

Intégration du service d'interactions au sein d'un serveur d'EJB

Audrey Ocello

Ecole Supérieure en Sciences Informatiques

Septembre 2001

Encadreurs :

Mireille Blay, Anne-Marie Dery, Michel Riveill

Table des matières

1	Introduction	3
2	Présentation du projet	3
2.1	Contexte	3
2.2	Cahier des charges	4
2.3	Un langage de spécification des interactions	5
2.4	Un service	6
3	Mise en oeuvre	7
3.1	Planning	7
3.2	Une architecture	8
3.3	Un modèle fusionnel	9
3.4	Un environnement de programmation	12
4	Conclusion	14
5	Annexe 1 : Le planning	16
6	Annexe 2 : Les Diagrammes de classes	17
7	Annexe 3 : Mapping objet relationnel	24
8	Annexe 4 : Bibliothèque de schémas	25
9	Annexe 5 : Schémas d'interactions de l'interface graphique	26

1 Introduction

La finalité du stage est de poursuivre et compléter le travail réalisé durant le projet de fin d'études d'ESSI3 à savoir l'intégration du service d'interactions sur un serveur EJB[5] à fin de diffusion. La réalisation de cet objectif tient en 3 étapes.

La première tâche est la définition et la mise en oeuvre du modèle d'interactions dans Java (i.e. fournir le langage de description des interactions et mettre en place les mécanismes de base de la gestion automatique de ces interactions). La seconde tâche consiste en l'intégration dans la plate-forme de développement de manière extensible, adaptable et efficace. Enfin, la dernière étape correspond à la conception d'un outil de développement permettant de démontrer à la fois l'intérêt et la fiabilité du service.

Ce rapport s'articule en 3 parties. La première partie présente le contexte dans lequel a émergé le projet, décrit les tâches à réaliser en s'appuyant sur les spécifications du projet et introduit le concept d'interactions au lecteur. La seconde partie expose le travail réalisé en mettant en évidence les points forts du modèle et les difficultés rencontrées. La dernière partie conclut sur les apports du stage et les perspectives sur lesquelles il débouche.

2 Présentation du projet

2.1 Contexte

Dans un monde où les objets sont de plus en plus souvent distribués, un nouveau problème a émergé. Les systèmes répartis sont confrontés à la faible capacité d'adaptation des objets déployés sur différentes plates-formes qui imposent des contraintes telles que l'intégration de la sécurité ou encore l'observabilité.

Pour réaliser cette adaptation, la première idée venant à l'esprit consiste en l'implémentation de différentes versions d'un même objet. Chaque version mélange code fonctionnel (méthodes "business") et code non fonctionnel (se chargeant des propriétés techniques telles que la persistance, les transactions, la sécurité, la notification, ...). Le déploiement d'un objet se fait en sélectionnant la version correspondant au type de contraintes imposées par le système. Cette approche n'est pas viable. D'une part, il faut prévoir autant de versions qu'il y a de configurations possibles, c'est un problème combinatoire qui explose au bout d'un moment et est source d'erreurs. D'autre part, une application est difficilement adaptable et réutilisable à cause de l'entrelacement entre code fonctionnel et non fonctionnel.

Les approches par composants tentent de répondre à cette adaptabilité statique par l'usage de fichier de configuration et de génération de code. Le projet ARCAD[2] propose quant à lui de concevoir un environnement d'exécution reparti permettant de déployer des applications par assemblage de composants logiciels adaptables et de réagir aux variations du contexte par modification dynamique des configurations. L'intégration des travaux de diverses équipes

françaises de recherche à la plate-forme ObjectWeb[9] permettra d'obtenir une plate-forme distribuée open source destinée à l'enseignement mais aussi à l'industrie.

Un autre problème est que les outils actuels ne permettent pas d'extraire la sémantique de la communication entre objets. La collaboration des objets imposent la modification du code existant pour les connecter. Ceci réduit considérablement l'évolutivité et l'adaptabilité d'un système réparti dont l'une des principales caractéristiques devrait être la capacité à interagir dynamiquement avec les autres objets de son environnement.

Le service d'interactions développé par l'équipe RAINBOW[11] du laboratoire I3S répond à ce besoin en s'attaquant à l'adaptabilité dynamique. Par l'expression de simples règles d'interaction entre objets, il permet de modifier dynamiquement le comportement des objets sans aucune modification de ces derniers et ceci de façon réversible. Les interactions permettent d'exprimer la communication entre les objets de façon externe. Dans le cadre du projet ARCAD, il s'agissait d'adjoindre aux modèles de composants, un modèle d'interactions. Le serveur d'EJB[5] JOnAS[8] faisant partie de la plate-forme ObjectWeb a été choisi pour réaliser l'implémentation. Ce travail devait être mené à la fois dans un souci d'adaptabilité dynamique des objets, de consistance des interactions et d'efficacité dans les communications.

Plusieurs implémentations des interactions existent déjà et reposent sur la Méta programmation avec Smalltalk, OpenC++ et OpenJava[7] et utilisent CORBA[4] comme support aux applications distribuées. La plate-forme Dico Java est une implémentation du service d'interactions écrite en Java et utilisant CORBA. Dans le cadre du projet, il était très intéressant d'exploiter ce code car une partie de ce qui a été fait pouvait être réutilisée.

2.2 Cahier des charges

Différents axes de travail ont été identifiés.

A. Consolidation du service d'interactions existant

- Rendre le service conforme aux spécifications attendues
- Rendre le serveur interagissant
- Développer le serveur d'interactions en tant que Enterprise Bean pour permettre la persistance des données

B. Evolution du modèle d'interactions

- Introduction d'éléments génériques de syntaxe
- Gestion de la cohérence par l'usage de transactions

C. Réalisation d'une GUI pour l'utilisation du service d'interactions

D. Fournir une bibliothèque de schémas d'interactions pour l'introduction dynamique de services tels que :

- L'authentification
- Une forme de synchronisation simplifiée (i.e. la pose d'un verrou)
- La gestion d'un historique listant les appels des méthodes
- La sauvegarde et la restauration de l'état d'un objet durant les phases critiques.

Le but de cette partie n'est pas de proposer des services évolués mais de montrer comment utiliser le service d'interactions pour introduire des services dynamiquement et de manière intuitive.

