette même du composant. Les COTs permettent ainsi d'étendre les moyens classiques d'introspection en ajoutant à la description de leurs méthodes la sémantique propre à leur modèle de tâches. La compréhension du fonctionnement du composant s'en trouve augmentée. Elle facilite son contrôle lors de son intégration, puisque le modèle de tâches contextualise les méthodes permettant de le piloter et indique directement les tâches accessibles et pilotables par un environnement intégrateur. L'activité de traçage est également facilitée dans le sens où la sélection des méthodes à tracer ne s'effectue plus en étudiant le code du composant, mais au travers de son modèle de tâches et par identification directe des tâches à tracer, ce qui est plus en accord avec le niveau d'abstraction de cette activité.

L'approche OT est aujourd'hui en partie supportée par l'outil STOrM (Simple Task Oriented Modeler) dédié à la création et la manipulation de COTs. STOrM, comme l'approche OT, est encore sujet à évolution. Comme nous l'avons souligné, ces premiers résultats ouvrent de nombreuses et excitantes perspectives. Enfin, il est important de rappeler que ces travaux s'inscrivent dans une démarche globale dédiée au principe de co-évolution. STOrM a de ce fait été réalisé en tant que plug-in Eclipse intégrable à la plateforme de développement coopérative CooLDev. Nous espérons ainsi encore mieux supporter cette nouvelle approche pluridisciplinaire de conception pour de meilleurs environnements logiciels en symbiose avec les propriétés intrinsèques des activités humaines qu'ils tentent de supporter.

***Remerciements***

*Les auteurs remercient la Direction de la Recherche pour l'ACI CooLDEv, le FEDER (Fonds Européen de Développement Régional), ainsi que le programme TAC financé par la Région Nord/Pas-de-Calais et l'Etat dans le cadre du CPER pour les projets MIAOU et EUCUE.*

## 8 Références

Abed, M., Ezzedine, H. (1998). Vers une démarche intégrée de conception-évaluation des systèmes Homme-Machine. *Journal of Decision Systems*, vol. 7, 147-175.

Augustin, L., Bressler, D., Smith, G. (2002). Accelerating software development through collaboration. In *Proceedings of the 24rd International Conference on Software Engineering*, ICSE 2002, 559-563.

Bardram, J. E. (1998). Designing for the dynamics of cooperative work activities. In *Proceedings of The 1998 ACM Conference on Computer Supported Cooperative Work, Seattle, Washington, USA.*

Baron, M., Lucquiaud, V., Autard, D., Scapin, D. L. (2006). K-MADe: un environnement pour le noyau du modèle de description de l'activité. In *Actes de la 18ème Conférence Francophone sur l'Interaction Homme-Machine IHM'06*, New York, NY, USA. ACM Press, 287-288.

Bedny, G., Meister, D. (1997). *The Russian theory of activity, Current Applications to Design and Learning.* Lawrence Erlbaum Associates.

Blevins, D. (2001). Overview of the Enterprise JavaBeans Component Model. In (Heineman et Councill, 2001), 589-606.

Booth, D., Liu, C.K. (2006). W*eb Services Description Language (WSDL) Version 2.0.* Disponible à : http://www.w3.org/TR/wsdl20-primer

Bourguin, G., Derycke, A. (2005). Systèmes interactifs en co-évolution, réflexions sur les apports de la théorie de l'activité au support des pratiques collectives distribuées. *Revue d'Interaction Homme-Machine (RIHM)*, vol. 6, num. 1, 1-31.

Bourguin, G., Derycke, A., Tarby, J.C. (2001). Beyond the Interface: Co-evolution Inside Interactive Systems – A proposal Founded on Activity Theory. *People and Computer vol. 15 – Interaction without Frontiers, Proc. of HCI'2001*, Blandford, Vanderdonckt, Gray (eds), Springer Verlag, 297-310.

Bruins, A. (1998). The Value of Task Analysis in Interaction Design. In *Task to Dialogue: Task-Based User Interface Design, Workshop, CHI'98*, Los Angeles, April 18-23.

