



Groupe SODIFRANCE

Nantes



Université de Nantes

Reverse engineering C# and .Net

Mathieu ALBERT

Stage de fin d'études
MASTER PROFESSIONNEL ALMA
2006-2007

Maître de stage : M. BONNET
Tuteur : M. SUNYÉ



Table des matières

Table des matières	1
Introduction	3
1 Organisation du Stage	4
1.1 L'Entreprise Sodifrance	4
1.2 Le Stage	7
2 Les outils et techniques utilisées	11
2.1 La grammaire	11
2.2 La transformation	12
2.3 Présentation des outils	13
3 Construction d'une grammaire C#	18
3.1 Un peu de cours	18
3.2 La grammaire	19
3.3 Le méta modèle C#	23
4 Rétro-ingénierie	25
4.1 Un peu de cours	25
4.2 Les outils existants	26
4.3 La création du modèle	27
5 Migration C# - Java	32
5.1 Un peu de cours	32
5.2 La comparaison C# - Java	33
5.3 Les règles de transformation	35
Conclusion	39
Bibliographie	40
Index	43
Table des figures	45



ANNEXES	46
A Abréviations	46
B Glossaire	49
C Fonctionnement du Common Language Infrastructure	50
D Un exemple d'utilisation de Reflector for .NET	52
E Étude du langage C#	54
F Une comparaison C# vs Java	86
G C# vs Java : Les mots clés	113
H Un résumé des plateformes .NET & J2EE	114



Introduction

Du 2 avril 2007 au 14 septembre 2007, j'ai effectué un stage au sein de l'entreprise Sodifrance (située à Nantes). Au cours de ce stage dans le département R&D, j'ai pu m'intéresser à l'étude du langage C# ainsi qu'aux méthodes de recherche et développement.

Plus largement, ce stage a été l'opportunité pour moi d'appréhender les difficultés rencontrées lors de projets professionnels. Ce stage a été aussi l'occasion de comprendre le fonctionnement interne d'une société informatique (éditrice de logiciel et SSII) avec toutes ses difficultés.

Au-delà d'enrichir mes connaissances en informatique, ce stage m'a permis de comprendre dans quelle mesure la mise en place de méthodes et d'organisation est importante dans la réussite d'un projet, ainsi que le contact avec ses collègues, ses supérieurs hiérarchiques et le cas échéant les clients.

En m'intéressant de près à l'étude d'un langage (le langage C#) et à son outil de cartographie, j'ai pu apprécier le travail des équipes de recherche et développement. C'est dans cette direction que mon projet professionnel s'oriente.

En vue de rendre compte de manière fidèle et analytique des 5 mois passés au sein de la société Sodifrance, il apparaît logique de présenter à titre préalable l'environnement économique du stage et le cadre du stage (chapitre 1, page 4). Les outils fournis par Sodifrance (chapitre 2, page 11) m'ont permis de réaliser les 3 tâches de mon stage, à savoir l'étude du langage (chapitre 3, page 18), l'analyse des fichiers sources (chapitre 4, page 25) et la migration vers la langage java (chapitre 5, page 32).

Remerciements :

Je tiens à remercier le Groupe Sodifrance et ses équipes pour m'avoir accueilli pour ce stage.

Je tiens à remercier plus particulièrement :

- M. Patrice BONNET, qui m'a suivi et guidé durant tout le stage.
- M. Jean-Michel PETIT, qui a su écouter mes attentes en adaptant le sujet du stage.
- M. Gwenaél POINTEAU, qui m'a fait une place dans son bureau et qui m'a aidé à maîtriser l'environnement Sodifrance.

Je voudrais également remercier M. Gerson SUNYÉ, mon tuteur, pour ses conseils et son aide.

1 Organisation du Stage

1.1 L'Entreprise Sodifrance

1.1.1 Présentation

Depuis sa création en 1986, Sodifrance accompagne les entreprises dans leurs projets de développement et de modernisation de leurs systèmes d'information, depuis la prise en compte de leurs besoins jusqu'à la réalisation des projets et le transfert des compétences.

Ses atouts en font un partenaire fort pour ses clients avec des offres technologiques reconnues, la capacité à mener les plus grands projets, des compétences techniques importantes, et de nombreuses références.

Les offres et savoir-faire :

- Un positionnement unique combinant offres technologiques spécialisées et services informatiques de proximité
- Une avance technologique reconnue sur des domaines ciblés : Legacy Modernization, MDA, Évolution vers les Nouvelles Technologies
- Une expérience importante des grands projets capitalisée dans une méthodologie éprouvée
- Une activité de Recherche et Développement permanente et soutenue, garantissant l'excellence des solutions proposées
- Des équipes de haut niveau, intervenant tout au long du cycle des projets
- La maîtrise de technologies multiples, la double culture Main-Frame - Nouvelles Technologies

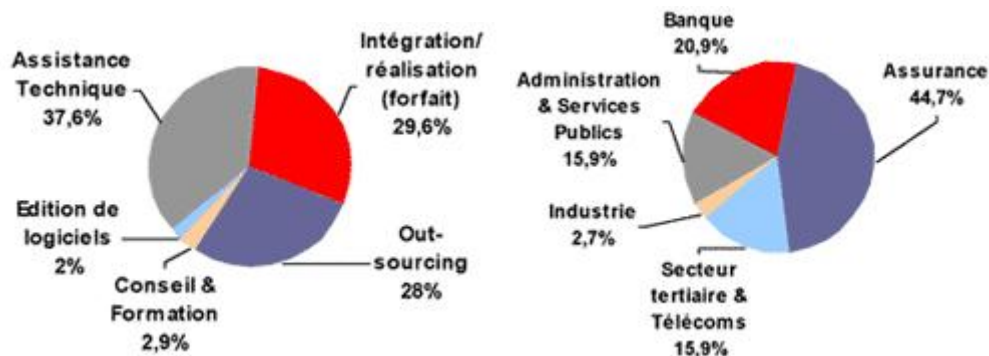


FIG. 1.1 – Répartition du CA 2006 par métier et par secteur d'activité.

1.1.2 Carte d'identité

Date de création	1986
Structure juridique	SA à Directoire et Conseil de surveillance
Chiffres d'affaires 2006	45 Millions d'euros (cf. figure 1.1)
Président du directoire	Franck MAZIN
Activités	Société de Services et d'Ingénierie Informatiques, avec : - une expertise reconnue en Modernisation de patrimoines applicatifs (Legacy Modernization) - une filiale d'édition de logiciels dédiés à l'optimisation du cycle de vie des applications : Mia-Software
Effectifs	570 collaborateurs au 31 décembre 2006, présents en France et en Belgique. 835 collaborateurs au 1er août 2007
Cotations	Eurolist d'Euronext Paris compartiment C ISIN : FR0000072563 Reuters : SDIF.PA Bloomberg : SDIF FP
Implantations	En France (Ouest) : Rennes (siège social), Nantes, Brest, Bordeaux, Niort, Le Mans En France (Est) : Paris, Orléans, Toulouse En Belgique : Bruxelles (cf. figures 1.2)

Assurance	AG2R, AGF, AXA FRANCE, AXA BELGIUM, CIMUT, CNP-ASSURANCES, DEXIA INSURANCE, INDEPENDANT INSURANCE, MAAF, MACIF, MAIF, IMA, MMA, MNAM, MEDERIC, P&V ASSURANCES, SMABTP, REUNICA, ...
Banque, Finance	BANQUE POPULAIRE, BNP, CAISSE DES DÉPOTS, CRÉDIT AGRICOLE, CRÉDIT INDUSTRIEL DE L'OUEST, CRÉDIT MUTUEL, CRÉDIT DU NORD, CRÉDIT LYONNAIS, FORTIS BANK, GE MONEY BANK, ING BELGIUM, SOCIÉTÉ GÉNÉRALE, PORTZAMPARC, ...
Industrie, Défense, Services	APEX, ARTIS, EADS, ERAM, CARRE BLANC, CONDAT, JANSSEN CILAG, LECLERC, OUEST-FRANCE, MAISONS DU MONDE, SAGESS, SYSTÈME U, TERMINAUX DE NORMANDIE, SIEMENS, THALES, VALEO, YORK FRANCE, YVES ROCHER, ...
Public, Para-public	CELAR, COMMUNAUTÉ URBAINE DE NANTES, ENSP, EQUANT, DCN, ÉTAT DE GENÈVE, LA POSTE, SNCF, MINISTÈRE DE LA JUSTICE, MINISTÈRE DE L'ÉDUCATION NATIONALE, PARLEMENT EUROPÉEN, MIPIH, TRANSPAC, ...

TAB. 1.1 – Clients du Groupe Sodifrance.

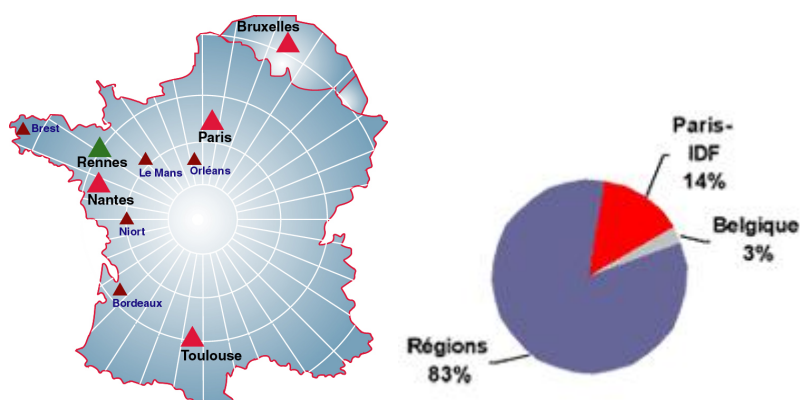


FIG. 1.2 – Répartition des effectifs 2006 en France et en Belgique.

1.1.3 Historique

1986	Création de Sodifrance par Francis Mazin. La société se positionne initialement sur des activités de monétique et d'infogérance.
1988	Sodifrance crée des activités d'ingénierie informatique.
1992	Sodifrance crée des activités de migration des systèmes d'information.
1994-98	Sodifrance se développe : acquisitions en France et au Luxembourg.
Avril 1999	Sodifrance est introduite au second marché de la bourse de Paris.
Janvier 2000	La gestion de Sodifrance est modifiée avec la mise en place d'un Conseil de Surveillance et d'un Directoire. Les activités du groupe sont recentrées sur les services informatiques, avec une expertise en migration, et sur la monétique.
Décembre 2001	Cession de TGD, la filiale de traitement du chèque.
Décembre 2003	Cession de C2S, la filiale monétaire.
2006	Recrutement de 140 collaborateurs
2007	Prévisions : Recrutement de 150 collaborateurs
Avril 2007	Acquisition de la société OneXt (spécialiste de la gestion de contenu open source)
Juillet 2007	Acquisition de la société API Group (propose aux grands comptes des prestations d'ingénierie et de Tierce Maintenance Applicative et capitalise sur une double compétence Mainframe-Nouvelles Technologies)

