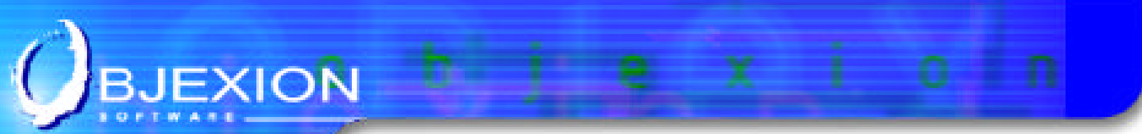


SA au capital de 500 000 F
Siret 421 565 565 00015
APE 722Z
Téléphone : 03 89 35 70 75
Télécopie : 03 89 35 70 76

L'embarcadère
5, rue Gutemberg
68 800 Vieux-Thann, France

Création d'un langage d'action pour un logiciel MDA.

Frédéric FONDEMENT
DRT GEII – 2nde année

Michel HASSENFORDER
Responsable DRT

Philippe STUDER
Directeur Technique - ObjeXion Software
Maître de Stage

Sommaire

Introduction p. 2

I - Netsilon p. 3
- Présentation p. 3
- Modélisation p. 4

II - Xion p. 6
- Qualités requises p. 6
- Chaîne de compilation p. 7
- Contrôle des Types p. 8
- Arbre Concret p. 10

III - Optimisation SQL p. 12
- Problème de performances p. 12
- Solution adoptée p. 14
- Exemple p. 18

IV - Editeur de code p. 20
- Un nouveau langage p. 20
- Coloration lexicale p. 21
- Complétion sémantique p. 22

V - Gestion automatique des objets Métier p. 23
- Gestion des données p. 23
- Réalisation p. 24

Conclusion p. 26

Références p. 27

Introduction

Le Diplôme de Recherche Technologique (DRT) est un diplôme de troisième cycle. Il sanctionne des travaux de recherche technologique d'une durée de dix-huit mois visant à la résolution d'un problème relevant du secteur industriel.

ObjeXion Software est une entreprise informatique dont l'objectif est de développer et de commercialiser une ligne de produits dans le domaine de l'aide à la modélisation objet des systèmes d'information.

Model Prototyper est un logiciel de prototypage de modèles UML d'ObjeXion. Il permet de mettre au point des modèles métier, sans programmation. A partir de diagramme de classes est générée (ou mise à jour) une base de données relationnelle chargée du stockage des objets métier, ainsi que de leurs champs (attributs, relations). Un second logiciel, FacSimile, permet lui de nourrir cette base, par la création et l'édition d'objets. Au-delà des diagrammes de classe, UML définit un langage de contraintes, l'Object Constraint Language – OCL. La première partie de mon stage consista donc à développer un interpréteur pour ce langage. Ceci m'a permis d'en comprendre les intérêts et les limites. Une précédente présentation traite de cette partie, c'est pourquoi elle ne sera pas développée dans le présent document.

Fort de cette technologie ObjeXion décida de créer une second logiciel, Netsilon, qui utilise cette technique de stockage d'objets au sein d'une application internet. La description de l'application se base donc également sur des diagrammes de classe pour la description du modèle métier, mais aussi sur un nouveau type de modélisation, le modèle de navigation, décrivant les relations entre pages HTML ainsi que les interactions avec les objets métiers. A partir de ces seuls éléments, Netsilon se charge de créer (ou maintenir) la base de données relationnelle ainsi que les programmes (scripts PHP, JSP ou Servlets) décrivant ce site, donc utilisant cette base.

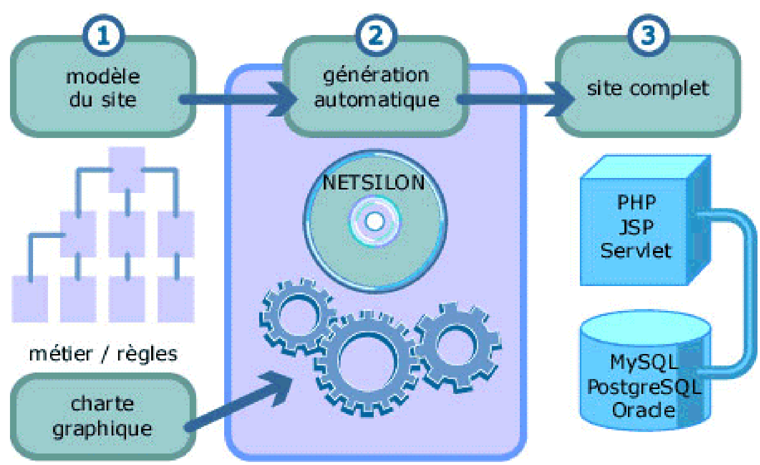
Outre la réalisation de ce nouveau modèle de navigation, il était nécessaire de créer un langage d'action chargé de décrire les méthodes métier ainsi que les interactions du modèle de navigation. Ce fut là la seconde partie de mon stage, l'expérience acquise avec OCL me permettant de mieux cerner les nécessités d'un langage niveau modèle. En plus de la syntaxe et des règles sémantiques, il me fallut réaliser un compilateur efficace, ainsi qu'un éditeur de code simplifiant la production. Enfin, j'ai eu l'occasion de créer une interface de gestion des objets métiers, à l'instar de FacSimile.

Dans une première partie, nous décrirons plus précisément le logiciel sur lequel j'ai travaillé, Netsilon, puis nous poursuivrons par une présentation de ce nouveau langage d'action, ainsi que la réalisation du compilateur. Une troisième partie traitera d'une technique dont le but est de rendre les accès à la base plus efficaces. Une quatrième partie montrera la réalisation de l'éditeur de code. Enfin, la cinquième partie décrira l'interface de gestion des objets métier.

I - Netsilon

Présentation

Netsilon est un environnement de développement complet pour la modélisation et la réalisation d'applications web dynamiques de troisième génération, c'est-à-dire d'applications informatiques accessibles par l'intermédiaire d'un navigateur Internet, et dont le contenu et la forme des pages web peuvent changer en fonction de données ou de règles.



Fonctionnement général de Netsilon

Netsilon automatise la fabrication des logiciels qui s'exécutent du coté serveur : configuration et maintenance des bases de données telles Oracle ou MySQL, génération et maintenance des scripts comme PHP ou Java. Les éléments graphiques (HTML, Flash, images animées…) manipulés par les scripts générés par Netsilon sont créés à l'extérieur de Netsilon, de manière traditionnelle, avec les logiciels du marché (Dreamweaver, GoLive…) puis importés dans les projets gérés par Netsilon.

Netsilon comprend un outil de modélisation qui permet de représenter de manière graphique (avec la notation UML, Unified Modeling Language) les objets et les règles métiers, ainsi que le schéma de navigation (l'enchaînement des pages) qui composent une application web, puis de générer automatiquement 100 % du code correspondant à cette description, dans un environnement de déploiement web donné. La persistance des données est assurée par une base de données que le logiciel se chargera de formater, et le code généré d'utiliser : le programmeur n'a plus à s'en occuper.

Les modèles assurent une parfaite séparation du QUOI (type de contenu et cheminement dans ce contenu) du COMMENT (ingénierie informatique). La forme (l'aspect visuel) est intégrée sous la forme de fragments HTML, composés entre eux de manière dynamique lors de l'exécution de l'application.

Je vous invite à consulter l'annexe I (p. 2) et II (p. 4) pour de plus amples renseignements sur l'utilité et le fonctionnement de Netsilon. L'annexe III (p. 9) décrit une prise en main de l'outil.

Modélisation

Netsilon est un logiciel qui respecte le paradigme MDA, c'est à dire qu'il est capable de générer des applications web, indépendamment des technologies de déploiement à partir d'une modélisation du site.

Le modèle métier décrit les fonctions internes du site. Il est donné par des diagrammes de classes UML, il est donc nécessaire de connaître la modélisation objet. Nous vous invitons à lire des ouvrages comme *Modélisation objet avec UML* (ed. Eyrolles) de P-A.Muller et N.Gaertner, ou à visiter le site http://www.omg.org/uml/ pour appréhender ou compléter vos connaissances sur cette technique.

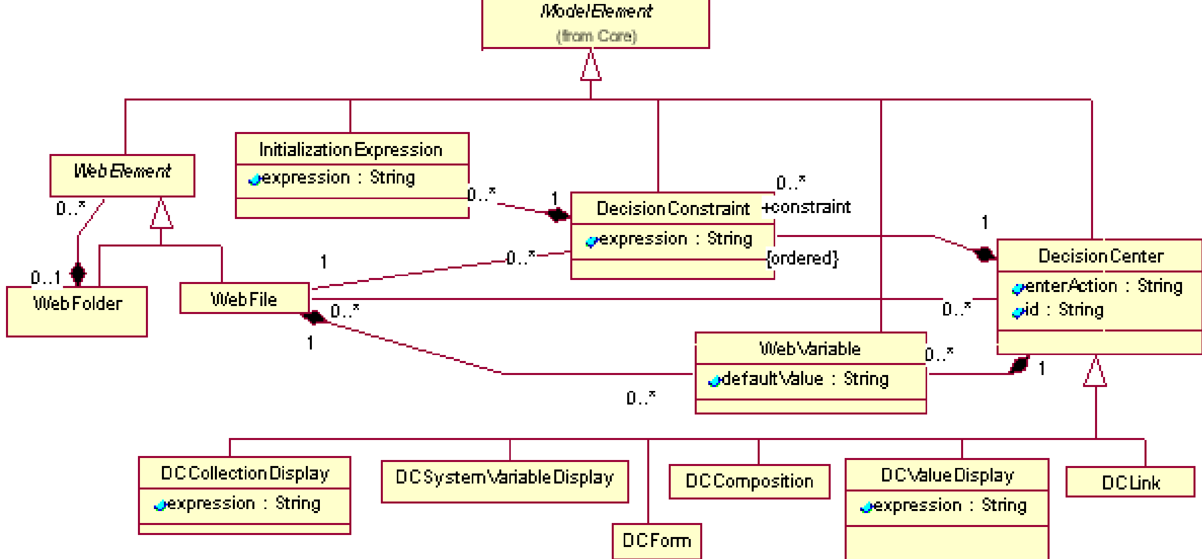
Le modèle de navigation est responsable de la cohésion entre la forme (donnée par une collection de fichiers HTML) et le modèle métier. Cette navigation est décrite par des extensions au métamodèle UML. Par exemple, chaque fichier du projet correspond à un objet instance de la classe *WebFile* du métamodèle étendu. La classe *WebFile* sera dite métaclasse. Un fichier est lié à un autre par l'intermédiaire de centres de décision (*DecisionCenter*) de différentes sortes. Au moment voulu, ces centres de décision choisiront une de leur décision (*DecisionConstraint*) suivant un critère donné (attribut *expression*) ; cette décision conduira alors au fichier lié. Les centres de décision possibles sont :

- les lieurs (*DCLink*) pour un lien hypertexte,
- les composeurs (*DCComposition*) qui incluent un fichier dans un autre,
- les afficheurs de collection (*DCCollectionDisplay*) qui itèrent sur une collection dont l'expression est donnée par le paramètre *expression*,
- les formes (*DCForm*) qui travaillent sur des formulaires HTML,
- les afficheurs de valeur (*DCValueDisplay*) qui affichent la valeur résultat (sans faire de lien à un autre fichier) de l'évaluation d'une expression contenue dans le paramètre *expression*,
- les afficheurs de valeur système (*DCSystemValueDisplay*) qui ne font pas non plus de lien avec un autre fichier.

La plupart de ces centres de décision peuvent également effectuer des opérations lors de leur appel par exemple lors du clic sur un lien pour un lieur, avant la composition pour un composeur... Cette opération est donnée par l'expression contenue dans le paramètre *enterAction* de la métaclasse *DecisionCenter*.

On peut également noter l'existence de variables (*WebVariable*). Ces variables sont liées à l'exécution d'un fichier ou d'un centre de décision (ainsi que ses décisions). Il s'agit en fait de paramètre qui peuvent avoir une valeur par défaut (attribut *defaultValue*), consultables et modifiables à volonté. On remarquera que ces paramètres peuvent également être transmis par une décision appelante (*InitializationExpression*).

Voici un extrait de l'extension du métamodèle adapté correspondant au modèle de navigation :



Extension au métamodèle UML : Centres de Décision

Je vous invite à consulter l'annexe III (p. 9) pour avoir un aperçu de l'utilisation de l'outil.

Cette description nous conduit alors à une réflexion sur la puissance d'UML. Il n'est en effet pas assez riche pour décrire des opérations complexes de manière précise. Existent les diagrammes d'état, de collaboration et de séquence mais en aucun cas ils ne représentent de façon exhaustive un protocole opératoire. Or nous avons besoin, pour réaliser Netsilon, d'une représentation de ces opérations complexes. On peut parler de l'implémentation des méthodes du modèle métier, des actions en entrée des centres de décision, de l'expression des valeurs utilisées dans les centres de décision afficheurs de valeur et de collection, de l'initialisation des variables, par défaut ou par transmission… Décision fut prise d'adopter une représentation textuelle de ces opérations, soit un langage de programmation, plus classique et expressif que les solutions proposées par UML.

Il est alors pertinent de remarquer q'UML fournit un langage textuel : l'Object Constraint Language – OCL. Ce langage – proposé par UML lui-même – est alors parfaitement adapté à toutes les subtilités d'UML contrairement à d'autres langages, comme C++, Java, Smalltalk, etc.… Cependant OCL, par essence, ne peut décrire que des contraintes. De plus, il s'agit d'un langage sans effet de bord, c'est à dire qu'il est incapable de modifier quoi que ce soit au sein des objets ou même des variables.

Les autres langages objets existants, comme C++, Java, Ada, etc. , ne semblent pas non plus convenir. En effet, ils offrent des possibilités incompatibles avec UML (comme les pointeurs) et ne supportent pas certaines constructions comme les associations navigables, les classes-association, les multiplicités…

En cas de doute, je vous invite à consulter le site http://www.umlactionsemantics.org/.

La tâche qui m'a été confiée fut de créer un nouveau langage textuel pour la définition des actions. De plus, il m'incombait de créer les premières couches du compilateur (analyses syntaxiques, lexicales, contrôle des types, un début d'optimisation), ainsi qu'un éditeur de code simplifiant la phase d'apprentissage de ce nouveau langage.

II - Xion

Qualités requises

Tel que nous l'avons décrit au chapitre précédent, un langage d'action est nécessaire pour la modélisation précise d'une application web. Nous avons décidé de l'appeler Xion.

Ce langage doit répondre à plusieurs attentes, et principalement :

- la description de protocoles opératoires complexes avec effet de bord,
- la navigation possible dans tout le modèle métier (classes et relations),
- pouvoir être traduit dans les langages ciblés par que Netsilon, soient Java, JSP et PHP.

De plus, ce langage doit être le plus simple possible à appréhender.

Plusieurs solutions existent déjà pour chacun de ces points, aucune cependant capable de répondre à tous. Java, par exemple, peut répondre au premier point, est déjà bien connu de beaucoup de monde et est un bon langage objet. OCL, quant à lui, est capable de répondre au second. Ces deux langages sont de plus reconnus comme d'excellentes solutions dans leurs spécialités respectives. Nous avons donc décidé que Xion serait une dérivation des deux, en les limitant au strict nécessaire, d'une part car certaines fonctionnalités ne correspondent pas au problème posé (définition de classe alors qu'elle doit être présente dans le modèle métier, définition d'une contrainte invariante, etc.), et d'autre part pour demeurer compilable (PHP ne définit pas de gestion des exceptions par exemple).

La syntaxe de ce nouveau langage est donc celle de Java diminuée de :

- les définitions de paquetage, d'imports, de classes, d'interfaces, d'attributs, d'opérations
- les type prédéfinis (`boolean`, `char`, etc.) ainsi que les classes prédéfinies (`java.lang.Object`, tableaux, etc.),
- les déclarations de classes à la volée

```
new ActionListener(){public void actionPerformed(ActionEvent ae) {}};
```

- la gestion des exceptions (clause `try … catch`, instruction `throw`),
- le nommage d'une instruction,
- `break`, `continue`, les retours d'affectation (`a=b=c`) qui sont des sources d'erreurs de programmation,
- `switch` qui ne serait pas généré de façon efficace,
- la gestion transactionnelle (`synchronized`) que PHP ne connaît pas,
- le décalage non signé (`>>>`),
- certaines constructions qui perdent leur sens (`.class`, `.this`, `.new`, `.super`).

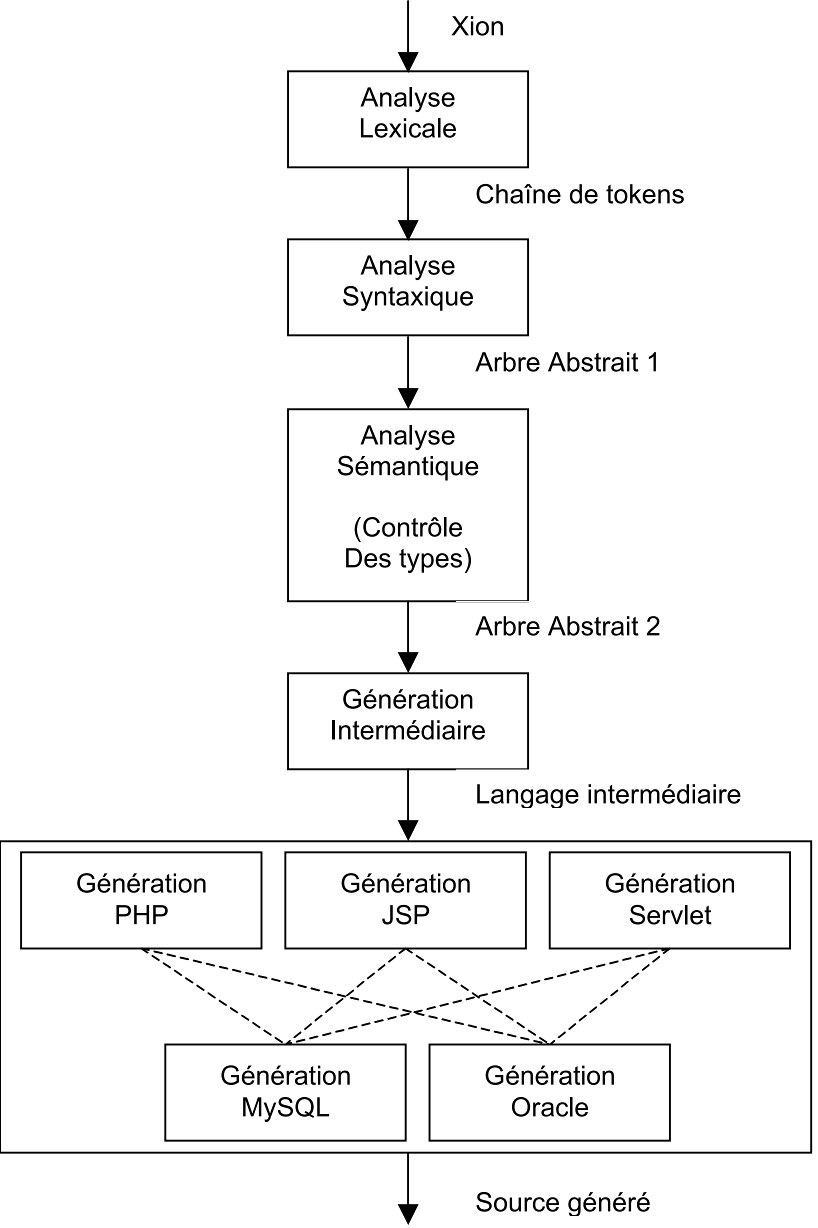
Les classes prédéfinies deviennent alors celles d'OCL, ainsi que la navigation dans le modèle (opérateurs '`.`' et '`->`', types de retour des associations). A ces classes prédéfinies nous avons ajouté `Date` et `Time`, ainsi que `OclObject` qui remplacera `java.lang.Object`.

Je vous invite à lire l'annexe IV (p. 20) pour découvrir la grammaire de ce nouveau langage, et annexe V (p. 24) pour en avoir les spécifications fonctionnelles, ainsi que l'annexe VI (p. 48) pour une liste exhaustive des classes et opérations prédéfinies.

Chaîne de compilation

Netsilon est écrit en Java, pour des raisons de portabilité (write once, run everywhere), et des raisons de qualité de programmation (pas de pointeur ni de gestion mémoire). Il était donc nécessaire d'avoir un compilateur Java. Nous avons choisi d'utiliser le compilateur de compilateur ANTLR. Je vous invite à consulter l'annexe VIII (p. 71) pour en savoir plus sur cet outil. Ce choix vient du fait qu'il s'agit d'un produit « Open Source », écrit en Java et générant en Java. Il est donc possible de l'intégrer à Netsilon (l'analyseur généré a besoin des classes ANTLR), voire de le modifier, notamment pour l'internationalisation des messages d'erreur.

Netsilon doit être capable de générer en plusieurs langages, et pour différentes bases de données, d'où la nécessité de rester le plus générique possible. C'est pourquoi la génération se fait dans un langage intermédiaire de bas niveau, qui est traduit ensuite dans le langage cible et pour la bonne base.



Chaîne de compilation du code Xion

Ma responsabilité était de fournir les analyseurs lexicaux, sémantiques et syntaxiques et un parcours de l'arbre généré, squelette de la génération intermédiaire.

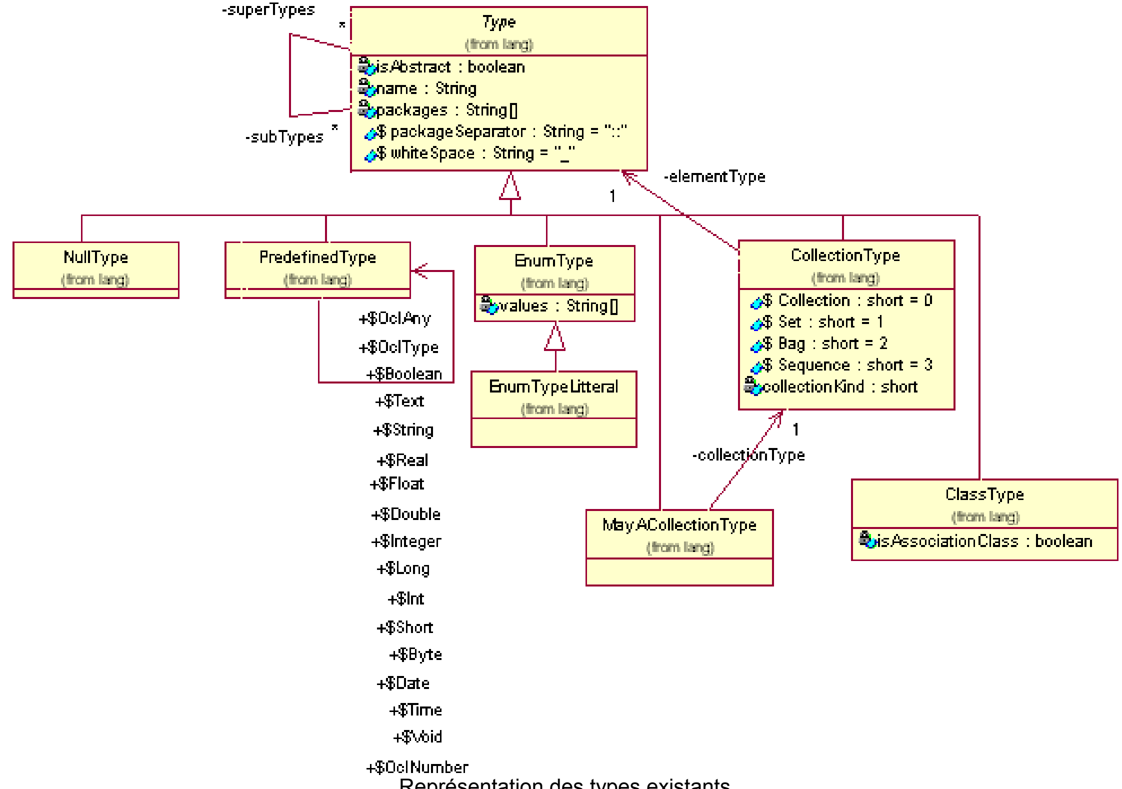
Ce schéma, pour les 4 premières étapes, se justifie principalement par le fonctionnement d'ANTLR. Je vous invite à vous reporter à l'annexe VIII (p. 71) pour de plus amples détails.

L'analyse lexicale et syntaxique, tout comme l'arbre intermédiaire 1, est générée simplement à partir de la grammaire Xion (fournie en annexe IV p. 20) grâce à ANTLR. Un « parseur d'arbre » ANTLR va alors analyser ce premier arbre. C'est à cet endroit que se situe le gros du travail. Il est en effet chargé d'une part du contrôle des types et d'autre part de la transformation de l'arbre intermédiaire 1 en un second arbre intermédiaire, beaucoup plus détaillé, sensé être compilable sans erreur, c'est à dire qu'à ce stade, toutes les erreurs de code devront avoir déjà été signalées.

La génération intermédiaire est un second parseur d'arbre ANTLR qui va le transformer en un langage intermédiaire, lequel ne concerne pas cette présentation.

Contrôle des types

Les types disponibles du langage sont les types prédéfinis, accompagnés de leurs opérations, ainsi que les types décrits par le modèle métier. Les objets métier décrivant cette dernière partie sont des instances du métamodèle UML. Or ce métamodèle est peu adapté à un contrôle de types. En effet, les mécanismes de recherche d'opération, de recherche de types, etc., n'y sont pas simples. De plus, les types prédéfinis ainsi que leurs opérations ne s'y trouvent pas. Enfin, certaines incohérences peuvent être données par l'utilisateur (héritage circulaire…). Vu leur nombre, il n'est pas pensable de les traiter dans l'analyse au cas par cas. J'ai donc dû développer un nouveau métamodèle. Une analyse préalable réalise la transformation de la description UML en cette nouvelle description, tout en vérifiant sa validité. Il s'agit d'une photographie du modèle métier, visible pendant toute la génération.



Représentation des types existants

Type est la métaclasse abstraite pour tous les types. Elle décrit les relations d'héritage entre les différents types, les noms des paquetages, le nom du type lui-même, s'il est abstrait et liste ses champs. On y trouve notamment des opérations de recherche de champs sur critère, d'équivalence entre types…

NullType représente le type de la valeur `null`.

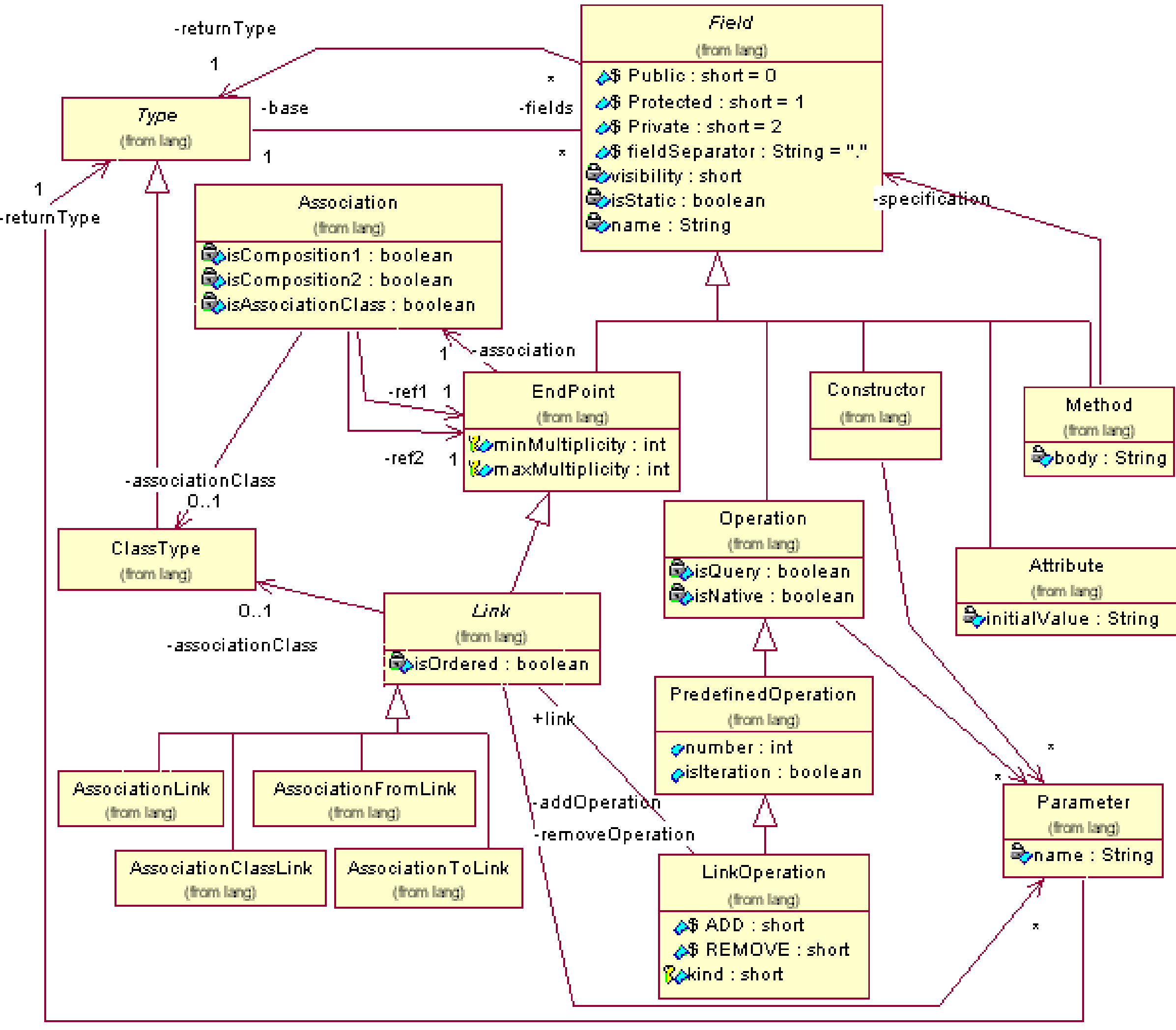
PredefinedType représente la métaclasse de tous les types prédéfinis du langage, tels `OclAny`, `Integer`…

EnumType est la métaclasse pour les types énumérés. Elle retient la liste des valeurs possibles pour cet énuméré. *EnumTypeLitteral* correspond au type d'une valeur littérale d'énuméré. En effet, les littéraux d'énumérés se font par `#ValeurEnumeree`. Or, lors de la phase du contrôle de type, il est impossible de savoir à quel type énuméré il appartient, car deux types énumérés différents peuvent partager un même nom de valeur.

CollectionType s'applique aux collections. Il définit son genre (`Collection`, `Set`, `Bag` ou `Sequence`) ainsi que le type sur lequel il s'applique.

Une lecture attentive des spécifications de Xion indiquera qu'un lien de multiplicité 0..1 sera vu indifféremment comme un type simple ou un type collection. Ceci signifie que par défaut le type de retour de l'appel d'un tel lien est simple, mais si on y appelle une opération prédéfinie sur la collection (par l'intermédiaire de l'opérateur '->'), il sera vu comme une collection. C'est le rôle d'un type de métaclasse *MayACollectionType*, qui simulera le comportement du type simple, mais sur lequel le contrôle des types permettra d'appeler des opérations du type collection auquel il se réfère.

Enfin, *ClassType* représente la métaclasse de tous les types définis dans le modèle métier.



Représentation des champs des types existants

Chaque type possède une série de champs, représentés par la métaclasse abstraite *Field*. Un champ a un certain nom, une certaine visibilité, un certain type de retour, et peut être statique.

Attribute est la métaclasse pour les attributs, lesquels peuvent avoir une valeur par défaut.

Constructor est la métaclasse de l'opération qui sera appelée à la construction d'un objet. Il peut avoir un certain nombre de paramètres.

Operation représente toute opération disponible dans le langage, soient les opérations prédéfinies et les opérations du modèle métier. Une opération possède aussi une liste de paramètres. Elle peut être marquée comme statique ou native.

Les opérations prédéfinies sont représentées par *PredefinedOperation*. Chacune d'elles correspond à un élément d'arbre abstrait (n°2), dont le type est donné par un certain numéro. Une opération prédéfinie peut être une opération d'itération, c'est à dire opérant sur une collection et appliquant une expression élément par élément, comme `select` ou `collect`. Ce marqueur indique à l'analyseur sémantique s'il est possible de nommer une variable d'itération (`collection->select(i : i.aGarder())`), et au contrôle de type le type où rechercher l'opération (`collection->select(aGarder()`). La plupart des opérations prédéfinies sont définies de manière statique, telle `Integer.max(Integer)` qui gardera quoi qu'il arrive la même signature. On peut remarquer, en lisant attentivement la documentation, que certaines opérations sur les collections dépendent du type sur lequel elles s'appliquent, ce qui est typique d'OCL dont Xion est issu. Par exemple l'opération `getOne`, qui retourne un élément au hasard dans une collection. Il est nécessaire que le contrôle des types trouve que `getOne` (cf. annexe VI p. 51) retourne un entier sur une collection d'entiers. Ces opérations sont redéfinies sur chaque nouveau type collection avec la signature qui convient. D'autres opérations posent un problème encore plus complexe : leur type de retour, voire leur existence même, est remis en cause par le type de leur argument. Il s'agit de

`OclType.allInstances` (annexe V p. 73) qui retourne une collection différente suivant le type sur lequel elle est appelée, `OclAny.OclAsType(OclType)` (p. 28) qui retourne une valeur du type qui lui est passé en argument, `Set.collect` `Bag.collect` et `Sequence.collect` (p. 42) qui retourne une collection sur le type de retour du paramètre, `Collection.sortedBy` (annexe VI p. 51-52) qui n'existe que si son argument a un type définissant l'opération < sur lui-même. Aucune solution générique n'a été trouvée, cependant ces exceptions sont assez peu nombreuses pour les traiter au cas par cas dans le contrôle des types.