2.3 Un langage de spécification des interactions

De nombreux nouveaux modèles de programmation tels que la programmation par aspects[1] visent à séparer le code fonctionnel du code non fonctionnel au sein des applications. Le service d'interactions développé par le projet Rainbow permet d'extraire la communication inter-objets du code fonctionnel par modification du comportement à la réception d'un message.

Prenons l'exemple de la traçabilité d'un programme. Pour debugger une application, un programmeur a tendance à insérer ici et là dans le code des " `println()` " pour afficher l'état d'un objet avec une multitude d'instructions conditionnelles si on s'intéresse à tracer une seule instance d'une classe. Ceci alourdit le code inutilement. D'autre part, l'utilisation d'un debugger avec un objet réparti ne permet de tracer que les appels de méthodes locaux au debugger, on ne voit qu'une partie de ce qu'il se passe. Avec le service d'interactions, il suffit de créer une classe `Trace` qui gère l'affichage de différents appels puis de lier un objet de cette classe à un autre objet dont on veut surveiller le comportement au cours de l'exécution. Les objets peuvent être liés et déliés dynamiquement à l'exécution. Ce service rend donc les applications plus faciles à écrire, à comprendre et à réutiliser. En terme d'interactions, cet exemple s'écrira¹ de la manière suivante :

```
interaction Tracabilité(Object O, Traceur T) {
    O.* -> T.trace(_call); O._call; T.trace(_call)
}
```

Le service utilise un langage de spécification nommé ISL[3] (Interaction Specification Language) pour décrire les schémas d'interactions. Il est basé sur une grammaire simple, incluant des opérateurs conditionnels (`if..then..else`), de délégation ainsi que des opérateurs réactifs (`;` et `//` respectivement pour les appels synchrones et asynchrones). Ce langage est indépendant du langage d'application. Les contrôles interviennent à la réception des requêtes qui peuvent être interdites, transformées ou impliquer de nouvelles requêtes.

¹Le caractère "*" indique que la règle s'applique à toutes les méthodes "métier" de l'objet O. Selon le contexte d'utilisation, le terme `_call` désigne soit l'appel à la méthode cible de l'interaction soit le message lui-même sous forme réifié

Les implémentations existantes de ce service sont basées sur l'interception de l'exécution des méthodes destinées à l'objet sur lequel une ou des interactions ont été posées, puis sur l'évaluation des règles correspondantes. Cette évaluation est basée sur l'invocation dynamique à distance des méthodes. Par ailleurs, si un utilisateur ajoute de nouvelles interactions sur un objet, les arbres de règles correspondants sont recalculés.

Un schéma d'interaction est composé :

- d'un nom,
- d'une liste de couples "type / nom de variable", c'est à dire les paramètres du schéma
- d'une ou de plusieurs règles.

Une règle est composée :

- d'un message interactif (partie gauche de la règle), nommé notifiant,
- d'une liste d'actions à exécuter (partie droite de la règle), nommée réaction.

Soit le schéma d'interactions Authentification permettant de lier des objets de type Object ou un de ses sous types à des objets de type Authenticator. Supposons que l'objet o1 soit lié à l'authentificateur a1 par ce schéma. Lorsque une méthode d'o1 est invoquée, a1 vérifie que le client a le droit d'utiliser la méthode en cours sur l'objet spécifié. Si c'est le cas, la méthode est exécutée, sinon a1 affiche un message d'erreur.

Ce schéma s'écrit de la manière suivante :

```
interaction Authentification(Object O, Authenticator A) {
  O.* -> if A.authenticate(O, _call) then O._call
        else A.printError() endif
}
```

Lorsqu'un schéma est instancié, c'est à dire qu'il lie plusieurs instances, on obtient une nouvelle interaction. Cette opération est appelée la "pose d'interaction". Chaque règle du schéma est alors ajoutée à la liste de règles de l'objet notifiant (i.e. l'objet récepteur du message notifiant). Lorsque plusieurs règles concernent la même méthode d'un objet, une fusion des réactions est réalisée. Aussi, à l'exécution d'un message c'est la fusion des règles d'interactions qui est exécutée.

2.4 Un service

Le schémas de la figure 1 récapitule l'ensemble des étapes d'utilisation du service et les éléments qui entrent en jeu à ces différentes étapes. Lorsqu'un schéma est créé, son contenu est d'abord analysé par le parser qui le reconstruit à partir d'entités réifiées. A titre d'exemple, la réaction associée à chaque règle est transformée en arbre de syntaxe. Chaque schéma réifié (dont les arbres de syntaxe) sont transmis au serveur d'interactions qui les stocke. Avant de pouvoir poser des interactions entre objets, ceux-ci doivent s'enregistrer auprès du serveur d'interactions. Lorsque des objets sont liés par un schéma (i.e. la pose),

chaque arbre est “instancié”² et transmis à l’objet cible de la règle d’interaction. Ensuite la communication entre les objets interagissants est directe, elle ne passe pas par le serveur d’interactions.

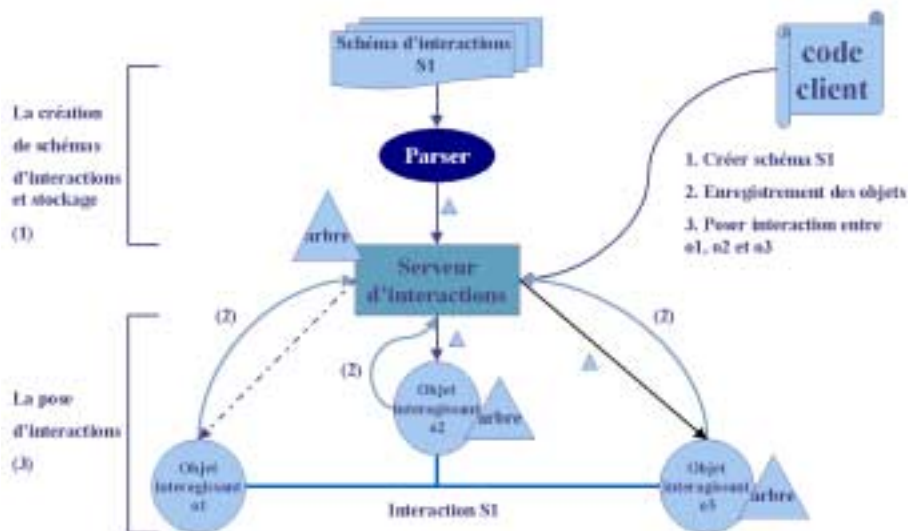


FIG. 1 – Schéma récapitulatif

3 Mise en oeuvre

Cette partie se limite à la description du travail que j’ai réalisé. En effet, s’il existe une grande partie du code, mon travail a consisté à réorganiser l’ensemble, à l’homogénéiser.