Clerckx, T., Luyten, K., Coninx, K. (2004). DynaMo-AID: a Design Process and a Runtime Architecture for Dynamic Model-Based User Interface Development. In *The 9th IFIP Working Conference on Engineering for Human-Computer Interaction, jointly with the 11th International Workshop on Design, Specification and Verification of Interactive Systems*, Trems-büttel Castle, Hamburg, Germany, July 11-13, Pre-Proceedings, 142-160.

Clerckx, T., Vandervelpen, C., Luyten, K. Coninx, K. (2006). A Task-Driven User Interface Architecture for Ambient Intelligent Environments. In *International Conference on Intelligent User Interfaces*, Sydney, Australia, 29 January -1 February.

Delotte, O., David, B.T., Chalon R. (2004). Task Modelling for Capillary Collaborative Systems based on Scenarios. In (Slavík et Palanque, 2004), 25-31.

Diaper, D., Stanton, N. (2004). *Handbook of Task Analysis for Human-Computer Interaction.* Lawrence Erlbaum Associates Pubs, London.

Dougiamas, M. (2001). *Moodle: open-source software for producing internet-based courses.* Disponible à : http://moodle.com

Ferris, C., Farrel, J. (2003). What are web services? *Communications of the ACM,* vol. 46, num. 6, 31.

Filman, R. E., Elrad, T., Clarke, S., Akit, M. (2005). *Aspect-Oriented Software Development.* Addison-Wesley, Boston.

Heineman, G.T., Councill, W.T. (2001). *Component-based software engineering: putting the pieces together.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Hilbert, D., Redmiles, D. (2000). Extracting usability information from user interface events. *ACM Computing Surveys*, vol. 32, num. 4, 384-421.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G. (2001). An overview of AspectJ. In *ECOOP 2001 - Object-Oriented Programming: 15th European Conference, Budapest, Hungary*. Springer-Verlag, 327-353.

Kiniry, J.R. (2003). Semantic Component Composition. In *Third International Workshop on Composition Languages, in conjunction with 17th European Conference on Object-Oriented Programming (ECOOP)*, Darmstadt, Germany.

Lewandowski, A. (2006). *Vers de meilleurs supports aux activités cooperatives en accord avec la co-évolution — application au développement logiciel coopératif*. Thèse de l'Université du Littoral Côte d'Opale, LIL, décembre.

Lu, S., Paris, C., Vander Linden, K., Colineau, N. (2003). Generating UML Diagrams From Task Models. In *Proc. of CHINZ'03, July 3-4, Dunedin, New Zealand.*

Luyten, K., Clerckx, T., Coninx, K., Vanderdonckt, J. (2003). Derivation of a Dialog Model from a Task Model by Activity Chain Extraction. In *Proceedings of Interactive Systems: Design, Specification, and Verification, 10th International Workshop DSV-IS*, Funchal, Madeira Island, Portugal, June, 2003. Lecture Notes in Computer Science, Springer, vol. 2844, 203-217.

Luyten, K., Vandervelpen, C., Coninx, K. (2005). Task Modeling for Ambient Intelligent Environments: Design Support for Situated Task Executions. In *The 4th International Workshop on TAsk MOdels and DIAgrams for user interface design (TAMODIA 2005)*, Gdansk, Poland, 87-94.

Maes, P. (1987). Concepts and experiments in computational reflection. In *OOPSLA'87: Conference proceedings on Object-oriented programming systems, languages and applications*, New York, USA, 147-155.

Medjahed, B., Bouguettaya, A., Elmagarmid, A.K. (2003). Composing Web services on the Semantic Web. *The International Journal on Very Large Data Bases*, vol. 12, num. 4, 333-351.

Mørch, A. (1997). Three levels of end-user tailoring: customization, integration, and extension. In *Method and Tools for Tailoring Object-Oriented Applications: An Evolving Artifacts Approach*. PhD thesis, Dept of Informatics, University of Oslo, 41-51.

Mørch, A. I., Stevens, G., Won, M., Klann, M., Dittrich, Y., Wulf, V. (2004). Component-based technologies for end-user development. *Communications of the ACM*, vol. 47, num. 9, 59-62.

Mori, G., Paterno, F., Santoro, C. (2002). CTTE: Support for Developing and Analysing Task Models for Interactive System Design. *IEEE Transactions on Software Engineering*, vol. 28, num. 8, 797-813.