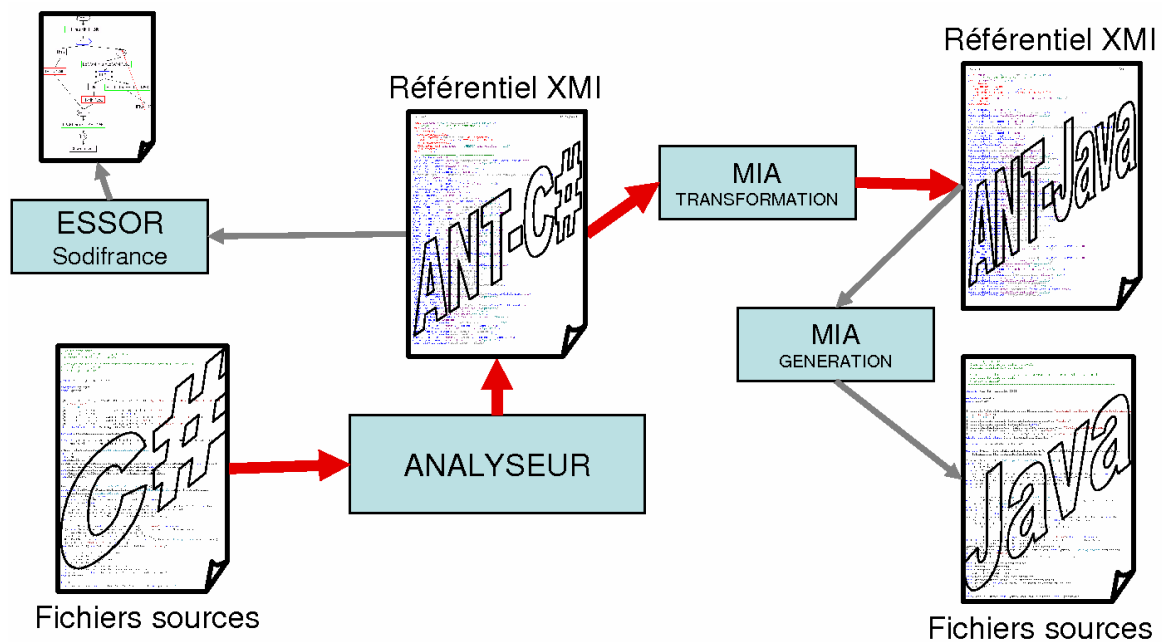


FIG. 1.3 – Schéma synthétique des différents composants et leurs échanges de données.

1.2 Le Stage

1.2.1 Le sujet

Le stage est intitulé : "Reverse engineering C# and dot.Net".

L'objectif est de proposer des briques de solutions de reverse engineering d'applications écrites en .Net.

Pour la société Sodifrance, ce stage est intéressant car il permet d'alimenter leur logiciel de cartographie. En effet, Essor est un produit de rétro documentation d'applications qui a pour fonction d'inventorier et de cartographier les composants et objets informatiques du système d'information. Cet inventaire se fait par une analyse statique régulière de l'ensemble des sources d'un site informatique. Le résultat des analyses est consolidé dans un référentiel. Le référentiel est ensuite accessible par un outil de consultation proposant notamment des fonctions graphiques (graphe d'utilisation d'une classe, graphe de dépendances ...).

Plus concrètement, le stage c'est deux étapes (comme on le voit sur la figure 1.3). La première est la création d'un analyseur de source C#, la seconde est une migration d'une technologie à une autre. On remarque sur cette même figure, comment s'intègre le projet dans l'environnement Sodifrance, et plus particulièrement l'outil ESSOR.

1.2.2 Les différentes parties

Le stage est découpé en plusieurs parties. Avant de toucher au contenu du stage, il est nécessaire de se familiariser avec les outils et les méthodologies utilisés par Sodifrance. Pour le déroulement du stage, comme indiquez précédemment, deux étapes sont nécessaires. La première est la

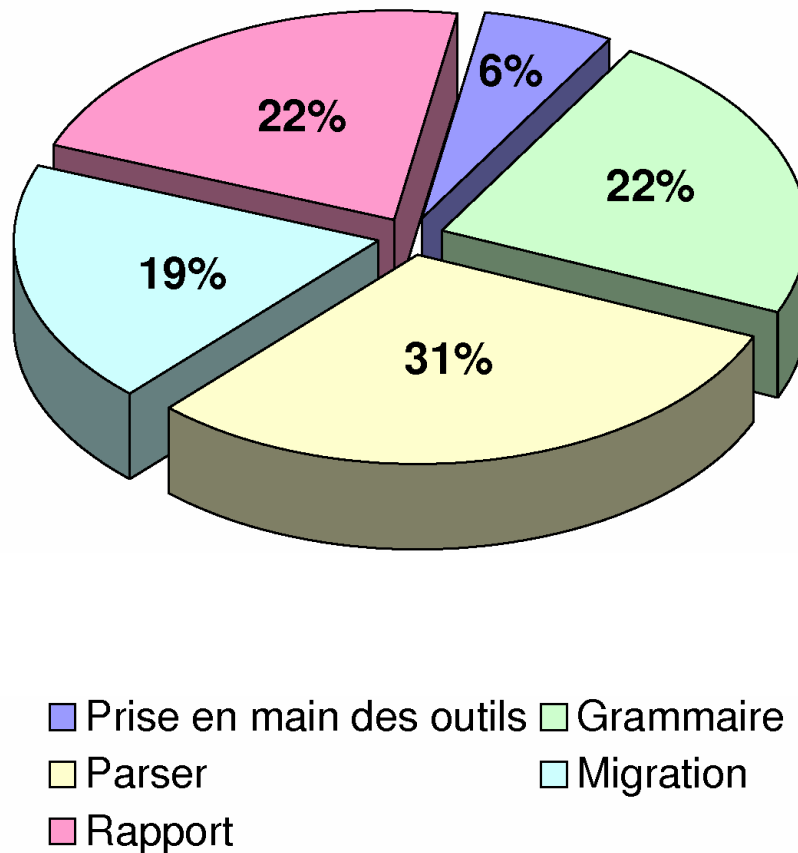


FIG. 1.4 – Découpage du temps de travail.

création d'un parseur. Celle ci est elle même composée de sous étapes : La modélisation et la partie de rétro ingénierie. La seconde étape est la migration C# vers Java. Pour finir et compléter ce stage, des ressources sont allouées pour la constitution d'un rapport.

Prise en main des outils

La première partie consistait à la prise en main des outils. En effet, pour la réalisation de ce stage, de nouveaux outils se sont présentés à moi. Tout d'abord, la programmation s'est effectué en C++, il m'a fallu maîtriser Visual C++ [1]. Pour la construction du parser à partir de la grammaire, j'ai utilisé un logiciel propre à Sodifrance T-gen. Ce dernier crée par Justin O. Graver [2] n'est plus soutenu et à été repris par Sodifrance qui l'a amélioré afin de pouvoir l'utiliser pour ses applications. Enfin, MIA Studio a été utilisé pour générer le code et effectuer la transformation [3].

Tâches	Estimation ¹		Réévaluation ¹
Prise en main des outils	9 HJ	↘	6 HJ
Méta modélisation et grammaire C#	19 HJ	↗	24 HJ
Implémentation des règles	28 HJ	↗	31 HJ
Solution de migration C# vers Java	28 HJ	↘	20 HJ
Rapport Soutenance	19 HJ	→	22 HJ

FIG. 1.5 – Durée établie pour chacune des tâches.

Méta modélisation et grammaire C#

La deuxième partie est l'étude du langage C# avec la modélisation du langage et la création de la grammaire du langage.

Implémentation des règles

La troisième partie est l'implémentation des règles de la grammaire afin d'alimenter le modèle. Le modèle ainsi créé sera exporter au format xmi.

Solution de migration C# vers Java

La quatrième partie est une solution de migration du code source C# vers Java.

Rapport Soutenance

La dernière partie permet de réaliser le rapport de stage ainsi que la préparation à la soutenance.

1.2.3 Le planning

Dans le but de suivre une organisation claire afin de mener à bien l'ensemble des tâches prévues nous avons établi des durées prévisionnelles pour chacune des parties.

Après avoir séparé les 5 étapes du stage, nous avons chiffré le temps nécessaire à leur réalisation. Pour cela, nous avons considéré que l'"Implémentation des règles" et la "Solution de migration" avaient besoin d'une durée importante : 6 semaines. La "Méta modélisation et grammaire" et le rapport seront effectués en 4 semaines et enfin, la prise en mains des outils en 2 semaines. Avec une estimation de 4,5 jours par semaine, nous avons établi l'estimation de la figure 1.5.

Pendant le déroulement du stage, nous nous sommes aperçu que l'évaluation méritait d'être revue. En effet, certaines parties (comme l'"Implémentation des règles") nécessitaient plus de temps, et d'autres moins de temps. Nous avons donc réalisé une réévaluation pour chacune des parties (figure 1.5) en rajoutant des HJ dans certaines parties, et en enlevant dans d'autres.

¹Le coût pour chaque étape du projet est défini en HJ : Homme Jour (cf. définition ANNEXE A page 47.)

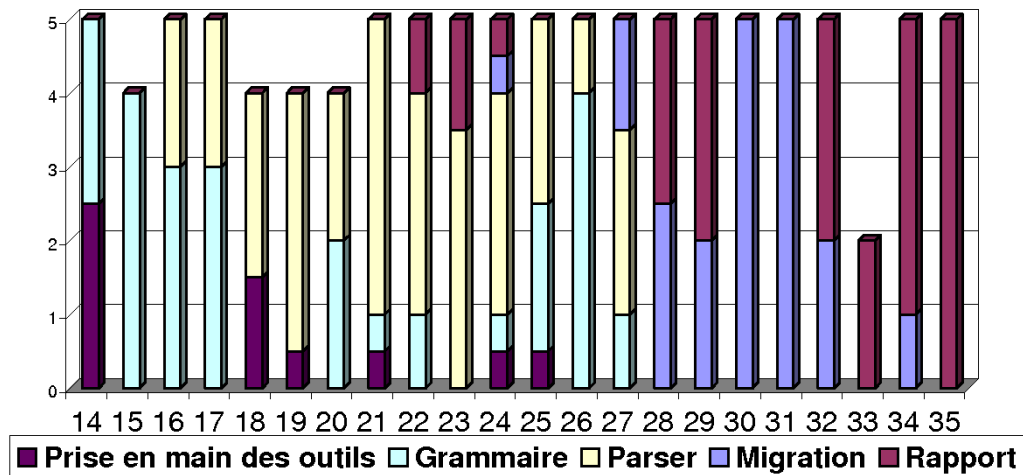


FIG. 1.6 – Répartition des tâches en jours par semaine.

Pour l'évolution des tâches et pour permettre de rester dans les temps, j'ai mis en place un suivi au jour le jour. Chaque temps passé sur une tâche est alors comptabilisé (à la demi journée près). Dans la figure 1.6, on remarque semaine par semaine le découpage des tâches.

2 Les outils et techniques utilisées

Durant le stage, j'ai été amené à manipuler des outils, pour la plupart propriétaires dont Sodifrance a l'habitude d'utiliser. Mais toutes ces applications possèdent des alternatives.

2.1 La grammaire

Pour la grammaire nous avons utilisé l'outil T-gen utilisé depuis plusieurs années par Sodifrance et adapté aux besoins de leurs projets.

2.1.1 Le choix d'un générateur d'analyseur

Pour créer des compilateurs, il existe de nombreux outils :

- Lex et Yacc : Lex et yacc sont des outils de génération d'analyseurs lexicaux (Lex) et syntaxiques (Yacc) en langage C
- Flex et Bison : Flex et bison les implémentations GNU de Lex et yacc.
- T-gen : T-gen permet la génération des parseurs LL et LR, qui seront automatiquement générés dans des arbres de dérivation des arbres de syntaxe abstraits ou d'objets arbitraire Smalltalk.
- LRgen : LRgen est un système de construction de parseurs, traducteurs, et compilateurs pour les langages informatiques. Il est basé sur la nouvelle forme normale TBNF[4]. il contient : un générateur de lexer, un générateur de parseur, un constructeur pour la table des symboles, un constructeur d'arbres syntaxiques abstraits. La sortie peut s'effectuer en C++ ou dans un autre langage.