3.1 Planning

Tout au long du projet des réunions ont été organisées avec les encadreurs et d’autres groupes d’étudiants travaillant sur des projets communs à l’équipe RAINBOW. Pasin Marcia, Pham Mihn et un groupe d’étudiants d’ESSI2 font partie des personnes avec lesquelles j’ai travaillé. Les dernières semaines du stage ont été consacrées aux tests et à la préparation de la réunion ARCAD. Le détail du planning se trouve en annexe 1.

²les paramètres formels sont remplacés par les paramètres réels par exemple.

3.2 Une architecture

Des entités distribuées A partir de l'implémentation Dico Java, une réification complète du service a été réalisée. Des entités devaient être accessibles à distance (par copie ou référence selon les cas). Des entités telles que les schémas, les instances de schémas (i.e. interactions), les modèles de règles et les instances de règles ou encore les paramètres interagissants (i.e. paramètres d'un schéma) et les objets interagissants (objets liés à une interaction) ont donc été réifiées. Cette séparation entre modèle et instance s'est répercutée sur l'ensemble des classes. Ainsi un " RulePattern " s'instancie en " RuleInstance ".

Cette réification du service a permis de rendre le modèle plus clair, plus facilement réutilisable et évolutif.

Un service hétérogène L'existence de plusieurs versions du service d'interactions incompatibles les unes avec les autres et ne proposant pas des services identiques était difficile à maintenir. Un travail d'abstraction a été réalisée afin d'obtenir une architecture capable de supporter l'hétérogénéité et de faire cohabiter les différentes versions du service.

Quelle que soit l'implémentation du serveur (local, RMI, EJB)³, une grande partie du code du service d'interaction peut être réutilisé. En effet, le parsing, la gestion de l'arbre, la fusion sont autant d'éléments communs à toutes les versions réalisées jusqu'à présent. La motivation est de mettre en commun toutes ces versions qui diffèrent uniquement par les entités modélisant le serveur d'interaction à proprement dit. Il faut également que les différentes implémentations du serveur évoluent parallèlement mais toujours dans le même sens, c'est à dire qu'elles respectent une spécification précise (via des interfaces).

Pour répondre à ces besoins, une remodelisation de l'architecture du service a été réalisée. Chaque objet d'implémentation d'un type de serveur donné implémente une interface commune pour un soucis de cohérence entre les versions. D'autre part, la création des objets modélisant le serveur est déléguée et contrôlée par une "fabrique" qui sait instancier les objets en fonction du type du serveur. Enfin, les objets interagissants qui peuvent être des objets locaux, des objets RMI ou encore des beans doivent être manipulés de la même façon. Aucune différence ne doit transparaître pour permettre une transparence vis à vis du serveur. L'appel de méthodes sur de tels objets est délégué à une classe d'objet qui décrit ce qu'est un objet interagissant. Elle doit permettre de récupérer l'objet réel. Cette classe est donc dérivée en sous classes représentant un type d'objets particulier.

Les impacts de cette nouvelle architecture sont positifs. Le code client reste inchangé puisque les points d'entrées du service sont données par les interfaces. Un fichier de configuration⁴ décrivant le type du serveur est utilisé. A ce niveau, la fabrique renvoie une référence sur le bon type de serveur. D'autre part,

³Un serveur local est composé d'objets java "pur", un serveur RMI est composé d'objets distribués java RMI, un serveur EJB est un serveur composé d'Enterprise Bean

⁴Le fichier de configuration est lu comme un ensemble de variables d'environnement (nom du serveur, url, type ...)

l'utilisateur peut choisir d'utiliser un type d'implémentation particulier selon les besoins de l'application visée. Cela permet d'éviter l'installation de bibliothèques supplémentaires lorsque ce n'est pas nécessaire.

Des objets interopérables La pose d'interactions est possible sur des objets locaux au serveur d'interactions et sur des objets distants déclarés auprès du serveur. La gestion des interactions en réaction est toujours possible pour autant que les objets aient été déclarés auprès du serveur d'interactions. Les objets interagissants peuvent donc interagir indépendamment de leur type.

Ceci laisse penser qu'à l'avenir, l'architecture supportera les objets CORBA[4] moyennant une communication RMI-IIOP[6].

Bilan Le service se découpe en 7 packages⁵ :

- le package *parser* permet de lire des schémas d'interactions et de les transformer en un ensemble d'entités réifiées,
- le package *tree* définit la structure d'une réaction⁶,
- le package *process* est le cerveau du service, il décrit des mécanismes tels que la fusion de règles d'interactions,
- le package *server* contient les interfaces décrivant le serveur d'interactions. Ces interfaces constituent une spécification du service d'interaction. Les sous packages du package *server* correspondent aux implémentations en java pur (package *localserver*), en RMI (package *rmiserver*) et en Enterprise Beans (package *beanserver*),
- le package *common* comporte un ensemble de classes utilitaires souvent utilisées, par exemple une classe permettant de lire le fichier de configuration ou celle opérant la sérialisation,
- le package *thread* permet de mettre en place une gestion de threads pour exprimer le parallélisme entre instructions,
- le package *exceptions*.

3.3 Un modèle fusionnel

Le nouveau modèle d'interactions puise ses richesses au sein du modèle EJB et ceci a deux niveaux. Il utilise le modèle de persistance et de transactions des EJB pour améliorer la consistance du système. Il se sert des mêmes points d'entrée que les autres services pour s'intégrer de manière efficace. D'une certaine manière, le modèle d'interaction est client du modèle EJB et dans un même temps, étend ce dernier (est une extension).

Des objets d'interposition L'intégration du service au sein du modèle EJB s'est fait par le biais des objets d'interposition. Ces objets représentent le point de contrôle pour la réception des messages et permettent par voie de conséquence l'acquisition transparente de services au niveau du bean. Dans JOnAS, ils sont

⁵Voir les diagrammes de classes en annexe 1

⁶la partie gauche d'une règle d'interactions (i.e. le code à exécuter)

obtenus par l'outil de génération de code GenIC. Le code généré dépend des informations décrites dans le fichier de déploiement (écrit en XML[13]). Pour permettre l'acquisition du service d'interactions à la plate-forme de JOnAS, la grammaire XML sur laquelle est basé le fichier de déploiement a été étendue⁷.