Il existe un deuxième type d'opération prédéfinie qui permet d'ajouter ou de retirer des liens lorsque ces derniers sont multiples (cf. annexe V p. 45). Ces opérations sont des *LinkOperation* avec un certain genre (ajout ou retrait) s'appliquant sur un lien donné.

Une association entre deux classes est décrite par *Association*. Chacune de ces deux classes verra l'autre à travers un champ qui sera un *EndPoint*. L'association peut être reliée par une classe qui sera la classe-association en question. L'*EndPoint* a une multiplicité minimum et maximum. Un *Link* est un *EndPoint* dont la navigabilité permet à la classe de base de le voir.

Il existe quatre sortes de liens. Les liens simples sont des *AssociationLink*. Les liens d'une classe vers une autre à travers une classe-association est une *AssociationClassLink*. Une *AssociationToLink* joint une classe de base vers la classe-association et une *AssociationFromLink* joint la classe-association vers une classe de base.

Cette architecture permet au parseur d'arbre chargé du contrôle des types de trouver simplement des types par leurs noms, de vérifier des compatibilités entre types, de rechercher des champs ainsi que leurs droits d'accès (attributs, liens, opérations, constructeurs) avec leur signature.

Arbre concret

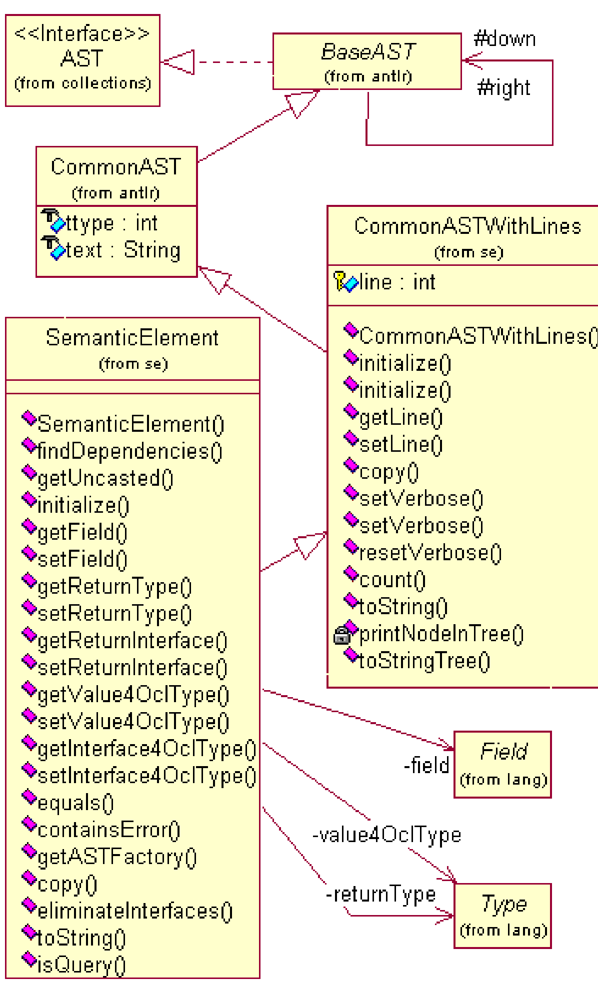
Cet arbre est une description complète et simplifiée de l'instruction ou de l'expression Xion analysée. La liste des nœuds de l'arbre abstrait est donnée en annexe VII p. 68. On y trouve notamment :

- des littéraux (un pour chaque type prédéfini sauf Date et Time), comme *SE_LitteralInteger* ou *SE_LitteralString*,
- les appels aux variables prédéfinies *SE_This* ou *SE_Litteral_null*,
- l'appel à variable *SE_VariableCall*, associé à la provenance de cette variable donné par *SE_ FromIteratingDefinition*, *SE_ FromLocalDefinition*, *SE_ FromParameter*, *SE_ FromDecisionCenterContext*, *SE_ FromWebFileContext*, *SE_ FromSessionContext* ou *SE_SIDVar*,
- les appels à champs *SE_CallAttribute*, *SE_CallLink*, *SE_CallConstructor*, *SE_CallOperation*,
- l'assignation *SE_Assign*,
- les structures de contrôle *SE_Block*, *SE_While* et *SE_If*,
- le retypage *SE_Cast*,
- les instructions particulières *SE_Return*, *SE_Write*, *SE_WriteRaw*, *SE_DestroySession*, *SE_DoNothing*.

De plus, chaque opération prédéfinie possède un certain token propre. Par exemple l'opération prédéfinie `Real.max(Real)` correspond au token *SE_Real_max*.

La génération de l'arbre concret se doit de transformer la représentation sous forme d'arbre du code source en un arbre constitué de ces différents types de nœud. Cette première représentation n'est pas décrite dans ce document mais est une représentation très proche du code source entré. On remarquera que l'arbre généré ne dispose pas de certaines fonctionnalités du langage. Les instructions do..while, for, l'incrémentation (++ et --), les assignations composées (+=, *=, etc.) et les opérateurs (+, -, *, /, etc.) sont donc à construire à partir des éléments disponibles. De plus, il est nécessaire d'y ajouter certaines informations telles que le rendu explicite des casts implicites, le type de retour exact d'une expression, le champ appelé, etc…

L'arbre abstrait adopté est dérivé d'ANTLR, ce qui permet d'implémenter la génération en langage intermédiaire par un nouveau parseur d'arbre (cf. annexe VIII p. 71). Il a cependant été modifié pour sauvegarder ces dernières informations.



Arbre abstrait

AST, *BaseAST* et *CommonAST* proviennent d'ANTLR. Une première classe *CommonASTWithLines* permet de retrouver la ligne qui a généré le nœud. Elle offre également des opérations qui permettent le copiage complet, de compter le nombre de nœuds et d'imprimer (sous forme textuelle) un arbre. C'est la classe des nœuds de l'arbre abstrait 1.

SemanticElement a été spécifiquement développé pour l'arbre abstrait décrit ci-dessus. Il retient le type de retour pour chaque nœud (*returnType*). Le cas échéant, il peut aussi retenir le champ appelé (*field*). Dans le cas d'un littéral type, il retient également la valeur de ce type. En outre, des opérations permettent de trouver les dépendances de l'arbre (ce qui permet une compilation partielle lors de changements), de comparer deux arbres, de savoir si l'arbre est avec ou sans effet de bord…

On comprend alors bien pourquoi la génération de l'arbre concret en phase d'analyse sémantique se fait en même temps que le contrôle de types. Ces informations sont ensuite réutilisées en phase de génération de langage intermédiaire.

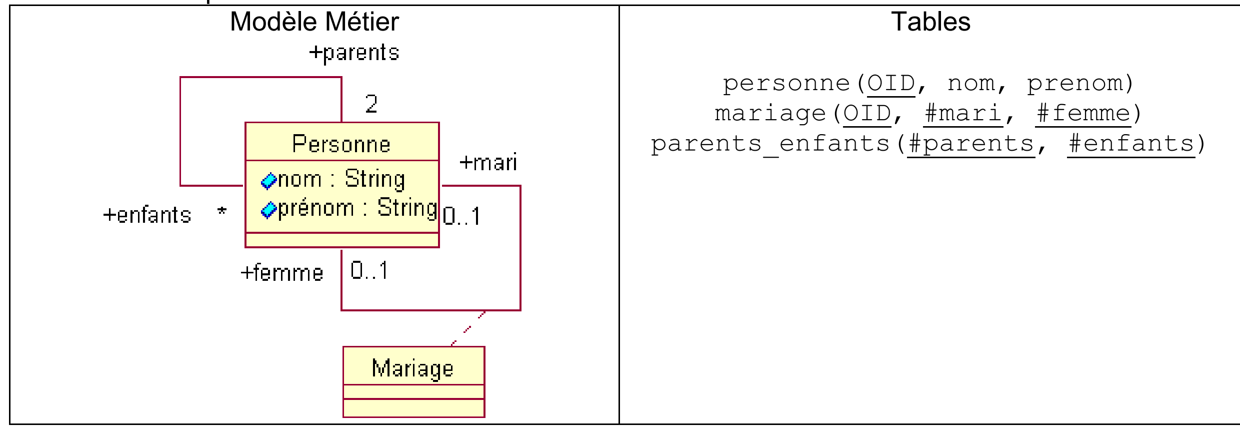
III - Optimisation SQL

Problème de performances

La génération se fait dans un langage de script interprété côté serveur. Or un langage de script n'assure pas la persistance des données. C'est pourquoi ces dernières sont sauvegardées dans une base de données. En cas de besoin, les scripts vont y chercher les informations qui leurs sont nécessaires. Ces données concernent les objets utilisés, leurs liens et leurs attributs, ainsi que les paramètres mis en session.

La génération du format de la base de données est assurée par Netsilon lui-même. Elle ne fait pas l'objet de ce document mais il est bon d'en avoir un aperçu. Chaque classe du modèle métier donne lieu à une table où seront stockés les objets. Pour identifier ces objets, cette table contiendra une clé primaire unique, générée lors de la création de l'objet, souvent nommée OID. Cette table contiendra également les valeurs de chacun des attributs de la classe, que chaque objet devra définir. Les associations sont contenues soit dans la table elle-même (en cas de multiplicité maximum 1 dans un sens), soit dans une table externe, les objets étant retenus par leur clé primaire. Les classes-association sont vues comme des classes avec trois clés primaires : la leur propre, et celles des deux objets liés. En cas d'attributs ou de liens statiques, une seconde table est générée pour les retenir. Un enregistrement, et un seul, permet d'y retenir les valeurs de ces champs statiques. En cas d'héritage, les objets sont présents dans chacune des tables correspondant au type dont ils sont instance, chacune des tables retenant la valeur des attributs et liens dont elles sont responsables.

Par exemple :



Les classes du modèle métier sont ensuite générées en classes du langage de script. Ce sont elles qui iront rechercher la valeur de leurs attributs qui ne seront donc accessibles que grâce à des accesseurs, dans lesquels se trouvent les expressions SQL de recherche. Le code Xion d'accès à un attribut génère lui un appel à ces accesseurs. Un appel à lien ou un accès à variable de session est généré directement par le code SQL de requête à la base de données.

<table>
<tr>
<td><pre>classe Personne
  attribut oid : String

  fonction get_nom : String
    retourne execute_SQL(
         'SELECT personne.nom
         FROM personne
         WHERE pesonne.OID = ' + oid
)
  fin get_nom

  fonction set_nom (nom:String)
    execute_SQL(
         'UPDATE personne
         SET nom = ' + nom + '
         WHERE OID = ' + oid
)
  fin set_nom

  fonction get_prenom : String
    retourne execute_SQL(
         'SELECT personne.prenom
         FROM personne
         WHERE pesonne.OID = ' + oid
)
  fin get_prenom

  fonction set_prenom (prenom:String)
    execute_SQL(
         'UPDATE personne
         SET prenom = ' + nom + '
         WHERE OID = ' + oid
)
  fin set_prenom

  …

fin classe</pre></td>
<td>Pour appeler nom sur une personne (maPersonne.nom)
<pre>maPersonne.get_nom</pre>
Pour changer la valeur du prénom (maPersonne.prenom = 'toto')
<pre>maPersonne.set_nom('toto')</pre>
Pour trouver les enfants (maPersonne.enfants)
<pre>execute_SQL(
        'SELECT enfants
        FROM parents_enfants
        WHERE parents = '
        + maPersonne.oid
)</pre>
Pour ajouter un parent (maPersonne.addparent(unParent))
<pre>execute_SQL(
        'INSERT INTO parent_enfants_
        (enfants, parents)
        VALUES ('
        maPersonne.oid + ',' + '
        unParent.oid + ')'
)</pre>
Les noms de ses enfants (maPersonne.enfants.nom)
<pre>Ensemble(Personne) tmp1 =
execute_SQL(
        'SELECT enfant
        FROM parents_enfants
        WHERE parents = '
        + maPersonne.oid
)
Ensemble(String) tmp2 = Ensemble vide
Enumération e = tmp1.éléments
Tant que e a des éléments faire
    tmp2.ajoute(e.suivant.get_nom)
Fin tant que</pre></td>
</tr>
</table>

Ceci a pour conséquence d'opérer une requête à la base de donnée à chaque appel d'attribut, de lien ou de variable de session, c'est à dire chaque élément retenu par celle-ci. Dans l'exemple précédent, on veut connaître les noms de ses enfants. Une première requête va chercher la liste des enfants, puis, une seconde, exécutée sur chaque enfant trouvé, donne le nom de chaque enfant. SQL aurait permis de trouver ce nom en une seule requête par :

```
SELECT personne.nom
FROM personne
WHERE personne.OID in (
      SELECT parent_enfant.enfant
      FROM parent_enfant
      WHERE parent_enfant.parent = <oid de maPersonne>)
```

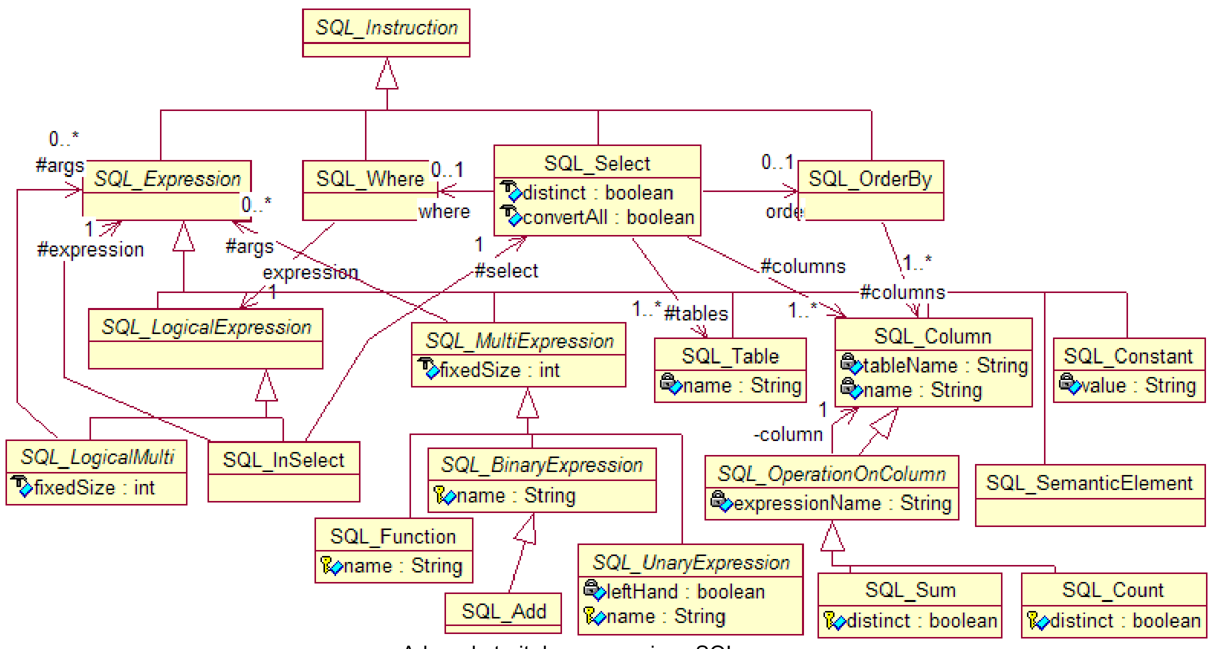
Or une requête à la base n'a rien d'anodin. D'abord, il s'agit d'une opération lente, traitée plus ou moins efficacement suivant les serveurs, les tables et les données. Il faut également prévoir un certain temps de communication entre le serveur d'application, chargé d'exécuter les scripts côté serveur http, et le serveur de base de données (qui ne cohabitent d'ailleurs pas toujours sur la même machine). Donc, moins il y a de requêtes, moins le serveur de base de données est chargé, et moins les serveurs d'application et de base auront à communiquer. Ensuite, les recherches sous forme SQL sont (normalement) traitées de façon optimale par le serveur de base. Le serveur d'application, lui, dépend du programme et réclame beaucoup plus d'opérations, donc de temps et de mémoire, pour réaliser une recherche équivalente et n'est pas initialement prévu pour le faire. L'exemple précédent en est d'ailleurs une bonne illustration.

Mon problème fut donc de réaliser un composant capable de limiter le nombre de requêtes à la base en « optimisant » celles nécessaires pour leur faire réaliser le plus d'opérations possibles.

Solution adoptée

Lorsqu'on connaît Xion, basé sur OCL, on pressent que certaines expressions se prêteraient volontiers à une traduction vers un langage comme SQL, comme les opérations prédéfinies *select* ou *reject* (annexe V p. 41). Il est donc intéressant de réaliser une analyse en amont, encore proche du code source, plutôt qu'en aval, comme une classique optimisation. Une seconde raison est que le code SQL n'est appelé par le script que ponctuellement, et seule une analyse fine et délicate serait à même de trouver les bonnes factorisations. Il faut cependant avoir accès au schéma de la base de données avant d'espérer trouver le meilleur code SQL possible. Il a donc été décidé que ce composant opèrerait sur le second arbre intermédiaire, qui suit bien les opérations prédéfinies Xion, et fournit les types de chaque élément de l'arbre, grâce auxquels il sera possible de déterminer les tables et les champs nécessaires.

Il a alors été nécessaire de créer un nouveau type de nœud (de type *SE_SQL_Expression*) correspondant à une classe dérivée de *SemanticElement*, capable de retenir l'expression SQL résultat de l'optimisation sous forme d'arbre (voir ci-après). Ce nœud se trouvera alors à la place de l'expression optimisée, le générateur de langage intermédiaire devant naturellement en tenir compte…



Arbre abstrait des expressions SQL

SQL_Instruction est la classe-mère de toute expression SQL. Elle définit notamment les opérations de clonage, de vérification de validité et de génération d'expression sous forme textuelle.

SQL_Select représente l'expression de requête SQL *SELECT*. Elle retourne au moins une colonne trouvée dans une ou plusieurs tables (jointes). Un *SELECT* peut être conditionné par une clause *WHERE*, et ordonnée par une clause *ORDER BY*. Un *SELECT* peut être *DISTINCT*.

SQL_Where représente la clause *WHERE*, et est donné par une certaine expression logique.

SQL_OrderBy représente la clause *ORDER BY* et s'applique sur une collection ordonnée de colonnes.

SQL_Expression est la classe mère pour toutes les expressions.

SQL_LogicalExpression est la classe mère pour toutes les expressions logiques. Une bonne partie des classes-filles génériques de *SQL_Expression* trouve un clone fils de *SQL_LogicalExpression* à l'instar de *SQL_MultiExpression* et *SQL_LogicalMultiExpression*.

SQL_MultiExpression représente les fonctions SQL à arguments de nombre défini ou non. Elles sont différenciées en SQL_UnaryExpression pour les opérations unaires (comme *NOT*), SQL_BinaryExpression (comme l'addition), SQL_Function pour les fonctions à paramètres (comme *SUBSTRING(chaîne, début, fin)*).

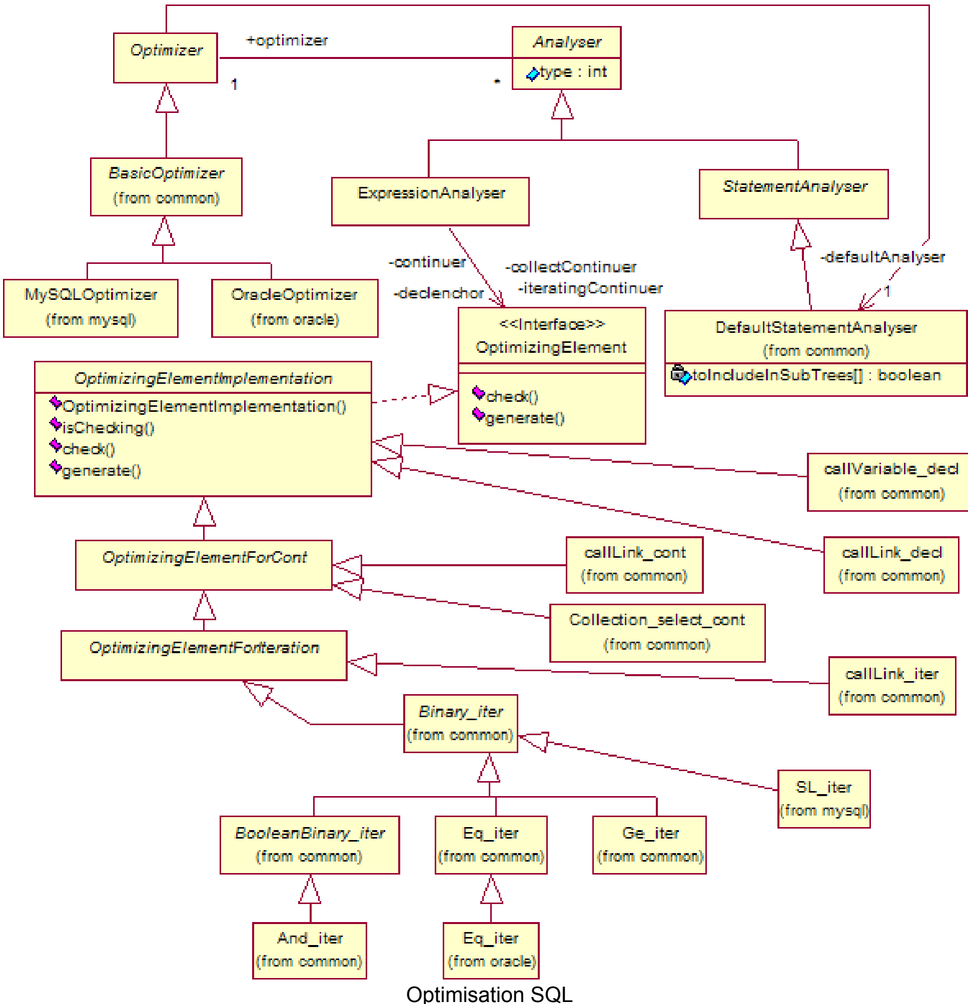
SQL_Table représente une table et *SQL_Column* une colonne dans la table. Une colonne peut être une opération sur une colonne (dans une clause *SELECT*) donnée par *SQL_OperationOnColumn*,, comme les fonctions *SUM* et *COUNT*.

SQL_Constant représente une constante SQL.

SQL_SemanticElement permet d'insérer des expressions données par un arbre abstrait 2 dans une expression SQL, par exemple une valeur de variable.

Viennent ensuite toute une liste de classes représentant chacune les opérations définies dans SQL. Il faut cependant noter que Netsilon génère pour plusieurs types de base de données, à ce jour MySQL et Oracle. Or ces bases ne permettent ni tout SQL ni que SQL. Par exemple, MySQL ne définit pas les *SELECT* imbriqués (*IN (SELECT …)*), mais autorise des opérations non standard comme le modulo (*MOD*) qu'il serait intéressant d'utiliser. C'est pourquoi chacun de ces éléments seront donnés par une « factory » qui donnera le bon objet (sous Oracle une demande de *SELECT* fournira un *SQL_Select* standard, alors que sous *MySQL*, elle en donnera un capable d'éliminer les *SELECT* imbriqués) ou les accès aux opérations non standard de la base utilisée.

Cette représentation n'est ni précise, ni exhaustive, mais elle est suffisante pour décrire toutes les optimisations trouvées.



Optimisation SQL

L'optimisation travaille directement sur l'arbre abstrait, sans passer par le parseur d'arbre ANTLR. Ceci permet d'utiliser toutes les fonctionnalités objet.

L'élément chargé de l'analyse est représenté par la classe *Optimizer*. Il a notamment accès au schéma de la base et appelle la bonne factory pour la création de nouveaux éléments d'arbre SQL. Une première sous-classe *BasicOptimizer* est spécialisée dans la génération pour SQL 92. *MySQLOptimizer* et *OracleOptimizer* génèrent aussi pour un SQL le plus souvent proche, mais optimisé respectivement pour les serveurs MySQL et Oracle.

Chaque *Optimizer* possède une collection d'*Analyser*, spécialisés dans l'analyse d'un type de nœud de l'arbre intermédiaire 2. Ces analyseurs analysent soit des instructions (*InstructionAnalyser*), soit des expressions (*ExpressionAnalyser*). Un analyseur d'instruction par défaut se contente de demander à son *optimizer* d'analyser chacun de ses sous-arbres. Ceci permet de visiter tout l'arbre sans avoir à définir un *Analyser* par type de nœud, puis de le référencer dans les *Optimizer*. Il est d'ailleurs possible de lui demander de ne pas s'occuper de certains sous-arbres (suivant leur cardinalité) ; c'est notamment le cas pour l'analyseur d'affectation (qui n'optimise que le contenu, pas le conteneur).

L'analyseur d'expression a quatre modes d'analyse que savent explorer quatre *OptimizingElement* :

- le *declenchor* qui va savoir comment optimiser quoi qu'il arrive l'arbre en cours d'analyse,
- le *continuer* qui va savoir comment optimiser une partie de l'arbre, charge à un autre *continuer* ou un autre *declenchor* d'en optimiser le reste,
- l'*iteratingContinuer* va servir à un *continuer* spécialisé dans une opération d'itération (telles *select* ou *sortedBy*) pour obtenir la signification SQL de son paramètre,
- le *collectContinuer* va savoir comment optimiser le paramètre d'un *collect* ; il s'agit là d'un *iteratingContinuer* particulier à *collect*.

Les analyseurs n'ont hélas pas tous tout les quatre éléments d'optimisation. L'expression SQL donnée est celle que le premier de ces éléments d'optimisation est capable de manipuler. L'ordre de priorité par défaut est *continuer*, *declenchor*, *collectContinuer*, *iteratingContinuer*, mais il peut être modifié à l'appel, par exemple dans le continuer de *collect*, qui va retrouver l'analyseur capable d'optimiser son argument et lui demander d'utiliser de préférence son *collectContinuer*. Les expressions retournées sont indifféremment des instructions ou des expressions SQL. Il est à la charge de l'élément d'optimisation les utilisant de contrôler s'il est capable d'utiliser le résultat. L'intérêt de cette technique est la séparation des méthodes d'optimisation pour un même type d'élément d'arbre suivant le contexte où il a pu être rencontré. Par exemple un appel à lien ne représente pas la même chose au sein d'une opération d'itération ou non.

Il existe des types spécialisés d 'éléments d'optimisation continueurs (*OptimizingElementForCont*) et itérateurs (*OptimizingElementForIteration*).

OptimizingElementForCont permet de trouver l'expression SQL donnant l'objet sur lequel s'applique l'expression qu'on tente d'optimiser ; par exemple comment trouver l'objet sur lequel on essaye d'appeler un lien. Il est aussi capable de retenir des expressions SQL en leur donnant un nom et un type de retour. Ceci est utilisé dans les itérations pour stocker la valeur de la variable d'itération (qu'elle soit nommée ou non).

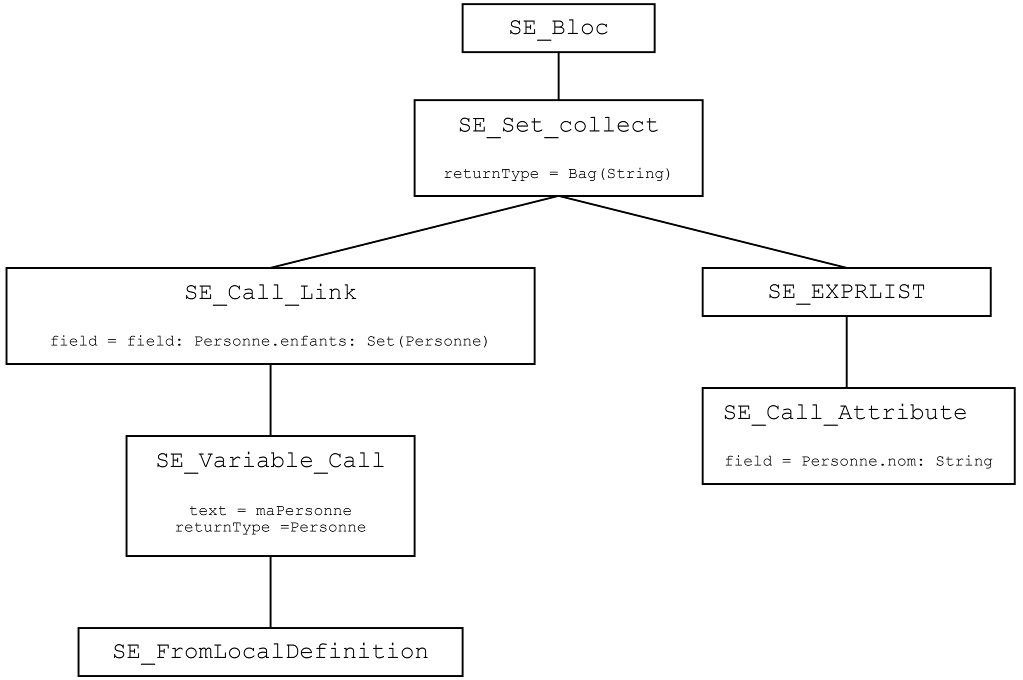
OptimizingElementForIteration est spécialisé pour les *iteratingContinuer*. Il hérite de *OptimizingElementForCont* pour son mécanisme de retenue de variable. Il permet en outre de retenir le besoin d'une jointure de table, soit la table à joindre et (éventuellement) l'expression de jointure.

Enfin, de multiples éléments d'optimisation sont disponibles, correspondant chacun à un certain type de nœud à compiler et à un certain mode de traitement. On peut citer en exemple *callLink_decl* qui optimise un appel à lien où qu'il soit, *callLink_cont* qui optimise un appel à lien si l'objet sur lequel on l'appelle est optimisé par un *SELECT*, ou *callLink_iter* qui optimise un lien au sein d'une opération d'itération s'il parvient à trouver une colonne qui donne l'objet sur lequel il est appelé. Certains élément factorisent le comportement commun d'autres tel *Binary_iter* qui s'occupe de toute opération binaire comme la comparaison. Il est possible que les éléments d'optimisation communs ne conviennent pas à certaines bases à l'instar de la comparaison qui, sous Oracle, est incapable de comparer des colonnes de texte long (*blob*). La classe générale *Eq_iter* est alors sous-classée, et un nouvel analyser d'expression égalité l'intégrant est créé et référencé dans l'optimiseur *OracleOptimizer*. Le nouveau *Eq_iter* peut alors contrôler qu'on ne compare pas deux textes longs avant de demander à son ancêtre de compiler.

La liste des types de nœuds pris en charge se trouve à l'annexe IX (p.74).

Exemple

Pour illustrer l'optimisation, reconsidérons notre exemple précédent : `maPersonne.enfants.nom`, autrement écrit `maPersonne.enfants->collect(nom)` (le '`.`' est un raccourcit de l'opération `collect` lorsqu'il est appelé sur une collection, cf. annexe V p.36),qui se traduit par l'arbre concret 2 en :



Arbre concret non optimisé

C'est cet arbre qui sera envoyé à l'optimisation. Nous prendrons l'exemple d'Oracle, plus proche du langage SQL standard.

Il n'existe pas d'*Analyser* pour *SE_Bloc* sous Oracle, par contre il en existe un commun (à toutes les bases – SQL standard). C'est donc lui qui va être appelé pour optimiser le programme. Il s'agit d'un *DefaultStatementAnalyser*, on ne tiendra donc pas compte des différents types d'éléments d'optimisation. Cet analyseur va itérer sur les enfants, dans notre cas le *SE_Set_collect*, et tenter de l'optimiser. Il va alors demander à son *Optimizer* de lui fournir l'*Analyser* adéquat, qui est un *ExpressionAnalyser* commun (il n'en existe pas de spécifiques pour Oracle), donc qui peut posséder des éléments d'optimisation continueurs, déclencheurs ou itérateurs.

L'analyseur de *SE_Bloc* va alors vérifier que l'analyseur de *SE_Set_collect* est capable de faire jouer soit son continueur, soit son déclencheur. En effet, un itérateur ne correspondrait à rien ici. L'analyseur de *SE_Set_collect* possède un élément d'optimisation continueur et va donc lui transmettre le *SE_Set_collect* à transformer.

Le continueur est écrit pour travailler sur une expression SQL correspondant à l'objet sur lequel est appelée l'opération (ici l'objet est l'arbre de nœud principal *SE_Link_Call* représentant `maPersonne.enfants`, et l'opération est le `collect`). Il va donc demander à l'*Optimizer* de lui trouver un *Analyser* pour un *SE_Link_Call*. Il trouvera un *ExpressionAnalyser* commun.

L'*ExpressionAnalyser* trouvé pour les *SE_Link_Call* va donc tenter de faire jouer son continueur. Pour ce continueur là, l'objet est `maPersonne` et l'opération l'appel au lien `enfants`. A l'instar du précédent, il va demander à l'*Optimizer* de lui trouver un l'*Analyser* pour *SE_Variable_Call*.

Il trouvera un nouveau *ExpressionAnalyser* dans les analyseurs communs, qui ne pourra pas faire jouer son continueur, ce dernier n'existant pas. Le déclencheur, lui, est écrit pour traduire uniquement les variables de session stockées dans la base de données, et refusera le travail, de même pour l'itérateur de collect et l'itérateur simple (écrits tous deux pour traduire les variables d'itération). Il va donc tenter de faire jouer l'optimiseur par défaut de l'*Optimizer*, qui n'aura aucun effet car il n'existe pas d'*Analyser* pour *SE_FromLocalDefinition*.