Dans le tableau suivant, on remarque chaque outil permet de créer l'analyseur lexical et syntaxique. On remarque par ailleurs que le seul outil qui permet la génération d'arbres abstraits est l'outil Tgen

	Analyseur lexical	Analyseur syntaxique	Génération d'arbres abstraits
ANTLR	X	X	
BISON		X	
FLEX	X		
LEX	X		
LRGEN			
T-GEN	X	X	X
YACC		X	

Le tableau ci dessous montre les différents choix possibles ainsi que quelques informations (algorithmes, langages de sortie).

	Algorithme	Langage de Sortie	Plateforme
ANTLR	LL(k)	C++, C#, Java et Python	Toutes plateformes (java)
BISON	LALR, GLR	C, C++	Unix, Linux, Win
LRGEN	SLR, LALR, LR(1)	C++ ¹	Win
T-GEN	LL, LALR	Tous ²	Win
YACC	LALR	C, C++	Unix

2.1.2 Le choix T-gen

T-gen permet la génération des parseurs LL et LR, qui seront automatiquement générés dans des arbres de dérivation des arbres de syntaxe abstraites ou des objets arbitraire Smalltalk. La simple interface graphique améliore l'apprentissage, la compréhension et l'utilisation de T-gen. T-gen est écrit en Smalltalk.

Sodifrance travaille depuis de nombreuses années avec cet outil. Il a prouvé son efficacité. De plus, il a été repris en interne afin de répondre plus particulièrement aux besoins de la société.

T-gen permet la génération du parseur en classes smalltalk et en fichier xmi. Ce dernier est exploitable par n'importe quel outil de génération auquel on applique nos propres règles. C'est le cas de l'outil de Mia Software : Mia Génération. Un module de génération en C++ de Mia Génération a été créé par les équipes de recherches de Sodifrance.

2.2 La transformation

2.2.1 Le choix d'un outil de transformation

Pour la transformation, le nombre d'outils est important. Le domaine est en plein développement. Pour limiter la comparaison des outils, j'ai sélectionné les plus courants, en respectant la parité libre et propriétaire. Parmi ceux qui sont référencés dans le tableau suivant, on remarque qu'une majorité prend en compte le standard de l'OMG QVT[5].

Outil	Société	Licence	Commentaires
ATL	LINA	EPL	Implémentation de QVT
MDWORKBENCH	Sodius	Propriétaire	Utilisation d'ATL
MIA STUDIO	MIA Software	Propriétaire	Utilisation possible de plusieurs langages pour la description des règles : Java, MIATL, RL-TL
MODFACT	Lip6	GPL LIP6	Implémentation de QVT
OPENARCHITECTURE WARE	openArchitecture Ware	EPL	Le langage utilisé est xTend
SMARTQVT	France Telecom R&D	EPL	Implémentation de QVT
TOGETHER 2006	Borland	Propriétaire	Conforme aux standards QVT

¹Tous les langages peuvent être générés, à condition de créer le squelette correspondant.

²L'outil T-gen génère un fichier xmi. Ce fichier permet la génération dans le langage désiré (C++, Java, ...)

2.2.2 Le choix de MIA Transformation

Le logiciel

MIA Transformation est un module de la suite MIA Studio. Cette dernière permet l'automatisation des développements logiciels grâce à l'ingénierie des modèles MDA. Cette suite permet ainsi de :

- **Accroître la productivité des développements** tout en augmentant la qualité du code produit ;
- **Baisser les coûts de développement** des nouvelles applications ;
- **Réduire les coûts de maintenance** des applications développées.

MIA Transformation est un outil qui réalise des transformations personnalisées à partir d'un modèle source (en xmi) vers un modèle cible.

Chaque règle de transformation est composée de deux parties : une partie "requête" et une partie action. Ces deux parties peuvent être écrites en trois langages différents : Java, RL-TL et MIA-TL.

Ses points forts

Le choix s'est porté sur le logiciel MIA Studio. Une raison de ce choix est la compatibilité avec les outils utilisés pour la création du parseur MIA Génération. Le second choix est l'utilisation habituelle de ce logiciel par les équipes de R&D. Les méthodes de développement pour MIA Transformation sont très utilisées et donc fiables.

MIA DSL (Domain-Specific Language) permet d'intégrer n'importe quel méta modèle (créer par vos soins) à cet outil de transformation. Les méta modèles peuvent être décrit à partir de Rose, Poseidon, ...

La dernière raison est le choix du langage de description des règles. Java est un langage de programmation non spécifique à la transformation qui permet de décrire tout problème non lié spécifiquement à la transformation. Les deux autres langages sont développés pour les transformations. La description des règles de transformation est alors très simple. La combinaison des deux types de langages fournit un outil complet pour la transformation.

2.3 Présentation des outils

Après une petite étude des différents outils existant sur le marché, j'ai choisi un outil pour chacune des étapes. Ces deux outils ont été choisis par les critères cités ci-dessus mais aussi par le fait que ces outils sont connus dans le service R&D de Sodifrance qui fournit une assistance importante de ces deux outils.

2.3.1 T-gen

L'outil T-gen est un générateur de traducteur "string-to-object". Il est composé de plusieurs parties, comme on le voit sur la figure 2.1. Les deux cadres de gauche sont plus intéressants.

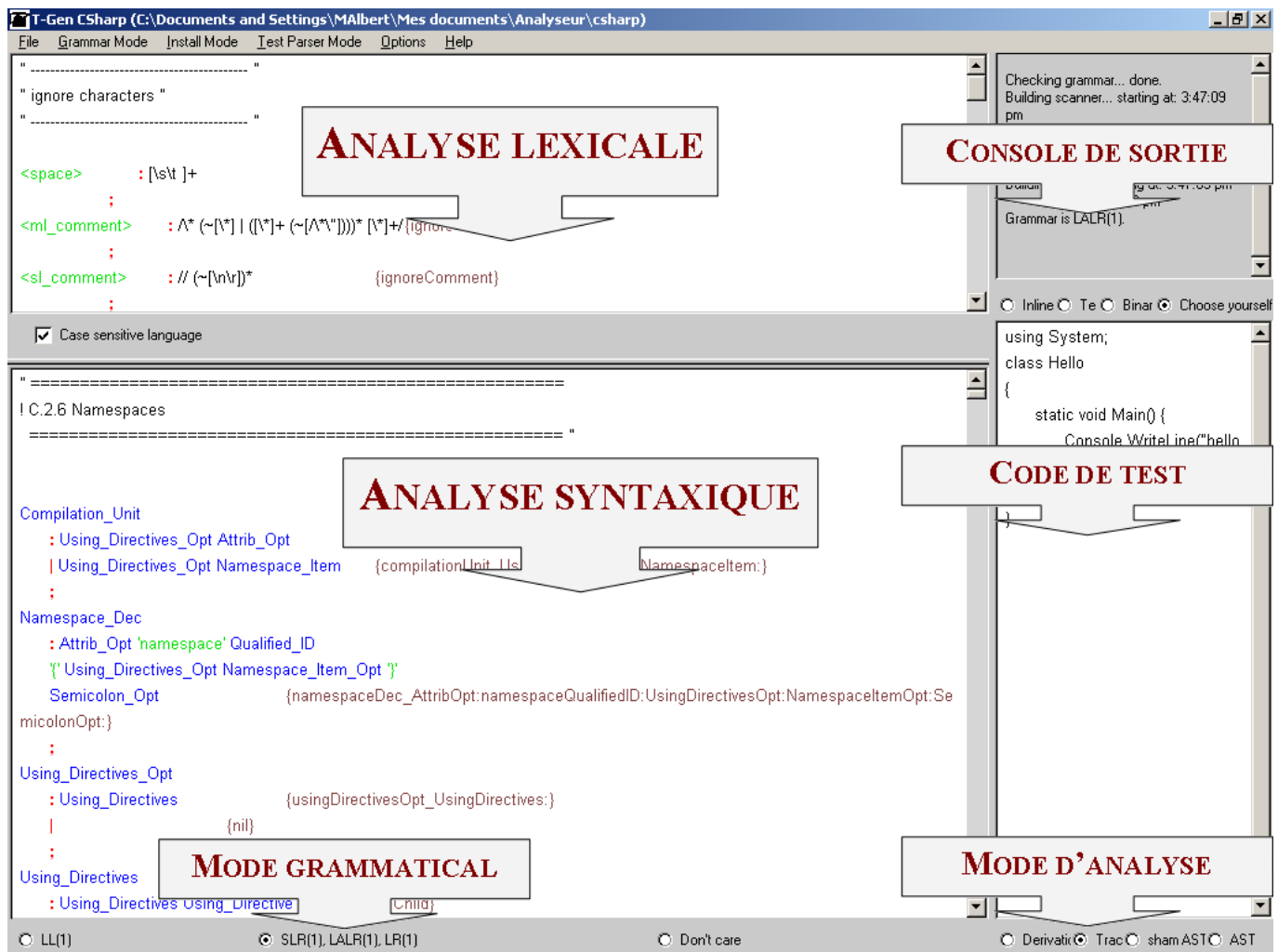


FIG. 2.1 – TGEN : l'outil.

Les deux cadres de droite sont :

- la console qui affiche les messages à l'attention de l'utilisateur.
- l'exemple qui permet de vérifier que la grammaire accepte bien l'exemple.

Analyseur lexical

ANALYSE LEXICALE : Elle lit les caractères d'entrée et produit comme résultat une suite d'unités lexicales que l'analyseur syntaxique aura à traiter. En plus, l'analyseur lexical réalise certaines tâches secondaires comme l'élimination de caractères superflus (commentaires, tabulations, fin de lignes, ...).

Analyseur syntaxique

ANALYSE SYNTAXIQUE : L'analyseur syntaxique reçoit une suite d'unités lexicales de la part de l'analyseur lexical et doit vérifier que cette suite peut être engendrée par la grammaire du langage.