L'intégration aurait pu se faire par modification du bean au moyen de la Méta programmation[7]. Cette technique est employée pour rendre interagissants des objets java quelconques. Dans le cas des EJB, ce n'est pas le meilleur moyen étant donné que toute la logique de composition de services est située au niveau des objets d'interposition.

Des objets interagissants Quelque soit le type d'objet (java pur, RMI, Enterprise Bean), des modifications de code doivent être réalisées pour permettre à l'objet de supporter les interactions. Dans le cas d'un objet java pur ou RMI, le code de l'objet est transformé par Meta programmation. D'autre part, l'utilisateur doit lui même spécifier que l'objet hérite d'une interface spécifique dont les fonctionnalités sont destinées au serveur d'interactions. Dans le modèle EJB, les beans ne sont pas modifiés, ce sont les objets d'interposition générés qui supportent les interactions. L'utilisateur n'a pas à faire hériter son bean de l'interface citée ci-dessus car c'est l'objet d'interposition qui implémente cette interface.

L'interface d'un objet interagissant⁸ comporte un minimum de méthodes permettant au serveur d'interactions de communiquer avec les objets enregistrés.

Gestion de la persistance Etant donné que les Entity beans[5, 12] sont des composants persistants, il semble logique que leurs interactions soient persistantes de la même manière. Le serveur d'interactions implémenté à l'aide d'enterprise beans a permis la persistance des données et par voie de conséquence une bonne gestion des interactions des Entity beans interagissants. D'autre part, la persistance des schémas d'interactions est une chose essentielle pour un système robuste. Certains objets du serveur d'interactions écrits à partir de beans sont donc eux même des Entity beans.

Le modèle de persistance du modèle EJB est basé sur les bases de données relationnelles. Tout un ensemble de problèmes complexes à résoudre sont liés à la correspondance entre le modèle objet de Java et le modèle relationnel. Des problèmes qui ne se poseraient pas si les objets étaient stockés dans des bases de données orientées objet. Ainsi, la plupart des objets constituant le serveur d'interactions ont pu être transformés en Entity beans⁹.

Lorsque le mapping objet-relationnel est impossible car trop lourd à mettre en oeuvre, la sérialisation devient une technique appréciable. Coté serveur, la sérialisation a été utilisée pour stocker les réactions associées à chaque modèle de règles. Elle a également été utilisée pour stocker les références aux objets interagissants. Coté client, le contexte d'interactions de chaque objet interagissant,

⁷ Ce travail a été effectué pendant le projet de fin d'études d'ESSI3

⁸ cf interface InterfaceISL en Annexe 1, figure 6

⁹ Le détail du mapping objet-relationnel mis en place se trouve en annexe 3

c'est à dire l'état de l'objet d'un point de vue "interactions", est sérialisé ce qui permet une gestion.

La conception d'Entity beans implémentant de multiples interfaces pose problème. En effet, toutes les méthodes accessibles aux clients doivent être déclarées dans l'interface RemoteInterface (dont l'objet d'interposition RemoteObject est l'implantation). Si on veut un bean offrant de multiples facettes, il est nécessaire de créer autant de Session beans que d'interfaces. Les Session beans délèguent ensuite la méthode à l'Entity bean qui possède les implémentations des méthodes de toutes les interfaces. Tous les Entity beans du serveur ont donc leur homologue en Session bean.

Des opérations transactionnelles Les transactions ont été utilisées à plusieurs niveaux. Elles rendent le système plus consistant. Une transaction est mise en place à chaque création ou destruction d'un schéma d'interactions. De cette façon, un schéma ne peut exister dans le système que si les éléments qui le constituent ont pu être correctement construits. Un schéma ne peut être détruit que s'il n'est plus instancié (i.e. toutes les interactions de ce schéma ont été détruites). D'autres transactions sont également nécessaires à la pose d'une interaction ainsi qu'à sa suppression. Une interaction existe dans le système que si toutes les règles de l'interaction ont été correctement¹⁰ posées.

L'utilisation de transactions devient très intéressante pour contrôler l'exécution. En effet, une transaction est embarquée non plus pour un enchaînement d'appel de méthodes selon le flot de séquences classique mais pour la communication inter-objets. Au lieu d'avoir un modèle de transaction locale à une règle d'interaction, on obtient un modèle de transaction globale. Ceci permet de conserver la consistance du système lors de la propagation des interactions.

Prenons un exemple, soit le schéma d'interactions :

```
interaction Achat(Buyer B, Account A) {
  1. B.buy(int x) -> A.retrieve(x); B.buy(x)
  2. A.retrieve(int y) -> A.checkAccount(y); A.retrieve(y)
}
```

Soit l'acheteur b1 et le compte en banque a1 liés par ce schéma. Supposons que le modèle de transaction locale à une règle soit utilisé. Chaque réaction associée à une règle est englobée par une transaction. A l'appel de la méthode *buy(2)* sur b1, le code exécuté est :

```
transaction1.begin();
transaction2.begin();
a1.checkAccount(2);
a1.retrieve(2);
transation2.end();
b1.buy(2);
transaction1.end();
```

¹⁰Si une fusion de règles échoue, l'interaction ne sera pas prise en compte

Si la transaction 2 est réussie et si *buy(2)* lève une exception alors la transaction 1 échoue mais les changements effectués par la transaction 2 sont irréversibles. Le système devient inconsistant car le compte est débité alors que l'acheteur n'a pas pu conclure son achat!

Supposons maintenant que le modèle de transaction utilisé soit le modèle global. Dans ce cas, une seule transaction est utilisée. A l'appel de la méthode *buy(2)* sur *b1*, le code exécuté est :

```
transaction1.begin();
a1.checkAccount(2);
a1.retrieve(2);
b1.buy(2);
transaction1.end();
```

Si *buy(2)* lève une exception alors la transaction 1 échoue. Le système reste stable car l'achat n'ayant pas eu lieu, le compte en banque n'a pas été débité.