Après cet échec (l'analyse par défaut ne retourne jamais d'arbre SQL), l'*ExpressionAnalyser* trouvé pour les *SE_Link_Call* va donc tenter de faire jouer son déclencheur. Celui-ci va réussir en empaquetant l'appel à variable *SE_Variable_Call* dans un *SQLSemanticElement*, élément d'arbre SQL décrit en page 15, qui permet d'appeler un arbre concret dans une expression SQL. L'arbre SQL qu'il retourne alors est l'équivalent de :

```
(1)  SELECT parent_enfant.enfant
     FROM parent_enfant
     WHERE parent_enfant.parent = <appel à maPersonne>
```

En effet, le lien se trouve dans la table `parent_enfant`, le parent étant donné par la colonne `parent` et l'enfant par la colonne `enfant`.

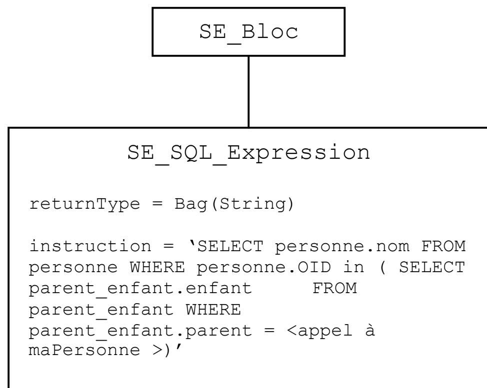
Cette expression 1 remonte alors au continueur de *SE_Set_collect*, qui est satisfait car l'expression retournée est bien un `SELECT`. Il va ensuite repérer le type de l'objet qu'il trouvera à chaque itération (dans notre cas, une *Personne*) et le garder en mémoire. Son paramètre est un *SE_Call_Attribute*, ce qui conduira l'*Optimizer* à lui fournir un nouveau *ExpressionAnalyser* auquel sera transmis le type d'itération ainsi que l'expression SQL 1 qui correspondra à l'appel de la variable d'itération (qui ici n'a pas de nom).

Après échec de l'itérateur de `collect` (il n'est pas défini pour les *SE_Call_Attribute*), le continueur va renvoyer l'arbre SQL correspondant à l'expression :

```
(2)  SELECT personne.nom
     FROM personne
     WHERE personne.OID in (
           SELECT parent_enfant.enfant
           FROM parent_enfant
           WHERE parent_enfant.parent = <appel à maPersonne >)
```

Un continueur a besoin d'un objet sur lequel s'appuyer, or ici, il n'existe pas, ce qui est caractéristique d'un appel à champ sur une variable (anonyme) d'itération. C'est pourquoi, le lien étant bien appelé sur une *Personne*, il s'est basé sur l'expression 1 précédemment transmise. L'attribut se trouve dans la table `personne`, l'objet étant donné par la colonne `OID` et l'attribut nom par la colonne `nom`.

Le propre d'un `collect` est de renvoyer la liste des résultats d'une expression appliquée sur chacun des objets de la collection sur laquelle l'opération s'applique. Pour la base, un appel à attribut sur un objet trouvé par une fonction `SELECT` ou toute une collection aussi trouvée par une expression `SELECT`, revient au même. C'est pourquoi il suffit au continueur de *SE_Set_collect* de renvoyer cette expression, laquelle sera empaquetée dans un *SQLSemanticElement*. L'*Analyser* adopté pour *SE_Bloc* n'a plus qu'à modifier l'arbre concret initial en :



Arbre concret optimisé

Le générateur de langage intermédiaire transformera alors directement cet arbre en un appel SQL, dont le résultat sera correctement interprété, grâce au type de retour, toujours accessible dans le nœud *SE_SQL_Expression*.

IV – Editeur de code

Un nouveau langage

Netsilon est un outil MDA, c'est à dire que la modélisation du site telle qu'elle est décrite dans Netsilon suffit à le décrire entièrement. Pour respecter cette règle, un nouveau langage, Xion, a été développé. L'intérêt discuté ci-avant provient de son pouvoir sémantique, capable de décrire des actions proprement dites en fonction des éléments de la modélisation. Ce langage intervient à de multiples endroits dans la modélisation. On a parlé de la description des méthodes, des actions en entrée des fichiers et centres de décision, des valeurs à afficher dans les afficheurs de valeur, des collections sur lesquelles itérer dans les centres de décision collection, des valeurs par défaut ou transmises aux variables de la session ou d'un fichier…

Or Netsilon est un produit commercial, qui plus est fortement innovant. L'effort à fournir pour l'appréhender est non négligeable. Une fois les généralités d'architectures choisies pour l'application Internet générée comprise, il faut également comprendre l'intérêt d'un modèle métier, ainsi que le rôle et le fonctionnement du modèle de navigation complètement nouveau. Les pages HTML elles-mêmes ont une fonction différente de celle qui leur est classiquement attribuée.

Xion fait partie de ces nouveautés. Il est donc nécessaire de rendre l'effort d'apprentissage le plus léger possible. Cet état de fait a déjà été pris en compte lors de la création du langage : il a donc été rendu semblable à un langage très connu du moment – Java qui lui-même est basé sur la syntaxe du non moins connu langage C++ - ainsi qu'un langage faisant partie intégrante d'UML, norme sur laquelle se base le modèle métier, – OCL –, qui est déjà bien connu des experts. Cependant il reste nouveau et doté d'un certain nombre de particularités. C'est pourquoi il m'a été confié la tâche d'écrire un éditeur de code Xion destiné non seulement à rentabiliser la production, mais aussi à simplifier l'apprentissage du langage.

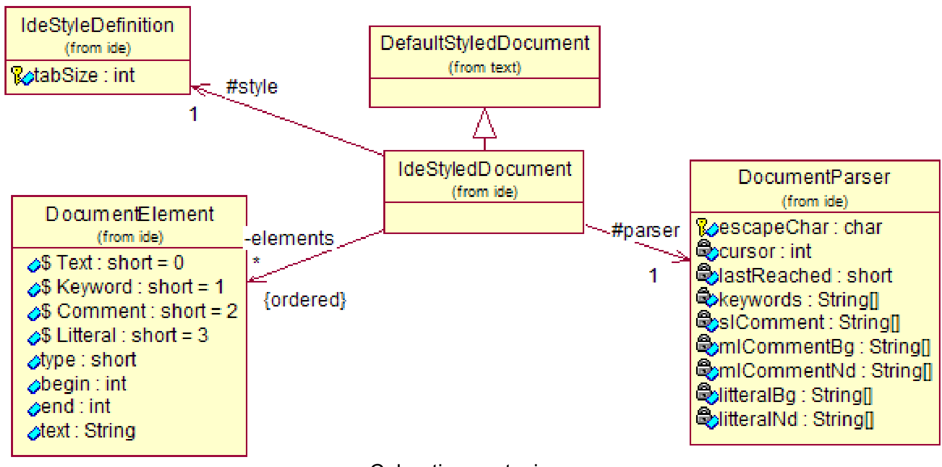
Pour cela, toute une série d'outils classiques ont été implémentés, tels la sauvegarde, le défaire / refaire, le rechercher / remplacer, le saut à la ligne et le couper / copier / coller. Un saut automatique de tabulation (d'espace réglable) fut aussi réalisé pour simplifier la production d'un code simple à relire. Enfin, le choix des couleurs apporte un certain confort visuel, et une internationalisation des commandes une meilleure compréhension…

De plus, une fonction de vérification de code tente de faire fonctionner la chaîne de compilation précédemment décrite jusqu'à la production de l'arbre 2, c'est à dire une vérification lexicale, syntaxique et un contrôle des types. Elle en récupère les éventuels messages d'erreurs qu'elle affiche dans une liste sur lesquels il suffit de cliquer pour surligner la ligne concernée. Une modification d'ANTLR a permis de retrouver ces lignes jusque dans le contrôle des types (classe *CommonASTWithLines* précédemment décrite). Une seconde modification a quant à elle rendu possible l'internationalisation des messages d'erreurs fournis par l'analyse syntaxique d'ANTLR, le contrôle des types ayant déjà été écrit pour lancer des messages internationaux. Pour l'instant, seuls l'anglais et le français ont été écrits, mais la réalisation de message dans d'autres langues demanderait un effort minimum (technologie fournie par Java à base de paires clés / valeurs, assortie d'une détection automatique de région).

Furent également réalisées deux autres fonctions, inspirées des éditeurs de code actuels, qui sont la coloration syntaxique, qui change la police du texte entré suivant sa valeur syntaxique, ainsi que la complétion sémantique, qui indique les solutions de code possible à certains endroits du texte. Ces deux dernières fonctions sont décrites dans le présent document.

Coloration syntaxique

La coloration syntaxique est la fonctionnalité qui permet le changement des attributs d'une partie de texte, soient sa fonte, sa coloration et sa taille, suivant la signification syntaxique de ce mot. Quatre types de signification ont été identifiés : les commentaires, les mots-clés, les littéraux et le texte simple.



Coloration syntaxique

L'éditeur de code est une sous-classe de la classe prédéfinie java *javax.swing.JTextPane*. Cet éditeur va donc être une simple vue d'un document décrivant un certain texte accompagné de certains attributs d'affichage comme sa police, etc.. Il suffit donc de sous-classer ce document pour y adjoindre les comportements de coloration automatique ; c'est ce qui est fait par la classe *IdeStyledDocument*, les calculs de style étant intégrés aux fonctions d'ajout et d'effacement de partie de texte.

Les préférences de l'utilisateur sont sauvegardées dans un fichier texte stocké dans son répertoire dit « home ». La classe *IdeStyleDefinition* se charge de les relire et de les sauvegarder. Le document s'y réfère pour appliquer les styles voulus.

L'analyse du texte doit être particulièrement rapide pour éviter des lenteurs lors de la production de code. De plus, seuls quatre types de tokens sont nécessaires. Le lexeur de Xion précédemment développé ne convient donc pas car il est trop complet, donc nécessite trop de ressources. C'est pourquoi a été développée la classe *DocumentParser*, qui est capable de tokenizer un texte façon rapide, c'est à dire de la transformer en une suite de *DocumentElement*. Il est cependant nécessaire de lui donner quelques informations sur le langage en cours d'édition :

- les mots-clés
- les chaînes de commencement de commentaire simple ligne
- les chaînes de commencement et de fin de commentaire multi lignes
- les chaînes de commencement et de fin de littéraux
- le caractère d'échappement (qui permet d'ignorer la signification d'un caractère de fin de littéral).

On peut ainsi imaginer le configurer pour lui décrire d'autres langages. Les tokens produits enregistrent :

- leur type
- leurs positions de début et de fin
- le texte auquel ils correspondent.

Après tests, il s'est avéré que cette technique était suffisamment rapide. Cependant, la modification des attributs de morceaux de textes est très lente en Java… C'est pourquoi le document retient les tokens à chaque calcul. Lors d'une insertion ou d'une destruction de texte, ce dernier sera mis en tokens. Cette nouvelle liste de tokens sera alors comparée à l'ancienne, et seules les différences entre les deux modifieront les attributs des éléments de texte du document.

Complétion sémantique

Toutes les fonctionnalités que nous venons de décrire correspondent à un certain confort de production de code mais ne résolvent que trop peu le problème de l'apprentissage. Or il s'agit du problème principal qu'est sensé résoudre cet éditeur. De nombreuses solutions existent dans ce but, mais il m'a semblé que celle adoptée généralement par des IDE modernes tels Microsoft Visual Studio ou Borland JBuilder était la plus efficace. Il s'agit d'une liste de choix possibles qui apparaît lors de la production de texte, qui donne les différentes possibilités de code à l'endroit où elles se trouvent.

Mais si JBuilder y parvient, de manière plus ou moins fiable d'ailleurs, à tout endroit du texte, Visual Studio, lui, se contente d'indiquer la navigation au sein des objets (la liste de ses champs accessibles). Prétendre implémenter une complétion sémantique seul étant déjà ambitieux, je décidai de me limiter à cette dernière solution. En effet, la syntaxe de Xion est très proche des langages de programmation les plus connus. Ses principales différences proviennent de ses types prédéfinis et des opérations prédéfinies qui les accompagnent, lesquelles s'appellent exactement comme les opérations définies dans le modèle métier. Les noms de ces opérations étant suffisamment explicites (ils ont en effet pu être discutés lors de la création et des différentes évolutions de la norme OCL dont elles sont issues), les lister aux bons moments est une solution intéressante. De plus, cette solution présente l'avantage supplémentaire de simplifier la navigation dans le modèle métier. Cette complétion apparaît donc lors d'un appel à champ, c'est à dire lorsqu'on tape un '`.`' ou une '`->`'. Pour apprendre Xion, il serait alors toujours nécessaire de lire la documentation l'accompagnant (cf. annexe V p. 24), mais il serait inutile d'avoir en permanence recours à la description des opérations prédéfinies (cf. annexe VI p. 48). Les plus gros problèmes restant à la charge du producteur de code se limiteraient alors à l'apprentissage de la sémantique de base (la déclaration de variables, les structures de contrôle, la différence entre '`.`' et '`->`',…).

Le principal problème fut donc de retrouver le type de l'élément sur lequel s'applique le champ. La meilleure manière d'y parvenir était d'utiliser le contrôle des types qui réalisait déjà une analyse fine du texte et en extrayait déjà le type de chaque élément sémantique, et qui présentait l'avantage de fournir une réponse juste à coup sûr. La sémantique du langage signifie qu'un appel à champ est possible que sur une variable, un type, `self` / `this`, une constante, un littéral, une expression ou un précédent appel à champ. Il suffit dont de donner à l'analyse de type le début du texte jusqu'à l'appel du champ. Le dernier nœud de l'arbre (le dernier fils du bloc, du while ou du if) correspondant à l'objet sur lequel est appelé le champ. Ce nœud fournit son type auquel il est possible de demander la liste des opérations disponibles en vue de la proposer au producteur de code.

Cependant, pour qu'il puisse répondre, le texte entré doit être juste. Ce fut là le plus gros problème à résoudre, car outre les erreurs éventuelles du code entré, lors d'un appel à champ ledit code ne peut en aucun cas être juste. La technique adoptée est de reconstruire la dernière instruction (celle pendant laquelle l'appel à la complétion a été demandé) de manière à en extraire uniquement l'objet sur lequel le champ est appelé, et d'y ajouter les terminateurs nécessaires ('`;`', '`}`', ou '`)`' lors d'une définition d'un type collection). Par exemple, l'expression

```
Integer i = 0 ;
this.setAttribute(i.max(21).
```

sera modifiée en, pour rendre possible le contrôle des types responsables de la complétion :

```
Integer i = 0 ;
i.max(21) ;
```

Le dernier nœud étant un appel à une opération prédéfine de type de retour `Integer`, la liste des opérations de ce type sera alors proposée.

C'est cette reconstruction qui a demandé le plus de travail dans cette partie du projet. Le principe est de reconstruire la chaîne d'appels de droite à gauche. Une fois ceci fait, une vérification de présence de variable d'itération remplace les éventuelles itérations par une déclaration de variable du bon type.

V – Gestion automatique des objets métier

Gestion des données

Un site Internet sans état n'est autre qu'un site statique, autrement dit un ensemble de pages au format HTML sans notion de dynamisme. L'état d'une application Internet est fourni par ses données. Donc pour en modifier l'état, il faut en modifier les données. Les données d'une application générée par Netsilon sont représentées par les objets métier, instances des classes du modèle métier. Ainsi, pour modifier l'état d'une application Internet Nesilon, il faut en modifier les objets métier.

Les objets métier sont stockés dans une base de données relationnelle. Or, ce type de base de données n'est pas adapté au stockage d'objet. On peut souligner plusieurs problèmes :

- nécessité d'un identificateur unique (clé primaire),
- l'absence de notion d'héritage entre tables, donc pas de polymorphisme,
- l'absence de constructeur,
- les types prédéfinis des attributs n'ont pas de correspondance directe avec les types disponibles dans les bases (d'ailleurs différents pour chaque base),
- pas de notion de multiplicité des liens,
- pas de possibilité de liens circulaires.

C'est pourquoi le schéma de la base généré par Netsilon ne correspond pas tout à fait au modèle métier. Ceci implique que l'utilisateur de l'application générée sera certainement incapable de d'ajouter ou de supprimer à la main des objets, ni même de modifier la valeur d'un attribut ou d'ajouter et supprimer des liens. Quand bien même il aurait l'accès aux solutions adoptées par Netsilon, la modification directe des données à la base serait désagréable (Netsilon utilise des chaînes de 32 caractères uniques à tout le site pour identifier des objets, un objet est présent dans plusieurs tables qui dépendent du modèle, les liens sont parfois dans une table séparée, parfois dans la table correspondant à une classe, etc.) et source d'erreurs difficiles à identifier et réparer.

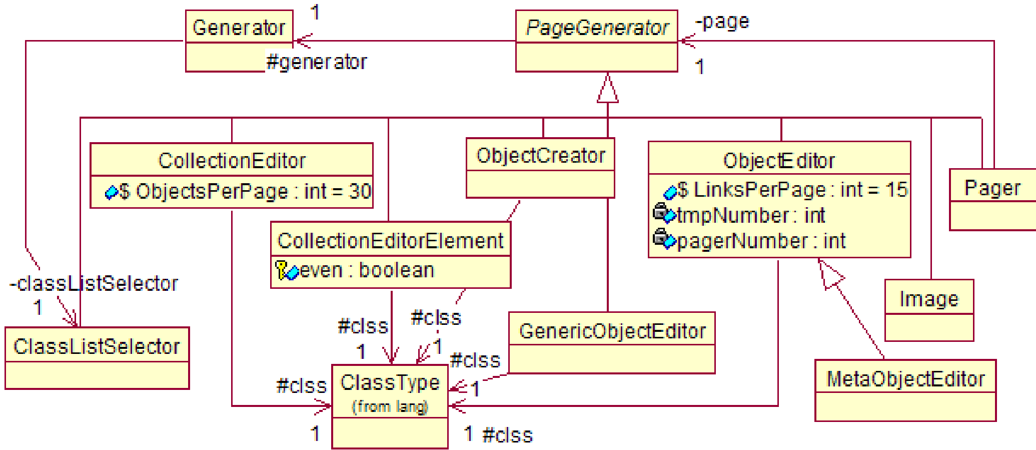
A priori, la seule solution utilisable pour gérer les objets métier est d'utiliser le langage prévu à cet effet : Xion. Or, ceci implique la création d'un modèle de navigation spécialisé, ainsi que toutes les pages HTML correspondantes. Par expérience, il s'agit d'un travail long, donc source de nombreuses erreurs et peu intéressant. La force de Netsilon, qui est de créer rapidement des applications Internet sans se soucier des détails technologiques, s'en trouve nettement amoindrie.

Or, pour peu que la charte graphique indiffère, il s'agit d'une tâche qui trouve toutes ses informations dans le modèle métier : elle en devient tout à fait automatisable. C'est là l'objet du dernier travail qui prendra place dans ce document.

Réalisation

La méthode la plus simple pour générer le page de ce qu'on appellera la gestion de contenu est d'utiliser Netsilon lui-même. Le but est donc de créer la partie de modèle de navigation accompagnée de la bonne collection de pages HTML à même de résoudre le problème posé. On travaille alors au niveau MDA, ce qui permet de garder la même implémentation quelques soient le site de déploiement et la base utilisée, tout en bénéficiant de l'optimisation SQL précédemment décrite, ici nécessaire car il s'agit d'une partie d'introspection qui charge particulièrement la base. L'utilisateur peut ainsi visualiser le résultat de la génération de contenu, voire la dupliquer en vue de modification si celle-ci ne lui convient pas tout à fait. La génération se base sur le métamodèle décrit en page 8.

Cette technique présente cependant un défaut : puisqu'il s'agit d'un modèle de navigation comme un autre, les champs privés ou protégés ne peuvent être affichés ou modifiés. Cependant, il est plausible que si le modeleur les a marqués comme tels, c'est qu'il ne veut pas donner la possibilité de les éditer.



Génération automatique de gestion de contenu

Generator est la factory capable de générer la gestion de contenu. Elle référence les pages déjà produites (grâce à une table de hachage) et les fournit à la demande. Si une demande échoue, la page demandée est créée.

PageGenerator est le type abstrait pour toute génération de page. Elle connaît le *Generator* qui l'a créée, donc peut l'utiliser et impose à ses sous-classes de savoir générer ladite page, ainsi que la transmettre.

ClassListSelector est la *PageGenerator* spécialisée dans la génération de la page qui donnera le choix du type des objets à administrer. C'est la page principale de la gestion de contenu, à partir de laquelle il est possible d'appeler toutes les *CollectionEditor* existantes de l'application, soit une par classe métier.

CollectionEditor est la *PageGenerator* spécialisée dans l'affichage de la liste des objets d'une classe, éventuellement triée suivant la demande de l'utilisateur. Il existe une page de ce type par classe du modèle métier. Chaque ligne représentant un objet est donnée par l'inclusion du fichier fragment généré par *CollectionEditorElement*, lui aussi dépendant de cette classe métier. Elle donne en outre la possibilité de détruire les objets et inclut également la page responsable de l'édition des attributs et liens statiques (*MetaObjectEditor*). Ce type de page inclut des fonctions de paginations, la page affichée étant transmise par une variable de valeur par défaut correspondant à la première.

CollectionEditorElement, on l'a dit, génère la page fragment responsable de l'affichage d'un objet de type donné dans une liste. Il en existe deux par classe métier, une claire et l'autre foncée, ce qui permet un affichage dit « pyjama » de la collection des objets à afficher. Cette page donne la possibilité de détruire l'objet quelle affiche, ou d'appeler la page spécialisée dans l'édition des objets (générée par *GenericObjectEditor*).

ObjectCreator intervient lors de la création d'un objet d'une classe donnée. Il existe donc une page de ce type par classe métier. Cette page n'est générée, et donc appelée, que lorsque cette classe possède un constructeur différent du constructeur par défaut, ou lorsque le constructeur par défaut est marqué comme privé. Elle permet de choisir le constructeur qui créera la nouvelle instance, tout en lui fournissant les paramètres. Malheureusement, aucune solution faisant intervenir des paramètres de type classe ou collection n'a pour l'instant été trouvée. Les constructeurs y faisant appel sont donc ignorés.

GenericObjectEditor génère la page spécialisée dans l'édition d'un objet (qui lui est naturellement passée en paramètre). Son fonctionnement est cependant très simple puisqu 'elle se contente de composer la véritable page d'édition (générée par *ObjectEditor*), choisie suivant le véritable type de l'objet édité. Ceci explique qu'il n'existe qu'une page de ce type par classe métier de base, en pratique `OclObject`.

ObjectEditor génère la véritable page d'édition d'objet. Il en existe une par type concret. Elle permet (dans un formulaire HTML) de modifier la valeur de chaque attribut, ainsi que d'ajouter ou retirer des liens, affichés par composition itérative de la page générée par *CollectionEditorElement*, dont la fonction de destruction d'objet est remplacée par la fonction de destruction de lien. Pour ce faire, un nouveau paramètre de valeur par défaut nulle contenant l'objet de base du lien est ajouté. Lorsque cette page est appelée, si tous les paramètres de ce type son nuls, l'objet doit être détruit. Par contre si l'un d'eux est non nul, l'objet doit être retiré du lien correspondant à ladite variable. Une vérification de multiplicité de lien peut afficher un message d'erreur le cas échéant. Pour sélectionner l'objet du lien, l'affichage de collection du type correspondant au lien est utilisé. Là aussi, pour effectivement ajouter le lien, un nouveau paramètre de valeur par défaut nulle est transmis à cette dernière page indiquant le lien à mettre à jour, ce qui a pour effet de modifier la destruction de l'objet en mise à jour de lien. Comme la page d'affichage des objets à sélectionner est celle responsable de l'affichage des objets liés (toujours celle générée par *CollectionEditorElement*), un second paramètre doit donc lui être ajouté pour savoir différencier le mode ajout ou destruction de lien. Ce type de page offre une pagination par lien multiple.

MetaObjectEditor est l'*ObjectEditor* pour l'édition des attributs et liens statiques. La page fragment générée n'a donc pas besoin de l'objet sur lequel il s'applique. Les paramètres qui doivent être transmis pour la sélection/destruction de lien sont alors des booléens, la sélection ne se faisant plus sur valeur nulle ou non.

Image génère un fichier non HTML qui correspond à une image que la gestion de contenu utilise.

Pager génère un fichier fragment capable d'appeler une page par un lien, en transmettant chacun de ses paramètres avec une valeur inchangée, hormis celui indiquant la page à afficher auquel on donne un numéro défini. On rappelle que chaque éditeur de collection (généré par un *CollectionEditor*) a une variable de pagination, et que chaque éditeur d'objet (généré par un *ObjectEditor*) en a autant que de liens multiples. Comme les éditeurs d'objets sont des fragments, la variable de pagination est retransmise par l'éditeur d'objet générique (généré par *GenericObjectEditor*) auquel il se rapporte.

Conclusion

Je tiens tout d'abord à remercier toute l'équipe d'ObjeXion Software et notamment Philippe Studer, directeur technique, pour sa patience et sa mise en commun de connaissances, Pierre-Alain Muller pour ses idées et bien sûr m'avoir accueilli au sein de son équipe ainsi que les Oliviers pour leurs nombreux conseils et leurs tests de contraintes.

L'avenir de Xion est très intimement lié à Netsilon. Il est en effet incontournable pour modéliser une application Internet à l'aide de ce produit du fait de ses nombreuses utilisations.

Cependant, du fait de son architecture basée sur OCL, il est capable de s'adapter à toutes les configurations de classes qu'il est possible de rencontrer dans les diagrammes de classes UML. De plus, contrairement à OCL, il est capable d'exécuter des actions à effet de bord. Il pourrait donc être intéressant de l'adapter à une utilisation au sein même d'UML, à l'instar d'OCL, comme langage d'action textuel.

Il est utile de rappeler qu'un langage d'action est en cours d'étude depuis longtemps à l'OMG, organisme normalisateur d'UML, et tarde à voir le jour. A l'heure actuelle, seul l'arbre abstrait est en cours de réalisation. L'adaptation de Xion à UML consisterait donc à la compilation de ce langage en l'arbre une fois qu'il sera défini. Son principal attrait, par rapport aux autres langages existants dans d'autres outils eux aussi créés pour des besoins spécifiques tels Kabira Action Semantics, résidant dans sa proximité du langage Java, déjà bien connu et maîtrisé par bon nombre de développeurs, ainsi que sa simplicité qui lui permet d'être compilé dans des langages aussi basiques que PHP.

Les principaux défauts de Xion proviennent de la pauvreté des langages de génération et des bases ciblés par Netsilon (je vise ici PHP et MySQL) : il n'existe ni gestion d'erreurs de type exception, ni gestion de transaction. De plus, certaines spécificités ont été ajoutées telle la destruction de session (mot-clé *DestroySession*).

Enfin, pour être totalement assimilable à UML, il serait également nécessaire d'intégrer au mécanisme de profilage UML (lui aussi en cours de définition) une augmentation de la sémantique du langage d'action, tant au niveau textuel que de l'arbre abstrait y correspondant. Un métamodèle de langage serait à même de résoudre ce genre de problème…

Références

Unified Modeling Language (UML):
http://www.omg.org/uml

Muller P-A., Gaertner N: Modélisation Objet avec UML
Eyrolles, 2000.

Model Driven Architecture (MDA) :
http://www.omg.org/mda

Object Constraint language (OCL):
http://www.klasse.nl/ocl

Warmer J., Kleppe A.: The Object Constraint Language.
Addison-Wesley, 1999.

Richters M., Gogolla M.: A Metamodel for OCL
University of Bremen, 1999.

Mapping OCL to SQL:
Schmidt A.: Untersuchungen zur Abbildung von OCL-Ausdrueken auf SQL.
Technische Universitaet Dresden, Diplomarbeit, 1998.

Demuth B., Hussmann H.: Using UML/OCL constraints for relational database design
Technische Universitaet Dresden, 1999.
http://www.inf.tu-dresden.de/TU/Informatik/ST2/ST/papers/uml99draft.pdf

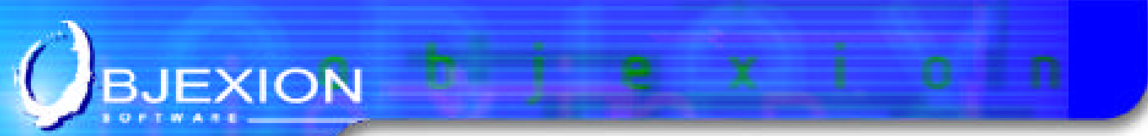
Argo UML
http://www.argouml.com

Action Language for UML
http://www.umlactionsemantics.org

Mellor S., Tockey S., Arthaud R., Leblanc P.: An action language for UML
Proposal for a Precise Execution Semantics
1998

Java
http://java.sun.com

ANTLR
http://antlr.org



SA au capital de 500 000 F
Siret 421 565 565 00015
APE 722Z
Téléphone : 03 89 35 70 75
Télécopie : 03 89 35 70 76

L'embarcadère
5, rue Gutemberg
68 800 Vieux-Thann, France

Création d'un langage d'action pour un logiciel MDA.

ANNEXES

Frédéric FONDEMENT
DRT GEII – 2nde année

Michel HASSENFORDER
Responsable DRT

Philippe STUDER
Directeur Technique - ObjeXion Software
Maître de Stage

Sommaire

I - Netsilon - Integration to the development process p. 2

II - Netsilon - Survol technique p. 4

III - Netsilon - Prise en main p. 9

IV - Xion - Grammaire p. 20

V - Xion - Documentation p. 24

- Utilisations p. 24
- Sémantique de base p. 27
- Types prédéfinis p. 29
- Classes p. 44

VI - Xion - Opérations prédéfinies p. 48

- Bag p. 48
- Boolean p. 49
- Byte p. 49
- Collection p. 51
- Date p. 52
- Double p. 53
- Float p. 54
- Int p. 55
- Integer p. 56
- Long p. 57
- OclAny p. 58
- OclObject p. 59
- OclType p. 59
- Real p. 60
- Sequence p. 61
- Set p. 62
- Short p. 63
- String p. 64
- Text p. 65
- Time p. 66

VII - Compilation - Liste des tokens de l'arbre intermédiaire p. 68

VIII - ANTLR - Présentation p. 71

IX - Liste des optimisations SQL p. 74

- Toutes les bases p. 74
- MySQL p. 76
- Oracle p. 77

I - Netsilon - Integration to the development process

Draft

Pierre-Alain Muller, CEO, ObjeXion Software
www.objexion.com

Introduction

Netsilon is a web application development environment. Netsilon has been designed specifically to address the challenge of building complex, dynamic web applications.

Netsilon is compatible with the way people work currently. Netsilon consolidates the information provided by the people in charge of the development of the web application, and uses this information to generate the server side code.

User roles

The following items describe the various roles played by the people in charge of the development of a typical dynamic web application. Some combination of these roles are often played by the same person.

- The web consultant

Responsible for the design of the overall site architecture accordingly to the customer's requirements. **Using Netsilon.**

- The business analyst

Responsible for the analysis of the application domain, and for the description of the business objects and rules. **Using Netsilon.**

- The graphic designer

Responsible for the look and feel of the web site, using tools like Adobe Photoshop or Macromedia Flash.

- The HTML integrator

Responsible for building the HTML files, of file fragments, that correspond to the grapliic design elaborated by the graphic designer, and for making sure that these HTML files render correctly on the various kind of browsers. Using tools like Macromedia Dreamweaver or Adobe GoLive, and **Netsilon**.

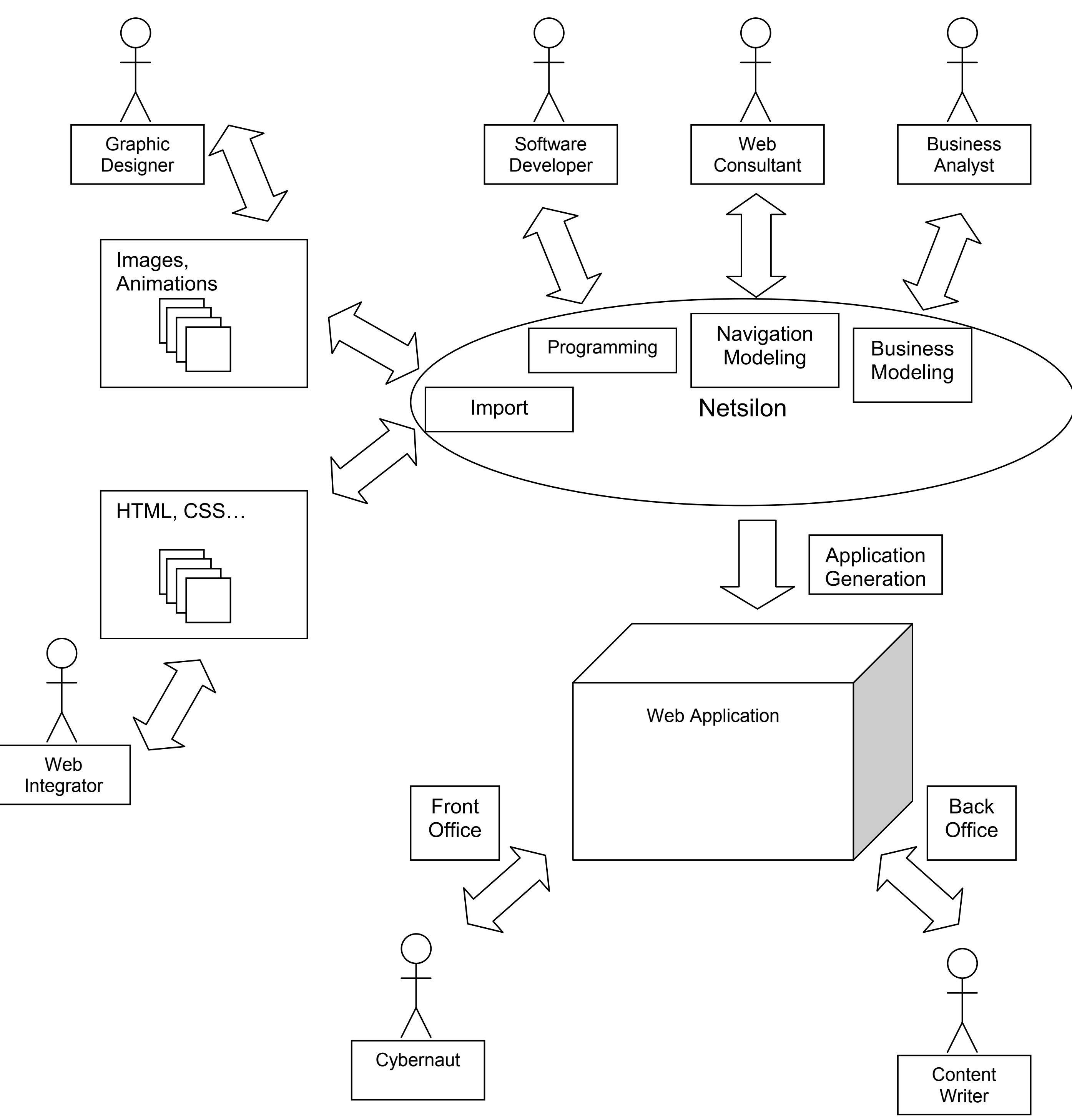
- The software developer and integrator

Responsible for writing programs that Will implement the dynamic behaviours required by the web application. This includes scripts that access databases and generate web pages on the fly. **Using Netsilon.**

- The content writer

Responsible for authoring content that will be displayed by the web application. This content may be created off-fine (with document processing tools) or on-fine (in which case a content management facility has to be provided by the web application). Using interfaces created by Netsilon.