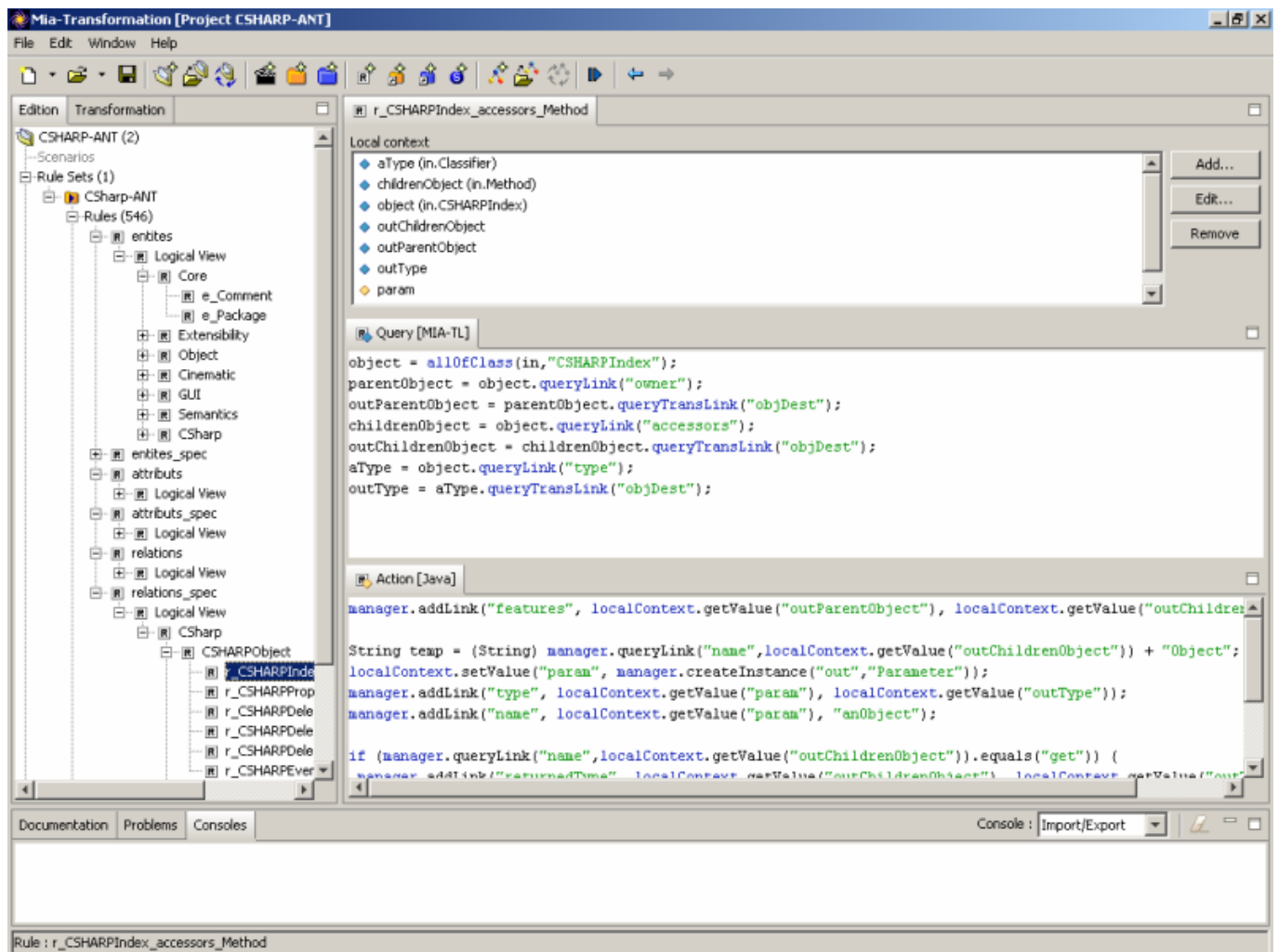


FIG. 2.2 – MIA-Transformation : les règles de transformation.

2.3.2 MIA-Transformation

L'outil MIA Transformation est utilisé pour :

1. Effectuer une transformation
2. Visualiser les modèles

La transformation

Dans la partie gauche de la fenêtre (figure 2.2), une arborescence présente les différentes règles. La règle sélectionnée est détaillée dans la partie droite à l'aide de 3 cadres : le contexte local, la requête et l'action :

Le CONTEXTE LOCAL permet de définir les variables manipulées par la règle courante (en lecture ou en écriture) et/ou ses sous règles (en lecture seulement), pour stocker des éléments particuliers du modèle ou des données de type primitif (booléens, chaînes, etc.) ou toute classe Java accessible. Elles sont soit alimentées par la requête, soit par l'action, soit privées (non accessibles aux sous règles).

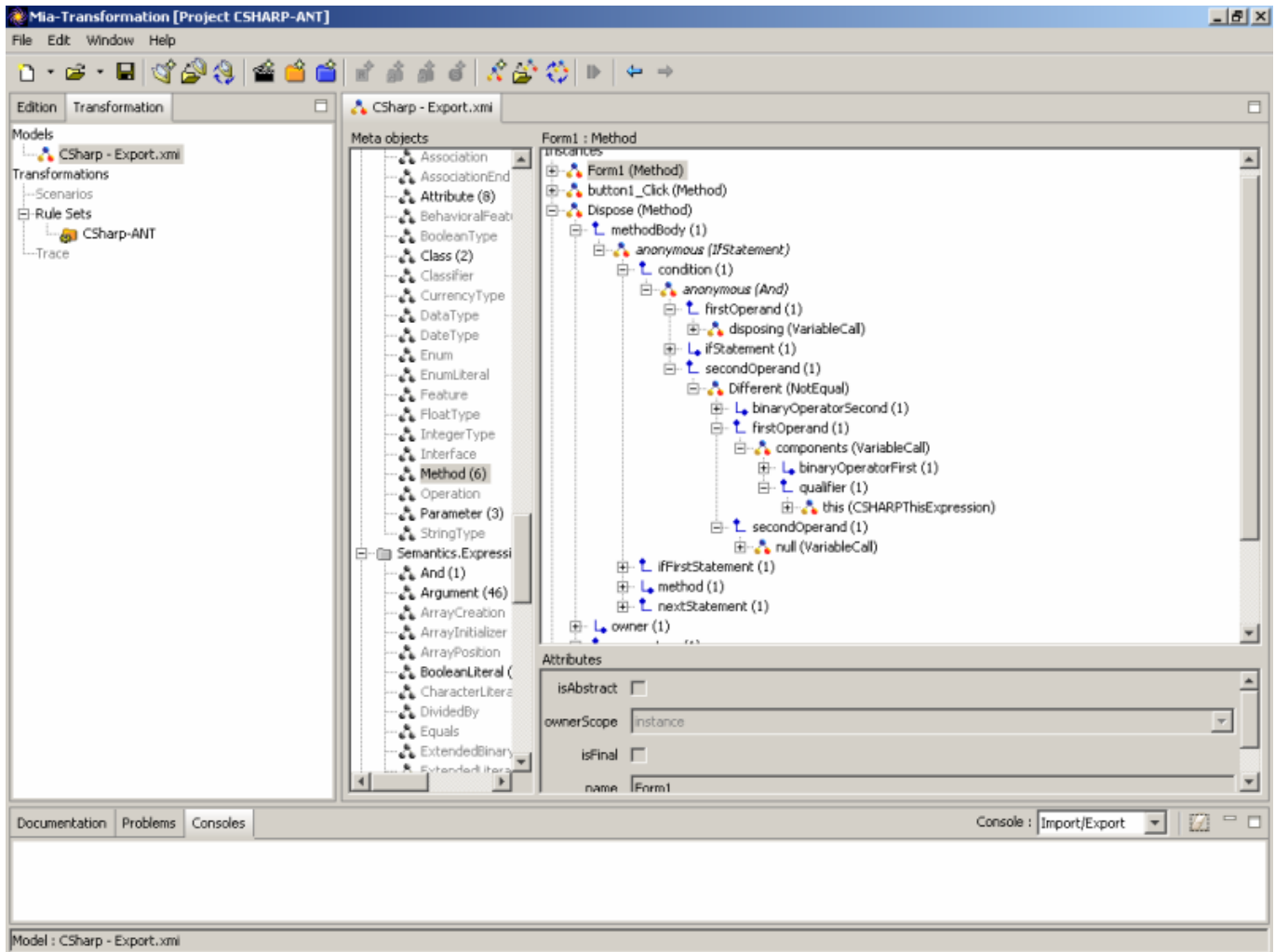


FIG. 2.3 – MIA-Transformation : la lecture de modèles.

La REQUÊTE est chargée de chercher les ensembles d'objets du modèle qui vérifient certaines conditions. Elle peut être exprimée dans 3 langages différents : Java, MIA-TL et RL-TL. Ces 2 derniers sont des langages proches d'un langage d'inférence utilisés pour des requêtes simples.

L'ACTION définit le traitement à appliquer sur les éléments fournis par la requête. Elle est exprimée dans un des 3 langages précédents (pas obligatoirement le même que pour la requête).

Une règle de transformation consiste à créer de nouvelles instances et à recopier les éléments d'une instance vers une autre. Les attributs et les associations définies dans le métamodèle sont tous considérés comme des liens par MIA-Transformation. Ainsi, pour spécifier la valeur d'un attribut par exemple, on ajoute un lien entre l'attribut et cette valeur.

La transformation d'un modèle consiste à exécuter l'ensemble des règles définies dans un scénario pour lequel on précise le ou les modèles d'entrée (et donc les métamodèles correspondants) et le ou les modèles de sortie (éventuellement correspondant à d'autres métamodèles).



La lecture des modèles

Dans le but de lire les modèles sources et cibles, MIA-Transformation affiche le modèle de façon graphique.

Ainsi dans la partie gauche (figure 2.3) est affiché les modèles chargés, les transformations du projet et optionnellement une trace de l'exécution (pour faciliter la résolution des problèmes). Ici un modèle en entrée est sélectionné et l'arborescence centrale présente les différents éléments du métamodèle correspondant (ici avec une présentation alphabétique) avec pour chacun le nombre d'instances dans le modèle.

Les instances de l'élément en surimpression sont détaillées à droite avec une hiérarchie de ses liens. Les attributs de l'objet sélectionné sont alors affichés en dessous. Les modèles sont stockés soit dans un format XMI, soit directement dans le format d'applications de modélisation comme Rational Rose par exemple.

L'application donne également accès à la consultation de l'ensemble des métamodèles disponibles à l'aide d'un browser.

3 Construction d'une grammaire C#

3.1 Un peu de cours

3.1.1 Définitions

La grammaire est l'étude des règles qui régissent un langage.

Grammaire

Une grammaire est quadruplet $G = \langle V_N, V_T, S, R \rangle$. Les différents composants de la grammaire sont :

- V_N : le vocabulaire non terminal ;
- V_T : le vocabulaire terminal ;
- S : l'ensemble du vocabulaire non terminal en partie gauche, ($S \in V_N$) ;
- R : l'ensemble des règles. Une règle est un couple qu'on note : $\varphi \rightarrow \psi$, où $V_N \cap V_T = \emptyset$, $\varphi \in V_N$ et $\psi \in (V_N \cup V_T)^*$

Vocabulaire terminal

Le vocabulaire terminal est l'ensemble des littéraux que l'on trouve dans les sources C#. on trouve alors :

- Les mots clés, qui sont des mots réservés avec un sens particulier (au nombre de 77) ;
- Les chiffres ;
- Les identifiants ;
- Les chaînes de caractères (entourées par " ") ;

L'étude de ce langage permet d'établir une grammaire. Même si la spécification ECMA existe, le travail de construction n'est pas simple pour autant.

3.1.2 L'étude du langage

Le langage C# créé par Microsoft® permet de développer des applications pour les systèmes d'exploitation Microsoft Windows® et le Web en se basant sur l'architecture .NET™.

Ce langage a été conçu dans le but d'être indépendant de la plate-forme et de l'environnement d'exécution. Normalement, le compilateur C# devrait pouvoir fonctionner partout où les types et fonctionnalités spécifiés dans l'ECMA-334 [6] sont correctement prises en charge par l'environnement d'exécution.

Actuellement, l'environnement d'exécution du langage commun (CLR : Common Language Runtime) de la technologie .NET est évidemment, le domaine de prédilection du langage C#. Mais il



existe d'autres environnements en cours de développement. Ils sont libres, on peut citer par exemple le projet mono [7].

Le mécanisme de traitement du code source écrit en C# par la plateforme Microsoft.NET, produit lors de la phase de compilation, un pseudo code dénommé MSIL (MicroSoft Intermediate Language), lequel sera interprété ensuite par la machine virtuelle (CLR) chargé de la transformation en instructions exécutable par la machine.

Il a été conçu afin de concurrencer Java dans la plateforme J2EE. Reprenant la syntaxe générale et les concepts de Java, C# reste quand même proche du C++. En effet, ce langage est une savante combinaison des langages C, C++ et Java en expurgeant la plupart des sources d'erreurs et en améliorant certaines fonctionnalités.