3.4 Un environnement de programmation

La mise en oeuvre du modèle pour l'application visée doit offrir aux utilisateurs de la plate-forme une bibliothèque de schémas et d'objets adaptée à leurs problèmes. Les utilisateurs peuvent dynamiquement associer et dissocier à un groupe d'objets de leur choix des schémas d'interactions. Cette bibliothèque¹¹ peut être vue comme un ensemble de services extensible permettant aux utilisateurs de définir de nouvelles formes de communication selon leurs besoins, c'est à dire d'introduire de nouveaux services dynamiquement. Pour établir cette bibliothèque, il a fallu étendre le langage d'interactions ISL. Par exemple, l'introduction de l'"étoile" permet d'exprimer qu'une règle concerne toutes les méthodes métiers d'un objet. Une interface graphique a été conçue pour faciliter l'utilisation du service. Elle permet entre autre de créer à l'exécution de nouveaux schémas, de les instancier, de supprimer des schémas, des interactions ou encore de désenregistrer des objets. L'interface graphique a elle même été reliée au serveur d'interactions via des schémas d'interactions¹². Voici quelques captures d'écrans provenant de l'interface :

¹¹ cf. voir les schémas d'interactions en annexe 4

¹² voir schémas en annexe 5

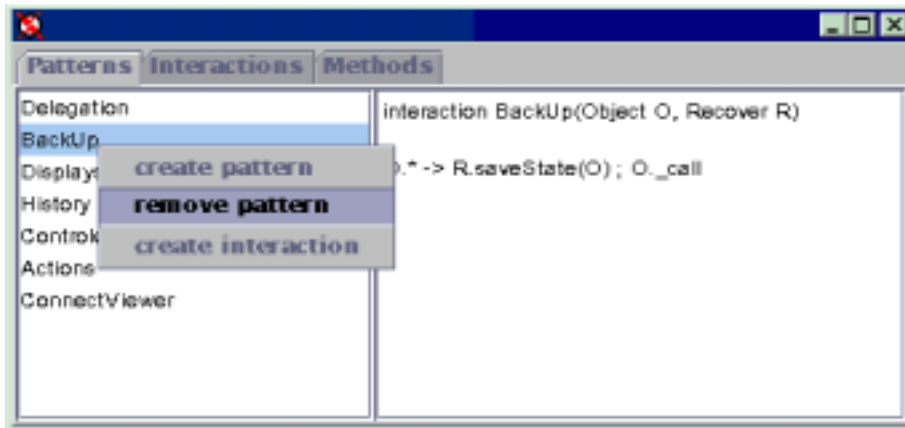


FIG. 2 – Vue schémas d'interactions

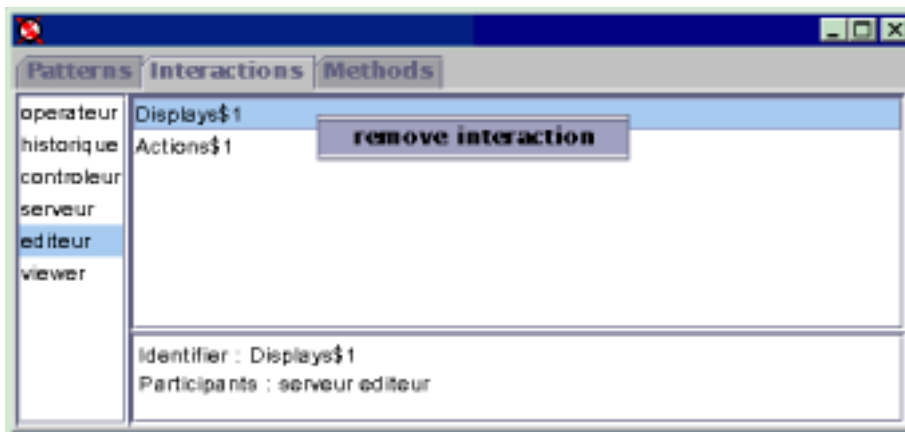


FIG. 3 – Vue interactions entre objets

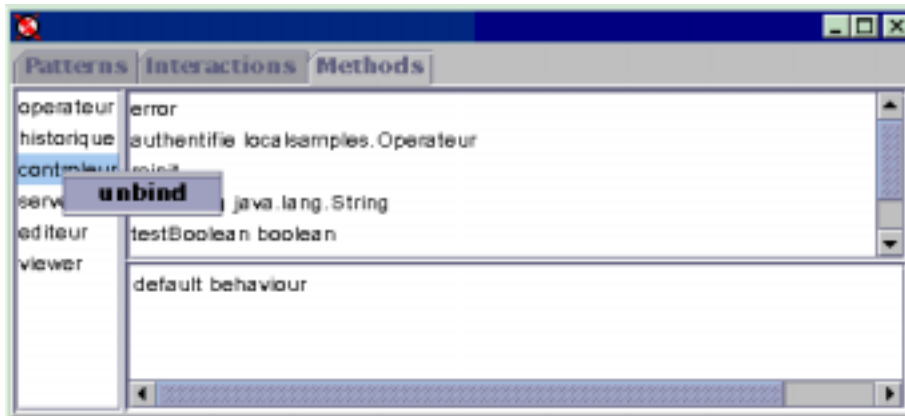


FIG. 4 – Vue méthodes et comportement associé

4 Conclusion

Le sujet était très intéressant et motivant sous différents aspects. Le projet était à la fois un projet de recherche et de développement.

Du point de vue recherche, il m'a permis d'appréhender de nouveaux concepts, de collaborer avec une équipe de chercheurs. Ainsi, j'ai participé à la rédaction d'articles, assisté à des conférences. J'ai également eu l'occasion de pressentir le métier d'enseignant lorsque j'ai pris en charge l'encadrement d'un groupe d'élèves d'ESSI2 chargé d'étendre le langage d'interactions.