Overal picture



II - Netsilon - Survol Technique

Un environnement pour modéliser et construire des applications web dynamiques.

Introduction

Netsilon est un environnement de développement complet pour la modélisation et la réalisation d'applications web dynamiques de troisième génération, c'est-à-dire d'applications informatiques accessibles par l'intermédiaire d'un navigateur Internet, et dont le contenu et la forme des pages web peuvent changer en fonction de données ou de règles.

Netsilon automatise la fabrication des logiciels qui s'exécutent du coté serveur (configuration et maintenance des bases de données (Oracle, MySQL…), génération et maintenance des scripts (PHP, Java)). Les éléments graphiques (HTML, Flash, images animées…) manipulés par les scripts générés par Netsilon sont créés à l'extérieur de Netsilon, de manière traditionnelle, avec les logiciels du marché (Dreamweaver, GoLive…) puis importés dans les projets gérés par Netsilon.

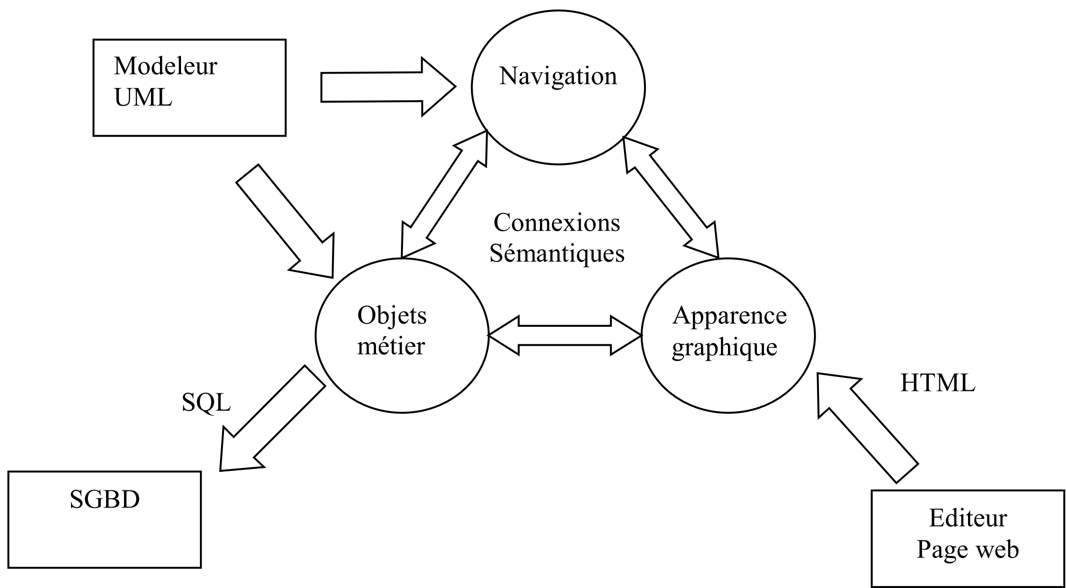
Netsilon comprend un outil de modélisation qui permet de représenter de manière graphique (avec la notation UML, Unified Modeling Language) les objets et les règles métiers, ainsi que le schéma de navigation (l'enchaînement des pages) qui composent une application web, puis de générer automatiquement 100 % du code correspondant à cette description, dans un environnement de déploiement web donné.

Les modèles assurent une parfaite séparation du QUOI (type de contenu et cheminement dans ce contenu) du COMMENT (ingénierie informatique). La forme (l'aspect visuel) est intégrée sous la forme de fragments HTML, composés entre eux de manière dynamique lors de l'exécution de l'application.

Modélisation d'une application web

Une application web est constituée des trois espaces suivants :

- Le métier, c'est-à-dire les objets qui appartiennent fondamentalement au domaine d'application, ainsi que les règles métiers qui s'appliquent à ces objets.
- L'apparence graphique, c'est-à-dire tous les éléments graphiques visibles par les utilisateurs dans les navigateurs Internet.
- La logique de navigation dans l'application web, c'est-à-dire les règles qui gouvernent le comportement de l'application web, en fonction du cheminement et des caractéristiques combinées de l'internaute et des objets métiers.



Modélisation d'une application web en trois espaces distincts.

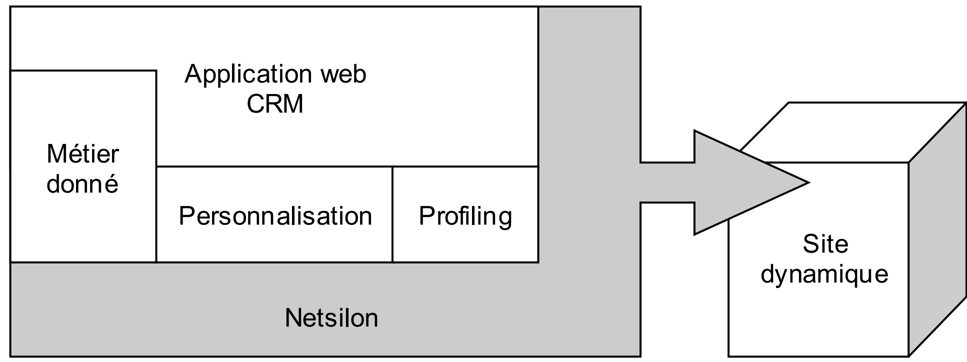
Netsilon permet de faire évoluer séparément ces trois espaces (par exemple pour modifier la charte graphique indépendamment de la navigation) tout en garantissant à tout moment la cohérence globale du modèle et donc de l'application. En particulier, les relations entre les espaces sont validées formellement par Netsilon (par exemple afin de prévenir tout problème de passage de paramètres entre pages dynamiques lors de l'exécution de l'application).

Netsilon respecte la notation normalisée UML pour la modélisation des objets métiers. Un éditeur graphique permet de créer des diagrammes de classes, afin de décrire de manière formelle les abstractions fondamentales de l'application, leurs attributs et opérations, ainsi que leurs relations. Cette représentation est totalement indépendante des choix technologiques, et a pour but de décrire précisément les éléments et concepts métiers autour desquels s'articulera l'application web. Des modèles métiers existants peuvent être importés dans Netsilon, au moyen d'un import XMI (format XML normalisé pour l'échange de modèles).

Netsilon utilise un graphe orienté pour la modélisation de la navigation. Un éditeur graphique permet de décrire complètement la cinématique de l'application web, en décorant le graphe au moyen d'un concept de centres de décision pour exprimer les règles qui gouvernent les transitions entre les pages (page source, page cible et paramètres échangés), la composition d'une page à partir de fragments ainsi que l'affichage ou la saisie de valeurs.

Les règles métiers et les méthodes des objets sont exprimées dans un langage d'action (Xion) qui combine les syntaxes des langages OCL (Object Constraint Language, normalisé avec UML) et Java. Un éditeur syntaxique dédié à Xion, à complétion sémantique (en fonction du modèle UML) facilite la rédaction des règles et de méthodes. Les énoncés Xion sont compilés en cours d'édition, de sorte que le code Xion est correct par construction.

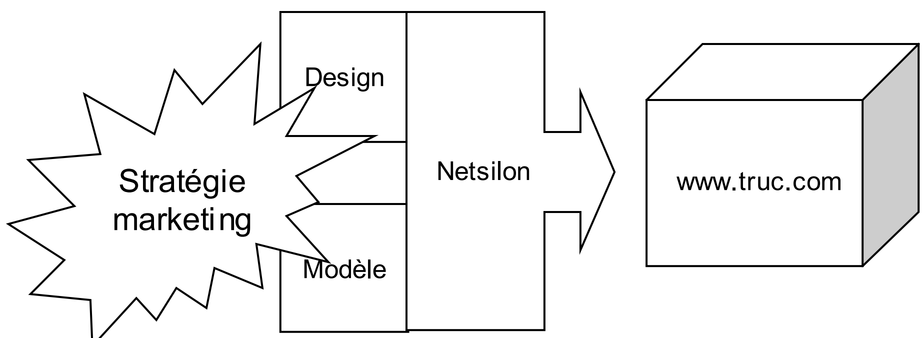
Netsilon encourage très fortement la réutilisation de composants d'une application à une autre. Ces composants peuvent fournir des fragments de solutions métier (géomètre, pharmacien…) ou apporter des services web transversaux (Profiling, gestion d'alertes…). Netsilon peut être vue comme une plate-forme d'intégration, qui permet la construction d'applications web, par empilage de composants (exprimés par des éléments de modèles). Ces composants-modèles apportent un très haut niveau de réutilisation, car ils se concentrent sur l'expression d'un savoir-faire métier, et restent totalement indépendants des techniques de réalisation.



Réalisation d'une application web par empilage de composants-modèles.

Démarche de réalisation de sites intelligents

Netsilon est conçu pour fabriquer des sites qui sont avant tout le reflet d'une stratégie marketing. Netsilon s'inscrit dans une démarche transformationnelle, c'est-à-dire que l'application est générée intégralement depuis les spécifications marketing (capturée dans les modèles) et que toute modification sur les modèles est automatiquement propagée sur le site déployé.



Netsilon génère une application web à partir de l'expression de la stratégie marketing.

La réalisation d'une application en ligne débute par l'analyse précise des exigences de l'exploitant du site, et par la modélisation du domaine de l'application. Le schéma de navigation du site peut être modélisé très en amont, dès lors que des zones ont été définies pour décrire la structure des pages du site. La charte graphique peut être développée en parallèle, et les fichiers HTML qui en résultent sont alors associés aux différentes zones définies dans le modèle de navigation.

Netsilon n'impose pas de méthode de travail, et ne contraint en aucune manière les concepteurs du site. Toutefois, Netsilon est particulièrement bien adapté pour les méthodes objets, itératives et incrémentales, depuis l'analyse des exigences, jusqu'à la génération de code et le déploiement. En particulier, Netsilon permet de modifier de manière incrémentale, une application en ligne.

Les techniques de réalisation interviennent en tant que paramètres lors de la génération de l'application, de sorte que les modèles d'analyse ne risquent pas d'être corrompus par des considérations de réalisation. Un changement de technologie de déploiement n'affecte pas les modèles ; il est ainsi possible de migrer par exemple d'une architecture PHP-MySQL, vers une architecture Java-Oracle, par simple paramétrisation de la génération de code.

Persistance des objets métiers

Netsilon assure la persistance des objets métiers dans une base de données relationnelle (Oracle, MySQL…) de manière totalement transparente. Netsilon se connecte à la base de données, puis prend en charge les opérations de création et de mises à jour du schéma de la base. Les classes et relations présentes dans les diagrammes de classes sont traduites en tables, selon un schéma de traduction optimisé pour l'extensibilité des applications.

Les opérations d'accès à la base, ainsi que les requêtes SQL sont intégralement générées par Netsilon à partir du modèle métier et des expressions Xion.

En cas d'évolution du modèle de classes, Netsilon est capable de modifier incrémentalement une base de donnée peuplée, sans pertes de données.

Intégration HTML

Netsilon génère l'intégralité des scripts (PHP, Java) qui s'exécutent sur le serveur, et qui fabriquent les pages dynamiques en cours d'exploitation du site. Une page dynamique peut être vue comme une mosaïque composée de multiples fragments, statiques ou dynamiques, constituée à la fois d'informations en provenance des objets métiers (attributs ou résultats d'opérations) et d'éléments graphiques prédéfinis.

Netsilon permet de décrire la nature des informations qui entrent dans la composition d'une page dynamique, indépendamment des informations de présentation. Le lien entre la logique de l'application web et le rendu graphique est effectué de manière dynamique, de sorte qu'il est possible de modifier complètement le look d'une application web, sans aucun impact sur la mécanique informatique.

Selon les cas, il est possible de rationaliser la construction des pages dynamiques ; par exemple, afin de ne pas régénérer systématiquement une page dynamique lorsque les éléments sources n'ont pas été modifiés depuis le dernier appel.

Netsilon manipule les fichiers HTML, et en extrait les fragments pertinents pour construire les scripts qui généreront les pages dynamiques. Netsilon peut être vu comme un générateur d'HTML au vol, piloté par les modèles de l'application web.

Prototypage rapide et mise au point

Netsilon peut générer une application web générique à partir du modèle, selon un design graphique simplifié, à destination du concepteur et non de l'utilisateur final. Cette application générique permet de prototyper très rapidement une application et ainsi de mettre au point un modèle de manière itérative. Une fois l'application terminée, l'application générique reste une méthode très efficace pour les opérations d'administration du site et constitue également une solution de gestion de contenu simplifiée.

Caractéristiques techniques des sites générés

Netsilon intervient totalement en boite blanche. Tous les composants de l'application web sont générés en langage source, comme s'ils avaient été écrits manuellement.

Déploiement : multi-sites, transfert automatique des fichiers modifiés, création et mise à jour du schéma de la base de données, possibilité de séparer serveurs de fichiers, de base de données et d'exécution de scripts.

Environnements de déploiement : Apache, IIS, PHP, Java Servlet (en cours de développement)

Persistance des données : Système de gestion d'objets, persistance des objets et des relations, mapping automatique vers les bases de données relationnelles (Oracle, MySQL…)

Sessions : multi-paramètres, durée de vie paramétrable, réalisation par cookie ou par URL longue.

Import de fichiers : HTML, CSS, XMI (format d'échange XML pour modèles UML).
Interface avec l'existant : par appel de fonction native

III - Netsilon - Prise en main

Cette documentation décrit Netsilon, d'ObjeXion Software. Il s'agit d'une documentation pour une prise en main rapide du logiciel.

Intérêt

Netsilon est un générateur d'applications internet. Son fonctionnement est basé sur trois éléments fondamentaux :

- Un modèle métier, qui fournit les mécanismes de base pour le fonctionnement de l'application. Il est représenté par un diagramme de classes UML.
- Un ensemble de fichiers HTML, qui fournit l'aspect de chaque page.
- Un modèle de navigation, qui compose les pages HTML entre elles.

A partir de ces trois éléments, *Netsilon* va générer des scripts pour un serveur Internet, qui utiliseront une base de données (pour la persistance), formatée automatiquement. Le langage de génération et le type de la base de données sont choisis par l'utilisateur. La liste des choix est en perpétuelle évolution.

Installation

Netsilon est fourni sous la forme d'un fichier ZIP. Pour l'installer, il suffit de l'extraire dans un répertoire de votre choix (environ 10 Mo).

Exécution

Netsilon est écrit en Java, compilé pour une version 1.3. Vous pouvez donc lancer le logiciel sur n'importe quelle plate-forme supportant une machine virtuelle Jsava compatible. Nous vous conseillons les machines virtuelles:

- Sun Java 2 Runtime Environment Standard Edition 1.3 (JRE) pour Windows et Solaris(http://java.sun.com/)
- IBM Runtime Environment Release 1.3 (http://www.ibm.com/java/) pour Linux et autres plate-formes.

Remarque: *Netsilon* inclut certaines librairies pour son fonctionnement, comme Xerces, Xalan, et autres librairies d'accès aux bases de données. Si vous constatez un disfonctionnement lors de l'ouverture de *Netsilon* (du type "*No class def found error*"), où lors de la lecture ou l'écriture d'un projet, vérifiez que ces bibliothèques n'existent pas par ailleurs. C'est notamment le cas de la machine virtuelle d'IBM pour Windows, qui inclut en standard la librairie Xerces. La mauvaise librairie est prise en compte et *Netsilon* ne peut lire ou sauvegarder un projet.

Netsilon nécessite quelques ressources mémoire. C'est pourquoi nous vous conseillons d'indiquer à la machine virtuelle Java une taille mémoire maximum en fonction de votre configuration. La valeur optimale correspond aux 2/3 de votre mémoire disponible. Une mémoire de 128 Mo est un minimum, mais 256 Mo sont confortables (la plupart de machines virtuelles Java limitent la mémoire qu'elles utilisent à 64 Mo). Pour indiquer à votre machine virtuelle la taille de cette mémoire, il existe une option (non standard) généralement comprise par les machines virtuelles `-Xmx???m` où ??? est cette taille en Mo. Référez-vous à la documentation de votre machine virtuelle si elle ne comprend pas cette option.

Netsilon se présente sous la forme d'un fichier JAR exécutable, accompagné de fichiers XML nécessaires à son exécution. Si votre machine virtuelle la supporte, indiquez l'option `-jar` à la ligne de commande puis le nom du fichier jar (habituellement *WebProject.jar*) pour exécuter Netsilon. Si votre machine virtuelle ne la supporte pas, exécuter la classe *uci.uml.Main* en ayant pris soin d'indiquer le nom (absolu) du fichier jar dans la variable d'environnement `CLASSPATH`.

Typiquement, les lignes de commande seront

- avec l'option jar:

```
java -Xmx256m -jar CheminDInstallationDeNetsilon/WebProject.jar
```

- sans l'option jar sous Windows

```
set CLASSPATH=%CLASSPATH%;CheminDInstallationDeNetsilon\WebProject.jar
java -Xmx256m uci.uml.Main
```

- sans l'option jar sous un système UNIX

```
set CLASSPATH=$CLASSPATH:CheminDInstallationDeNetsilon/WebProject.jar
java -Xmx256m uci.uml.Main
```

Netsilon peut aussi se lancer avec quelques options qui lui sont propres. Il suffit alors de placer un tiret suivi du nom de l'option en fin de la ligne de commande. Par exemple l'option *help* sera prise en compte (avec l'option jar) par

```
java -Xmx256m -jar CheminDInstallationDeNetsilon/WebProject.jar -help
```

Les options suivantes sont disponibles:

- *help*: affiche la liste des options disponibles.
- *big* / *huge*: utilise de grande/petites polices.
- *nosplash*: évite l'écran de démarrage.
- *errorfile <nomDeFichier>*: Détourne la sortie d'erreur vers le fichier donné. En cas de "bug", nous vous demanderons de nous renvoyer ce fichier.

Projets

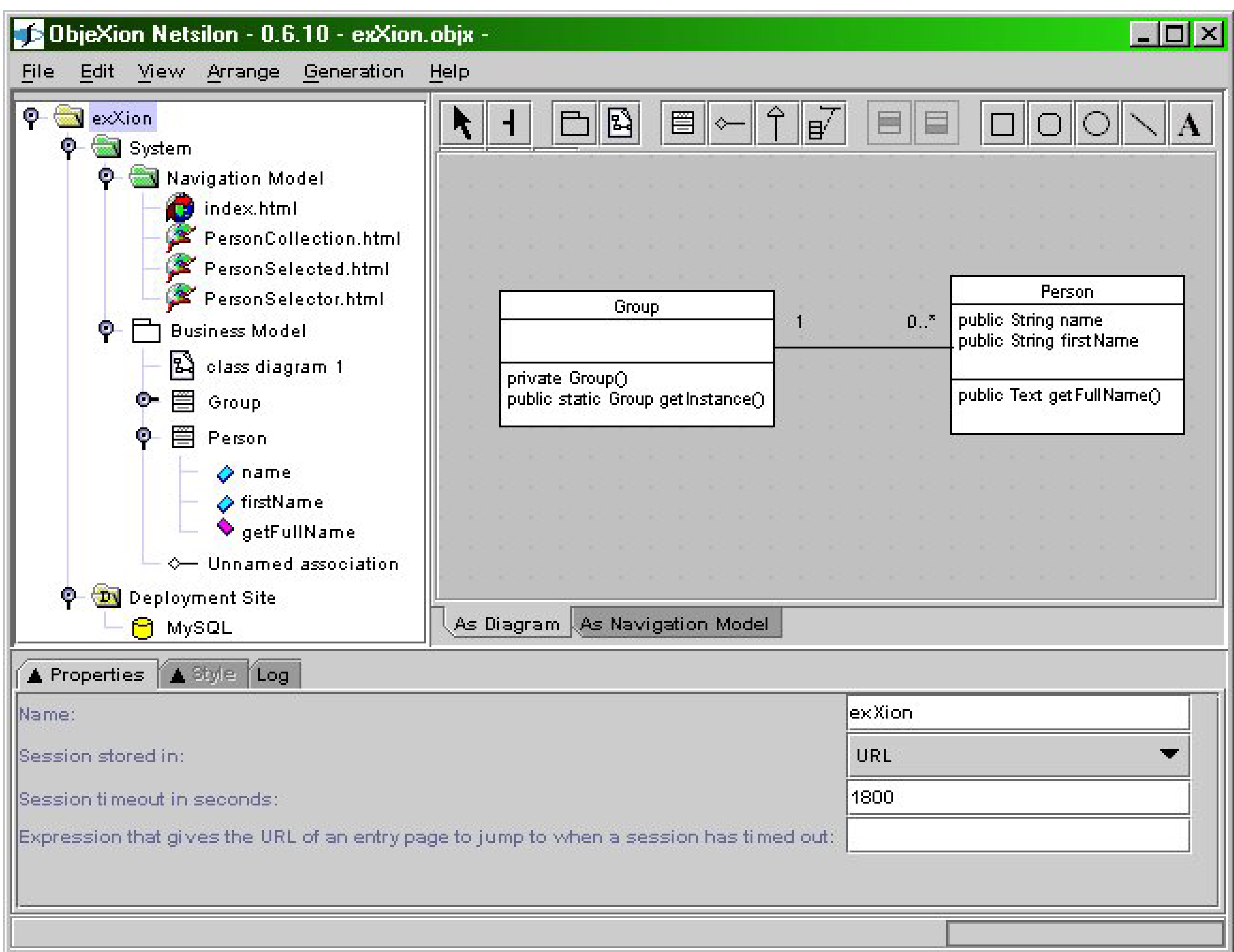
Netsilon vous demandera toujours de travailler sur un projet. A l'ouverture du programme il vous sera proposé soit d'ouvrir un projet récemment ouvert (*Recently opened projects*), soit d'ouvrir un projet existant (*Open project*), soit d'en créer un nouveau (*New project*). Une annulation entraîne l'abandon de l'ouverture de *Netsilon*. Les projets portent toujours l'extension `objx`.

Dans un répertoire de projet, on retrouve l'arborescence suivante:

- `htdocs`, qui contient les fichiers HTML.
- `trash`, où se trouvent les fichiers effacés sous Netsilon (jusqu'à 10 sauvegardes).
- `generation`, répertoire de travail où *Netsilon* stocke ses fichiers générés.

Si vous utilisez une version de démonstration pour déployer sur un site particulier vous devrez demander à ObjeXion un fichier `netsilon.key`, qui décrit le site de déploiement, dans ce cas unique.

Interface



L'interface est composée d'un arbre de navigation comportant la plupart des éléments du projet. A droite se trouve un panneau spécialisé dans l'édition détaillée de certains de ces éléments, soit les diagrammes de classe (dans l'exemple ci-dessus), soit le modèle de navigation. Enfin, en bas, un panneau de propriétés où s'éditent les paramètres de chacun des éléments contenus dans le projet. Ce dernier panneau peut être remplacé par un panneau de style qui indique la manière de dessiner l'élément sélectionné, ou par un panneau d'informations (Log) chargé d'afficher les messages d'analyse, de génération et de déploiement. Un message d'information simple est précédé de , un message d'avertissement de , et un message d'erreur de .

Chacun des élément est représenté par une icône:

- pour un projet.
- pour un système.
- le modèle de navigation, ou un dossier du modèle de navigation (si destiné principalement à contenir des fichiers statiques, ou si destiné aux fichiers générés également transmis).
- pour un fichier, pour un fichier directement appelable, pour un fragment, pour un fichier statique (non généré), pour un fichier non-html.
- pour une zone, pour une zone directement appelable, pour un fragment.
- pour un centre de décision.
- pour le modèle métier ou un paquetage.
- pour un diagramme de classes.
- pour une classe.
- pour un attribut public (si statique), pour un attribut protégé (si statique), pour un attribut privé (si statique).
- pour une opération publique (si statique), pour une opération protégée (si statique), pour une opération privée (si statique).
- pour une méthode publique (si statique), pour une méthode protégée (si statique), pour une méthode privée (si statique).
- pour un lien.
- pour un site de déploiement.
- pour une base de donnée.

Pour une édition détaillée, dans l'arbre de navigation, double-cliquez sur un fichier, ou cliquez sur un diagramme de classe. Pour voir apparaître les propriétés d'un élément, cliquez dessus. Pour ajouter des éléments à un élément, cliquez à droite sur cet élément.

Le menu *File* permet notamment de quitter le logiciel, d'ouvrir un autre projet (le projet courant étant alors fermé), ou de sauvegarder un projet en cours. Il est aussi possible d'imprimer un diagramme de classes. Le menu *Edition* de copier / coller / couper du texte ou des classes (vers un diagramme), de détruire les centres de décisions qui ne sont pas utilisés et de détruire des éléments (il s'agira toujours de l'élément décrit dans le panneau de propriétés). Le menu *View* permet lui de chercher un centre de décision par son nom et de changer l'affichage de la grille de diagrammes de classe. Le menu *Generation* permet d'analyser le modèle métier, de générer le site, de régénérer l'intégralité du site, de sélectionner un site de déploiement, et de déployer un site. Le menu *Help* donne accès aux mentions légales.

Générer un site

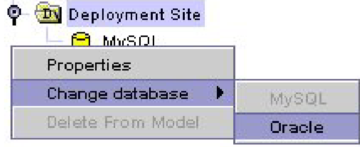
Netsilon permet de définir plusieurs sites de déploiement. Chacun de ces sites est décrit par un ensemble de paramètres, et utilise une base de donnée à définir.

Name: le nom du site de déploiement.
HTTP server name(*): le nom du serveur HTTP dudit site.
Root Path(*): le chemin de base.
Target Language(*): le langage des fichiers générés.
Target Language Extension: l'extension des fichiers générés.
Transfert Mode: le mode de transfert des fichiers du site.
Copy script files to directory: le chemin où les fichiers du site seront physiquement copiés.

Dans cet exemple, l'adresse du site Internet sera `http://localhost/exXion/`. Le langage des scripts généré côté serveur sera PHP, et auront tous une extension .php. Enfin, tous les fichiers nécessaires seront copiés (*Transfert Mode* à *File Copy*) dans le répertoire désigné par `C:\Easyphp\www\exXion`. Il est possible de ne pas transférer les fichiers (*Transfert Mode* à *No Transfert*), ou d'utiliser le protocole *ftp*, auquel cas il vous sera aussi demandé:

- le serveur ftp.
- le port du serveur ftp (par défaut 21).
- l'identifiant de connexion (*login* - optionnel).
- le mot de passe de connexion (*password* - nécessaire s'il y a un *login*).
- le répertoire sur le serveur ftp (*ftp remote directory*).

Il est aussi nécessaire de configurer la base de données. Le changement de type de base de données se fait par le menu contextuel du site de déploiement:



Name:	MySQL		
RDBMS:	MySQL		
Database or sid:	exXion		
Prefix all tables with:			
DB access in the IDE		*DB access in the generated scripts*	
Server Name (hostname.com[:port]):	localhost	Server Name (hostname.com[:port]):	localhost
User:		User:	
Password:		Password:	

Name: le nom de la base de données (au sein du projet Netsilon).

RDBMS: le type du serveur de base de données.

Database or sid(*): le nom de la base pour le serveur.

Prefix all tables with(*): un préfixe pour les noms de table au sein de la base; chaque table avec ce préfixe sera considérée comme faisant partie du site.

DB access in the IDE: le serveur de base de données vu par *Netsilon*, c'est à dire vu par la machine de développement.

DB access in the generated scripts: le serveur de base de données vu par les scripts générés, c'est à dire vu par le serveur HTTP.

Server name(*): le nom du serveur de base de données.

User: le nom de l'utilisateur du serveur de base de données.

Password: le mot de passe de l'utilisateur.

*: Si vous avez une version de démonstration, veuillez contacter ObjeXion en indiquant la valeur de chacun de ces champs. Vous recevrez alors un fichier qui les renseignera à copier dans le répertoire du projet.

La sélection du site de déploiement se fait par *Select deployment site*.

Quelques options de génération sont données par *Generation directives...*:

- *Check for null object* va générer dans le site des messages d'erreurs lors de l'invocation de méthode ou de l'accès aux attributs ou liens d'un objet `null` (inexistant) en indiquant où se trouve l'erreur.
- *Optimize SQL* fait intervenir un compilateur améliorant la qualité des requêtes à la base de données pour décharger le serveur HTTP.
- *Disallow cache* interdit au navigateur client la mise en mémoire cache des pages du site; cette option est pratique lors du développement, mais doit être décochée lorsque le site entre en production.

Netsilon dispose de plusieurs niveaux d'analyse. Ces analyses génèrent des messages dans le panneau de "Log". Elles sont appelées par le menu *Generation*:

Analyse construit une représentation mémoire du modèle métier (types disponibles). Elle permet une vérification de sa sémantique, sans toutefois vérifier le corps des méthodes. Cette action est très souvent effectuée de manière automatique par *Netsilon*.

Rebuild all appelle la fonction *Analyse*, puis génère tous les fichiers "dynamiques", c'est à dire les classes du modèle métier, les fichiers du modèle de navigation non marqués comme statiques, ainsi que les centres de décisions. Le résultat de cette génération se trouve dans le répertoire du projet objx, généralement sous `generation\<NomDuSiteDeDéploiement>\<LangageCible>`. ***Rebuild***, lui, fait la même opération en ne compilant que les fichiers nécessaires (les éléments correspondants ayant changé ou les fichiers ayant été effacés).

Deploy appelle la fonction *Rebuild* et copie (suivant l'état de l'option *Transfert Mode* du site de déploiement) l'intégralité des fichiers nécessaires au site, c'est à dire le résultat de la génération et les fichiers du modèle de navigation non-html et html demandant le transfert. Seuls les fichiers modifiés depuis le dernier déploiement sont transférés. Pour déployer en recompilant tout, appelez *Rebuild all*, puis *Deploy*. Lors de son appel, plusieurs options sont possibles:

Include content management pages:	inclut les fichiers de gestion de contenu automatique.
Generate content management pages:	(re)construit les fichiers de gestion de données. automatique.
Skip online database update:	ne vérifie la validité ni du format, ni du contenu de la base de données.
Generate SQL Script:	les requêtes SQL de création du schéma de la base; dans `SQL Script.txt` dans le répertoire du projet.
garbage collector:	récupère la mémoire inutilisée après avoir généré X fichiers.

Pour la gestion des données de la base, il est possible de demander la création automatique de fichiers qui seront capables de faire vivre les objets métier, c'est à dire de les créer (les constructeurs ayant pour paramètre des objets ou des collections ne sont pas gérés), les détruire, et modifier leurs attributs ou leurs liens publics. Ces fichiers sont créés au sein du projet *Netsilon* sous le répertoire `contentmanagement`, la page de garde étant `content.html`. Ces fichiers faisant alors partie du projet sont générés au même titre que les autres fichier HTML du projet si l'option *Generate content management page* est sélectionnée. Il est impératif de ne pas modifier ces fichiers. L'arborescence des répertoires est conservée à chaque génération, toute la gestion de contenu se retrouve dans ce répertoire, qu'il est alors possible de protéger.

Lors du déploiement une mise à jour incrémentale du schéma de la base de données est effectué de manière à conserver autant que faire se peut le contenu de la base lors d'un changement dans le modèle métier. Une vérification de conformité des objets avec leur type est également effectuée. Il arrive que par Internet, il soit impossible d'accéder directement au serveur. C'est l'objet de l'option *Skip online database update*. Dans ce cas, le script SQL généré par *Generate SQL Script* indique comment formater la base.

La dernière option permet de forcer la récupération de mémoire inutilisée dans le cas de machines à configuration limitée.

Le modèle métier

Le modèle métier décrit les fonctions internes du site. Il est donné par un diagramme de classes UML, il est donc nécessaire de connaître la programmation objet. Nous vous invitons à lire des ouvrages comme *Modélisation objet avec UML* (ed. Eyrolles) de P-A.Muller et N.Gaertner, ou à visiter le site http://www.omg.org/uml/ pour appréhender où compléter vos connaissances sur cette technique. Les liens qualifiés ne sont pas disponibles, et une classe ne peut avoir qu'un seul ancêtre.

A l'ouverture de *Netsilon*, l'éditeur détaillé montre le premier diagramme de classes du paquetage de base. Dans cet éditeur, en plus du diagramme de classes, se situe une barre d'outils. Pour savoir à quoi correspond chacun de ces outils, laissez le pointeur de la souris sur celui qui vous intéresse pour avoir son nom.