Nunes, N. Falcão e Cunha, J. (2000). Towards a UML profile for interaction design: the Wisdom approach. In *Proceedings of the UML 2000 Workshop*. Lecture Notes in Computer Science, Springer, vol. 1939, 117-132.

Object Technology International (OTI), Inc. (2003). *Eclipse Platform Technical Overview*. Disponible à : http://www.eclipse.org

O'Neill, E., Johnson, P. (2004). Participatory task modelling: users and developers modelling users' tasks and domains. In (Slavík et Palanque, 2004), 67-74.

Paris, C., Colineau, N., Lu, S., Vander Linden, K. (2005). Automatically generating effective online help. *International Journal on Elearning*, vol. 4, num. 1, 83-103.

Paterno, F. (2004). ConcurTaskTrees: An Engineered Notation for Task Models. In (Diaper and Stanton, 2004), 483-501.

Pinheiro da Silva, P. (2002). *Object Modelling of Interactive Systems: The UMLi Approach*. Ph. D. Thesis, University of Manchester, United Kingdom.

Reichart, D., Forbrig, P., Dittmar, A. (2004). Task models as basis for requirements engineering and software execution. In (Slavík et Palanque, 2004), 51-58.

Richard, J.F. (1983). *Logique de fonctionnement et logique d'utilisation*. Rapport de recherche INRIA n° 202, Avril.

Ruault, JR. (2002). UML and interactive systems, another step forward. In *Computer-aided Design of User Interface III*, Kolski C., Vanderdonckt J. (Eds.), Kluwer Academic Publishers, Dordrecht, 243-256.

Schuler, D., Namioka, A. (1993). *Participatory Design: Principles and Practices*. Lawrence Erlbaum Associates, Inc., Mahwah, NJ.

Scogings, C., Phillips, C. (2004). Linking Task and Dia-logue Modeling: Toward an Integrated Software Engineering Method. In (Diaper and Stanton, 2004), 551-566.

Settouti, L. S., Prié, Y., Mille, A., Marty, J.C. (2006). Système à base de trace pour l'apprentissage humain. In *Colloque International TICE 2006 «Technologies de l'Information et de la Communication dans l'Enseignement Supérieur et l'Entreprise»*, INP Toulouse.

Slavík, P., Palanque, P. (2004). *Proceedings of the Third International Workshop on Task Models and Diagrams for User Interface Design - TAMODIA 2004*. Prague, Czech Republic, November.

Sun Microsystems (1997). *JavaBeans API Specification, Version 1.01-A*. Disponible à : http://java.sun.com/products/javabeans/docs/spec.html

Szyperski, C., Pfister, C. (1997). Workshop on component-oriented programming, summary. In *Special Issues in Object-Oriented Programming - ECOOP 96, Workshop Reader*, Muehlhaeuser, M. (Ed.), Heidelberg, Gemany.

Tarby, J.C. (2006). Évaluation précoce et conception orientée évaluation. In *Actes de la 10ème Conférence Internationale Ergo'IA 2006*, Bidart/Biarritz, France, 11-13 Octobre, Brangier, E., Kolski, C., Ruault, J.-R. (Eds.), ISBN: 2-9514772-6-0, 343-346.

Van der Aalst, W. (2003). Don't go with the flow: Web services composition standards exposed. *Trends Controversies Jan/Feb 2003 issue of IEEE Intelligent Systems.*

Van Welie, M., van der Veer, G.C., Eliëns, A. (1998). Euterpe - Tool support for analyzing cooperative environments. In *Proceedings of the Ninth European Conference on Cognitive Ergonomics, Limerick, Ireland.*

Vicente, K. J. (2000). HCI in the global knowledge-based economy: designing to support worker adaptation. *Communications of the ACM*, vol. 7, num. 2, 263-280.

Wang, N., Schmidt, D.C., O'Ryan, C. (2001) Overview of the CORBA Component Model. In (Heineman et Councill, 2001), 557-572.

Won, M., Stiemerling, O., Wulf, V. (2005). Component-Based Approaches To Tailorable Systems. In *End-User Development*, Lieberman, H., Paternò, F., Wulf, V., (Eds.), Human-Computer Interaction Series, Kluwer Academic, 115-142.