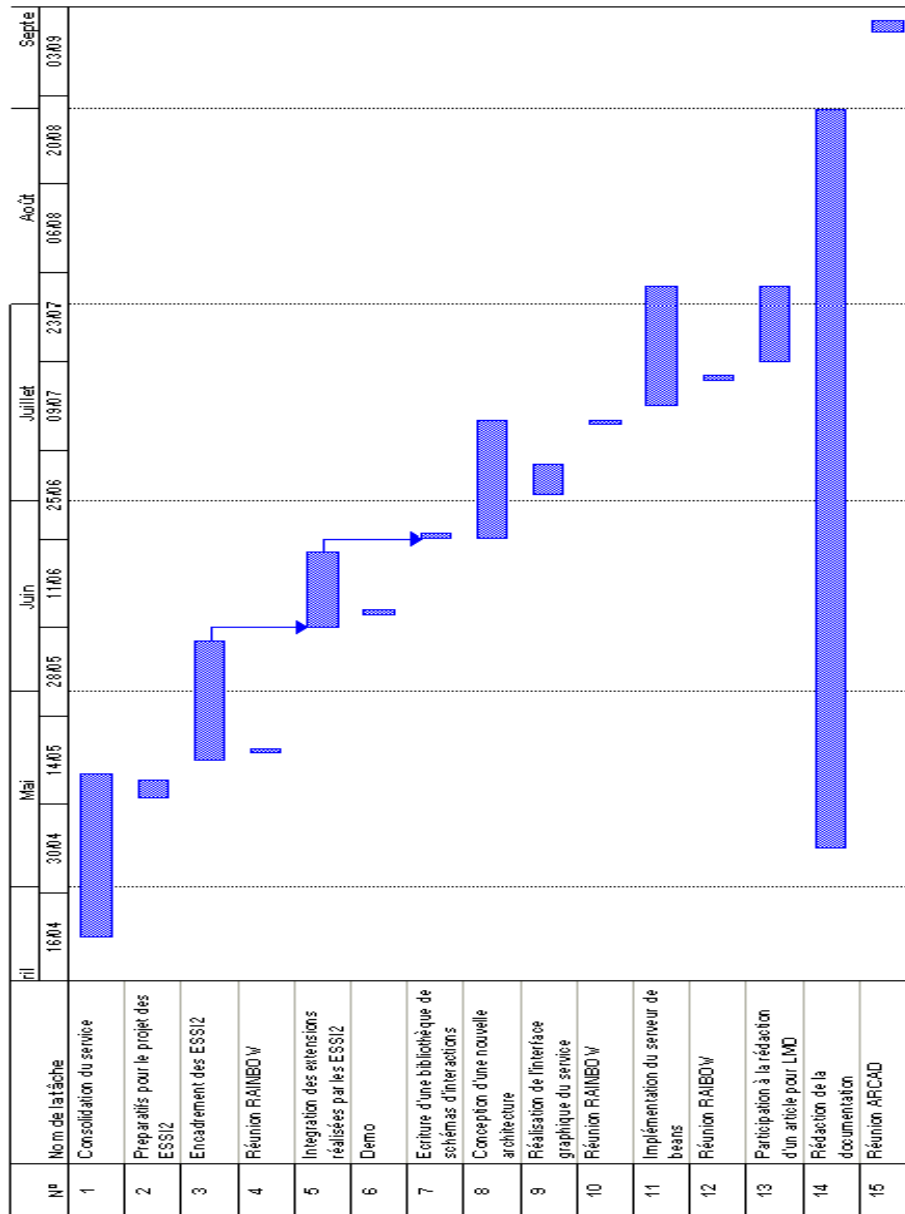
Du point de vue développement, ce projet a été enrichissant. Atteindre des objectifs est d'autant plus valorisant lorsque le résultat doit être livrable. Le modèle d'interactions enrichit les modèles de composants tels que le modèle EJB, les rend plus adaptables.

Ce travail s'inscrit dans le cadre des travaux développés par l'OMG[10] avec CORBA. Il les complète en exprimant de manière externe la communication entre les objets et en permettant l'adaptabilité dynamique de ses caractéristiques. Cela pourrait entraîner la soumission d'un service d'interactions à l'OMG.

Références

- [1] Site web consacré à la programmation par aspects en aspectj. <http://www.aspectJ.org>, 2001.
- [2] Architecture répartie extensible pour composants adaptables. <http://www.essi.fr/riveill/recherche/00-02-rntl-arcad>, 2001.
- [3] A.M. Dery and M. Blay. An interaction service : isl langage and the dico* architecture. ESSI3/I3S.
- [4] Object Management Group. The common object request broker : architecture and specification. version 2.1. <http://www.omg.org>, August 1997.
- [5] Sun Microsystem Inc. Enterprise javabeans specification. version 1.1. <http://java.sun.com/products/ejb/docs.html>, janvier 2000.
- [6] Sun Microsystem Inc. Site web sur rmi-iiop. <http://java.sun.com/products/rmi-iiop/>, 2000.
- [7] Openjava. <http://www.csg.is.titech.ac.jp/mich/openjava/>.
- [8] ObjectWeb. Jonas : javatm open application server. <http://www.objectweb.org/jonas>.
- [9] ObjectWeb. Objectweb home page. <http://www.objectweb.org/>.
- [10] Object management group. <http://www.omg.org>.
- [11] Rainbow. <http://www.essi.fr/rainbow/index.html>.
- [12] Ed Roman. *Mastering Enterprise Java Bean*. Wiley Computeur Publishing, 2000. à partir de la page 175.
- [13] Michel J. Young. *Formation à XML*. Microsoft, 2001. eXtensible Markup Language.