- (*Select*) et (*Broom*) indiquent le mode de sélection.
- (*Package*) ajoute un paquetage (il faut alors cliquer dans le diagramme sur l'endroit où doit apparaître le paquetage).
- (*ClassDiagram*) ajoute un diagramme de classes.
- (*Class*) ajoute une classe (il faut alors cliquer dans le diagramme sur l'endroit où doit apparaître le paquetage).
- (*Association*) ajoute un lien (il faut alors lier les deux classes en appuyant su le bouton droit de la souris sur la première et le relâchant sur la seconde).
- (*Generalization*) ajoute une relation d'héritage (il faut lier les deux classes en appuyant sur le bouton droit de la souris sur la classe fille et le relâchant sur l'ancêtre).
- (*AssociationClass*) ajoute une classe-association (il faut lier les deux classes en appuyant sur le bouton droit de la souris sur la première et le relâchant sur la seconde).
- (*Add Attribute*) ajoute un attribut à la classe sélectionnée.
- (*Add Operaion*) ajoute une opération à la classe sélectionnée.
- les autres outils permettent d'ajouter des éléments graphiques non significatifs.

Pour éditer les propriétés d'un élément, sélectionnez-le dans le diagramme de classes ou dans l'arbre de navigation. Attention, différence est faite entre les méthodes et les opérations (une opération est définie une fois pour toute, les méthodes les implémentant dans la classe elle-même ou dans les classes-filles). Les corps de méthodes (accessibles en double-cliquant sur la méthode ou l'opération si cette méthode est au sein de la classe elle-même, ou dans le menu contextuel de ces mêmes éléments) sont écrits en langage *Xion*.

Le modèle de navigation

Le modèle de navigation s'appuie sur les fichiers HTML pour en définir les relations. Ce type de modèle est spécifique à *ObjeXion Netsilon*. Les fichiers sont séparés en différentes catégories:

- Les fichiers non-HTML ()
- Les pages HTML
  - Les fichiers HTML
    - Les fichiers statiques ()
    - Les fichiers d'entrée ()
    - Les fichiers fragments ()
    - Les fichiers contextuels ()
  - Les zones
    - Les zones d'entrée ()
    - Les zones fragments ()
    - Les zones contextuelles ()

Les **fichiers non-HTML** sont des fichiers nécessaires à l'exécution d'un site qui ne sont pas au format HTML. Il peut s'agir d'images, de feuilles de style...

Les **pages HTML** sont des pages au format HTML. Ils se différencient en **fichiers HTML** et en **zones**. Les fichiers HTML sont des fichiers logiques, alors que les zones vont chercher leur contenu de façon dynamique, soit par une url, soit en retrouvant le corps de l'HTML. Ces fichiers peuvent être dits **d'entrée**, c'est à dire qu'ils pourront directement être appelés par un client sans passage de paramètre. C'est typiquement l'état du fichier *index.html*. Un **fragment** est un fichier destiné à être contenu dans un autre fichier HTML. Il peut bien sûr être contenu dans un autre fragment... Une **page contextuelle** désigne une dernière catégorie, non composable (à l'instar des fichiers d'entrée), mais qui ne peut être exécutée sans paramètre. On peut considérer les **fichiers HTML statiques** qui s'apparentent aux fichiers **non-HTML**, transmis sans être compilés.

La différentiation *non-HTML*, *fichier HTML* et *zone* se fait à la création. Un *fichier HTML* se différenciera d'un *non-HTML* par son extension (.html ou .htm ou non); une zone sera explicitement demandée. La création se fait soit au sein de *Netsilon* par le menu contextuel d'un répertoire, soit, hormis les zones, en copiant (ou créant) les fichiers concernés dans l'arborescence du répertoire `htdocs` (répertoire qui se trouve au même endroit que votre projet).

Une zone peut retrouver son contenu de différentes manières:

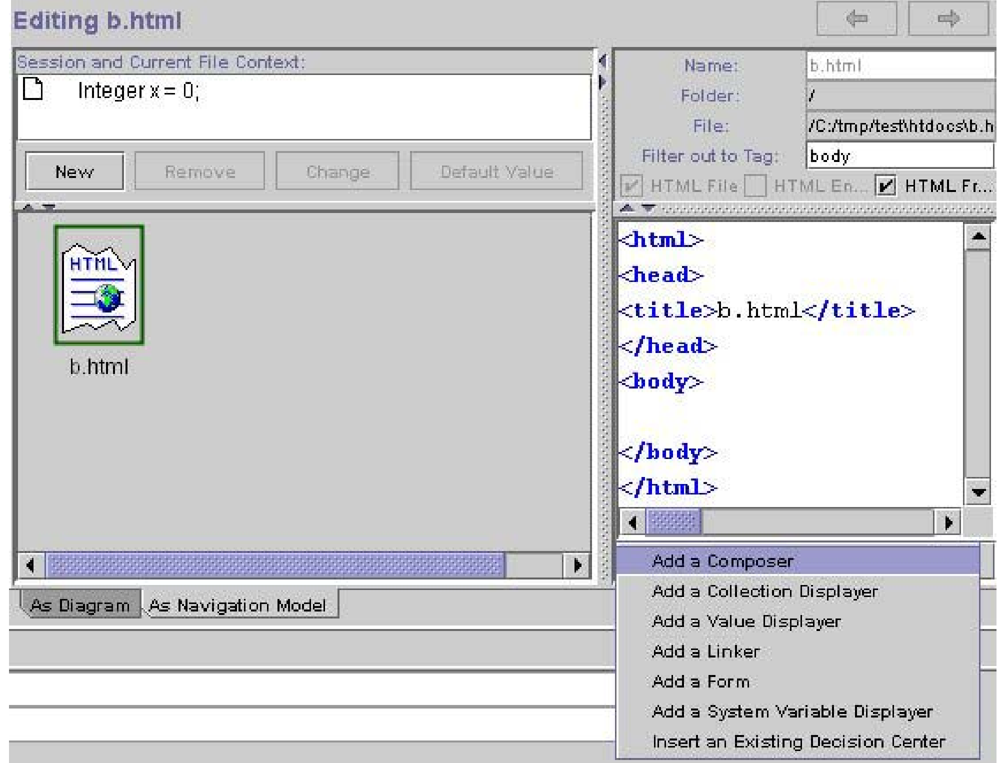
- à partir d'un fichier présent lors de la compilation (*Source From Compile-Time HTML File*).
- à partir d'une url complète (*Source From URL*).
- à partir d'une expression Xion retournant un *Text* donnant le source du HTML à afficher (par exemple *'<h1>Hello <i>World</i>…</h1>' - Source From Expression*).
- à partir d'une expression Xion retournant un *String* donnant l'url complète à afficher (*URL From Expression*).
- à partir d'une url relative au chemin de base - `root path` (*Source From relative Path*).
- à partir d'une expression Xion retournant un *String* donnant l'url relative à afficher (*URL From Expression*).

Attention: Les affichages par url ne tiennent pas compte des Headers lorsque la zone est un fragment...
Une URL relative est relative au chemin de base (`root path`) du site de déploiement.

Chaque page peut posséder un contexte, soit une liste de variables disponibles lors de l'exécution de la page. Ces variables sont caractérisées par un nom, un type, et éventuellement par une expression représentant leur valeur par défaut. Une page (non-fragment) d'entrée est une page qui n'a pas de variable de contexte, ou dont toutes les variables de contexte ont une valeur par défaut, sinon il s'agit d'une page contextuelle. Existent aussi les variables de session qui sont, elles, partagées entre toutes les pages. Un appel par url va créer la session, cette session étant transmise à toutes les pages appelées. Il est possible d'éditer ces variables de session dans le panel *Session and Current File Context* de la fenêtre de détails d'un fichier HTML.

Les éléments dynamiques au sein de ces pages sont représentés par différents centres de décision, spécialisés dans un domaine précis:

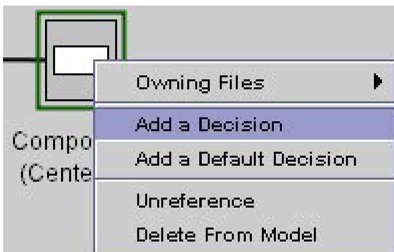
- les **composeurs** (*Composer*): ils intègrent un fragment à la page.
- les **afficheurs de collection** (*Collection Displayer*): ils itèrent sur une collection en intégrant autant de fois un fragment.
- les **afficheurs de valeur** (*Value Displayer*): ils affichent une valeur d'un type simple (Text, String, Boolean, Integer, Real, Double, Float, Long, Int, Short, Byte).
- les **lieurs** (*Linker*): ils lient une page à une page d'entrée ou contextuelle - généralement en paramètre `href` du tag `a`.
- les **exécuteurs de formulaire** (*Form*): ils lient une page à une page d'entrée ou contextuelle - nécessairement en paramètre `action` du tag `form`.
- les **afficheurs de valeurs système** (*System Variable Displayer*): ils affichent des variables particulières, qui peuvent être:
    - le nom du serveur HTTP (HTTP Server Name).
    - le nom du chemin du site derrière le serveur (Root Path).
    - les deux précédents (HTTP Server Name And Root Path).
    - l'extension des fichiers générés (Target Language Extension).
    - l'url de la gestion de contenu automatique (Content Management URL) - déclenche un message de warning à la génération si la gestion de contenu automatique n'est pas incluse.



La situation d'un centre de décision se mémorise au sein du fichier HTML lui-même par un tag spécifique. Pour inclure un centre de décision, placez le curseur dans la fenêtre de texte où le centre de décision devra prendre effet. Un clic sur le bouton *Add a Decision Center* vous donnera la liste des choix possibles. En ce qui concerne les Zones, ce même bouton ajoutera le centre de décision à une liste. Cette liste représente les tags interprétables à l'exécution de la zone. Il vous appartient de les insérer à besoin dans le source HTML que vous désirez afficher. Pour les autres pages, les tags apparaissent en vert. Un clic sur ce tag ou dans le panel de visualisation (à droite, en bas) fait apparaître les détails de ce centre de décision.

Les centres de décision (hormis les afficheurs de valeur système) sont accompagnés d'une action d'entrée (*Entry Action*): une suite d'instructions Xion exécutées en même temps que les centres de décision (avec la page ou à l'appel pour les *lieurs* et les *exécuteurs de formulaire*). On peut aussi leur définir des variables contextuelles qui devront être initialisées dans l'action d'entrée. Les variables de sessions sont toujours accessibles. Comme un même centre de décision peut être utilisé par plusieurs fichiers, il ne sera capable d'accéder qu'aux variables communes à ces fichiers, c'est à dire de même nom, et dont le plus grand ancêtre commun n'est pas *OclAny* ou *Collection(OclAny)*. Par exemple si le centre de décision 1 est utilisé par les fichiers `a.html` de contexte `v` de type `Integer` et `b.html` de contexte `v` de type `Double`, la variable `v` sera vue dans ce centre comme étant du type `Real`.

Les centres de décision composeurs, afficheurs de collections, lieurs et formes ont tous besoin d'un fichier cible. Il est possible de choisir de façon dynamique ce fichier cible à l'aide de décision définie par une expression en Xion retounant une valeur Boolean . Les décisions sont ordonnées et évaluées tour à tour. C'est la première à vrai qui sera choisie. Une décision particulière est la décision par défaut, qui sera élue si aucune autre ne l'a été. Ajouter une décision (ou un décision par défaut) se fait par le menu contextuel dans le panneau de visualisation:



Le menu contextuel de la décision (dans le panneau de visualisation) permet de modifier la priorité de la décision, de la rendre par défaut ou de la détruire (ce qui ne détruit évidemment pas le fichier lié). L'expression de la décision (si elle n'est pas par défaut) se fait dans le panel *Expression of the Constraint*. Le passage de paramètres au fichier lié se fait dans le panneau *Context of the called file*. Un résumé des variables accessibles est fait dans le panneau *Accessible Variables*.

IV - Xion - Grammaire

Il s'agit ici de la grammaire utilisée qui parse le langage Xion au format EBNF. Cette grammaire a été implémentée à l'aide de l'outil ANTLR 2.7.0. Les caractères sont entre ' et les mots-clé entre ". Un ? représente une option, un * une répétition, un + une répétition d'au moins une occurrence, un | sépare différentes possibilités, un ~ est l'anti-règle et la suite .. donne un itervalle entre deux caractères. Les caractères blanc, saut de ligne, retour chariot, saut de page, tabulation sont ignorés, tout comme la règle comment.

```
program::=   (initializer)?   (statement)*

initializer::=   "super"   '('   ( exprList )?   ')'

compoundStatement::=   '{'   ( statement )*   '}'

statement::=
      compoundStatement
   |  expression ';'
   |  declaration   ';'
   |  "if"   '('   expression   ')'   statement   ( "else"   statement )?
   |  "for"   '('   forInit   ';'   forCond   ','   forIter   ';'   ')'   statement
   |  "while"   '('   expression   ')'   statement
   |  "do"   statement   "while"   '('   expression   ')'   ';'
   |  "return"   (expression)?   ';'
   |  "write"   (expression)?   ';'
   |  "writeRaw"   (expression)?   ';'
   |  ';'

forInit::=   ( declaration   |   expressionList )?

forCond::=   ( expression )?

forIter::=   ( expressionList )?

declaration::=   type   variableDefinitions

variableDefinitions::=   variableDeclarator   ( ','   variableDeclarator )*

variableDeclarator::=   IDENT varInitializer

varInitializer::=   ( '='   expression )?

type::=
      identifier
   |  builtInType
   |  collectionType.
```

```
builtInType::=
        "Void"
    |   "Boolean"
    |   "Integer"
    |   "Real"
    |   "String"
    |   "Date"
    |   "Time"
    |   "OclAny"
    |   "OclType"
    |   "Double"
    |   "Float"
    |   "Long"
    |   "Int"
    |   "Short"
    |   "Byte"

collectionType::=    collectionKind    '('    type    ')'

collectionKind::=
        "Collection"
    |   "Set"
    |   "Bag"
    |   "Sequence"

identifier::=    IDENT    ( "::"    IDENT )*

expression::=    assignmentExpression

expressionList::=    expression    ( ','    expression )*

assignmentExpression::=
       conditionalExpression
       (
          ( '=' | "+=" | "-=" | "*=" | "%=" | ">>=" | "<<=" | "&=" | "^=" | "|=" )
          assignmentExpression
       )?

conditionalExpression::=    logicalOrExpression    ( '?'    assignmentExpression    ':'
conditionalExpression )?

logicalOrExpression::=    logicalAndExpression    ( "||"    logicalAndExpression )*

logicalAndExpression::=    inclusiveOrExpression    ( "&&"    inclusiveOrExpression )*

inclusiveOrExpression::=    exclusiveOrExpression    ( '|'    exclusiveOrExpression )*

exclusiveOrExpression::=    andExpression    ( '^'    andExpression )*

andExpression::=    equalityExpression    ( '&'    equalityExpression )*

equalityExpression::=    relationalExpression    ( ( '=' | "!=" )    relationalExpression )*
```

```
relationalExpression::=    shiftExpression    (
          ( ( '<' | '>' | "<=" | ">=" )    shiftExpression )*
      |   "instanceof"    type
      )

shiftExpression::=    additiveExpression    ( ( "<<" | ">>" )    additiveExpression )*

additiveExpression::=    multiplicativeExpression    ( ( '+' | '-' )    multiplicativeExpression )*

multiplicativeExpression::=    unaryExpression    ( ( '*' | '/' | '%' ) unaryExpression )*

unaryExpression:
      "++"    unaryExpression
   |  "--"    unaryExpression
   |  '-'    unaryExpression
   |  '+'    unaryExpression
   |  unaryExpressionNotPlusMinus

unaryExpressionNotPlusMinus::=
      '~'    unaryExpression
   |  '!'    unaryExpression
   |  '('    type    ')'    unaryExpression
   |  postfixExpression

postfixExpression::=    primaryExpression    ( ( '.' | "->" )    featureCall )*    ( ( "++" | "--" ) )?

featureCall::=
      ( identifier    ( '('    ( expressionList )?    ')'    |    ( '['    IDENT    ']' )?    '['    expressionList    ']' )?)
   |  ( ( '=' | "==" | '!' | '~' | "!=" | "<>" | '/' | '+' | '-' | '*' | '%' | ">>" | '>' | ">=" | "<<" | "<=" | '<' | '^' | '|' | '||' |
"&&" | '&') ( '('    expressionList    ')' )? )

primaryExpression::=
      newExpression
   |  constant
   |  "true"
   |  "false"
   |  "this"
   |  "self"
   |  "null"
   |  '('    assignmentExpression    ')'
   |  builtInType
   |  collectionType
   |  featureCall

newExpression::=    "new"    type    '('    ( expressionList )?    ')'

constant::=
      NUM_INT
   |  STRING_LITERAL
   |  NUM_FLOAT
   |  '#' IDENT
   |  litteralCollection

litteralCollection::=    collectionKind    '{'    ( expressionListOrRange )?    '}'
```

expressionListOrRange::= expression ((',' expression)+ | ".." expression)?

DIGIT::= '0' .. '9'

ALPHA::= 'a' .. 'z' | 'A' .. 'Z' | '_' | '$' | '\u00C0' .. '\u00D6' | \u00D8' .. '\u00F6' | '\u00F8' .. '\u00FF'

IDENT::= ALPHA (ALPHA | DIGIT)*

NUM_INT::= (DIGIT)+

NUM_FLOAT::= (DIGIT)+ ('.' (DIGIT)+ (EXPONENT)? | EXPONENT)

EXPONENT::= ('e' | 'E') ('+' | '-')? ('0' .. '9')+

ESC::= '\' ('n' | 'r' | 't' | 'b' | 'f' | '"' | ''' | '\' | ('0' .. '3') (('0' .. '7') (('0' .. '7'))?)? | ('4' .. '7') (('0'..'7'))?

STRING_LITERAL::= ''' (ESC | ~('\' | '''))* ''' | '"' (ESC | ~('\' | '"' '))* '"'

V - Xion - Documentation

Utilisations

Xion permet de décrire tout comportement. Il est donc utilisé afin de définir chacun des comportements, tant au niveau modèle métier que de la navigation. Il faudra distinguer les instructions des expressions. Les instructions sont l'ensemble des requêtes Xion disponibles, et les expressions représentent cet ensemble, diminué de l'affectation, des blocs, des boucles et branchements (*if*, *for*, *while*, *do*), du *return*, du *write* ,du *writeRaw* et des déclarations de variables.

Dans le Modèle Métier

Le modèle métier définit les classes qui décrivent les objets que représente le site. Chacune de ces classes a un comportement. Ce comportement est décrit par des méthodes et des constructeurs, décrits par une suite d'instructions en Xion.

Définition des Méthodes

Une méthode est la description de ce que réalise une opération. Cette méthode possède des paramètres typés et nommés et un retour typé.

Chacun des paramètres est contenu dans la variable du nom et du type décrit.

La description d'une méthode se fait par une suite d'instructions en Xion.

Un retour se fait par l'instruction *return* suivi d'une expression décrivant la valeur retournée par la méthode si le type de retour n'est pas *Void*, lequel représente le type "vide". Cette instruction met donc fin à l'exécution de la méthode.

L'accès à l'objet sur lequel s'applique cette méthode est fait par les variable *this* ou *self*.

On remarquera qu'une opération peut être marquée *isQuery* ou non. Une méthode modifiant un attribut ou un lien, ou encore créant un objet, ne peut pas implémenter une opération marquée *isQuery*.

Définition des Constructeurs

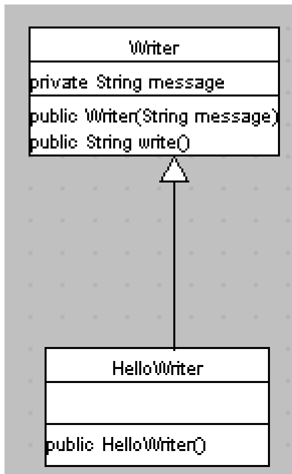
Un constructeur est une méthode automatiquement appelée à la création d'un objet. Elle est généralement utilisée pour initialiser les attributs et les liens de cet objet. Définir un constructeur n'est pas obligatoire, mais conseillé, pour être sûr qu'aucune valeur (d'attribut ou de lien) n'est fantaisiste. Il ne peut donc pas être marqué *isQuery*. Un constructeur est une méthode dont le nom est celui de la classe, et dont le type de retour est *Void*. Par exemple, pour une classe *MaClasse*, il faut créer une opération (*Add/Operation*) qu'on nommera *MaClasse*, en veillant à ce que le type de retour est bien *Void*. Un constructeur peut posséder un certain nombre d'arguments. S'il n'en a pas, il est dit "constructeur par défaut".

En cas d'héritage, il peut être nécessaire d'indiquer quel constructeur utiliser. En effet, l'objet créé est aussi du type de la classe ancêtre. Plusieurs cas se distinguent:

Si la super-classe (ou classe ancêtre) ne possède pas de constructeur: on ne peut donc pas appeler le constructeur de la super-classe !

Si la super-classe possède un ou plusieurs constructeurs: on doit donner quel constructeur est appelé parmi ceux de la super-classe à l'aide de l'instruction *super*. Par exemple si ma super-classe possède un constructeur ayant pour paramètres *Int, Boolean*, et que je souhaite l'appeler, la première instruction du constructeur sera *super(1, true)*. Si un des paramètre nécessite un calcul lourd, une astuce consiste à définir une méthode statique de la classe qui fera ce calcul.

Un petit exemple



Méthode *public Writer(String message)*: "this.message = message;"

Méthode *public String write()*: "return this.message;"

Méthode *public HelloWriter()*: "super("Hello World");"

Pour écrie "Hello World", il faut demander à un objet de type *HelloWriter* de faire *write()*, par exemple *new HelloWriter().write();*

Dans le Modèle de Navigation

Un centre de décision permet un certain nombre de choix quant à l'affichage ou la navigation entre pages. Un centre de décision a pour rôle de décider quel partie de page sera affichée ou quelle nouvelle page sera activée, suivant un certain nombre de critères de choix. Il peut aussi être nécessaire d'effectuer quelque calcul préalable à la décision. On rappelle qu'une page, lorsqu'elle appelle un centre de décision lui envoie un certain contexte (entrant dans le centre), souvent soit pour les calculs du centre de décision, soit pour les passer à la page suivante. De plus, avant d'activer une nouvelle page, il est nécessaire de lui envoyer un nouveau contexte (sortant du centre), qu'il faudra initialiser. Le contexte entrant est celui de la page d'appel, ou l'ensemble des variables de même nom et de même type s'il s'agit de plusieurs pages. Existe aussi un contexte dit session.

Définition des Critères de choix

Un critère de choix permet à un centre de décision de choisir la page appelée par une autre page. Un centre de décision possède un certain nombre de critères, chacun relié à une page. Ces critères sont ordonnés et retournent un résultat booléen. Si le premier est vrai, alors la page qui lui est liée est appelée, sinon, on teste le suivant, etc... Si aucun critère n'est vérifié, une page par défaut (si elle existe) est appelée.

Un critère est défini par une suite d'instruction en Xion. Évidemment *this* et *self* ne représentent rien.

Définition des Actions préalables

Il peut être nécessaire d'effectuer certains calculs avant de calculer les critères. Ceci est possible par l'intermédiaire de la définition des actions préalables, toujours en instructions Xion. A noter que chacune des variables définies dans le centre de décision (hormis celles correspondant à des formes) devra être affectée à une certaine valeur.

Accès au Contexte entrant

Un contexte qui entre dans un centre de décision (c'est à dire envoyé par la page qui l'appelle) est une suite de variables d'un certain nom, d'un certain type et d'une certaine valeur. Un simple appel à cette variable est possible dans les actions préalables, les critères de choix ou la définition du contexte sortant.

Construction du Contexte sortant

Lorsqu'une page a été choisie par le centre de décision, son appel nécessite la construction d'un nouveau contexte à envoyer à l'appel de cette dernière. On l'a dit, un contexte est une suite de variables d'un certain nom, d'un certain type et d'une certaine valeur. Ceci est simplement fait en Xion en affectant chaque variable à la valeur désirée. Il faut évidemment définir chacune de ces variables (ce qui n'est pas obligatoire lorsqu'elle possède une valeur par défaut).

Destruction de la Session

Il est possible de déclarer un contexte dit session, soit une suite de variables définies pour chaque entrée d'un client dans le site. Ces variables sont sauvegardées sur le serveur. Il peut donc être utile, pour des raisons de sécurité, ou plus simplement de place, de détruire cet enregistrement. Ceci est possible grâce au mot-clé du langage *DestroySession*.

Sémantique de base

Il faut noter que ce langage différencie les majuscules des minuscules.

Commentaires

Il existe deux types de commentaires: les commentaire sur une ligne qui commencent pas les symboles *//* et qui finissent à la fin de la ligne, et les commentaires indépendants, débutant sur les symboles */** et finissant sur les symboles **/*.

Exemple:

//Créons ici une variable de type WelloWriter
HelloWriter hello*/* = new HelloWriter()*/*; */*L'initialisation est ici ignorée mais le point-virgule est nécessaire*/*

Séparateurs d'instructions

Un programme en Xion est une suite d'instructions Xion (une ou plusieurs). Chaque instruction est séparée des autres par un *;*, sauf dans le cas de bloc d'instructions. Les gestionnaires de déroulement if , else, while et for sont séparés des autres instructions par celle qu'ils pilotent (exactement comme en en C ou en Java).

Blocs

Il est possible de grouper un certain nombre d'instructions Xion en les englobant par les caractères *{* et *}*. Ceci peut avoir plusieurs intérêts, notamment dans les instructions if, else, while, do et for ou afin de déclarer des variables qui ne seront visibles qu'à l'intérieur même dudit bloc. Un bloc n'est pas une expression.

Variables

Les variables sont déclarées à la manière de C/Java. Tout d'abord on indique le type souhaité, puis, séparés par des virgule, les noms des variables éventuellement suivies par leur valeur d'initialisation après un =. Une déclaration de variables est considérée comme une instruction, et non comme une expression.

Exemple: *Integer i = 0, j = ++i, k = i + j + 3;*

Assignations

Pour affecter une variable ou un champ accessible, il suffit d'utiliser le symbole =. Une assignation n'est pas une expression.

Exemple: *Integer i; i = 123456789;*

Instructions conditionnelles

Il existe plusieurs instructions qui permettent de contrôler l'exécution du code à l'aide d'expression booléennes. Il ne s'agit pas d'expressions. Sont disponibles:

- le *if*: qui permet d'exécuter une instruction (pu un bloc) seulement si sa condition est vraie
  ex: *if (opNeeded) op()*; où *opNeeded* est une variable booléenne
- le *else*: qui permet d'xécuter une instruction (ou un bloc) dans le cas d'un *if* évalué à faux précédemment
  ex: *if (isEven) doEven(); else doOdd();*
- le *while*: qui répète une instruction (ou un bloc) tant que sa condition est vraie (la condition est évaluée avant l'instruction)
  ex: *while(i > 10) i = i - 1; //si i = 10 au départ, l'instruction n'est pas exécutée*
- le *do*: qui répète une instruction (ou un bloc) tant que sa condition est vraie (la condition est évaluée après l'instruction)
  ex: *do i = i - 1; while (i > 10); //si i = 10 au départ, l'instruction est exécutée une fois tout de même*
- le *for*: un *while* qui définit des pré-instructions, une condition de bouclage (évaluée avant l'exécution de l'instruction) et une instruction de fin de boucle, exécutée après l'instruction
  ex: *for (Integer i = 0; i < 10; ++i) do(); //Ecécute 10 fois la fonction do() - Equivalent à: Integer i = 0; while (i < 10) {do(); ++i;}*

Re-typage ou "Cast"

Il peut être nécessaire de changer le type d'une valeur sous forme d'expression, par exemple lorsqu'on est sûr de son type. Il existe deux moyens (dont les différences seront expliquées plus tard) pour cette action. Par exemple pour transformer un *Real* en *Integer*, lorsque ce *Real* n'a pas de partie réelle.

On notera qu'il n'est possible de retyper que des expressions qui sont du super-type du type cible (par exemple, une expression réelle a une chance de pouvoir être transformée en entier, mais pas une expression booléenne)...

Comme en en C/Java: on indique le type cible entre parenthèses; ex: *Real r = 1.0; Integer i = (Integer)r;*

Une opération définie sur tous les types: *oclAsType*: *Real r = 1.0; Integer i = r.oclAsType(Integer);*

A noter les casts automatiques qui transforment des types enfants en type parents. Par exemple pour la méthode *max(Real)* de *Real*, qui peut être appelée par *monReal.max(12)*, bien que *12* soit un *Integer*, type enfant de *Real* (on dit aussi sous-type). Ici, Xion a retypé *12* en *Real* sans que l'utilisateur aie eu besoin de le signaler.

Traçage

Pour des besoins de déverminage, par exemple, il peut être nécessaire d'afficher certaines valeurs, lors de l'exécution d'un script. Ceci est rendu possible par les instructions *write* et *writeRaw*. Ces instructions sont suivies par une expression dont la valeur est à afficher. *write*, en plus de *writeRaw*, fera intervenir un filtre HTML pour un affichage correct lors de la visualisation de la page. Par exemple, là où *write "toto
tutu";* affichera *toto
tutu* à l'écran, *writeRaw "toto
";* affichera sur une première ligne *toto*, et sur une seconde *tutu*, *
* étant la balise HTML de saut de ligne.

Types Prédéfinis

Pour pouvoir commencer à travailler, il est nécessaire d'avoir un certain nombre de types de base. Les types de base de Xion ne sont autres que des classes prédéfinies pour lesquelles seul l'opérateur *new* n'est pas accepté. Chacun de ces type est accompagné d'une série d'opérations prédéfinies. Il faut savoir qu'une opération prédéfinie sans paramètres s'appelle sans les parenthèses.

OclAny

OclAny est un type abstrait (qu'on ne peut instancier), père de tous les autres (hormis les collections). Les opérations prédéfinies dans ce type sont accessibles dans tous les autres, y compris sur les types énumérés et sur les classes du modèle métier. On trouve notamment:

- *oclIsTypeOf(OclType):Boolean*: pour déterminer si une valeur est du type donné
- *oclIsKindOf(OclType):Boolean*: pour déterminer si le type d'une valeur est du type, ou est un sous-type du type donné
- *= (OclAny):Boolean*: compare une valeur à une autre; vrai seulement s'il sagit de la même
- *<>(OclAny):Boolean*: l'opération opposée de =
- *oclAsType (OclType)*: pour changer définitivement le type d'une valeur; la virtualité, dans ce cas, ne jouera plus

Exemples:

```
Integer i = 0;
i.oclIsTypeOf(Real);//retourne faux
i.oclIsTypeOf(Integer);//retourne vrai
i.oclIsKindOf(Real);//retourne vrai
i.=(0);//retourne vrai
i.<>(0);//retourne faux
i.oclAsType(Real).+(1);//la valeur retournée est un Real
```

Types de Base

Ces types sont visibles pour le programmeur à tout moment. On ne les crée pas par un new mais par une valeur littérale.

Real

Comme son nom l'indique, il s'agit du type des nombres réels. Comme indiqué plus haut, *Real* est un sous-type de *OclAny*. Les valeur littérales qui lui correspondent sont du format

('0' .. '9')+

(('.' ('0' .. '9')+)

| (('e' | 'E')   ('+' | '-')?   ('0' .. '9')+   )

| ('.'   ('0' .. '9')+   ('e' | 'E')   ('+' | '-')?   ('0' .. '9')+   )

)

Exemples: 0.1   17.12   3.0   3e-9   3e7   3E+18   38.545e+12

Le type *Real* s'accompagne notamment des opérations prédéfinies:

- = et <>: opérations d'égalité et d'inégalité
- +, -, *, /: les quatre opérations de base
- *floor:Integer*: qui retourne la partie entière
- *abs:Real*: la valeur absolue; par exemple *(- 12.1753e-17).abs* donnera *12.1753e-17* (notez l'absence de parenthèses à l'appel de la fonction abs)
- *max* et *min*: qui retournent respectivement le maximum et le minimum entre le réel auxquels il s'applique et leur paramètre; par exemple *(12.5).max(-17.3)* retournera le réel *12.5*
- <, >, <=, >=: les quatre comparaisons de base

Comme *Real* est une classe fille (hérite) de *OclAny*, sont aussi accessibles des opérations définies dans *OclAny* telles que *oclAsType* ou *oclIsKindOf*. Une valeur de type *Real* peut aussi être re typée en *OclAny*.

Integer

Comme son nom l'indique, il s'agit du type entier. Le type entier est une sorte de réel, donc *Integer* est un sous-type de *Real*. Les valeurs littérales qui lui correspondent sont du format *('0' .. '9')+*

Exemples: 1 1234 45612348 0

Le type *Integer* s'accompagne notamment des opérations prédéfinies:

- +, -, *, /: les quatre opérations, ici définie entre deux entier; a noter que la division entre deux entiers retourne un résultat réel
- *div* et *mod*: respectivement la division entière et le modulo; par exemple *13.div(2)* retourne l'entier *6* et *13.mod(2)* retourne l'entier *1*
- *min* et *max*: le minimum et le maximum entre entiers

De plus, comme *Integer* hérite de *Real*, et que *Real* hérite de *OclAny*, on y trouve aussi des opération telles que *floor*, *oclAsType* ou *oclIsKindOf*. Une valeur de type *Integer* peut aussi être re typée en *OclAny* ainsi qu'en *Real*.

Boolean

Comme son nom l'indique, il s'agit du type booléen, donc a pour valeur possibles *true* et *false* (états vrai et faux). *Boolean* est un sous-type de *OclAny*.