Comme pour java, la technologie .NET met également à la disposition du programmeur de nombreuses classes du .NET Framework, comprenant une pléthore de méthodes et propriétés exploitables dans des programmes C#.

La gestion de la mémoire par l'utilisation des pointeurs a été maintenue tout en veillant à y intégrer un dispositif d'appel explicite par l'utilisation de mots-clés spécifiques.

Le concept d'héritage multiple dans la programmation orientée objet a été abandonné au profit de l'utilisation d'interfaces.

Le langage C# permet une programmation orientée objet simple et moderne. Dérivé des langages de programmation C et C++, C# semblera familier aux habitués du développement en C, C++ et Java et ne sera sans aucun doute, pas plus difficile à apprendre pour les autres.

Effectivement, le langage C# ne comporte que 77 mots-clés. De plus, la programmation en C# se révèle intuitive, familière et moins verbeuse produisant du code se démarquant par sa lisibilité et sa concision.

Vous retrouvez dans l'annexe E une présentation détaillée du langage C#.

3.2 La grammaire

La grammaire C# est très complexe. La mise en œuvre d'une grammaire LALR pour ce langage demande beaucoup de temps et d'énergie. La grammaire actuelle ne permet pas de reconnaître toutes les possibilités du langage, il reste encore du travail de ce côté ci.

3.2.1 La spécification ECMA

L'ECMA est un organisme de normalisation international qui compte parmi ses membres les plus grands organismes de production, de commercialisation et de développement de systèmes informatique. Parmi les membres, on peut citer : Microsoft, Intel, Canon, HP, Novell, Apple, et Borland. Citons aussi pour les membres NFP (not-for-profit) : GNOME Foundation, Mozilla Foundation, et l'Object Management Group (la liste complète des membres : <http://www.ecma-international.org/memento/members.htm>).

Ces principaux thèmes de normalisation sont :

- Communications : CSTA (Computer Supported Telecommunication Applications), UWD

- (Ultra-wideband)
- Langages : C# et CLI;
- Média : disques optiques et disques à cartouches, Holographic Information Storage Systems (HISS);
- Environnement : Design for Environment (DfE);
- Sécurité : "ECMA-287 - Hazard Based Safety Engineering for ICT & CE products";
- Format des données : Universal 3D (U3D), le format Office Open XML, XML Paper Specification;

L'ECMA a ratifié les spécifications du langage C# (prononcer C-Sharp) sous la référence ECMA 334 [6]. Cette ratification marque une étape importante dans la normalisation et l'indépendance du langage C# vis à vis des fournisseurs.

Cette spécification permet d'autres implémentations du langage C#. On peut citer Mono et DotGru. Il est nécessaire de préciser que Microsoft ne considère pas ces projets comme une menace. Aujourd'hui, une équipe de 28 développeurs (dont 6 permanents) travaillent sur le projet Mono, chez Microsoft ils sont plusieurs centaines (source : [8]).

Microsoft a demandé la normalisation à ECMA, qui est rendu à la 4^{ème} édition (Juin 2006). La première édition est sortie en décembre 2001 suite à la demande de normalisation de Microsoft pour le langage C# sorti en juin 2000.

3.2.2 La grammaire

Pour pouvoir arriver à fournir une grammaire convenable, je suis tout d'abord parti de la grammaire officielle établie par ECMA Internationale. Celle ci est ambiguë.

J'ai donc travaillé à la rendre non ambiguë.

Le premier travail a été de simplifier la grammaire afin de prendre en compte les objets comme les classes, les namespaces ou les méthodes. Puis petit à petit, j'ai procédé à l'ajout successif des concepts du langage.

Le langage C# tire ses concepts des langages C/C++ et Java, la construction de morceaux est évidente et souvent habituelle.

Prenons l'exemple de déclaration d'une classe.

Soit le bout de code suivant :

```
public class Kid
{
    private int age;
    private string name;

    // Default constructor:
    public Kid()
    {
        name = "N/A";
    }

    // Constructor:
    public Kid(string name, int age)
```

```
{  
    this.name = name;  
    this.age = age;  
}  
}
```

Chaque classe C# est déclarée à l'aide du mot clé `class`. Ce dernier est précédé de la visibilité de classe et est suivi par son nom. La construction d'une classe C# est simple et étrangement ressemblante à celle d'une classe Java. Ainsi on peut modéliser la règle concernant les classes comme ceci :

```
Class_Declaration  
    : Modifiers_Opt 'class' Identifier '{' ... '}'  
    ;
```

A cette règle, il est nécessaire d'ajouter quelques informations, comme les attributs C# (attention à ne pas confondre avec les attributs de classes) qui s'appliquent sur les concepts C#. Il manque aussi les déclarations d'héritage (classe mère ou implémentation d'interfaces) ainsi que le corps de la classe.

C'est ainsi que l'on obtient cette règle :

```
Class_Declaration  
    : Attributes_Opt Modifiers_Opt 'class' Identifier  
      Type_Parameter_List_Opt  
      Class_Base_Opt '{' Class_Item_Declarations_Opt '}' Semicolon_Opt  
    ;
```

Pour comprendre cette règle, chaque mot du vocabulaire non terminal est expliqué ci-dessous.

- `Attributes_Opt` définit l'ensemble des attributs applicables à la classe.
- `Modifiers_Opt` liste les modifieurs de la classe ainsi que sa visibilité.
- `Identifier` donne le nom de la classe.
- `Type_Parameter_List_Opt` est un survivant de C++, il permet de paramétrer une classe (template).
- `Class_Base_Opt` liste les interfaces implémentées et la classe dérivée.
- `Class_Item_Declarations_Opt` donne le contenu de la classe (méthodes, attributs, ...).

La première approche était donc de décrire à l'aide d'une grammaire les différents objets du langage. La seconde est de décrire la composition d'un fichier source (unité de compilation). Les fichiers sources C# dont l'extension est `cs`, ont la particularité de décrire plusieurs namespaces et classes. A l'inverse de Java, qui oblige une déclaration de classe par fichier. Comme pour Java, le fichier est composé de deux parties, la première partie est l'ensemble des références aux classes extérieures, la seconde aux déclarations de classes.

C'est pourquoi on obtient cette règle :

```
Compilation_Unit  
    : Using_Directives_Opt Namespace_Item  
    ;
```

Un fichier source peut être aussi, un fichier vide. C'est à dire que le fichier ne contienne aucune déclaration, il pourra contenir des attributs C#.

```
Compilation_Unit
: Using_Directives_Opt Attributes_Opt
| Using_Directives_Opt Namespace_Item
;
```

Afin de pouvoir établir une grammaire LALR et la plus proche de celle utilisée par les compilateurs C#, je me suis basé sur le travail réalisé et sur des grammaires C# trouvées en ligne basées pour la plupart sur les spécifications ECMA-334. En particulier sur le travail d'une équipe qui développe des parseurs indépendants [9] ou de développeurs qui enrichissent la collection de grammaires pour l'outil ANTLR (outil de compilation java) [10] [11].

Le résultat de mon travail est une grammaire qui reconnaît l'ensemble des concepts C# à l'exception des blocs `unsafe` (l'utilisation de pointeurs en C#).

Cette grammaire possède 500 règles, 150 axiomes et 150 éléments du vocabulaire non terminal. Cette grammaire n'est pas exhaustive, la grammaire définit seulement la sémantique et la syntaxe d'une partie du langage C#.

Un petit exemple

Pour cet exemple :

```
using System;
class Hello
{
    static void Main() {
        Console.WriteLine("hello, world");
    }
}
```

Il est nécessaire d'utiliser une quarantaine de règles différentes :

1. reconnaissance du symbole 'using' ;
2. reconnaissance de l'identifiant 'System' ! ;
3. réduction par la règle `Identifieur` : `<Identifieur>` ;
4. réduction par la règle `Qualified_ID` : `Identifieur` ;
5. reconnaissance du symbole ';' ;
6. réduction par la règle `Using_Directive` : 'using' `Qualified_ID` ';' ;
7. réduction par la règle `Using_Directives` : `Using_Directive` ;
8. réduction par la règle `Using_Directives_Opt` : `Using_Directives` ;

...

79. reconnaissance du symbole '}' ;
80. réduction par la règle `Semicolon_Opt` : `ε` ;

81. réduction par la règle

```
Class_Decl : Attrib_Opt Mod_Opt 'class' Identifier  
Type_Parameter_List_Opt  
Class_Base_Opt '{' Class_Item_Decs_Opt '}' Semicolon_Opt ;
```

82. réduction par la règle `Type_Decl : Class_Decl ;`

83. réduction par la règle `Namespace_Item_Decl : Type_Decl ;`

84. réduction par la règle `Namespace_Item : Namespace_Item_Decl ;`

85. réduction par la règle `Compilation_Unit : Using_Directives_Opt Namespace_Item ;`

3.3 Le méta modèle C#

3.3.1 Le méta modèle ANT

Sodifrance a écrit son propre méta modèle pour décrire une application N-Tiers : ANT pour Architecture N-Tiers. L'apprentissage des concepts présents dans le méta modèle ANT fut une étape importante de mon stage. La maîtrise de ce méta modèle est en effet indispensable : le méta modèle C# est une extension du méta modèle ANT, ils ont donc beaucoup de concepts en commun. De plus, je dois assurer la transformation de l'un vers l'autre.

Le méta modèle ANT est divisé en 6 parties :

- **Core** : c'est le noyau ANT. Il définit la racine des éléments (ModelElement) et les packages (Package).
- **Extensibility** : définit les éléments d'extension des modèles : par exemple TaggedValue et Stereotype.
- **Object** : définit les types de base (String, Float, etc.), les types spéciaux (tableaux, énumérations, etc.), les classes, les Feature (attributs, paramètres, etc.).
- **GUI** : définit tout ce qui concerne l'IHM (fenêtres, widgets, éléments graphiques de base, contrôles orientés base de données, etc.).
- **Cinematic** : définit tout l'aspect cinématique d'une application N-Tiers (transitions, événements, points d'entrée et de sortie, etc.).
- **Semantics** : c'est la partie code, c'est-à-dire les instructions (statements, expressions).

Au total, ce sont plus de 300 classes qui composent ce méta modèle. Le schéma ci-dessous montre un extrait de la modélisation UML de la partie GUI du méta modèle ANT.

3.3.2 Les particularités du méta modèle C#

Le méta modèle C# doit être une extension du modèle ANT. Ce dernier est orienté Java, comme la comparaison C# - Java (Annexe F) indique des changements peu importants, le méta modèle ANT est proche de celui attendu.

A l'aide de l'étude sur le langage C# (Annexe E), nous avons constaté que des éléments manquaient dans le modèle ANT. J'ai donc enrichi le méta modèle ANT pour créer le méta modèle ANT-C#.