5 Annexe 1 : Le planning



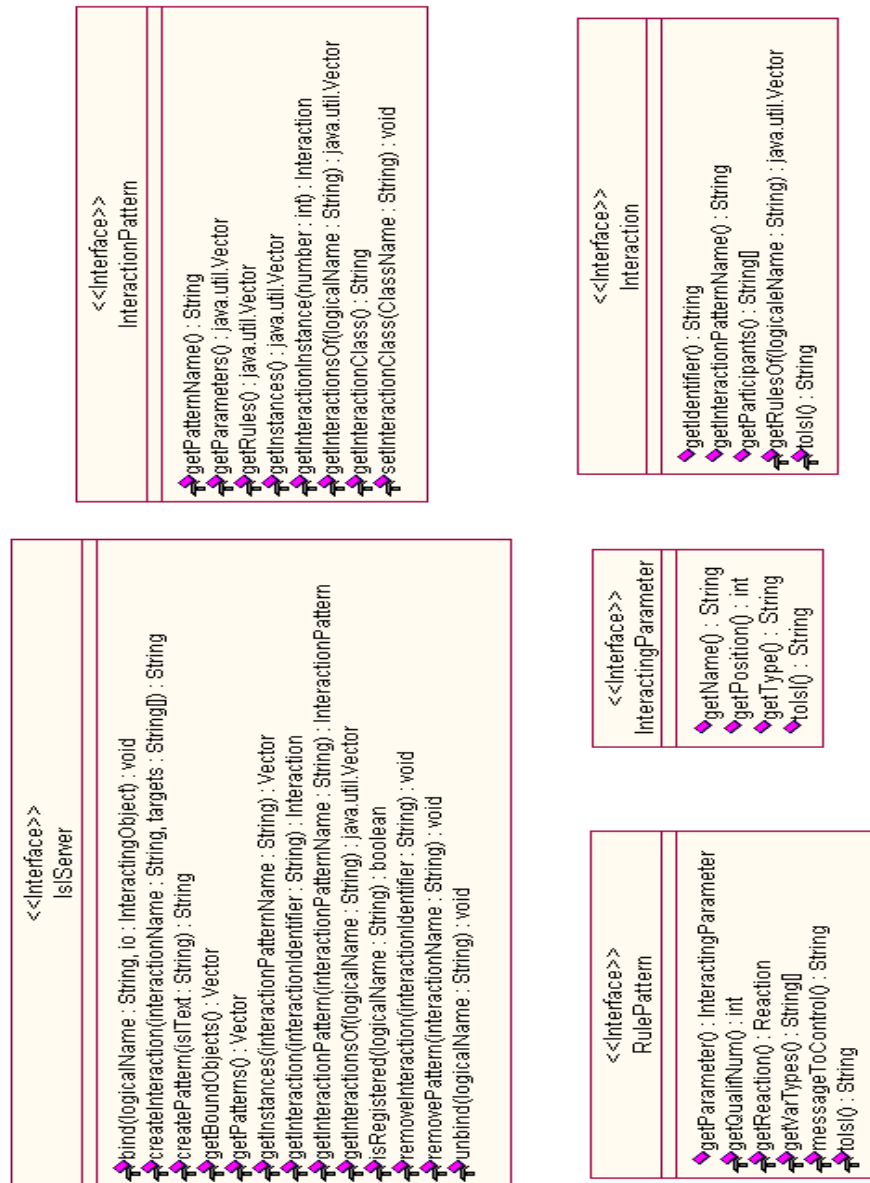


FIG. 8 – Le package server



FIG. 10 – Le package thread

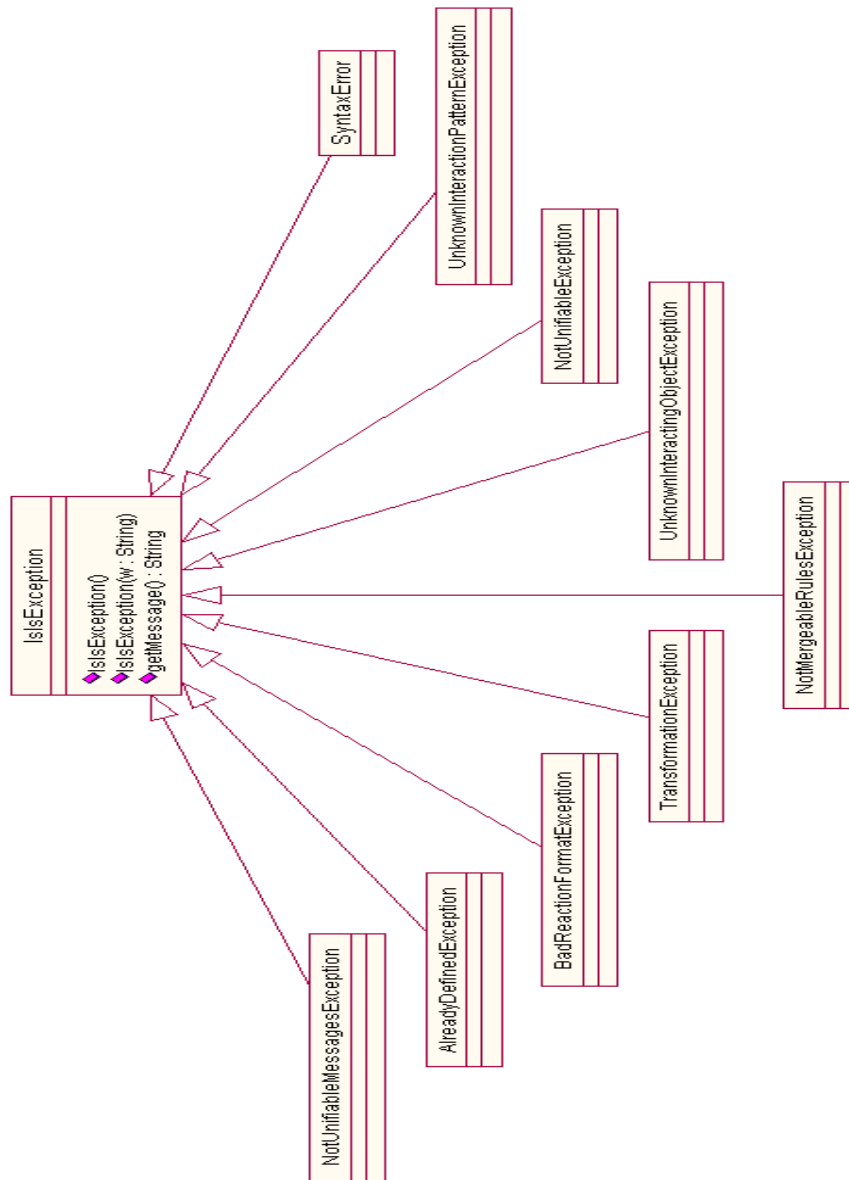


FIG. 11 – Le package exceptions

7 Annexe 3 : Mapping objet relationnel

Voici le script permettant de constituer les tables de la base de données (Oracle) du serveur d'interaction. A chaque table correspond un entity bean possédant les mêmes attributs persistants.

```
create table interactionpatterndata (
    patternname    varchar(30),    ninstance    integer not null,
    CONSTRAINT patternpk primary key (patternname));

create table interactingparameterdata (
    name           varchar(30),
    type          varchar(30),
    position      integer not null,
    patternname   varchar(30),
    CONSTRAINT parameterpk primary key (position, patternname),
    CONSTRAINT parameterfk foreign key (patternname) references
        interactionpatterndata(patternname));

create table interactiondata (
    logicalname   varchar(30),
    methodname   varchar(30),
    rulenumbers* integer not null,
    identifier**  varchar(30),
    islrule      varchar(255),
    patternname   varchar(30),
    CONSTRAINT interactionpk primary key
        (logicalname, methodname, rulenumbers, identifier),
    CONSTRAINT interactionfk foreign key (patternname) references
        interactionpatterndata(patternname));

create table patternsrelationship (
    patternname   varchar(30),
    extendedpattern varchar(30),
    CONSTRAINT relationshippk primary key (patternname, extendedpattern),
    CONSTRAINT relationshipfk foreign key (patternname) references
        interactionpatterndata(patternname));

create table registrydata (
    logicalname   varchar(30),    filename    varchar(30),
    CONSTRAINT registrypk primary key (logicalname));

create table islserverdata (
    servername    varchar(30),    crashed    integer,
    CONSTRAINT serverpk primary key (servername));
```