Le type *Boolean* s'accompagne notamment des opérations prédéfinies:

- *or*, *xor*, *and*, *not*: respectivement le ou inclusif, le ou exclusif, le et et le non
- *implies*: si l'objet auquel on l'applique est faux, le résultat est vrai; si l'objet est vrai, le résultat est le paramètre; par exemple *true.implies(false)* retourne la valeur *false*

Comme *Boolean* hérite de *OclAny*, sont aussi accessibles des opérations définies dans *OclAny* telles que *oclAsType* ou *oclIsKindOf*. Une valeur de type *Boolean* peut aussi être re typé en *OclAny*.

Text / String

Comme leurs noms l'indiquent, il s'agit des type chaînes de caractères. *Text* est un sous-type de *OclAny,* et *String* de *Text*. *Text* correspond aux chaînes de longueur indéfinie, et *String* à celles contenant au maximum 255 caractères. Elle est donnée indifféremment entre caractères *'* ou *"*, par exemple *'coucou'* ou *"hibou"*. Il existe un mécanisme de caractères dit échappatoires. Ces caractères sont commencés par le caractère \ et le code qui suit détermine ledit caractère:

- *\n* fournira un retour-chariot (changement de ligne); par exemple *'ligne 1\nligne2'*
- *\r* fournira un saut de ligne
- *\t* fournira une tabulation
- *\b* fournira un effacement à droite; par exemple *'Xia\bon'* s'afficherait *Xion* (mais est différent de *'Xion'*)
- *\f* fournira un saut de page
- *\"* et *\'* fourniront respectivement les caractères *'* et *"*, ce qui permet de ne pas finir la chaîne prématurément; par exemple *'une chaîne peut être entre caractères \"* serait affiché *une chaîne peut être entre caractères '*
- \\ fournira le caractère \
- *\0* à *\377* fournira le caractère unicode qui correspond au code octal donné; par exemple, sachant que le code unicode pour *é* est 351, *'\351'* est égal à *'é'*

Le type *Text* s'accompagne notamment des opérations prédéfinies:

- =: qui compare deux chaînes
- *size*: qui retourne le nombre de caractères dans la chaîne
- *concat*: qui concatène deux chaînes
- *toUpper* et *toLower* qui retourne la chaîne respectivement tout en majuscules ou en minuscules
- *substring*: qui retourne la chaîne comprise entre les balises données; par exemple *'coucou'.substring(0, 2)* donnera *'cou'*

Le type *String* s'accompagne notamment des opérations prédéfinies:

- *checkInteger*, *checkReal*, *checkDate*...: qui vérifient qu'il est possible de convertir une chaîne en un *Integer*, un *Real*, une *Date*...
- *parseInteger, parseReal, parseDate...*: qui convertissent une chaîne en *Integer*, en *Real*, en *Date*...

Comme *Text* hérite de *OclAny*, et *String* de *Text* sont aussi accessibles des opérations définies dans *OclAny* telles que *oclAsType* ou *oclIsKindOf*. Une valeur de type *String* peut aussi être re typé en *OclAny* ou en *Text*.

OclType

Comme son nom l'indique, il s'agit du type type. *OclType* est un sous-type de *OclAny*. Ce type peut avoir pour valeur chacun des types existant, que ce soient des types prédéfinis, des collections, des énumérés ou des classes du modèle métier.

Le type *OclType* s'accompagne notamment des opérations prédéfinies

- *name*: qui retourne le nom du type; par exemple *OclType t = OclAny; return t.name;//ceci retourne la chaîne 'OclAny'*
- *attributes*: qui retourne les noms de chacun des attributs
- *associacionEnds*: qui retourne les noms de chacun des liens
- *operations*: qui retourne les noms de chacune des opérations;
- *allInstances*: qui retourne chacune des instance créées ayant ce type; seuls les types (ou classes) du modèle métier (y compris les énumérés) peuvent retourner des instances

Comme *OclType* hérite de *OclAny*, sont aussi accessibles des opérations définies dans *OclAny* telles que *oclAsType* ou *oclIsKindOf*. Une valeur de type *OclType* peut aussi être re typé en *OclAny*.

Date

Comme son nom l'indique, il s'agit du type date. *Date* est un sous-type de *OclAny*. Il n'existe cependant pas de littéraux associés à ce type. La seule façon d'en créer est donnée par les opérations statiques *Date.getCurrent* et *Date.getDate(Integer year, Integer month, Integer day)*.

Par exemple: *Date.getCurrent* retoune la valeur d'aujourd'hui; *Date.getDate(2000, 7, 14)* retourne le 14 juillet 2000.

Le type *Date* s'accompagne notamment des opérations prédéfinies

- *addDay*, *addMonth* et *addYear*: qui ajoutent un certain nombre de jours/mois/années
- *getDay*, *getMonth*, *getYear*: qui retourne la cardinalité des jours/mois/années
- *min* et *max*: qui donne la valeur la plus vielle/récente
- <, >, <=, >=: qui compare les dates

Comme *Date* hérite de *OclAny*, sont aussi accessibles des opérations définies dans *OclAny* telles que *oclAsType* ou *oclIsKindOf*. Une valeur de type *Date* peut aussi être re typé en *OclAny*.

Time

Comme son nom l'indique, il s'agit du type heure. *Time* est un sous-type de *OclAny*. Il n'existe cependant pas de littéraux associés à ce type. La seule façon d'en créer est donnée par les opérations statique *Time.getCurrent* et *Time.getTime(Integer hour, Integer minute, Integer second)*.

Par exemple: *Time.getCurrent* retoune la l'heure courante; *Time.getime(16, 0, 0)* retourne l'heure du goûter.

Le type *Time* s'accompagne notamment des opérations prédéfinies

- *addHour*, *addMinute*, *addSecond* et *addMilli*: qui ajoutent un certain nombre d'heures/de minutes/de secondes/de millisecondes
- *getHour*, *getMinute*, *getSecond*, *getMilli*: qui retourne la cardinalité des heures/minutes/secondes/millisecondes
- *min* et *max*: qui donne la valeur la plus vielle/récente
- <, >, <=, >=: qui compare les heures

Comme *Date* hérite de *OclAny*, sont aussi accessibles des opérations définies dans *OclAny* telles que *oclAsType* ou *oclIsKindOf*. Une valeur de type *Date* peut aussi être re typé en *OclAny*.

Opérateurs et Règle de Précédence

La notation d'appel de méthodes peut sembler lourde ou déroutante, c'est pourquoi Xion définit un certain nombre d'opérateurs. Par exemple, pour éviter de taper *1.+(1)*, on peut écrire *1 + 1*. Comme dans tout langage, ces opérateurs on une certaine priorité relative aux autres. Par exemple 1 + 2 * 3 n'équivaudra pas à *1.+(2).*(3)* mais à *1.+(2.*(3))*. Voici sous forme de tableau la liste des opérateurs dans leur ordre de priorité (de la plus haute à la plus basse) . Il faut noter que la recherche des opérations se fait non pas sur un type particulier, mais sur le nom. L'opération appelée est recherchée sur le type de l'expression sur laquelle l'opérateur est appelé, avec un éventuel type de paramètre, qui est celui de l'expression qui suit l'opérateur. Ce mécanisme permet de redéfinir l'opération appelée pour pouvoir appeler l'opérateur correspondant. Si on défini par exemple une classe *A* avec l'opération *+(Integer):A*, il sera possible sur un objet *monA* de type *A* de faire *monA+1* et *monA*++.

Nom de l'opérateur	Opération recherchée	Exemple	Expression équivalente	priorité
+ (unaire)	+	+1	1.+	1
- (unaire)	-	-1	1.-	1
~ (unaire)	~	~1	1.~	1
! (unaire)	not	! true	true.not	1
*	*	1 * 2	1.*(2)	2
/	/	1 / 2	1./(2)	2
%	mod	1 % 2	1.mod(2)	2
+	+	1 + 2	1.+(2)	3
-	-	1 - 2	1.-(2)	3
<<	<<	1 << 2	1.<<(2)	4
>>	>>	1 >> 2	1.>>(2)	4
<	<	1 < 2	1.<(2)	5
>	>	1 > 2	1.>(2)	5
<=	<=	1 <= 2	1.<=(2)	5
>=	>=	1 >= 2	1.>=(2)	5
instanceof	oclIsKindOf	1 instanceof Integer	1.oclIsKindOf(Integer)	5
==	=	1 == 2	1.=(2)	6
!=	<>	1 != 2	1.<>(2)	6
&	&	1 & 2	1.&(2)	7
^	^	1 ^ 2	1.^(2)	8
|	|	1 | 2	1.|(2)	9
&&	and	true && false	true.and(false)	10
||	or	true || false	true.or(false)	11
implies	implies	true implies false	true.implies(false)	12

Il existe aussi des raccourcis de codes tel les opérateurs ++, --, +=, -=, *=, %=, <<=; >>=, &=, |=; ^=, qui correspondent respectivement sur *x* à *x = x + 1*, *x = x - 1*, et si on les appliques à *x* et *y*, *x = x + y*, *x = x - y*, *x = x * y*, x = *x % y*, *x = x << y*, *x = x >> y*, *x = x & y*, *x = x | y*, *x = x ^ y*. Il faut évidemment que les types concordent.

Un autre opérateur est en l'opérateur ternaire. Cet opérateur est conditionné par une expression booléenne, et va renvoyer le résultat d'une expression ou d'une autre suivant si elle est évaluée à vrai ou faux. Par exemple, on peut vouloir transformer un booléen *b* en chaîne de caractères:

String bAsString = b ? "true" : "false";

Collections

Xion ne définit pas de tableau mais donne les type collections. Une collection est un conteneur d'un certain type, par exemple de *Boolean*, d'une classe du modèle ou d'un énuméré. Il est possible de définir des collections de *OclAny*. Une collection ne peut contenir la valeur *null*: essayer de l'intégrer ne mènera à rien. Il faut principalement prendre garde à l'accès aux opérations de ces collections.

Accès aux Opérations

L'accès à une opération de collection est un peu différent de l'accès aux champs d'un objet ou des opérations prédéfinies sur les types simples. En effet, l'indirection ne se réalise plus par le caractère *.* mais par la flèche ->. Le point reste valide mais réservé à un autre usage. Il sert de raccourci au langage pour l'appel de l'opération *collect*, décrite plus bas.

Exemples:

- accéder à l'opération size sur un Set vide:
  Set {}->size
- accéder à l'opération *collect* sur la variable *x* de type *Set(Integer)*, pour obtenir un *Bag* contenant la valeur absolue de chacun des éléments (*abs* étant l'opération sur *Integer* pour en sortir la valeur absolue):
  x->collect(abs) ou *x.abs*

Cette méthode peut sembler compliquée à assimiler, cependant, l'habitude aidant, elle permet une certaine souplesse... Ne vous laissez donc pas décourager !

Collection

Collection est le type abstrait (non implémentable) pour toutes les collections. Il n'hérite pas de *OclAny*. Une collection est toujours donnée sur un certain type, par exemple sur des *Integer*. Dans ce cas, le type s'appelle *Collection(Integer)*. Sachant de surcroît que *Integer* hérite de *Real*, et que *Real* hérite de *OclAny*, *Collection(Integer)* héritera de *Collection(Real)*, et *Collection(Real)* héritera de *Collection(OclAny)*. Ceci est important à connaître pour retyper une valeur collection. Il n'est cependant pas possible de définir des collections sur des collections...

Le type *Collection* s'accompagne notamment des opérations prédéfinies:

- *size*: qui donne le nombre d'éléments inclus
- *includes* et excludes: qui déterminent si un objet est présent ou absent de la collection
- *count*: qui compte le nombre de fois où un objet est présent dans la collection
- *includesAll* et *excludesAll*: qui vérifient si la collection contient tous ou aucun des éléments de la collection passée en argument
- *isEmpty* et *notEmpty*: qui indiquent si la collection est vide

De plus, il existe plusieurs opérations dites "d'itérations", qui vont évaluer l'expression qui leur est passée en argument sur chacun des éléments que la collection possède. Par exemple pour la fonction *isUnique*, qui détermine si ladite expression retourne des valeurs différentes pour chaque élément de la collection.

Pour une *Collection(String)* contenant *"a"*, *"ab"*, *"abc"*, l'opération *isUnique(size)* retournera *true*. Si cette collection avait contenu *"ba"* en plus, le retour aurait été *faux*. En effet, dans le premier cas, *size*, qui aura été évalué sur chaque élément, aura retourné *1*, *2* et *3*. Dans le second cas, elle aura retourné *1*, *2*, *3* et *2*. isUnique aura alors vérifié que *2* égalait *2*.

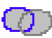
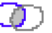
Set

Un *Set* est un ensemble au sens mathématique du terme, c'est à dire qu'il n'accepte en aucune façon de référencer deux fois un même objet. Il faut remarquer que deux éléments sont considérés différents à partir du moment où l'opération *OclAny.=(OclAny)* entre ces deux objets retourne faux. Le *Set* est une collection, donc hérite de *Collection*.

Pour initialiser un *Set*, il suffit de d'indiquer *Set { <ma première valeur>, <ma seconde valeur>, ...}*. Les valeurs peuvent être exprimées par des expressions Xion. Pour des *Set* d'entiers, on peut donner un intervalle: *Set { <mon premier entier> .. <mon dernier entier>}*. Le *Set* vide est donné par *Set {}*.

Si on prend un *Set* sur des *Integers* (défini par *Set(Integer)*), *Integer* héritant de *Real* et *Real* de *OclAny*, un *Set(Integer)* équivaut à un *Set(Real)*, un *Set(OclAny)*, une *Collection(Integer)*, une *Collection(Real)* et une *Collection(OclAny)*.

Le type *Set* s'accompagne notamment des opérations prédéfinies:

- *union*: qui réunit les éléments de deux *Set* ou d'un *Set* et un *Bag* 
- *intersection*: qui réunit les éléments commun à deux *Set* ou un *Set* et un *Bag* 
- -: qui réunint les éléments d'un *Set* qui ne sont pas dans un autre *Set*
- *symmetricDifference*: qui réunit les éléments non communs de deux *Set*
- *including* et *excluding*: qui retournent un Set incluant (sauf si elle est déjà existante) ou excluant la valeur donnée
- *select*, *reject* et *collect*
- *asSequence* et *asBag*: qui transforme le *Set* en une *Sequence* ou un *Bag*

Sont bien sûr accessibles les opérations définies dans *Collection*.

Bag

Un Bag est un conteneur où une même valeur peut apparaître plusieurs fois. Comme le *Bag* est une collection, il hérite de *Collection*.

Pour initialiser un *Bag*, il suffit de d'indiquer *Bag { <ma première valeur>, <ma seconde valeur>, ...}*. Les valeurs peuvent être exprimées par des expressions Xion. Pour des *Bag* d'entiers, on peut donner un intervalle: *Bag { <mon premier entier> .. <mon dernier entier>}*. Le *Bag* vide est donné par *Bag {}*.

Si on prend un *Bag* sur des *Integers* (défini par *Bag(Integer)*), *Integer* héritant de *Real* et *Real* de *OclAny*, un *Bag(Integer)* équivaut à un *Bag(Real)*, un *Bag(OclAny)*, une *Collection(Integer)*, une *Collection(Real)* et une *Collection(OclAny)*.

Le type *Bag* s'accompagne notamment des opérations prédéfinies:

- *union* ( ), *intersection* ( ): opérations sur les ensembles
- *including*, *excluding*: le même ensemble en y incluant (peut être une fois de plus) ou en y excluant (chaque occurrence) un valeur
- *select*, *reject* et *collect*
- *asSequence*, *asSet*: qui transforment le *Bag* en une *Sequence* ou un *Set*

Sont bien sûr accessibles les opérations définies dans *Collection*.

Sequence

Une *Sequence* est un ensemble de type *Bag* où les différents éléments (deux occurrences d'une même valeur sont deux éléments distincts) sont ordonnés, donc possèdent une place bien précise au sein de la collection; le premier élément d'une *Séquence* a pour cardinalité *0*. Comme la *Sequence* est une collection, elle hérite de *Collection*.

Pour initialiser une *Sequence*, il suffit de d'indiquer *Sequence { <ma première valeur>, <ma seconde valeur>, ...}*. Les valeurs peuvent être exprimées par des expressions Xion. Pour des *Sequence* d'entiers, on peut donner un intervalle: *Sequence { <mon premier entier> .. <mon dernier entier>}*. La *Sequence* vide est donné par *Sequence {}*.

Si on prend une *Sequence* sur des *Integers* (défini par *Sequence(Integer)*), *Integer* héritant de *Real* et *Real* de *OclAny*, une *Sequence(Integer)* équivaut à une *Sequence(Real)*, une *Sequence(OclAny)*, une *Collection(Integer)*, une *Collection(Real)* et une *Collection(OclAny)*.

Le type *Sequence* s'accompagne notamment des opérations prédéfinies:

- *union*: l'union de deux Séquence, l'une après l'autre
- *append* (ou *including*), *prepend*: ajoute un élément en début ou fin de liste
- *at*: l'élément d'une position précise
- *first*, *last*: le premier ou le dernier élément
- *subSequence*: un tronçon de la *Sequence*
- *excluding*: une *Sequence* sans aucune occurrence de la valeur donnée
- *select*, *reject* et *collect*
- *asSet*, *asBag*: qui transforment la *Sequence* en un *Set* ou un *Bag*

Sont bien sûr accessibles les opérations définies dans *Collection*.

exists et forAll

exists et *forAll* sont des opérations de la classe *Collection* dites d'itération. Ces opérations prennent en argument non pas une valeur mais une expression qui s'appliquera tour à tour sur chacun des éléments. Cette expression doit retourner un *Boolean*. *exists* vérifiera qu'il existe au moins un élément pour lequel cette expression est vraie, et *forAll* vérifiera que pour chacun des éléments cette expression est varie.

Exemples:

Soit *Voiture*, classe du modèle métier qui possède entre autre un attribut public *couleur* du type énuméré *Couleur*. Le type énuméré *Couleur* contient entre autre *rouge*. Soit *parking* une collection de *Voiture*.

On se demande si il y a une voiture rouge sur le parking: *parking->exists(color* == # *rouge)*

On se demande si toutes les voitures du parking sont rouges: *parking->forAll(color* == # *rouge)*

Remarques:

Les opérations, attributs et liens sont alors recherchés sur l'élément en cours, et uniquement sur celui-ci. Pour accéder au contexte courant, il peut donc être nécessaire d'appeler l'opérateur *this*.

Il est aussi possible de nommer l'élément en cours dans l'expression de la manière suivante: *parking->exists(voiture: voiture.color* == # *rouge)*.

select et reject

select et *reject* sont des opérations des classes *Set*, *Bag* et *Sequence* dites d'itération. Ces opérations prennent en argument non pas une valeur mais une expression qui s'appliquera tour à tour sur chacun des éléments. Cette expression doit retourner un *Boolean*. *select* construira une nouvelle collection incluant chaque élément dont le résultat sera vrai. *reject* fera lui la même opération, mais pour des résultats faux.

Exemples:

Soit *Voiture*, classe du modèle métier qui possède entre autre un attribut public *couleur* du type énuméré *Couleur*. Le type énuméré *Couleur* contient entre autre *rouge*. Soit *parking* une collection de *Voiture*.

On veut trouver la collection des voitures qui sont rouges: *parking->select(color* == # *rouge)* ou *parking->reject(color != # rouge)*

On veut trouver la collection des voitures qui ne sont pas rouges: *parking->reject(color == # rouge)* ou *parking->select(color != # rouge)*

Remarques:

Les opérations, attributs et liens sont alors recherchés sur l'élément en cours, et uniquement sur celui-ci. Pour accéder au contexte courant, il peut donc être nécessaire d'appeler l'opérateur *this*.

Il est aussi possible de nommer l'élément en cours dans l'expression de la manière suivante: *parking->select(voiture: voiture.color == # rouge)*.

collect

collect est une opération des classes *Set*, *Bag* et *Sequence* dite d'itération. Cette opération prend en argument non pas une valeur mais une expression qui s'appliquera tour à tour sur chacun des éléments. *collect* construit un *Bag* (ou une *Sequence* lorsqu'elle s'applique sur une *Sequence*) contenant chacun des résultats de l'évaluation de cette collection. Comme dit précédemment, il existe un raccourci pour cette opération qui consiste à remplacer *->collect* par '*.*'.

Exemple:

Soit *Voiture*, classe du modèle métier qui possède entre autre un attribut public *couleur* du type énuméré *Couleur*. Soit *parking* une *Sequence(Voiture)*.

On veut trouver la liste des couleurs des voitures sur le parking: *parking->collect(color)* ou *parking.color*

Remarques:

Les opérations, attributs et liens sont alors recherchés sur l'élément en cours, et uniquement sur celui-ci. Pour accéder au contexte courant, il peut donc être nécessaire d'appeler l'opérateur *this*.

Il est aussi possible de nommer l'élément en cours dans l'expression de la manière suivante: *parking->collect(voiture: voiture.color)*.

Types Enumérés

Les types énumérés sont définis dans le modèle métier. Une valeur littérale de ce type est précédée du caractère #. Vous pouvez vous référer au exemples des opérations *select*, *exists* ou *collect*. Les types énumérés héritent de *OclAny*, sont ainsi accessibles des opérations définies dans *OclAny* telles que *oclAsType* ou *oclIsKindOf*. De plus existe une opération *toString* qui retourne la valeur de l'énuméré sous forme de *String*.

Types Supplémentaires

Integer et Real sont des types génériques, et leur format ainsi que leur taille dépend du langage script cible. Or un base de données est manipulée par le site généré pour stocker les objets créés. Cette base de données est donc formatée pour les types contenus dans les objets, et définis dans les classes. Comme *Real* et *Integer* sont génériques, les types entier et réel les plus précis (donc importants en place) sont utilisés. C'est pourquoi, pour limiter la taille de ladite base de données, que ce soit pour des problèmes de performances ou de place de stockage, ont étés ajoutés des types entier et réel dont la taille est fixée.

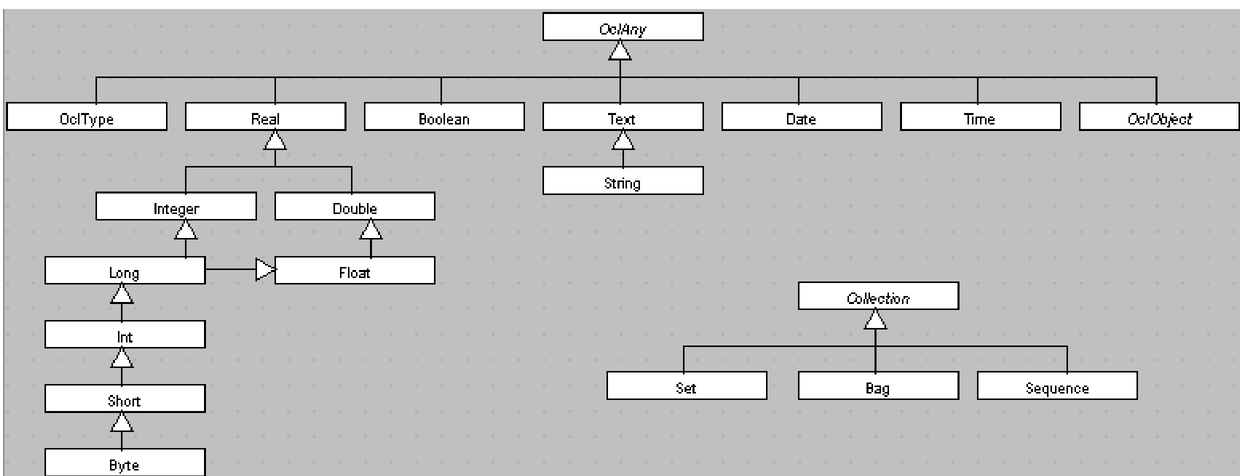
- Pour les entiers:

Byte est défini sur 8 bits, Short sur 16, Int sur 32 et Long sur 64. Sont aussi apparus les opérateurs sur les bits de ces valeurs telles que le et, le ou inclusif, le ou exclusif, ainsi que les décalages à gauche et à droite. De plus, des redéfinitions d'opérateurs y ont étré adjointes pour limier le grossissement des valeurs par exemple par un simple +. Les équivalences entre ces types sont données dans le paragraphe Relations entre Types.

- Pour les réels:

Float est défini sur 32 bits et Double sur 64. Des redéfinitions d'opérateurs y ont étré adjointes pour limier le grossissement des valeurs par exemple par un simple +. Les équivalences entre ces types sont données dans le paragraphe Relations entre Types.

Relations entre Types



Classes

Les classes, type des objets, sont définies la plupart du temps dans le modèle métier. Une classe peut être composée d'attributs, d'opérations, de liens qui peuvent être qualifiés, voire auxquels sont attachés des classes-associations: Xion a des mécanismes d'accès particuliers dans chacun de ces cas.

Instanciation d'Objet

L'objet est l'instance d'une classe. Pour créer un nouvel objet, il faut appeler l'opérateur *new*, suivi de la classe de l'objet, et enfin les paramètre du constructeur appelé. Si ladite classe ne possède pas de constructeur, les paramètres sont vides.

Exemple:

- Soit une classe A sans constructeur; créer un nouvel objet de classe A se fait par: *new A();*
- Soit une classe B avec un constructeur de paramètre *Integer*, *Boolean*; créer un nouvel objet de classe B se fait par: *new B(12, false);* Le constructeur de la classe B est alors immédiatement appelé sur le nouvel objet.

Un objet qui n'existe pas est donné par la constante *null*.

Unobjet est détruit (de la base) par l'opération *delete*.

Noms des Eléments

Certains noms dans le modèle, qu'ils appartiennent à des classes, des paquetages, ou des champs, peuvent comporter des blancs. Xion ne les supportent pas, donc ils on étés remplacés par le caractère _. Ainsi, si une classe s'appelle *Hello Writer*, dans le paquetage *Business Model*, son appel dans Xion se fera par *Business_Model::Hello_Writer*.

Accès aux différents Paquetages

Une classe appartient toujours à un paquetage et un paquetage peut appartenir à un autre paquetage. Xion définit un séparateur, *::*, qui, à l'instar du / Unix ou du \ DOS, permet d'accéder au contenu du paquetage défini. Par exemple, si vous voulez accéder à la classe *Field* du paquetage *reflect*, lui-même appartenant au paquetage *lang*, vous y accéderez par *lang::reflect::Field*.

Il n'est pas toujours nécessaire de donner le chemin complet d'accès (on dit chemin "absolu"), un chemin "relatif" peut suffire. Par exemple, si la classe Field de tout à l'heure était appelée par une méthode de la classe *Attribute* du même paquetage, *Field* aurait suffit. Et s'il avait sagit d'un appel par une méthode de la classe *Class* du paquetage *lang*, on aurait pu marquer *reflect::Field*.

Accès aux Champs d'une Classe

Qu'il soient attributs, liens ou opérations, une classe est composée de différents champs. Chacun de ces champs a une visibilité particulière. Cette visibilité peut être *public*, *protected* ou *private*.

- *public*: le champ est visible partout qu'il s'agisse du modèle métier ou des centres de décision
- *protected*: le champ n'est visible que par les sous-classes, ou encore les classes filles, qui héritent de la classe dans laquelle est défini le champ
- *private*: le champ n'est visible qu'au sein de la classe concernée

On accède aux champ d'un on objet à l'aide de l'opérateur . . Par exemple, pour un objet de classe *A* définissant l'opération publique *op()*, on invoquera cette dernière sur l'objet *o*, instance de cette classe, par *o.op()*.

Attention: n'essayez pas d'appeler un champ sur l'objet *null* !

Accès aux Champs Statiques d'une Classe

Une classe peut définir des champs statiques. Ceci veut dire que ces champs appartiennent à la classe et non à un objet en particulier. On y accède de deux manières:

- en le demandant, comme précédemment, à un objet; attention dans ce cas, car si vous modifiez ce champ, il le sera aussi pour chacune des instances de sa classe, puisque ce champ appartient à la classe et non pas à l'objet
- en le demandant directement à la classe, par exemple le champ *x* de la classe *A* sera appelé par *A.x*

On remarquera que les restrictions de visibilité décrits plus haut restent valides.

Attributs

Un attribut est un champ qui possède un type prédéfini du langage et un nom. Il s'agit en fait d'une variable définie pour tout l'objet qu'il convient d'initialiser dans un constructeur.

Opérations

Une opération est un champ qui possède un nom, un nombre indéfini de paramètres (nommés et typés) ordonnés, et un certain type de retour. Une opération est implémentée (c'est à dire que son comportement est donné) par une méthode. L'appel à une opération se fait par son nom, suivi (entre parenthèses) par les valeurs des paramètres qui lui sont transmis. Une méthode verra ces paramètres comme des variables.

Notons qu'il existe des opérations prédéfinies spécifiques aux classes permettant la gestion efficace des liens dont la multiplicité maximum est supérieure à 1. Ces opérations permettent d'ajouter un objet à cette collection d'objets liés, et d'en retirer. Elles commencent respectivement par *add* et *remove* suivies du nom du lien. Si on prend l'exemple donné dans le paragraphe liens, on peut ajouter l'enfant Deborah à Lucien par *Lucien.addchildren(Deborah)*. Si *Deborah* se fait adopter par la suite par une tierce personne, il sera nécessaire de la déréférencer de *Lucien* : *Lucien.removechildren(Deborah)*.

Surcharge et Virtualité

Il est possible de surcharger des méthodes. Il faut savoir qu'à, l'appel d'une opération, c'est sur l'objet auquel on l'applique que le message est envoyé. Par exemple, l'opération op() est définie dans une classe A. Cette opération est implémentée par une méthode de A et une méthode de B, classe fille de A. Dans le code suivant, c'est la méthode définie dans B qui est appelée :
A monA = new B(); //possible vu la conformance des types
monA.op();

Cette virtualité peut être cassée par le retypage dur de la fonction *oclAsType* de la classe prédéfinie *OclAny*. Comme chaque type existant en Xion hérite de celui-ci (sauf, bien évidemment, *OclAny* lui-même), cette opération est applicable sur n'importe quelle valeur. Elle retype de façon définitive ladite valeur. Dans l'exemple, si préalablement à l'appel de *monA.op()*, on avait appelé *monA.oclAsType(A)*, la méthode définie dans *A* aurait été appelée. Il est évident que cette opération n'est possible que si *A* n'est pas abstrait. Par contre, un cast simple, par *((A)monA).op()*, aurait tout de même déclenché la méthode *op()* de *B*.

Liens

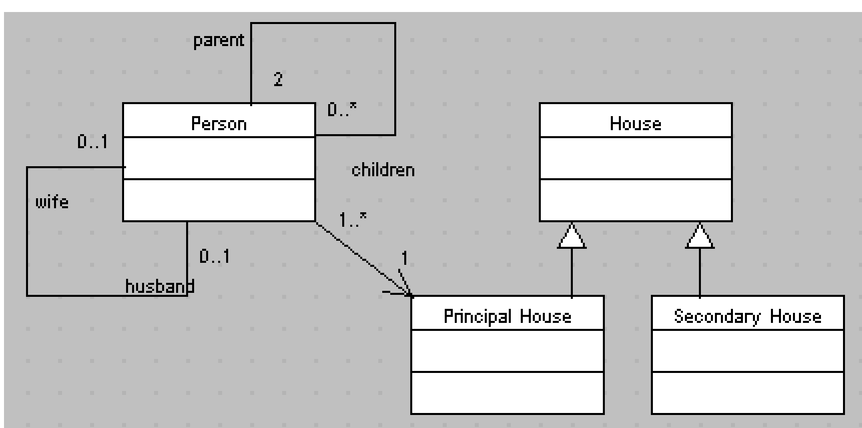
Un lien est un champ qui attache une classe à une autre. Un lien simple équivaut à un attribut dont le type est une classe du modèle métier. Pour accéder à un lien, il suffit de l'appeler par son nom de rôle. Si le nom de rôle n'est pas défini, et si la classe vers laquelle le lien sort a un nom commençant par une majuscule, le nom de rôle est donné par le nom de cette classe commençant par une minuscule. Dans aucun de ces cas, le lien n'est pas accessible.

A noter qu'un lien peut ne pas être navigable dans un sens ou l'autre. Dans ce cas, il est impossible d'y faire référence.

On prend l'exemple d'un lien entre une classe *A* et une classe *B*. Si un lien a une multiplicité de 1 *B*, l'appel à ce lien sur un objet de classe *A* est une expression du type *B*. Si cette multiplicité avait été de 1..* ou de 0..*, l'expression aurait été un *Set(B)*, ou une *Sequence(B)* si le lien était ordonné. En cas de multiplicité de 0..1, l'expression est de type *B*, mais peut aussi être interprété comme une des collections ci-dessus.

Pour affecter un lien de multiplicité maximum 1, une affectation suffit (si les droits d'accès se vérifient). Cette affectation peut porter sur *null*. Pour un lien de multiplicité supérieure existent les opérations *add* et *remove*, suffixées par le nom du lien.

Exemple:



```
Person father = new Person();
Person mother = new Person();
father.wife = mother; /*par là même, le lien mother.husband est affecté à father, lien de multiplicité 0..1: une affectation suffit; les opérations addprincipalHouse et removeprincipalHouse sont indisponibles*/
Person p = new Person(); //On suppose un homme
p.addparent(father); /*on donne à la personne p un de ses parent, donc un fils à father; lien de multiplicité 2: l'opération prédéfinie addparent est nécessaire*/
p.addparent(mother); /*une troisième affectation serait inopérante du fait de la multiplicité*/
Principal_House pricipalHouse = p.principal_House;
Set(Person) parents = p.parent;
Set(Person) children = p.children;
Person wife = p.wife; //null si p n'est pas marié !
Boolean isMarried = p.wife->notEmpty; /*notEmpty est une opération sur Collection; p.wife est unterprété comme un Set(Person)*/
principalHouse.person; //Indisponible car dans ce sens, le lien n'est pas navigable
```

Liens Qualifiés

On peut définir des liens génériques qui seront précisées (à l'appel) par des clés de types de base. Il est toujours possible d'appeler le lien sans ces clefs, ce qui aura pour résultat une valeur de type Set. Pour préciser ces clefs, on les transmet entre des *[ ]*, à l'instar de la transmission de paramètres vers une méthode par les caractères *( )* . Une fois les clefs déterminées, le type de retour est celui du lien décrit ci-dessus. Par exemple, un avion possède un certain nombre de places assises, chacune qualifiée par son numéro. On peut donc lier la classe *Avion* à la classe *PlaceAssise* par un lien qualifié de multiplicité 1 par un *Integer*, puisque chaque place a un numéro unique. Pour un objet de classe *Avion monAvion*, *monAvion.placeAssise* retourne un résultat de type *Set(PlaceAssise)*, et *monAvion.placeAssise[12]* un résultat de type *PlaceAssise*. Par contre, *monAvion.placeAssise[13]* retournera la valeur *null*.