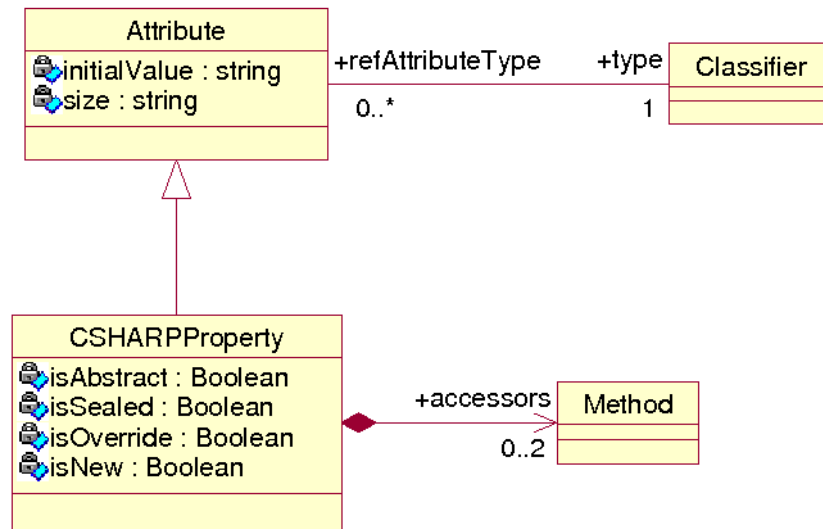


FIG. 3.1 – Extrait du méta modèle C#.

L'un des concepts qui n'est pas présent dans ANT est le concept de propriété. Les propriétés s'appliquent sur un attribut (au sens attribut de classe).

Par exemple, nous avons une classe `voiture` qui possède un attribut `tailleReservoir`

```

public class Voiture
{
    public int tailleReservoir;
}
    
```

Dans le but de manipuler cet attribut on crée une propriété. Ainsi deux méthodes sont possibles, le getter qui permet d'accéder à sa valeur et le setter de la modifier. Les deux ne sont pas obligatoires.

```

public class Voiture
{
    private int tailleReservoir;
    public int TailleReservoir {
        get { return tailleReservoir; }
        set { tailleReservoir = value; }
    }
}
    
```

La figure 3.1 montre un extrait du méta modèle C# avec en particulier la modélisation du concept de propriété présent en C#.

4 Rétro-ingénierie

4.1 Un peu de cours

4.1.1 Définitions

RÉTRO-INGÉNIERIE est l'activité qui consiste à étudier un objet pour en déterminer le fonctionnement interne.

Dans la vie des applications, la plus grande partie du temps est passé à maintenir et/ou à faire évoluer celles ci. Pour cela, il est nécessaire de connaître l'application, c'est à dire les équipes qui l'ont développé et qui l'ont documenté. Or la création, souvent rapide, ne permet pas toujours d'associer à l'application une documentation complète et une description détaillée. Cela pose un problème par exemple, lorsque les équipes viennent à changer.

C'est pour cela qu'il existe les méthodes de rétro-ingénierie. Elle permet de retrouver les composants structurants l'application. Associé à une rétro documentation, les nouvelles équipes peuvent faire évoluer ou maintenir l'application.

4.1.2 Travail

Dans le panel des outils Sodifrance, on trouve la suite ESSOR. ESSOR analyse automatiquement les différentes applications des systèmes d'information et met à disposition tous les outils nécessaires pour contrôler, évaluer et gérer les patrimoines.

La solution ESSOR offre :

- une vision globale et exhaustive du système d'information
- de puissantes capacités de cartographie et d'analyse.

L'intérêt d'utiliser un outil de cartographie et d'analyse est de résoudre tous les inconvénients des applications :

1. l'ancienneté des applications ;
2. une volumétrie importante ;
3. la diversité technologique ;
4. l'empilage des maintenances ;
5. le manque de documentation ;
6. le départ des sachants (équipes de développement, de conception, ...) ;
7. la difficulté à maintenir une base d'informations à jour.

Mon stage a aussi le but d'enrichir le référentiel de la suite ESSOR.

4.2 Les outils existants

4.2.1 Reflector for .NET

L'outil Reflector for .NET [12] crée par Lutz Roeder est un navigateur de classes et un décompilateur. En effet, il permet d'explorer, d'analyser et de consulter la documentation et le code des programmes .NET. Il travaille sur les assemblies C#, Visual Basic et MSIL.

L'outil possède une architecture à plugins. En effet, cet outil peut être surchargé de plugins. Pour développer un plugin pour Reflector, il suffit de créer un programme qui implémente l'interface IPackage. Cette interface possède deux méthodes à implémenter « load » et « unload ». Ces méthodes seront appelées lors du chargement du plugin (View | Add-Ins) ou du déchargement.

Les avantages de ce principe sont de permettre à chaque utilisateur différent d'ajouter les fonctionnalités désirées.

Reflector utilise CLI Header afin de récupérer les informations sur l'application.

Avec cet outil une application .NET peut être désassemblée. Le code d'un exécutable (exe ou dll) qui s'exécute dans la plateforme .NET peut être visible grâce à cet outil. On obtient aussi les dépendances avec les librairies et autres applications. La manipulation des données peut être effectuée de plusieurs manières soit le code est visible sous format texte standard soit l'application peut être vu de façon graphique ou interprétée.

La figure D.3 (page 53) montre le résultat de la décomposition d'une application .NET.

Certains plugins permettent d'afficher graphiquement les dépendances entre les classes et les namespaces. La figure D.2 (page 52) montre le résultat de l'un d'entre eux.

4.2.2 Autres applications

Il existe d'autres applications plus ou moins connues qui permettent de récupérer du code (soit intermédiaire soit haut niveau comme C#) à partir du code intermédiaire ou directement à partir de l'exécutable. Je n'ai testé aucune de ces applications (par manque de temps). La description que j'en fait, est celle trouvée sur les sites officiels et sur un site qui répertorie différents outils d'aide au développement .NET [13].

Si je devais en retenir que deux, c'est sûrement l'application de Microsoft et celle du projet Mono qui retiendrait mon attention, même s'il semble qu'elle ne permet que de trouver du code intermédiaire.

Nom	Entrée	Sortie	Commentaires
ILDASM	.EXE ou .DLL	IL	C'est l'utilitaire de Microsoft.[14] Il est fournis avec le SDK.
MONODIS	.EXE ou .DLL	IL	C'est l'utilitaire du projet Mono [15]
ANAKRINO ¹	MSIL	C#	Il s'applique seulement sur la plateforme .NET 1.1 [16]
SALAMANDER	.EXE ou .DLL	C#, VB.NET, ...	Convertis les applications en langage haut niveau [17]
ASMEX	.EXE ou .DLL	IL	[18]
.Net Explorer	.EXE ou .DLL	IL	[19]
SPICES.NET	.EXE ou .DLL	IL, C#, C++, VB.Net, J#, Delphi.Net	[20]
DIS#	.EXE ou .DLL	C#, VB.NET, Delphi.NET and Chrome	[21]

4.3 La création du modèle

4.3.1 Quelle approche

La rétro ingénierie consiste à retrouver les différentes caractéristiques d'un programme. Au sein de Sodifrance, il existe de nombreux outils permettant de travailler sur les sources de différents langages de programmation.

L'approche Sodifrance

La méthode d'approche pour ce cas est approximativement la même que pour les autres langages (approximativement puisque d'un langage à un autre, les mêmes concepts changent et leurs applications aussi, l'approche est alors adaptée à la situation).

Le but de cette approche est de créer à l'aide de l'outil crée par TGen un outil pour parser les codes sources d'application. Cette étape va permettre de créer une instance de notre méta modèle.

Cette instance permettra par la suite d'intégrer des applications de cartographie.

Nous possédons un méta modèle ainsi qu'une grammaire qui permet de valider de nombreux exemples de codes sources. Je vais appliquer à chaque règle de la grammaire une action quiinstanciera notre méta modèle.

Mon projet

Pour cette partie, mon approche consiste à instancier en premier les objets (autrement dit de créer des instances des objets haut niveau sans ses caractéristiques. Dans un second temps, j'enrichirai

¹Anakrino comme Reflector sont deux outils que Microsoft cite sur son site MSDN.

mon modèle en ajoutant à chaque objet ses caractéristiques.

Dans de grosses applications, le modèle obtenu est important, la manipulation de ce genre de modèle demande d'importantes ressources. Un moyen existe pour limiter le chargement en mémoire de l'ensemble de l'application. En effet, ce chargement peut se faire en deux étapes. La première consiste à charger tous les objets sans leurs caractéristiques. La seconde étape chargera en mémoire seulement (en plus du chargement de la première étape) le fichier source courant. Ainsi chaque référence pourra être résolu, et chaque fichier analysé.

Dans le projet, tel qu'il est actuellement, ces deux étapes sont confondues. Une erreur importante qui complique la résolution des références. Le problème de mémoire dans le cas des tests est accessoire car les applications manipulées ne contiennent pas plus de 100 fichiers sources chacune.

Nous choisissons de travailler en C++ dans le but d'étendre au maximum l'expérience multi-langages.

4.3.2 Le code

Les actions

Pour créer ce modèle, nous ajoutons aux règles de la grammaire des actions. C'est à dire, pour chaque règle qui le nécessite on crée une action. Ainsi la règle qui définit l'objet classe :

```
Class_Decl : Attrib_Opt Mod_Opt 'class' Identifier Type_Parameter_List_Opt  
           Class_Base_Opt '{' Class_Item_Decs_Opt '}' Semicolon_Opt
```

Cette dernière sera enrichi par l'action :

```
{ classDecl AttribOpt : ModOpt : ClassIdentifier : TypeParameterListOpt : ClassBaseOpt :  
  ClassItemDecsOpt : SemicolonOpt : }
```

Une action est une suite de mots clés représentant un élément du vocabulaire non terminal (c'est à dire l'appel à une autre règle de la grammaire). Chaque mot clé est suivi d'un "deux-points" (colon). L'ensemble des actions sont générés de façon automatique.

L'action sera traduite en une méthode C++, qui attendra du code C++, et renverra l'objet (ici, elle renvoie la classe créée).

Un exemple d'implémentation

Reprenons notre exemple. Lorsque cette règle est reconnue, c'est qu'une classe à été détectée. Dans notre modèle nous allons donc créer un objet C# `class`. Cet objet possède de nombreuses caractéristiques : son nom, son accessibilité, ses attributs C#, ses parents (classe ou interfaces), ses features. Toutes ses informations seront récupérées au moyen de variables. Ainsi le code de notre instantiation ressemble à ceci :

1. On crée un nouvel objet `class`

```
ClCSharpClass* pItem = new ClCSharpClass();
```

2. On y ajoute ses modifieurs

```
if ($2 != NULL) { // Mod_Opt
    list<TgenSemObject*> lst = ((TgenOrderedChildren*)$2)->asFlatList();
    pItem->SetVisibility(Mod_Visibility(lst));
    ...
}
```

3. On lui attribut son nom

```
pItem->SetName(((TgenSemString*)$3)->GetVal()); // Identifier
```

4. On lui ajoute la classe parente si elle existe

```
if ($5 != NULL) { // Class_Base_Opt
    list<TgenSemObject*> lst = ((TgenOrderedChildren*)$5)->asFlatList();
    while(!lst.empty()) {
        pItem->AddParentClassifier((ClCSharpClassifier*)lst.front());
        lst.pop_front();
    }
}
```

5. On lui ajoute ses fils (attributs et méthodes)

```
if ($6 != NULL) { // Class_Item_Decs_Opt
    list<TgenSemObject*> lst = ((TgenOrderedChildren*)$6)->asFlatList();
    while (!lst.empty()) {
        if (lst.front()->IsKindOf(RUNTIME_CLASS(ClCSharpFeature))) {
            pItem->AddFeatures((ClCSharpFeature*)lst.front());
        } else if (lst.front()->IsKindOf(RUNTIME_CLASS(ClCSharpClassifier))) {
            pItem->AddInnerClassifiers((ClCSharpClassifier*)lst.front());
        }
        lst.pop_front();
    }
}
```

6. Et on renvoie l'objet ainsi crée

```
return pItem;
```

L'instanciation

Cette partie est la plus grosse partie de mon travail, même si aujourd'hui elle n'est pas terminée. En effet, il est nécessaire de représenter toutes les informations contenues dans le code dans le méta modèle, de façon à retrouver tous les objets et liens entre les objets.