8 Annexe 4 : Bibliothèque de schémas

Historique :

L'objet H memorise les differents appels de methodes d'un objet O.

interaction History (Object O, Historique H) {
 O.* -> H.saveCall(_call); O._call
}

Sauvegarde :

L'objet R sauvegarde de l'etat courant d'un objet O et opère sa reinitialisation à partir de l'état sauvegardé après la fin de l'exécution de la méthode si celle-ci a levé une exception.

interaction Recoverable (Object O, Recover R) {
 O.* -> try {
 R.saveState(O); O._call
 } catch {exception e} {
 R.reinit(O)
 }
}

Sécurité :

L'authentificateur A demande à l'utilisateur de s'identifier avant de proceder à l'execution de la methode en cours sur l'objet O ou à l'envoi d'un message d'erreur.

interaction Security (Object O, Authenticator A) {
 O.* -> if A.authenticate(O, _call) then O._call
 else A.printError() endif
}

Synchronisation :

L'objet L permet la pose d'un verrou sur un objet O pour qu'il ne soit accessible que par une seule personne à la fois.

interaction Verrou (Object O, Lock L) {
 O.* -> if L.check() then
 L.lock(); O._call; L.release()
 endif
}

9 Annexe 5 : Schémas d'interactions de l'interface graphique

```
interaction Displays(server.IslServer ISL, gui.EditeurInterfaceRMI E) {
  E.getListPattern() ,
  java.lang.Vector v ->
  if E.hasChangedListPattern() then
    v := ISL.getPatterns(); E.setJListPattern(v) ;
    E.getListPattern() ;
    E.notifyChangedListPattern(FALSE)
  else E.getListPattern() endif
,
  E.getListObject1() ,
  java.lang.Vector v ->
  if E.hasChangedListObject1() then
    v := ISL.getBoundObjects(); E.setJListObject1(v) ;
    E.getListObject1() ;
    E.notifyChangedListObject1(FALSE)
  else E.getListObject1() endif
,
  E.getListObject2() ,
  java.lang.Vector v ->
  if E.hasChangedListObject2() then
    v := ISL.getBoundObjects(); E.setJListObject2(v) ;
    E.getListObject2() ;
    E.notifyChangedListObject2(FALSE)
  else E.getListObject2() endif
}
```

```
interaction Actions(server.IslServer ISL, gui.Editeur2 E) {
  E.connEtoM6(java.awt.event.ActionEvent arg1) ->
  E.setJLabel1(" Confirm pattern deletion?" ) ;
  E.setFocusOnPattern() ;
  E.openDialog2()
,
  E.connEtoM7(java.awt.event.ActionEvent arg1) ->
  E.setJLabel1(" Confirm object unregistration?" ) ;
  E.setFocusOnObject() ;
  E.openDialog2()
,
  E.connEtoM8(java.awt.event.ActionEvent arg1) ->
  E.setJLabel1(" Confirm interaction deletion?" ) ;
  E.setFocusOnInteraction() ;
  E.openDialog2()
}
```

```

,
E.connEtoM9(java.awt.event.ActionEvent arg1) ,
    java.lang.Vector v ->
    v := ISL.getBoundObjects(); E.getList1();
    E.setJList1(v) ; E.reinitList2() ;
    E.openDialog3()
,
E.connEtoM10(java.awt.event.MouseEvent arg1) ->
    E.list1ToList2()
,
E.connEtoM11(java.awt.event.MouseEvent arg1) ->
    E.list2ToList1()
,
E.connEtoM12(java.awt.event.ActionEvent arg1) ->
    ISL.createPattern(java.lang.String E.patternText()) ;
    E.notifyChangedListPattern(TRUE) ;
    E.connEtoM13(arg1)
,
E.connEtoM14(java.awt.event.ActionEvent arg1) ,
    server.InteractionPattern p ,
    server.Interaction i ->
    if E.hasFocusOnPattern() then
        p := E.selectedPattern() ;
        ISL.removePattern(java.lang.String p.getPatternName()) ;
        E.notifyChangedListPattern(TRUE)
    else if E.hasFocusOnObject() then
        ISL.unbind(java.lang.String E.selectedObject1()) ;
        E.notifyChangedListObject1(TRUE) ;
        E.notifyChangedListObject2(TRUE)
    else
        i := E.selectedInteraction() ;
        ISL.removeInteraction(java.lang.String i.getIdentifier())
    endif
    endif
    ;
    E.connEtoM15(arg1)
,
E.connEtoM16(java.awt.event.ActionEvent arg1) ,
    server.InteractionPattern p, java.lang.Vector v ->
    p := E.selectedPattern() ;
    v := E.selectedList2() ;
    ISL.createInteraction(java.lang.String p.getPatternName(), v) ;
    E.connEtoM17(arg1)
,
E.connEtoM18(java.awt.event.MouseEvent arg1) ,
    server.InteractionPattern p ->

```

```

        p := E.selectedPattern() ;
        E.setJEditorPanel1(java.lang.String p.toIsl())
    ,
    E.connEtoM19(java.awt.event.MouseEvent arg1) ,
        java.util.Vector v ->
        E.setJEditorPane2("") ;
        v := ISL.getInteractionsOf(java.lang.String E.selectedObject1()) ;
        E.setJListInteraction(v)
    ,
    E.connEtoM20(java.awt.event.MouseEvent arg1) ,
        server.Interaction i ->
        i := E.selectedInteraction() ;
        E.setJEditorPane2(java.lang.String i.toIsl())
    ,
    E.connEtoM21(java.awt.event.MouseEvent arg1) ,
        process.InteractingObject o ->
        o := ISL.lookup(java.lang.String E.selectedObject2()) ;
        E.setJListMethod(java.util.Vector o.getBussinessMethods())
    ,
    E.connEtoM22(java.awt.event.MouseEvent arg1) ,
        process.InteractingObject o, java.lang.String merge ->
        o := ISL.lookup(java.lang.String E.selectedObject2()) ;
        merge := o.getBehaviorOn(java.lang.String E.selectedMethod()) ;
        E.setJEditorPane3(merge)
}

```