Vers une Classe-Association

Une classe-association est une classe décrivant un lien. Il faut que cette dernière commence par une majuscule. On appelle cette classe par son nom commençant par une minuscule. Le résultat retourné sera du type de la classe association, dans les mêmes conditions que décrites dans le paragraphe liens. Il est interdit d'assigner une classe-association, ou de lier un objet par ce type de lien, par une affectation ou une opération *add*. Par contre l'opération *remove* et permise, tout comme l'affectation à *null* de l'objet lié (multiplicité maximum 1).

Depuis une Classe-Association

Au sein d'une classe association, on accède aux classes liées par leur nom de rôle, tel que défini dans le paragraphe sur les liens. On notera que les différentes parties sont toujours uniques et donc que le résultat ne sera jamais une collection. L'affectation des objets associés et permise. Les opération *add* et *remove* ne sont pas disponibles car il n'existe toujours qu'un objet lié pour chaque partie.

VI - Xion - Opérations prédéfinies

Type Bag

```
Collection
   |
   +--Bag
```

Un Bag est une collection où les duplicatas sont permis. Cela signifie qu'un élément peut apparaître plusieurs fois dans un même conteneur. Il n'y a pas de notion d'ordre dans un conteneur.

`Boolean` **=**`(Bag p0)`
Vrai si ce conteneur et p0 ont les mêmes éléments, le même nombre de fois.

`Bag` **union**`(Bag p0)`
Un conteneur ayant chacun des éléments de ce conteneur et chacun des éléments de p0. Si un même élément est contenu dans ce conteneur et p0, il est compté deux fois.

`Bag` **union**`(Set p0)`
Un conteneur ayant chacun des éléments de ce conteneur et chacun des éléments de p0. Si un même élément est contenu dans ce conteneur et p0, il est compté deux fois.

`Bag` **intersection**`(Bag p0)`
Un conteneur ayant les éléments présents dans ce conteneur et dans p0. Si un même élément est contenu deux fois dans cet élément et une seule fois dans p0, il n'est pris qu'une fois. Par contre, s'il existait seux fois dans p0, il aurait été pris deux fois.

`Bag` **intersection**`(Set p0)`
Un ensemble contenant les valeurs de p0 qui sont aussi dans ce conteneur.

`Bag` **including**`(OclAny p0)`
Un conteneur ayant chacun des éléments de ce conteneur et p0, même si p0 existe déjà dans ce conteneur.

`Bag` **excluding**`(OclAny p0)`
Un conteneur ayant chacun des éléments de ce conteneur, sauf toutes les occurences de p0.

`Bag` **select**`(Boolean_Expression p0)`
Un sous conteneur contenant les éléments de ce conteneur vérifiant à vrai l'expression p0.

`Bag` **reject**`(Boolean_Expression p0)`
Un sous conteneur contenant les éléments de ce conteneur vérifiant à faux l'expression p0.

`Bag` **collect**`(OclAny_Expression p0)`
Un conteneur contenant les résultats des évaluations de l'expression p0 sur chacun des éléments de ce conteneur.

`Integer` **count**`(OclAny p0)`
Le nombre de fois qu'est contenu p0 dans ce conteneur.
Surcharge: count in type Collection

`Sequence` **asSequence**
Une séquence contenant chacun des éléments de ce conteneur dans un ordre aléatoire.

`Set` **asSet**
Un ensemble qui contient chacun des éléments distincts de ce conteneur.

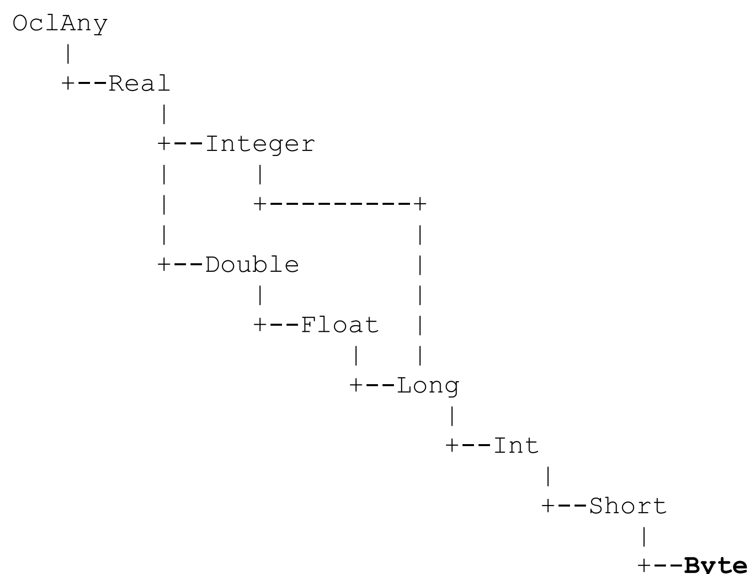
Type Boolean

```
OclAny
   |
   +--Boolean
```

Le type prédéfini Boolean représente les valeurs true/false.

`Boolean` **`=`**`(Boolean p0)`
Vrai si ce booléen est le même que p0.
`Boolean` **`<>`**`(Boolean p0)`
Vrai si ce Booléen est le complémentaire de p0.
`Boolean` **`or`**`(Boolean p0)`
Vrai soit si ce booléen est à vrai, soit si p0 est à vrai.
`Boolean` **`xor`**`(Boolean p0)`
Vrai si seulement ce booléen ou p0 est à vrai.
`Boolean` **`and`**`(Boolean p0)`
Vrai si ce booléen et p0 sont tout deux à vrai.
`Boolean` **`not`**`(Boolean p0)`
L'opposé de ce booléen.
`Boolean` **`implies`**`(Boolean p0)`
Vrai si ce booléen est à faux, ou si ce booléen est à vrai et que p0 est vrai.
`String` **`toString`**
Une chaîne représentant ce booléen.
Surcharge: toString in type java.lang.Object

Type Byte



```
OclAny
   |
   +--Real
         |
         +--Integer
         |     |
         |     +---------+
         |               |
         +--Double       |
               |         |
               +--Float  |
                     |   |
                     +--Long
                           |
                           +--Int
                                 |
                                 +--Short
                                       |
                                       +--Byte
```

Byte est le type prédéfini pour les entiers entre -128 à 127 inclus. Ce type décrit les octets.

Byte **floor**
Retourne cet octet.
Surcharge: floor in type Short

Byte **round**
Cet octet.
Surcharge: round in type Short

Boolean **=**(Byte p0)
Vrai si cet octet est égal à p0.

Boolean **<>**(Byte p0)
Vrai si cet octet n'est pas égal à p0.

Byte **+**
Cet octet.
Surcharge: + in type Short

Byte **-**
L'opposé de cet octet.
Surcharge: - in type Short

Byte **+**(Byte p0)
L'addition entre cet octet et p0.

Byte **-**(Byte p0)
La soustraction de cet octet par p0.

Byte *****(Byte p0)
La multiplication de cet octet et de p0.

Real **/**(Byte p0)
La division de cet octet par p0.

Byte **abs**
La valeur absolue de cet octet.
Surcharge: abs in type Short

Byte **div**(Byte p0)
La division entière de cet octet.

Byte **mod**(Integer p0)
Le modulo (ou reste de la division entière) de cet octet par p0.
Surcharge: mod in type Short

Byte **max**(Byte p0)
Le maximum entre cet octet et p0.

Byte **min**(Byte p0)
Le minimum entre cet octet et p0.

Byte **~**
La négation bit à bit de cet octet.
Surcharge: ~ in type Short

Byte **&**(Byte p0)
Le et bit à bit de cet octet avec p0.

Byte **|**(Byte p0)
Le ou inclusif bit à bit de cet octet et de p0.

Byte **^**(Byte p0)
Le ou exclusif bit à bit de cet octet et de p0.

Byte **<<**(Integer p0)
Un décalage à gauche des bits de cet octet de p0 crans.
Surcharge: << in type Short

Byte **>>**(Integer p0)
Un décalage à droite des bits de cet octet de p0 crans.
Surcharge: >> in type Short

Boolean **<**(Byte p0)
Vrai si cet octet est inférieur à p0.

Boolean **>**(Byte p0)
Vrai si cet octet est supérieur à p0.

Boolean **<=**(Byte p0)
Vrai si cet octet est inférieur ou égal à p0.

Boolean **>=**(Byte p0)
Vrai si cet octet est supérieur ou égal à p0.

`String` **`toString`**

Une chaîne de caractères décrivant cet octet.
Surcharge: toString in type Short

Type Collection

`Collection`
Sous types:
Bag, Sequence, Set

Le type Collection est le supertype abstrait de toutes les collections. Chaque occurence d'un objet dans une collection est appelé un élément. Si un objet est présent deux fois au sein d'une même collection, il compte pour deux éléments.

`Integer` **`size`**
Le nombre d'éléments présents dans cette collection.
`Boolean` **`includes`**`(OclAny p0)`
Vrai si p0 est un élément de cette collection.
`Boolean` **`excludes`**`(OclAny p0)`
Vrai si p0 n'est pas un élément de cette collection.
`Integer` **`count`**`(OclAny p0)`
Le nombre de fois qu'apparaît p0 dans cette collection.
`Boolean` **`includesAll`**`(Collection p0)`
Vrai si cette collection contient chacun des éléments de cette collection.
`Boolean` **`excludesAll`**`(Collection p0)`
Vrai si cette collection ne contient aucun des éléments de cette collection.
`Boolean` **`isEmpty`**
Vrai si cette collection ne contient aucun élément.
`Boolean` **`notEmpty`**
Vrai si cette collection contient au moins un élément.
`OclAny` **`getOne`**
Retourne un objet de la collection au hasard. Cet objet est null si la collection est vide.
`Integer` **`sum`**
Exécute la somme de chacun des éléments de cette collection. Le résultat est 0 si cette collection est vide. Cette opération n'est disponible que sur des collection dont les éléments ont un type qui supporte l'opération +, ayant pour paramètre ce même type et pour type de retour ce même type.
`Boolean` **`exists`**`(Boolean_Expression p0)`
Vrai si l'expression p0 appliquée sur chacun des éléments de cette collection retourne vrai au moins une fois.
`Boolean` **`forAll`**`(Boolean_Expression p0)`
Vrai si l'expression p0 appliquée sur chacun des éléments de cette collection retourne toujours vrai.
`Boolean` **`isUnique`**`(OclAny_Expression p0)`
Vrai si l'expression p0 appliquée sur chacun des éléments de cette collection retourne vrai une seule fois.
`Sequence` **`sortedBy`**`(OclAny_Expression p0)`
Trie les éléments de cette collection dans l'ordre croissant du résultat de l'expression p0 appliquée sur chacun de ses éléments. Cette opération n'est disponible que pour des collections dont le type des éléments supporte l'opération < avec pour paramètre ce même type et pour type de retour Boolean.

Sequence **sortedBy**(OclAny_Expression p0, Boolean p1)

Trie les éléments de cette collection du résultat de l'expression p0 appliquée sur chacun de ses éléments, dans l'ordre croissant si p1 est vrai, sinon décroissant. Cette opération n'est disponible que pour des collections dont le type des éléments supporte l'opération < avec pour paramètre ce même type et pour type de retour Boolean.

Type Date

```
OclAny
   |
   +--Date
```

Le type prédéfini pour les dates.

static Date **getCurrent**

La date courante.

static Date **getDate**(Integer p0, Integer p1, Integer p2)

La date pour laquelle p0 est l'année, p1 est le numéro du mois et p2 la cardinalité du jour dans le mois. Le mois doit être donné entre 1 et 12 et le jour entre 1 et 31.

Boolean **=**(Date p0)

Vrai si cette date correspond à p0.

Boolean **<>**(Date p0)

Vrai si cette date ne correspond pas à p0.

Integer **-**(Date p0)

Le nombre de jours entre cette date et p0. Cette opération est un alias pour daysFrom.

Integer **daysFrom**(Date p0)

Le nombre de jours entre cette date et p0.

Date **addDay**(Integer p0)

Cette date en y ajoutant p0 jours.

Date **addMonth**(Integer p0)

Cette date en y ajoutant p0 mois.

Date **addYear**(Integer p0)

Cette date en y ajoutant p0 années.

Integer **getDay**

La cardinalité du jour dans le mois de cette date.

Integer **getMonth**

Le numéro du mois de cette date.

Integer **getYear**

L'année de cette date.

Date **max**(Date p0)

La date la plus récente entre cette date et p0.

Date **min**(Date p0)

La date la plus ancienne entre cette date et p0.

Boolean **<**(Date p0)

Vrai si cette date est plus ancienne que p0.

Boolean **>**(Date p0)

Vrai si cette date est plus récente que p0.

Boolean **<=**(Date p0)

Vrai si cette date est plus ancienne ou équivalente à p0.

Boolean **>=**(Date p0)

Vrai si cette date est plus récente ou équivalente à p0.

Type Double

```
OclAny
   |
   +--Real
         |
         +--Double
```

Sous types:
Float

Le type Double est le type pour les valeur de réels en précision 64 bits.

`Integer` **floor**
Le plus grand entier inférieur ou égal à ce réel.
Surcharge: floor in type Real

`Integer` **round**
L'entier le plus proche de ce réel. L e plus grand en cas de conflit.
Surcharge: round in type Real

`Boolean` **=**`(Double p0)`
Vrai si ce réel est égal à p0.

`Boolean` **<>**`(Double p0)`
Vrai si ce réel est différent de p0.

`Double` **+**
Ce réel.
Surcharge: + in type Real

`Double` **-**
L'opposé de ce réel.
Surcharge: - in type Real

`Double` **+**`(Double p0)`
L'addition de ce réel et de p0.

`Double` **-**`(Double p0)`
La soustraction de ce réel par p0.

`Double` *****`(Double p0)`
La multiplication de ce réel et de p0.

`Double` **/**`(Double p0)`
La division de ce réel par p0.

`Double` **abs**
La valeur absolue de ce réel.
Surcharge: abs in type Real

`Double` **max**`(Double p0)`
Le maximum entre ce réel et p0.

`Double` **min**`(Double p0)`
Le minimum entre ce réel et p0.

`Boolean` **<**`(Double p0)`
Vrai si ce réel est inférieur à p0.

`Boolean` **>**`(Double p0)`
Vrai si ce réel et supérieur à p0.

`Boolean` **<=**`(Double p0)`
Vrai si ce réel est inférieur ou égal à p0.

`Boolean` **>=**`(Double p0)`
Vrai si ce réel est supérieur ou égal à p0.

`String` **toString**
Une chaîne représentative de ce réel.
Surcharge: toString in type java.lang.Object

Type Float

```
OclAny
   |
   +--Real
         |
         +--Double
               |
               +--Float
```

Sous types:
Long

Le type Float est le type pour les valeur de réels en précision 32 bits.

`Integer` **floor**
Le plus grand entier inférieur ou égal à ce réel.
Surcharge: floor in type Double

`Integer` **round**
L'entier le plus proche de ce réel. L e plus grand en cas de conflit.
Surcharge: round in type Double

`Boolean` **=**`(Float p0)`
Vrai si ce réel est égal à p0.

`Boolean` **<>**`(Float p0)`
Vrai si ce réel est différent de p0.

`Float` **+**
Ce réel.
Surcharge: + in type Double

`Float` **-**
L'opposé de ce réel.
Surcharge: - in type Double

`Float` **+**`(Float p0)`
L'addition de ce réel avec p0.

`Float` **-**`(Float p0)`
La soustraction de ce réel par p0.

`Float` *****`(Float p0)`
La multiplication de ce réel avec p0.

`Float` **/**`(Float p0)`
La division de ce réel par p0.

`Float` **abs**
La valeur absolue de ce réel.
Surcharge: abs in type Double

`Float` **max**`(Float p0)`
Le maximum entre ce réel et p0.

`Float` **min**`(Float p0)`
Le minimum entre ce réel et p0.

`Boolean` **<**`(Float p0)`
Vrai si ce réel est inférieur à p0.

`Boolean` **>**`(Float p0)`
Vrai si ce réel est supérieur à p0.

`Boolean` **<=**`(Float p0)`
Vrai si ce réel est inférieur ou égal à p0.

`Boolean` **>=**`(Float p0)`
Vrai si ce réel est supérieur ou égal à p0.

`String` **toString**
Une chaîne de caractères décrivant ce réel.
Surcharge:
toString in type Double

Type Int

```
OclAny
   |
   +--Real
         |
         +--Integer
         |     |
         |     +---------+
         |               |
         +--Double       |
               |         |
               +--Float  |
                     |   |
                     +--Long
                           |
                           +--Int
```

Sous types:

Short

Le type Int correspond aux entiers compris entre -2147483648 et 2147483647, soient aux mots de quatre octets (32 bits).

Int **floor**
: Ce mot.
: **Surcharge:** floor in type Long

Int **round**
: Ce mot.
: **Surcharge:** round in type Long

Boolean **=**(Int p0)
: Vrai si ce mot est égal à p0.

Boolean **<>**(Int p0)
: Vrai si ce mot est différent de p0.

Int **+**
: Ce mot.
: **Surcharge:** + in type Long

Int **-**
: L'opposé de ce mot.
: **Surcharge:** - in type Long

Int **+**(Int p0)
: L'addition de ce mot et de p0.

Int **-**(Int p0)
: La soustraction de ce mot par de p0.

Int *****(Int p0)
: La multiplication de ce mot et de p0.

Real **/**(Int p0)
: La division de ce mot par p0.

Int **abs**
: La valeur absolue de ce mot.
: **Surcharge:** abs in type Long

Int **div**(Int p0)
: La division entière de ce mot.

Int **mod**(Integer p0)
: Le modulo (reste de la division entière) de ce mot par p0.
: **Surcharge:** mod in type Long

Int **max**(Int p0)
: Le maximum entre ce mot et p0.

Int **min**(Int p0)
: Le minimum entre ce mot et p0.

Int ~
La négation bit à bit de ce mot.
Surcharge: ~ in type Long

Int &(Int p0)
Le et bit à bit de ce mot et de p0.

Int |(Int p0)
Le ou inclusif bit à bit de ce mot et de p0.

Int ^(Int p0)
Le ou exclusif bit à bit de ce mot et de p0.

Int <<(Integer p0)
Ce mot dont les bits on étés décalés p0 fois vers la gauche.
Surcharge: << in type Long

Int >>(Integer p0)
Ce mot dont les bits ont été décalés p0 fois vers la droite.
Surcharge: >> in type Long

Boolean <(Int p0)
Vrai si ce mot est inférieur à p0.

Boolean >(Int p0)
Vrai si ce mot est supérieur à p0.

Boolean <=(Int p0)
Vrai si ce mot est inférieur ou égal à p0.

Boolean >=(Int p0)
Vrai si ce mot est supérieur ou égal à p0.

String **toString**
Une chaîne de caractères décrivant ce mot.
Surcharge: toString in type Long

Type Integer

```
OclAny
   |
   +--Real
         |
         +--Integer
```

Sous types:
Long

Le type prédéfini Integer représente le concept mathématique d'entier.

Boolean =(Integer p0)
Vrai si cet entier est égal à p0.

Boolean <>(Integer p0)
Vrai si cet entier est différent de p0.

Integer +
Cet entier.
Surcharge: + in type Real

Integer +(Integer p0)
L'addition de cet entier avec p0.

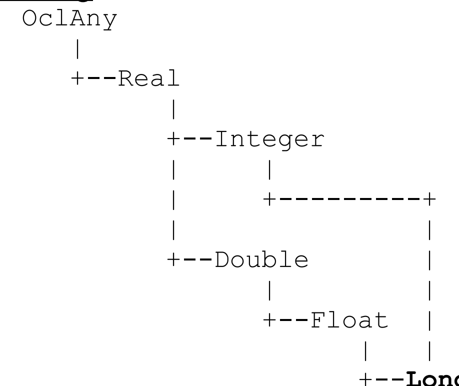
Integer *(Integer p0)
La multiplication de cet entier par p0.

Real /(Integer p0)
La division réelle de cet entier par p0.
See Also: div

Integer **abs**
La valeur absolue de cet entier.
Surcharge: abs in type Real

Integer **div**(Integer p0)
La division entière de cet entier par p0.
Integer **mod**(Integer p0)
Le modulo (ou reste de la division entière) de cet entier par p0.
Integer **max**(Integer p0)
Le maximum entre cet entier et p0.
Integer **min**(Integer p0)
Le minimum entre cet entier et p0.
Boolean **<**(Integer p0)
Vrai si cet entier est inférieur à p0.
Boolean **>**(Integer p0)
Vrai si cet entier est supérieur à p0.
Boolean **<=**(Integer p0)
Vrai si cet entier est inférieur ou égal à p0.
Boolean **>=**(Integer p0)
Vrai si cet entier est supérieur ou égal à p0.

Type Long



Sous types:
Int

Le type Int correspond aux entiers compris entre -9223372036854775808 et 9223372036854775807, soient aux mots de huit octets (64 bits).

Long **floor**
Ce mot.
Surcharge: floor in type Real
Long **round**
Ce mot.
Surcharge: round in type Real
Boolean **=**(Long p0)
Vrai si ce mot est égal à p0.
Boolean **<>**(Long p0)
Vrai si ce mot est différent de p0.
Long **+**
Ce mot.
Surcharge: + in type Integer
Long **-**
L'opposé de ce mot.
Long **+**(Long p)
L'addition de ce mot et de p0.
Long **-**(Long p0)
La soustraction de ce mot par p0.

Long *(Long p0)
La multiplication de ce mot et de p0.

Long /(Long p0)
La division de ce mot par p0.

Long **abs**
La valeur absolue de ce mot.
Surcharge: abs in type Integer

Long **div**(Long p0)
La division entière de ce mot par p0.

Long **mod**(Integer p0)
Le modulo (reste de la division entière) de ce mot par p0.
Surcharge: mod in type Integer

Long **max**(Long p0)
Le maximum entre ce mot et p0.

Long **min**(Long p0)
Le minimum entre ce mot et p0.

Long ~
La négation bit à bit de ce mot.

Long &(Long p0)
Le et bit à bit de ce mot et de p0.

Long |(Long p0)
Le ou inclusif bit à bit de ce mot et de p0.

Long ^(Long p0)
Le ou exclusif bit à bit de ce mot et de p0.

Long <<(Integer p0)
Ce mot dont les bits ont été décalés p0 fois à gauche.

Long >>(Integer p0)
Ce mot dont les bits ont étés décalés p0 fois vers la droite.

Boolean <(Long p0)
Vrai si ce mot est inférieur à p0.

Boolean >(Long p0)
Vrai si ce mot est supérieur à p0.

Boolean <=(Long p0)
Vrai si ce mot est inférieur ou égal à p0.

Boolean >=(Long p0)
Vrai si ce mot est supérieur ou égal à p0.

String **toString**
Une chaîne de caractères décrivant ce mot.
Surcharge: toString in type java.lang.Object

Type OclAny

OclAny

Sous types:
Boolean, Date, OclType, Real, Text, Time

Le type OclAny est le supertype de tous les types définis dans le modèle métier, des types prédéfinis et énumérés. Les opérations définies dans OclAny sont accessibles dans chaque objet, où qu'il soit traîté. Il est possible de surcharger dans le modèle une opération définie dans OclAny

Boolean =(OclAny p0)
Vrai si p0 est le même objet que celui-ci

Boolean <>(OclAny p0)
Vrai si p0 est un objet différent de celui-ci

`Boolean` **`oclIsKindOf`**`(OclType p0)`
Vrai si p0 est un des types ou supertypes (transitif) de cet objet. L'appel de *monObjet.oclIsKindOf(OclAny)* sera toujours vrai.

`Boolean` **`oclIsTypeOf`**`(OclType p0)`
Vrai si p0 est le type (vrai) de l'objet en cours. Si p0 est un type abstrait, comme OclAny, le résultat sera toujours faux.

`OclAny` **`oclAsType`**`(OclType p0)`
Retourne cet objet définitivement transformé comme étant du type p0.

`Void` **`delete`**
Détrtruit l'objet en cous, ainsi que ses liens, de la base de données. En cas de composition, l'objet possédé est aussi détuit. Après cette opération, l'ojet reste temporarement accessible, mais il est fortement déconseillé de l'utiliser. Cette opération n'est active que sur les objets du modèle métier (et non sur ceux de type prédéfinis).

Type OclObject

```
OclAny
   |
   +--OclObject
```

OclObject est le type dont héritent toutes les classes définies dans le modèle métier.

`String` **`getOID`**
Retourne une chaîne de caractères correspondant à un indentifiant unique de l'objet concerné. Cette opération est sencée être bijective (il est possible de retrouver de façon déterministe un object avec son OID).

Type OclType

```
OclAny
   |
   +--OclType
```

Tous les types, qu'ils soient définis dans le modèle métier, ou qu'il soit prédéfinis, collection ou énuméré, possède un type, instance de OcType.

`String` **`name`**
Le nom de ce type.

`Set` **`attributes`**
L'ensemble des noms des attributs de ce type.

`Set` **`associationEnds`**
L'ensemble des noms des liens navigables ce ce type.

`Set` **`operations`**
L'ensemble des noms des opérations définies dans ce type.

`Set` **`allInstances`**
L'ensemble de toutes les instances de ce type. Cette opération n'est pas autorisée que sur des types prédéfinis.

`String` **`toString`**
Retourne une chaîne de caractères correspondant au type.
Surcharge:
toString in type java.lang.Object

Type Real

```
OclAny
  |
  +--Real
```

Sous types:

Double, Integer

Le type Real représente le concept mathématique des réels. On remarque que Integer est un sous-type de Real, donc chaque paramètre de type Real peut être remplacé par un paramètre de type Integer.

`Boolean` **`=`**`(Real p0)`
Vrai si ce réel est égal à p0.

`Boolean` **`<>`**`(Real p0)`
Vrai si ce réel est différent de p0.

`Real` **`+`**
Retourne ce réel.

`Real` **`-`**
Retourne la valeur opposée de ce réel.

`Real` **`+`**`(Real p0)`
Retourne l'addition de ce réel et de p0.

`Real` **`-`**`(Real p0)`
Retourne la soustraction de ce réel par p0.

`Real` **`*`**`(Real p0)`
Retourne la multipication de ce réel par p0.

`Real` **`/`**`(Real p0)`
Retoune la division de ce réel par p0.

`Real` **`abs`**
Retourne la valeur absolue de ce réel.

`Integer` **`floor`**
Retourne le plus grand entier inférieur ou égal à ce réel.

`static Real` **`random`**
Retourne un réel entre 0 et 1.

`Integer` **`round`**
Retourne l'entier le plus proche de ce réel. En cas de conflit, il s'agit du plus grand.

`Real` **`max`**`(Real p0)`
Le maximum entre ce réel et p0.

`Real` **`min`**`(Real p0)`
Le minimum entre ce réel et p0.

`Boolean` **`<`**`(Real p0)`
Vrai si ce réel est inférieurà p0.

`Boolean` **`>`**`(Real p0)`
Vrai si ce réel est supérieur à p0.

`Boolean` **`<=`**`(Real p0)`
Vrai si ce réel est inférieur ou égal à p0.

`Boolean` **`>=`**`(Real p0)`
Vrai si ce réel est supérieur ou égal à p0.

Type Sequence

```
        Collection
           |
           +--Sequence
```

Une Sequence est une collection où les éléments sont ordonnés. Un même élément peut apparaître plusieurs fois dans une séquence.

`Integer` **`count`**`(OclAny p0)`
Le nombre de fois qu'apparaît p0 dans cette séquence.
Surcharge: count in type Collection

`Boolean` **`=`**`(Sequence p0)`
Vrai si cette séquence contient les mêmes éléments que p0, dans le même ordre.

`Sequence` **`union`**`(Sequence p0)`
Un séquence possédant chacun des éléments de cette séquence, suivie de chacun des éléments de p0.

`Sequence` **`append`**`(OclAny p0)`
Un séquence ayant chacun des éléments de cette séquence, suivi de p0.

`Sequence` **`prepend`**`(OclAny p0)`
Une séquence ayant p0, suivit de chacun des éléments de cette séquence.

`Sequence` **`subSequence`**`(Integer p0, Integer p1)`
Une sous séquence ce cette séquence commençant à l'élément numéro p0 et finissant à l'élément p1. Le premier élément d'une séquence est l'élément numéro 0. Si p0 < 0 ou p0 > p1 ou p1 > n avec n le numéro du dernier élément, cette sous séquence ne contient aucun élément.

`OclAny` **`at`**`(Integer p0)`
L'élément numéro p0. Le premier élément a le numéro 0. Si p0 < 0 ou p0 > n où n est le numéro du dernier élément, le résultat est null.

`OclAny` **`first`**
Le premier élément de la séquence. Si la séquence est vide, le résultat est null.

`OclAny` **`last`**
Le dernier élément de la séquence. Si la séquence est vide, le résultat est null.

`Sequence` **`including`**`(OclAny p0)`
Une séquence contenant tous les éléments de cette séquence suivie de p0.

`Sequence` **`excluding`**`(OclAny p0)`
Une séquence contenant tous les éléments de cette séquence sauf chacune des occurences de p0.

`Sequence` **`select`**`(Boolean_Expression p0)`
Une sous séquence contenant chacun des éléments pour lesquels l'expression p0 est évaluée à vrai.

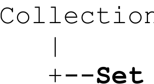
`Sequence` **`reject`**`(Boolean_Expression p0)`
Une sous séquence contenant chacun des éléments pour lesquels l'expression p0 est évaluée à faux.

`Sequence` **`collect`**`(OclAny_Expression p0)`
Une séquence contenant les résultats ordonnés de l'expression p0 sur chacun des éléments de cette collection.

`Bag` **`asBag`**
Un conteneur ayant pour éléments chacun des éléments de cette séquence.

`Set` **`asSet`**
Un ensemble contenant chacun des éléments distincts de cette séquence.

Type Set



Le Set est l'ensemble mathématique. Il contient des éléments non dupliqués. Il n'existe pas de notion d'ordre dans un ensemble. Cette contrainte d'unicité ignore les surcharges de OclAny.=(p0:OclAny):Boolean .

Set **union**(Set p0)
L'union de cet ensemble et de p0.

Bag **union**(Bag p0)
L'union de cet ensemble et de p0.

Boolean **=**(Set p0)
Vrai si cet ensemble contient les mêmes valeurs de p0. L'ordre est ici sans importance.

Set **intersection**(Set p0)
L'intersection de cet ensemble et de p0. Le résultat est un ensemble contenant les valeurs existant et dans p0, et dans cet ensemble.

Set **intersection**(Bag p0)
L'intersection de cet ensemble et de p0. Le résultat est un ensemble contenant les valeurs existant et dans p0, et dans cet ensemble.

Set **-**(Set p0)
Un ensemble contenant les éléments de cet ensemble qui ne sont pas dans p0.

Set **including**(OclAny p0)
Un ensemble contenant tous les éléments de cet ensemble et p0. Si p0 existe déjà, l'inclusion est ignorée.

Set **excluding**(OclAny p0)
Un ensemble contenant tous les éléments de cet ensemble sauf p0. Si p0 n'existe pas dans cet ensemble, l'inclusion est ignorée.

Set **symmetricDifference**(Set p0)
L'ensemble contenant tous les éléments qui sont soit dans cet ensemble, soit dans p0, mais pas dans les deux.

Set **select**(Boolean_Expression p0)
Le sous ensemble des éléments de cette collection qui évaluent à Vrai l'expression p0.

Set **reject**(Boolean_Expression p0)
Le sous ensemble des éléments de cette collection qui évaluent à Faux l'expression p0.

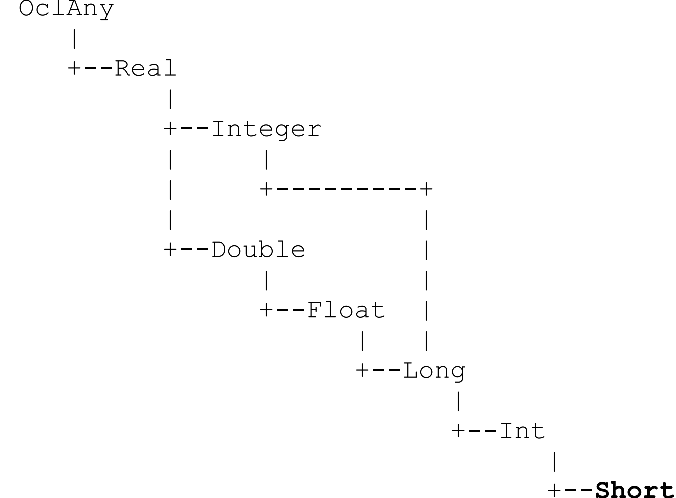
Set **collect**(OclAny_Expression p0)
Un conteneur contenant les résultat de l'évaluation de l'expression p0 sur chacun des éléments de cet ensemble.

Integer **count**(OclAny p0)
0 si p0 n'existe pas dans cet ensemble; 1 si p0 existe dans cet ensemble. On rappelle que les duplicatas ne sont pas permis dans un ensemble.
Surcharge: count in type Collection

Sequence **asSequence**
Une séquence contenant tous les éléments de cet ensemble, dans un ordre indéterminé.