Pour mener à bien ce travail, on doit :

1. Créer chaque objet et ses caractéristiques
2. Liés les objets entre eux :
 - la notion de "owner". Par exemple, relié l'attribut de classe à sa classe (à laquelle il appartient).
 - les notions d'héritage. Par exemple, relié la classe à sa classe abstraite.

- les résolutions de types (variables locales ou attributs). Par exemple, relié le type d'une variable avec la classe définie dans le même namespace.

3. Les appels de variables et de méthodes.

Pour la création des objets, le travail n'est pas très compliqué, dans la plupart des cas le `new` est suffisant. Pour les liaisons entre objets, cela se complique.

Les liaisons parent - fils le travail est maché par la règle. Avec l'exemple de l'attribut, on aura un attribut présent dans la liste `Class_Item_Decs_Opt`. Un ajout dans les features de la classe suffira.

Pour les liaisons de "types", plusieurs cas sont possibles. Soit le type est déclaré dans le même namespace et avant cette référence, dans ce cas, la référence est très rapide. Soit le type est plus "éloigné", il est alors nécessaire de consulter l'ensemble des définitions de types (classes, structures, ...) disponibles.

Pour résoudre les appels, le travail est plus compliqué. Pour les variables locales, l'algorithme suivant permet de retrouver de façon propre et sûr la déclaration de la variable.

On suppose que nous sommes dans un ensemble de statements. Il existe deux types de statements. Le premier est un statement simple qui n'implique que lui même, comme l'appel d'une méthode. Le second est un statement qui possède un corps comme le `for`. Ce second type de statement est important car il implique des sous blocs. On inclut dans les statements de bloc les déclarations de paramètres ainsi que les déclarations d'itérateurs.

Soit notre exemple suivant :

```
une_methode(Un_Type un_premier_parametre , Un_autre_type un_second_parametre) {  
    un_statement_A ;  
    un_statement_B ;  
    for_statement (entier un_iterateur) {  
        un_statement_C ;  
        un_statement_D ;  
        un_appel_x ;  
    }  
}
```

Pour résoudre l'appel x, nous appliquons l'algorithme qui suit :

```
Tant que ( trouvé == faux ) faire  
    On regarde les statements précédents .  
    Si le bloc courant est un sous bloc alors on remonte au bloc supérieur  
    .  
    Si le bloc courant est le corps d'une méthode alors on recherche dans  
    les attributs de la classe .
```

Dans la pratique, cet algorithme n'est pas si simple. En effet, dans notre cas, nous avons un méta modèle qui ne permet pas la navigation des liens qui ne sont pas nommés. Ce bémol implique que dans certaines associations, nous pouvons naviguer que dans un seul sens. Nous ne pouvons pas accéder alors au bloc supérieur.

Deux possibilités viennent à nous : La première est de remettre les liens anonymes au sein de notre méta modèle. La seconde est de dire, si on ne peut pas naviguer de bas en haut, alors naviguons de haut en bas. Dans cette seconde solution, le cout d'exécution risque de s'alourdir. De



plus la navigation de haut en bas ne nous permet pas de détecter la variable correspondante. Ainsi si deux variables ont une portée sur le même bloc, il est difficile de savoir, laquelle est la variable appelée.

- Si nous revenons à notre exemple, pour résoudre x dans la première solution :
 - On regarde si les statements D ou C sont des déclarations de x.
 - Si x n'est pas déclaré, on regarde l'itérateur du for.
 - Si x n'est pas déclaré, on remonte et on regarde les statements B et A.
 - Si x n'est pas déclaré, on regarde les paramètres de la méthode.
 - Si x n'est pas déclaré, on regarde les attributs de la classe.
- Pour résoudre x dans la deuxième solution :
 - On regarde si la variable est déclaré dans la méthode,
 - S'il existe une variable de même nom, on vérifie que sa portée corresponde à l'appel de la variable.
 - Si non, on regarde les attributs de la classe.

5 Migration C# - Java

5.1 Un peu de cours

5.1.1 Définitions

Une TRANSFORMATION prend un modèle source en entrée et fournit un modèle cible en sortie.

Il existe deux transformations différentes :

- Transformation endogène
 - Dans le même espace technologique
 - Les modèles source et cible sont conformes au même méta-modèle
 - ex : Transformation d'un modèle UML en un autre modèle UML
- Transformation exogène
 - Entre 2 espaces technologique différents
 - Les modèles source et cible sont conformes à des méta-modèles différents
 - ex : Transformation d'un modèle UML en programme Java
 - ex : Transformation d'un fichier XML en schéma de BDD

La transformation permet à une application de changer de technologie. Les raisons de ce changement permet de faire évoluer les outils avec les nouvelles technologies.

5.1.2 Travail

Mon travail est de transformer des applications développées pour les plateformes .NET écrites en C# vers des applications pour les plateformes J2EE écrites en Java. Les raisons qui pourraient demander un tel changement sont :

- une évolution possible vers un format open source
- un changement d'équipe
- ...

Cette transformation doit traduire les applications C# en Java. A l'aide de l'outil MIA-Transformation une transformation est réalisée.

5.2 La comparaison C# - Java

5.2.1 La différence d'un point de vue philosophique

Le langage C# est un langage créé par Microsoft pour sa plateforme .NET. Cette dernière a pour but de concurrencer le langage Java et les environnements concurrents. En effet, .NET est conçu pour permettre à n'importe quel langage de s'exécuter sur la plateforme. Microsoft veut créer une plateforme ouverte à tous. Chaque informaticien a son langage préféré et Microsoft fait le pari de tous les regrouper. Pour cela, il fait certifier par ECMA son CLI (cf. Annexe C).

Le langage Java est développé par Sun. Il existe trois versions différentes de plateforme : J2ME, J2SE et J2EE. Sur ces dernières, un seul langage est exécutable : le langage Java. Mais les plateformes sont développées pour une multitude de systèmes d'exploitations afin que les programmes écrits en java soit exécuté partout.

Même si Microsoft s'est appuyé sur Java pour créer C#, la base de ce langage est très nettement influencé par le C++.

5.2.2 La comparaison

La comparaison complète entre les deux langages demande beaucoup de temps. Ici je vais simplement citer les concepts qui sont communs ou différents. Dans l'annexe F, vous trouverez une comparaison plus "pratique".

Si vous voulez en savoir plus sur la comparaison entre ces deux langages, le Web est rempli d'articles sur le sujet [22][23][24][25][26], mais je vous conseille de consulter la page de Dare Obasanjo [27].

Ci dessous, un résumé de tous les idées de cette comparaison.

① Les concepts et caractéristiques presque identiques en C# et en java.

1. tout est objet (en java : Object, et C# : object)
2. des mots clés proches (cf. Annexe G)
3. des machines virtuelles et des langages d'exécution pour un fonctionnement similaire
4. un Garbage Collector et gestion de la mémoire identique
5. les mêmes tableaux
6. il n'existe pas de méthodes globales
7. les interfaces, oui, l'héritage multiple, non.
8. les chaînes de caractères ne peuvent pas être modifiées

9. les classes non extensibles
10. levée et capture d'exceptions
11. initialisation de membres et constructeurs statiques
12. *boxing* et *unboxing*

② Les concepts et caractéristiques qui diffèrent seulement par la syntaxe.

1. la méthode *main*
2. la syntaxe d'héritage
3. l'identification de type (is ou instanceof)
4. les espaces de noms
5. les constructeurs, les destructeurs et les finaliseurs
6. la synchronisation de méthodes
7. les accesseurs et modifieurs



8. l'outil "Reflection"
9. la déclaration de constantes
10. les types primitifs
11. les déclarations de tableaux
12. l'appel du constructeur de base
13. la liste de paramètres au nombre aléatoire
14. les types génériques
15. la boucle *foreach*
16. les méta données
17. les énumérations
12. Définition de plusieurs classes dans un fichier pour C#
13. l'importation de bibliothèques
14. les événements
15. l'interopérabilité des langages

③ **Les concepts et caractéristiques qui existent en C# et Java mais implémentés différemment.**

1. les classes imbriquées
2. les membres volatiles et les threads
3. la surcharge d'opérateurs
4. l'instruction *switch*
5. les assemblies
6. les collections
7. l'instruction de saut *goto* (non implémenté en java)
8. les méthodes virtuelles
9. la gestion des fichiers (lecture/écriture)
10. la sérialisation d'objets
11. la génération de documentation à partir de commentaires.

④ **Les concepts et caractéristiques qui n'existent pas en C#.**

1. la libération déterministe d'objets
2. les délégués
3. les structures
4. l'identification de type dynamique
5. les propriétés
6. les tableaux à plusieurs dimensions
7. les indexeurs
8. les directives de pré processeur
9. les alias
10. la génération de code dynamique
11. les pointeurs et code *Unsafe*
12. le passage par référence
13. les caractères spéciaux
14. la détection d'overflow
15. l'implémentation explicite d'interface
16. les itérateurs
17. les types partiels
18. les classes statiques
19. les types nullable
20. les méthodes anonymes

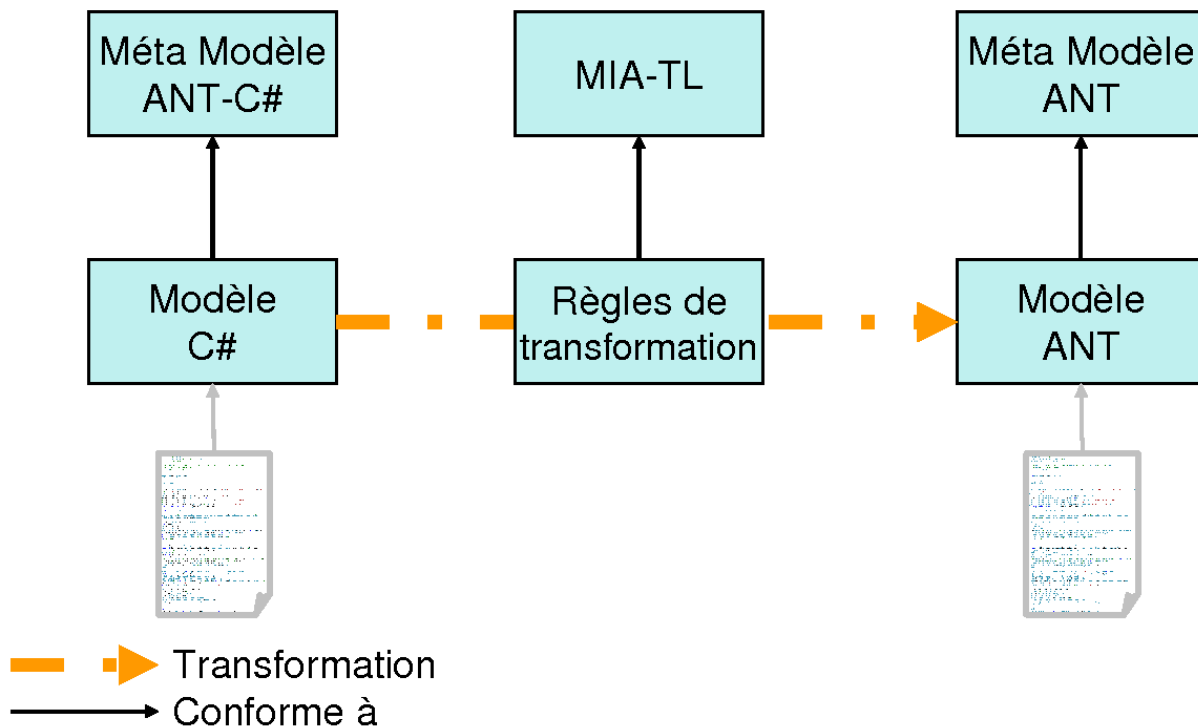


FIG. 5.1 – Schéma de la transformation.