Bag **asBag**
Un conteneur contenant tous les élément de cet ensemble.

Type Short



Sous types:
Byte

Le type Short correspond aux entiers compris entre -32768 et 32767, soient aux mots de deux octets (16 bits).

`Short` **`floor`**
Ce mot.
Surcharge: floor in type Int

`Short` **`round`**
Ce mot.
Surcharge: round in type Int

`Boolean` **`=`**`(Short p0)`
Vrai si ce mot est égal à p0.

`Boolean` **`<>`**`(Short p0)`
Vrai si ce mot n'est pas égal à p0.

`Short` **`+`**
Ce mot.
Surcharge: + in type Int

`Short` **`-`**
L'opposé de ce mot.
Surcharge: - in type Int

`Short` **`+`**`(Short p0)`
L'addition de ce mot et p0.

`Short` **`-`**`(Short p0)`
La soustraction de ce mot par p0.

`Short` **`*`**`(Short p0)`
La multiplication de ce mot et p0.

`Real` **`/`**`(Short p0)`
La division de ce mot par p0.

`Short` **`abs`**
La valeur absolue de ce mot.
Surcharge: abs in type Int

`Short` **`div`**`(Short p0)`
La division entière de ce mot par p0.

`Short` **`mod`**`(Integer p0)`
Le modulo (reste de la division entière) de ce mot par p0.
Surcharge: mod in type Int

`Short` **`max`**`(Short p0)`
Le maximum entre ce mot et p0.

Short **min**(Short p0)
Le minimum entre ce mot et p0.
Short **~**
La négation bit à bit de ce mot.
Surcharge: ~ in type Int
Short **&**(Short p0)
Le et bit à bit de ce mot et p0.
Short **|**(Short p0)
Le ou inclusif bit à bit de ce mot et p0.
Short **^**(Short p0)
Le ou exclusif bit à bit de ce mot et p0.
Short **<<**(Integer p0)
Ce mot dont les bits ont été décalés p0 fois à gauche.
Surcharge: << in type Int
Short **>>**(Integer p0)
Ce mot dont les bits ont été décalés p0 fois à droite.
Surcharge: >> in type Int
Boolean **<**(Short p0)
Vrai si ce mot est inférieur à p0.
Boolean **>**(Short p0)
Vrai si ce mot est supérieur à p0.
Boolean **<=**(Short p0)
Vrai si ce mot est inférieur ou égal à p0.
Boolean **>=**(Short p0)
Vrai si ce mot est supérieur ou égal à p0.
String **toString**
Une chaîne de caractères décrivant ce mot.
Surcharge: toString in type Int

Type String

```
OclAny
   |
   +--Text
         |
         +--String
```

Le type prédéfini String représente les chaînes de caractères ASCII d'une longueur maximum de 255 caractères.

Boolean **checkReal**
Vérifie si cette chaîne peut être transformée en Real.
Boolean **checkInteger**
Vérifie si cette chaîne peut être transformée en Integer.
Boolean **checkDouble**
Vérifie si cette chaîne peut être transformée en Double.
Boolean **checkFloat**
Vérifie si cette chaîne peut être transformée en Float.
Boolean **checkLong**
Vérifie si cette chaîne peut être transformée en Long.
Boolean **checkInt**
Vérifie si cette chaîne peut être transformée en Int.
Boolean **checkShort**
Vérifie si cette chaîne peut être transformée en Short.
Boolean **checkByte**
Vérifie si cette chaîne peut être transformée en Byte.

`Boolean` **checkBoolean**
Vérifie si cette chaîne peut être transformée en Boolean.
`Boolean` **checkDate**
Vérifie si cette chaîne peut être transformée en Date.
`Boolean` **checkTime**
Vérifie si cette chaîne peut être transformée en Time.
`Real` **parseReal**
Transforme cette chaîne en Real. En cas d'impossibilité, il s'agit d'une valeur quelconque (toujours la même).
`Integer` **parseInteger**
Transforme cette chaîne en Integer. En cas d'impossibilité, il s'agit d'une valeur quelconque (toujours la même).
`Double` **parseDouble**
Transforme cette chaîne en Double. En cas d'impossibilité, il s'agit d'une valeur quelconque (toujours la même).
`Float` **parseFloat**
Transforme cette chaîne en Float. En cas d'impossibilité, il s'agit d'une valeur quelconque (toujours la même).
`Long` **parseLong**
Transforme cette chaîne en Long. En cas d'impossibilité, il s'agit d'une valeur quelconque (toujours la même).
`Int` **parseInt**
Transforme cette chaîne en Int. En cas d'impossibilité, il s'agit d'une valeur quelconque (toujours la même).
`Short` **parseShort**
Transforme cette chaîne en Short. En cas d'impossibilité, il s'agit d'une valeur quelconque (toujours la même).
`Byte` **parseByte**
Transforme cette chaîne en Byte. En cas d'impossibilité, il s'agit d'une valeur quelconque (toujours la même).
`Boolean` **parseBoolean**
Transforme cette chaîne en Boolean. En cas d'impossibilité, il s'agit d'une valeur quelconque (toujours la même).
`Date` **parseDate**
Transforme cette chaîne en Date. En cas d'impossibilité, il s'agit d'une valeur quelconque (toujours la même).
`Time` **parseTime**
Transforme cette chaîne en Time. En cas d'impossibilité, il s'agit d'une valeur quelconque (toujours la même).

Type Text

```
OclAny
   |
   +--Text
```

Sous types:
String

Le type prédéfini Text représente les chaînes de caractères ASCII de longueur indéfinie.

`Boolean` **=**`(Text p0)`
Vrai si cette chaîne contient les mêmes caractères, et dans le même odre, que p0.
`Boolean` **<>**`(Text p0)`
Vrai si cette chaîne ne possède pas les mêmes caractères (ordonnés) que p0.
`Integer` **size**
Le nombre de caractères dans cette chaîne.
`Text` **concat**`(Text p0)`
La concaténation de cette chaîne et de p0.
`Text` **toUpper**
Cette chaîne dont tous les caractères minuscules sont transformés en majucules.

`Text` **`toLower`**

Cette chaîne dont tous les caractères majuscules sont transformés en minuscules.

```
Text substring(Integer p0,
               Integer p1)
```

La sous-chaîne contenue dans cette chaîne à commencer par le caractère numéro p0 et à finir par le caractère numéro p1. Le premier caractère a le numéro 0.

`Text` **`crypt`**

Cette chaîne de caractère cryptée. Le cryptage correspond à un hachage. Ceci permet notamment le stockage sécurisé de mots de passe.

```
Boolean <(Text p0)
Boolean >(Text p0)
Boolean <=(Text p0)
Boolean >=(Text p0)
```

Type Time

```
OclAny
   |
   +--Time
```

Le type prédéfini pour représenter les heures.

`static Time` **`getCurrent`**

Retourne l'heure courante.

```
static Time getTime(Integer p0,
                    Integer p1,
                    Integer p2)
```

Retourne une heure de p0 heures, p1 minutes et p2 secondes.

`Boolean` **`=`**`(Time p0)`

Vrai si cette heure est la même que p0.

`Time` **`<>`**`(Time p0)`

Vrai si cette heure est différente de p0.

`Time` **`addMilli`**`(Integer p0)`

Cette heure à laquelle on a ajouté p0 millisecondes.

`Time` **`addSecond`**`(Integer p0)`

Cette heure à laquelle on a jouté p0 secondes.

`Time` **`addMinute`**`(Integer p0)`

Cette heure à laquelle on a ajouté p0 minutes.

`Time` addHour(`Integer` p0)

Cette heure à laquelle on a ajouté p0 heures.

`Integer` **`getMilli`**

Le nombre de millisecondes depuis la dernière seconde de cette heure.

`Integer` **`getSecond`**

Le nombre de seconde depuis la dernière minute de cette heure.

`Integer` **`getMinute`**

Le nombre de minutes depuis la dernière heure de cette heure.

`Integer` **`getHour`**

Le nombre d'heures passées dans cette heure depuis minuit.

`Time` **`max`**`(Time p0)`

L'heure la plus avancée entre cette heure et p0.

`Time` **`min`**`(Time p0)`

L'heure la moins avancée entre cette heure et p0.

`Boolean` **`<`**`(Time p0)`

Vrai si cette heure est moins avancée que p0.

`Boolean` **`>`**`(Time p0)`

Vrai si cette heure est plus avancée que p0.

`Boolean` **<=**`(Time p0)`

Vrai si cette heure est moins avancée, ou identique à p0.

`Boolean` **>=**`(Time p0)`

Vrai si cette heure est plus avancée ou identique à p0.

VII - Compilation - Liste des tokens de l'arbre intermédiaire

ERROR
EOF
NULL_TREE_LOOKAHEAD
SE_EXPRLIST
SE_EXPRRANGE
SE_This
SE_Litteral_null
SE_Litteral_Integer
SE_Litteral_Real
SE_Litteral_Text
SE_Litteral_String
SE_Litteral_Boolean
SE_Litteral_Type
SE_Litteral_Enum
SE_Litteral_Set
SE_Litteral_Bag
SE_Litteral_Sequence
SE_Litteral_Double
SE_Litteral_Float
SE_Litteral_Long
SE_Litteral_Int
SE_Litteral_Short
SE_Litteral_Byte
SE_DoNothing
SE_OclAny__eq
SE_OclAny__ne
SE_OclAny_oclIsKindOf
SE_OclAny_oclIsTypeOf
SE_OclAny_evaluationType
SE_OclAny_oclAsType
SE_OclAny_delete
SE_OclType_name
SE_OclType_attributes
SE_OclType_associationEnds
SE_OclType_operations
SE_OclType_supertypes
SE_OclType_allSupertypes
SE_OclType_allInstances
SE_OclType_toString
SE_Real_rand
SE_Real__eq
SE_Real__ne
SE_Real__unaryPlus
SE_Real__unaryMinus
SE_Real__add
SE_Real__sub
SE_Real__mul
SE_Real__div
SE_Real_abs
SE_Real_floor
SE_Real_round
SE_Real_max
SE_Real_min
SE_Real__lt
SE_Real__gt
SE_Real__le
SE_Real__ge
SE_Real_toString
SE_Integer__eq
SE_Integer__ne
SE_Integer__add
SE_Integer__unaryPlus
SE_Integer__unaryMinus
SE_Integer__sub
SE_Integer__mul
SE_Integer__div
SE_Integer_abs
SE_Integer_div
SE_Integer_mod
SE_Integer_max
SE_Integer_min
SE_Integer__lt
SE_Integer__gt
SE_Integer__le
SE_Integer__ge
SE_Integer_toString
SE_Text__eq
SE_Text__ne
SE_Text_size
SE_Text_concat
SE_Text_toUpper
SE_Text_toLower
SE_Text__lt
SE_Text__gt
SE_Text__le
SE_Text__ge
SE_Text_substring
SE_Text_crypt
SE_String_parseReal
SE_String_parseInteger
SE_String_parseDouble
SE_String_parseFloat
SE_String_parseLong
SE_String_parseInt
SE_String_parseShort
SE_String_parseByte
SE_String_parseBoolean
SE_String_parseDate
SE_String_parseTime
SE_String_checkReal
SE_String_checkInteger
SE_String_checkDouble
SE_String_checkFloat
SE_String_checkLong
SE_String_checkInt
SE_String_checkShort
SE_String_checkByte
SE_String_checkBoolean
SE_String_checkDate
SE_String_checkTime
SE_Boolean__eq
SE_Boolean__ne
SE_Boolean_or
SE_Boolean_xor
SE_Boolean_and
SE_Boolean_not
SE_Boolean_implies
SE_Boolean_if
SE_Boolean_toString
SE_Date_getCurrent
SE_Date_getDate
SE_Date__eq
SE_Date__ne
SE_Date_daysFrom
SE_Date_addDay
SE_Date_addMonth
SE_Date_addYear
SE_Date_getDay
SE_Date_getMonth
SE_Date_getYear
SE_Date_max
SE_Date_min
SE_Date__lt
SE_Date__gt
SE_Date__le
SE_Date__ge
SE_Date_toString
SE_Time_getCurrent
SE_Time_getTime
SE_Time__eq
SE_Time__ne
SE_Time_addMilli
SE_Time_addSecond
SE_Time_addMinute
SE_Time_addHour
SE_Time_getMilli
SE_Time_getSecond
SE_Time_getMinute
SE_Time_getHour
SE_Time_max
SE_Time_min
SE_Time__lt
SE_Time__gt
SE_Time__le
SE_Time__ge
SE_Time_toString
SE_Collection_size
SE_Collection_includes
SE_Collection_excludes
SE_Collection_count
SE_Collection_includesAll
SE_Collection_excludesAll

SE_Collection_isEmpty
SE_Collection_notEmpty
SE_Collection_sum
SE_Collection_exists
SE_Collection_forAll
SE_Collection_isUnique
SE_Collection_sortedBy
SE_Collection_sortedBy_Directed
SE_Collection_getOne
SE_Set_union_Set
SE_Set_union_Bag
SE_Set__eq
SE_Set__ne
SE_Set_intersection_Set
SE_Set_intersection_Bag
SE_Set__sub
SE_Set_including
SE_Set_excluding
SE_Set_symmetricDifference
SE_Set_select
SE_Set_reject
SE_Set_collect
SE_Set_count
SE_Set_asSequence
SE_Set_asBag
SE_Bag__eq
SE_Bag__ne
SE_Bag_union_Bag
SE_Bag_union_Set
SE_Bag_intersection_Bag
SE_Bag_intersection_Set
SE_Bag_including
SE_Bag_excluding
SE_Bag_select
SE_Bag_reject
SE_Bag_collect
SE_Bag_count
SE_Bag_asSequence
SE_Bag_asSet
SE_Sequence_count
SE_Sequence__eq
SE_Sequence__ne
SE_Sequence_union
SE_Sequence_append
SE_Sequence_prepend
SE_Sequence_subSequence
SE_Sequence_at
SE_Sequence_first
SE_Sequence_last
SE_Sequence_including
SE_Sequence_excluding
SE_Sequence_select
SE_Sequence_reject
SE_Sequence_collect
SE_Sequence_asBag
SE_Sequence_asSet
SE_Byte_floor
SE_Byte_round
SE_Byte__eq
SE_Byte__ne
SE_Byte__add
SE_Byte__unaryPlus
SE_Byte__unaryMinus
SE_Byte__sub
SE_Byte__mul
SE_Byte__div
SE_Byte_abs
SE_Byte_div
SE_Byte_mod
SE_Byte_max
SE_Byte_min
SE_Byte__bnot
SE_Byte__band
SE_Byte__bor
SE_Byte__bxor
SE_Byte__sl
SE_Byte__sr
SE_Short_floor
SE_Short_round
SE_Short__eq
SE_Short__ne
SE_Short__add
SE_Short__unaryPlus
SE_Short__unaryMinus
SE_Short__sub
SE_Short__mul
SE_Short__div
SE_Short_abs
SE_Short_div
SE_Short_mod
SE_Short_max
SE_Short_min
SE_Short__bnot
SE_Short__band
SE_Short__bor
SE_Short__bxor
SE_Short__sl
SE_Short__sr
SE_Int_floor
SE_Int_round
SE_Int__eq
SE_Int__ne
SE_Int__add
SE_Int__unaryPlus
SE_Int__unaryMinus
SE_Int__sub
SE_Int__mul
SE_Int__div
SE_Int_abs
SE_Int_div
SE_Int_mod
SE_Int_max
SE_Int_min
SE_Int__bnot
SE_Int__band
SE_Int__bor
SE_Int__bxor
SE_Int__sl
SE_Int__sr
SE_Long_floor
SE_Long_round
SE_Long__eq
SE_Long__ne
SE_Long__add
SE_Long__unaryPlus
SE_Long__unaryMinus
SE_Long__sub
SE_Long__mul
SE_Long__div
SE_Long_abs
SE_Long_div
SE_Long_mod
SE_Long_max
SE_Long_min
SE_Long__bnot
SE_Long__band
SE_Long__bor
SE_Long__bxor
SE_Long__sl
SE_Long__sr
SE_Float__eq
SE_Float__ne
SE_Float__unaryPlus
SE_Float__unaryMinus
SE_Float__add
SE_Float__sub
SE_Float__mul
SE_Float__div
SE_Float_abs
SE_Float_floor
SE_Float_round
SE_Float_max
SE_Float_min
SE_Double__eq
SE_Double__ne
SE_Double__unaryPlus
SE_Double__unaryMinus
SE_Double__add
SE_Double__sub
SE_Double__mul
SE_Double__div
SE_Double_abs
SE_Double_floor
SE_Double_round
SE_Double_max
SE_Double_min
SE_Enum_toString
SE_Call_Link
SE_Variable_Call
SE_Call_Attribute
SE_Call_Operation
SE_Call_Constructor
SE_While
SE_If
SE_Block
SE_Return
SE_Declare_Local_Variable
SE_Variable_Name
SE_Assign

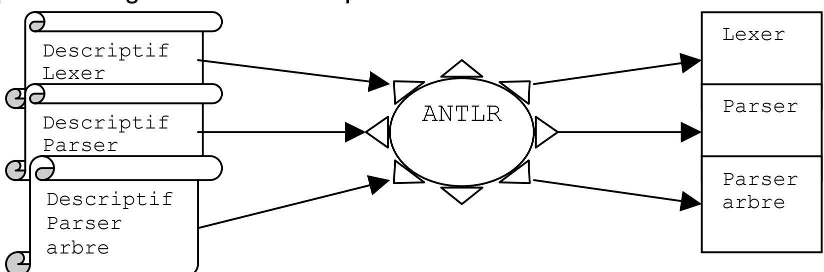
SE_AddLink
SE_RemoveLink
SE_Cast
SE_Write
SE_WriteRaw
SE_DestroySession
SE_SQL_Expression
SE_FromIteratingDefinition
SE_FromLocalDefinition
SE_FromParameter
SE_FromDecisionCenterContext
SE_FromWebFileContext
SE_FromSessionContext
SE_SIDVar

VIII - ANTLR – Présentation

ANTLR (http://www.antlr.org) est un outil de la famille des compilateurs de compilateur, à l'instar du couple Lex/Yacc ou JavaCC (http://www.metamata.com/). Un compilateur de compilateur est un outil qui lit des spécifications d'une grammaire dans un fichier et les convertit en programme (dans notre cas Java, Sather ou C++) capable de reconnaître un texte s'y conformant. En plus du générateur d'analyseur, ANTLR fournit d'autres possibilités standard liées à la construction d'arbre abstrait, la gestion des erreurs, l'insertion d'actions sous forme de code source… ANTLR permet également la génération de visiteurs d'arbre abstrait en introduisant la notion de grammaire d'arbre. Cet outil est la version Java de PCCTS.

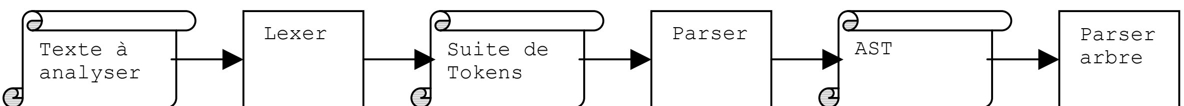
Les analyseurs générés sont de type LL(K) (où K est une constante qui peut être définie). Ceci permet notamment de travailler au format EBNF (*, +, ?, …), cependant, la récursivité gauche n'est pas permise. De plus, cette technologie permet un déverminage facilité, en comparaison au parseurs LR, aussi bien au niveau des messages d'erreurs donnés par le compilateur ANTLR que pour l'utilisation des débogueurs pour Java sur le code généré. Il est possible de définir des prédicats au sein de la grammaire lors d'un choix multiple. Ces prédicats sont soit syntaxiques (vérification q'une règle syntaxique peut être appelée sans erreur), ce qui permet d'augmenter localement la valeur de K, soit sémantique, c'est à dire donné en code Java.

La compilation de grammaire est simple :



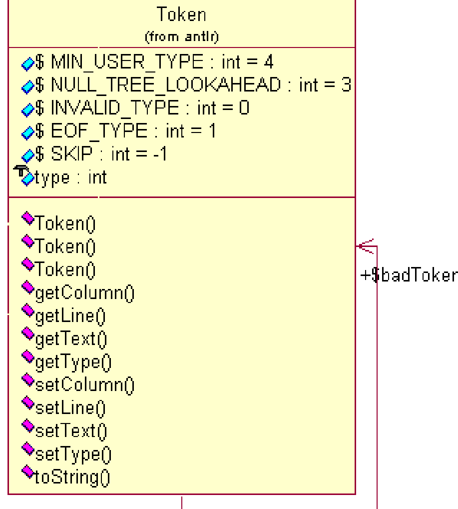
Chacun des éléments généré est donné sous forme de classe. En cas de génération pour Java, ces classes utilisent celles d'ANTLR ce qui rend nécessaire l'intégration d'ANTLR dans un logiciel l'utilisant. En effet, ANTLR est (en partie) généré par ANTLR, comme c'est souvent le cas dans ce genre d'outil.

L'analyseur généré est tout à fait classique :



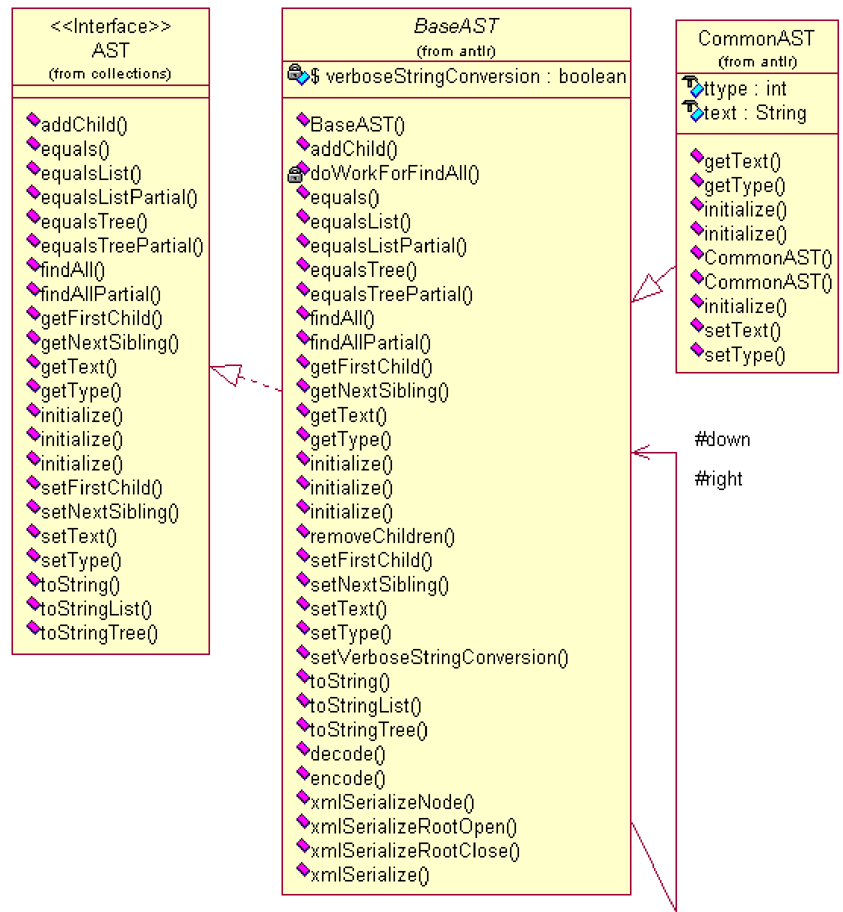
Ceci n'est qu'un aperçu des possibilités d'ANTLR. En effet, le lexer est capable de gérer plusieurs suites de Tokens, le parseur d'arbre est capable de générer un nouvel arbre, etc.…

Les tokens sont des classes simples :



Ils informent sur leur type (*get/setType*), leur ligne (*get/setLine*) et leur texte (*get/setText*).

Les arbres abstraits générés le sont également :



Les différents nœuds sont reconnus grâce à leur type (*get/setType*), soit un attribut numérique. A chaque nœud est aussi associé un texte (*get/setText*). Chaque nœud connaît son voisin (*get/setNextSibling*) et son premier enfant (*get/setFirstChild*).

Le Lexer

```
class MonLexer extends Lexer;
```

Il est possible :

Définir K

```
Options {k=2 ;}
```

Définir les tokens reconnus

```
GT : '>' ;
GE : ">= " ; //besoin de k=2
```

Définir des tokens internes

```
protected DIGIT : '1'..'9'; //Token privé
INT : DIGIT+ ;
```

Insérer du code source et accéder au token produit par `$`

```
WS : (' ' | '\t' | '\n' | ...)
      {$setType(Token.SKIP);}
```

Le Parser

```
class MonParser extends Parser;
```

Il est possible :

Construire un arbre
`^` indique le nœud principal
`!` indique un noeud ignoré

```
instr : ID AFFECT^ expression SEMI!
      | "if"^ LPAREN! expression RPAREN!
          instr ("else"! instr)?;
```

Fabriquer de nouveaux tokens

```
tokens {INSTRLIST ; UNARY_PLUS ;}
```

Redéfinir un arbre en insérant du code
`N` : nomme un élément
`#` indique un nœud de l'AST

```
factorExpr : p:PLUS^ {#p.setType(UNARY_PLUS);}
             basicExpr;
prog : (instr)*
       {#prog=#(#[INSTRLIST, "INTRLIST"], prog);}
       EOF!;
```

Le Parser d'arbre

```
class MonParserArbre extends TreeParser;
```

Il est possible :

Retourner des valeurs
`#(parent enfant1 enfant2 …)` représente la structure de l'arbre

```
expr returns [String s] {String e1,e2; s="";}:
      #(PLUS e1=expr e2=expr) {s=e1+e2+"Add";}
    | a:ID {s=a.getText();};
```

Accepter des arguments

```
test [String onTrue, String onFalse]
    returns [String s] {…;}:
      (
          #(EQ e1=expr e2=expr){s=e1+e2+"Equal";}
        | ...
      ) {s+="JumpIfTrue"+onTrue+"Jump"+onFalse;}
    | #(NOT e1=test[onFalse, onTrue]) {s=e1;}
    |…;
```

IX - Liste des optimisations SQL

Toutes les bases

SE_SQL_Expression
SE_WriteRaw
SE_Write
SE_Assign
SE_Declare_Local_Variable
SE_Return
SE_Block
SE_If
SE_While
SE_Call_Operation
SE_Call_Attribute
SE_Variable_Call
SE_Call_Link
SE_Enum_toString
SE_Double__div
SE_Double__mul
SE_Double__sub
SE_Double__add
SE_Double__unaryMinus
SE_Double__unaryPlus
SE_Double__ne
SE_Double__eq
SE_Float__div
SE_Float__mul
SE_Float__sub
SE_Float__add
SE_Float__unaryMinus
SE_Float__unaryPlus
SE_Float__ne
SE_Float__eq
SE_Long__div
SE_Long__mul
SE_Long__sub
SE_Long__unaryMinus
SE_Long__unaryPlus
SE_Long__add
SE_Long__ne
SE_Long__eq
SE_Int__div
SE_Int__mul
SE_Int__sub
SE_Int__unaryMinus
SE_Int__unaryPlus
SE_Int__add
SE_Int__ne
SE_Int__eq
SE_Short__div
SE_Short__mul
SE_Short__sub
SE_Short__unaryMinus
SE_Short__unaryPlus
SE_Short__add
SE_Short__ne
SE_Short__eq
SE_Byte__div
SE_Byte__mul
SE_Byte__sub
SE_Byte__unaryMinus
SE_Byte__unaryPlus
SE_Byte__add
SE_Byte__ne
SE_Byte__eq
SE_Sequence_asSet
SE_Sequence_asBag
SE_Sequence_collect
SE_Sequence_reject
SE_Sequence_select
SE_Sequence_excluding
SE_Bag_asSet
SE_Bag_asSequence
SE_Bag_collect
SE_Bag_reject
SE_Bag_select
SE_Bag_excluding
SE_Set_asBag
SE_Set_asSequence
SE_Set_collect
SE_Set_reject
SE_Set_select
SE_Set_excluding
SE_Collection_sortedBy_Directed
SE_Collection_sortedBy
SE_Collection_isUnique
SE_Collection_forAll
SE_Collection_exists
SE_Collection_sum
SE_Collection_notEmpty
SE_Collection_isEmpty
SE_Collection_excludesAll
SE_Collection_includesAll
SE_Collection_count
SE_Collection_excludes
SE_Collection_includes
SE_Collection_size
SE_Time__ge
SE_Time__le
SE_Time__gt
SE_Time__lt
SE_Time__ne
SE_Time__eq
SE_Date__ge
SE_Date__le
SE_Date__gt
SE_Date__lt
SE_Date__ne
SE_Date__eq
SE_Boolean_implies
SE_Boolean_not
SE_Boolean_and
SE_Boolean_xor
SE_Boolean_or
SE_Boolean__ne
SE_Boolean__eq
SE_Text__ge
SE_Text__le
SE_Text__gt
SE_Text__lt
SE_Text_concat
SE_Text__ne
SE_Text__eq
SE_Integer__ge
SE_Integer__le
SE_Integer__gt
SE_Integer__lt
SE_Integer__div
SE_Integer__mul
SE_Integer__sub
SE_Integer__unaryMinus
SE_Integer__unaryPlus
SE_Integer__add
SE_Integer__ne
SE_Integer__eq
SE_Real__ge
SE_Real__le
SE_Real__gt
SE_Real__lt
SE_Real__div
SE_Real__mul
SE_Real__sub
SE_Real__add
SE_Real__unaryMinus
SE_Real__unaryPlus
SE_Real__ne
SE_Real__eq
SE_OclType_allInstances
SE_OclAny__ne
SE_OclAny__eq
SE_Litteral_Byte
SE_Litteral_Short
SE_Litteral_Int
SE_Litteral_Long
SE_Litteral_Float
SE_Litteral_Double
SE_Litteral_Enum
SE_Litteral_Boolean
SE_Litteral_String
SE_Litteral_Text
SE_Litteral_Real
SE_Litteral_Integer

SE_Litteral_null
SE_This

MySQL

SE_Collection_count
SE_Collection_size
SE_Time_min
SE_Time_max
SE_Time_getHour
SE_Call_Attribute
SE_Call_Link
SE_Time_getCurrent
SE_Double_min
SE_Double_max
SE_Double_round
SE_Double_floor
SE_Double_abs
SE_Date_min
SE_Date_max
SE_Date_getYear
SE_Date_getMonth
SE_Date_getDay
SE_Float_min
SE_Float_max
SE_Float_round
SE_Date_getDate
SE_Float_floor
SE_Date_getCurrent
SE_Float_abs
SE_Boolean_if
SE_Long__sr
SE_Long__sl
SE_Long__bxor
SE_Long__bor
SE_Long__band
SE_Long__bnot
SE_Long_min
SE_Long_max
SE_Long_mod
SE_Long_abs
SE_Long_round
SE_Long_floor
SE_Int__sr
SE_Int__sl
SE_Int__bxor
SE_Text_substring
SE_Int__bor
SE_Int__band
SE_Int__bnot
SE_Int_min
SE_Int_max
SE_Text_toLower
SE_Int_mod
SE_Text_toUpper
SE_Int_abs
SE_Text_size
SE_Text__eq
SE_Integer_min
SE_Int_round
SE_Integer_max
SE_Int_floor
SE_Integer_mod
SE_Short__sr
SE_Short__sl
SE_Integer_abs
SE_Short__bxor
SE_Short__bor
SE_Short__band
SE_Short__bnot
SE_Short_min
SE_Short_max
SE_Short_mod
SE_Short_abs
SE_Real_min
SE_Real_max
SE_Real_round
SE_Real_floor
SE_Short_round
SE_Real_abs
SE_Short_floor
SE_Byte__sr
SE_Byte__sl
SE_Byte__bxor
SE_Byte__bor
SE_Byte__band
SE_Byte__bnot
SE_Byte_min
SE_Byte_max
SE_Real_rand
SE_Byte_mod
SE_OclType_allInstances
SE_Byte_abs
SE_Byte_round
SE_Byte_floor
SE_Sequence_subSequence

Oracle

SE_Set_excluding
SE_Collection_sortedBy
SE_Collection_excludesAll
SE_Collection_includesAll
SE_Collection_count
SE_Collection_excludes
SE_Collection_includes
SE_Collection_size
SE_Time_min
SE_Time_max
SE_Call_Attribute
SE_Double_min
SE_Double_max
SE_Double_round
SE_Double_floor
SE_Double_abs
SE_Date_min
SE_Date_max
SE_Date_addYear
SE_Date_addMonth
SE_Float_min
SE_Float_max
SE_Float_round
SE_Float_floor
SE_Float_abs
SE_Long__band
SE_Long_min
SE_Long_max
SE_Long_mod
SE_Long_abs
SE_Long_round
SE_Long_floor
SE_Text_substring
SE_Text__ge
SE_Int__band
SE_Text__le
SE_Text__gt
SE_Int_min
SE_Text__lt
SE_Int_max
SE_Text_toLower
SE_Int_mod
SE_Text_toUpper
SE_Int_abs
SE_Text_size
SE_Text__eq
SE_Integer_min
SE_Int_round
SE_Integer_max
SE_Int_floor
SE_Integer_mod
SE_Integer_abs
SE_Short__band
SE_Short_min
SE_Short_max
SE_Short_mod
SE_Short_abs
SE_Real_min
SE_Real_max
SE_Real_round
SE_Real_floor
SE_Short_round
SE_Real_abs
SE_Short_floor
SE_Byte__band
SE_Byte_min
SE_Byte_max
SE_Byte_mod
SE_Byte_abs
SE_Byte_round
SE_Byte_floor
SE_Sequence_excluding
SE_Bag_excluding