5.3 Les règles de transformation

5.3.1 Processus de transformation

La transformation du langage C# vers Java demande avant tout une étude des différences entre ces deux langages. Le langage Java est très utilisé dans la formation universitaire, son étude a donc été très rapide. A l'inverse le langage C# était inconnu. Son étude a été très intéressante, mais je ne peux pas prétendre tout connaître. Il existe encore quelques points qui me sont encore inconnus.

Après l'étude des 2 langages, il fut nécessaire de comparer ces deux langages et de comprendre comment des concepts qui existe en C# mais pas en Java doivent être traduit. De plus, la comparaison permet aussi de confronter la concurrence de deux langages, leur points forts et leurs points faibles. Il est d'ailleurs possibles pour chaque internaute, à l'aide de notre ami Google [28] de consulter tous les débats "C# - Java".

La comparaison des deux langages n'est pas complète dans ce rapport. L'annexe F permet de comprendre rapidement ce qui différencie les deux langages.

La comparaison terminée, il est nécessaire de passer au codage de la transformation. Pour cela, on va diviser les règles de transformation en trois parties :

- les entités
- les attributs
- les liens

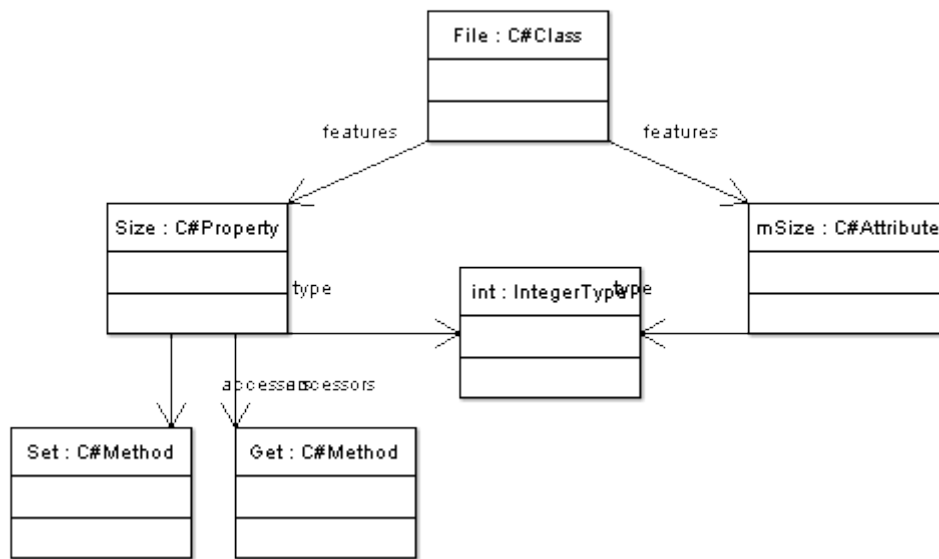


FIG. 5.2 – Schéma de l'exemple de modèle C#.

5.3.2 La transformation

Dans cette transformation de nombreux objets ne devront pas subir de changements. En effet, la transformation concerne deux méta modèles : ANT et ANT-C#. Le dernier méta modèle est une extension du méta modèle ANT c'est pourquoi de nombreux objets ne subiront pas de transformation.

Dans notre approche, nous commencerons à "cloner" tous les objets C# en objets ANT. Puis nous leur rajouterons leurs attributs, enfin nous recréerons tous les liens.

Bien sûr pour les objets qui diffèrent entre ANT et ANT-C# la méthode sera différente. Ainsi, des manipulations de l'arbre modèle et des changements dans les propriétés des objets seront effectués. De plus, de nouveaux objets seront créés.

L'exemple des propriétés

Pour illustrer la transformation, nous prenons ce petit exemple (le schéma obtenu avec l'analyseur, figure 5.2 - le corps des méthodes n'est pas représenté) :

C'est une classe appelée **File** qui possède un attribut **mSize** et une propriété.

```

public class File
{
    private int mSize ;
    public int Size
    {
        get { return mSize ; }
        set { if ( value < 0 ) mSize = 0 ; else mSize = value ; }
    }
}
    
```

La transformation de ce modèle :

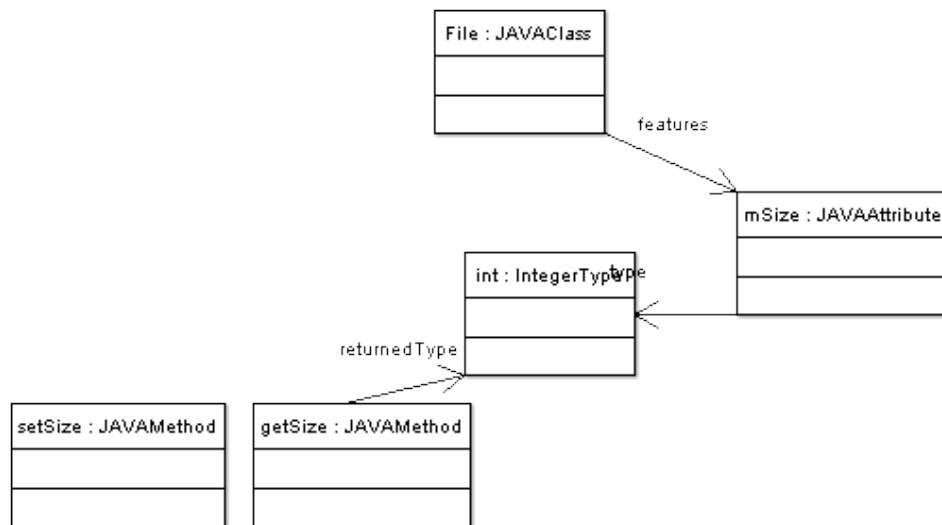


FIG. 5.3 – Schéma d'un extrait de l'exemple de modèle Java.

① Copie des éléments identiques. C'est à dire reproduction du modèle dans ce qui est commun au deux méta modèle, le résultat est obtenu en figure 5.3.

Le code qui permet de transformer une classe ANT-C# en ANT (simple recopie) possède deux variables *object* et *outObject*, reliés entre deux par un lien de transformation. Ce dernier sera utilisé dans les autres règles afin de retrouver les informations des deux modèles.

```

Local context
◆ object (in.Class)

Query [MIA-TL]
// selection de la classe
object = allOfClass(in,"Class");

Action [MIA-TL]
declare outObject;
// copie de la classe
outObject = createInstance(out,"Class");
addTransLink("objDest", object, outObject);
  
```

② Création de nouveaux éléments. C'est à dire, on transforme le modèle ANT-C# en ANT. Pour notre exemple, il s'agit de supprimer l'objet C# Proprety.

On va alors modifier les deux méthodes afin de les intégrer à notre nouveau modèle, le résultat obtenu est la figure 5.4.

Ainsi pour chaque objet spécifique à C#, une règle de transformation a été élaborée. Chacune de ces règles modifie l'arbre du modèle.

A la fin de la transformation, nous avons un modèle ANT (pseudo Java) permettant par la suite de générer le résultat. Le générateur Java utilisé pour ce stage est un générateur qui ne

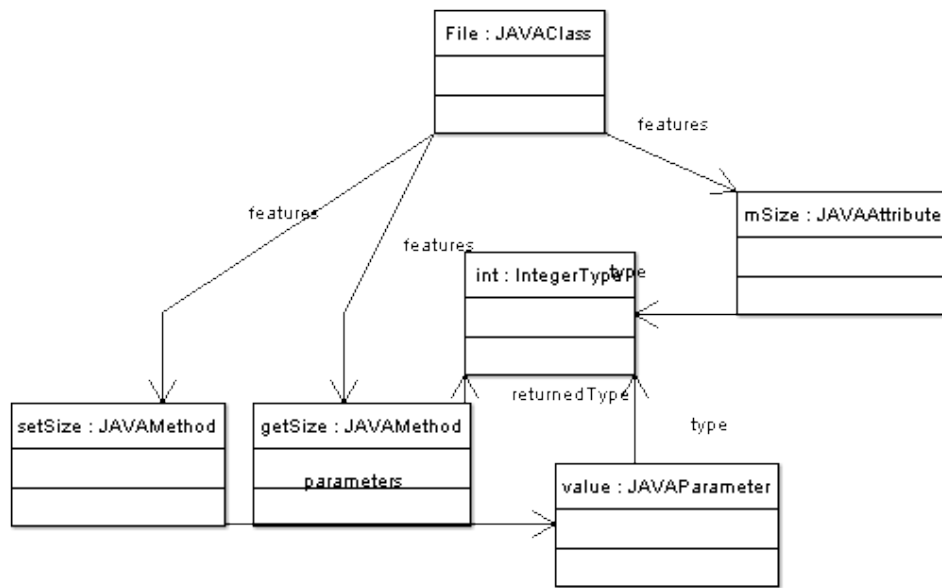


FIG. 5.4 – Schéma de l'exemple de modèle Java.

permet pas toutes la génération de tous les concepts Java 5. Pour une utilisation complète de cette transformation, un travail complémentaire est attendu pour la transformation ainsi que la génération.

Conclusion

Au cours de ce stage, j'ai beaucoup appris. Les apports que j'ai tiré de cette expérience professionnelle peuvent être regroupés autour de trois idées principales : les compétences acquises, les difficultés rencontrées et solutions apportées ainsi que la vie en société.

Les compétences acquises

Le stage m'a permis de prendre en considération tous les aspects d'un projet professionnel. C'est ainsi, que j'ai été informé des attentes des clients et de l'importance d'un travail réalisé avec méthodologie et précision. Même si un programme a toujours des bugs non identifié, il est important que l'application développées fonctionne pour de nombreux cas, et surtout pour tous les cas courant.

Les compétences acquises durant ce stage inclus l'utilisation d'outils. En effet, j'ai manipulé des outils spécifiques tel que T-gen et la suite MIA. J'ai pu aussi compléter ma formation et mon expérience au langage C++ à l'aide de l'outil Visual C++.

Ma connaissance des langages C# et Java s'est agrandie grâce à l'étude de ces langages et de leurs différences.

Les difficultés rencontrées et solutions apportées

Le stage ne possédait pas de difficultés particulière, mais comme tout projet des problèmes se sont posés.

La première des difficultés est de réussir à trouver une grammaire qui permet d'accepter le langage. Une grammaire tel que celle que décrit par ECMA[6] C# n'est pas LALR. Dans mon cas, la grammaire à utiliser ce devait d'être LALR, ce qui implique qu'un nombre de concept C# ne peut être accepté dans ma grammaire. En effet, il a fallu choisir et laisser des composants les moins utilisés.

La seconde des difficultés se trouve dans l'implémentation des règles ou plutôt dans le test de ces règles. A l'aide de plusieurs centaines de fichiers sources C# j'ai testé le parseur. Mais toutes les règles n'étaient pas appelées. C'est pourquoi, il a fallu identifier les règles qui n'étaient jamais utilisées. Suite à ce constat, je me suis aperçu que certaines règles ne seraient jamais appelées dans un fichier source bien formé.

La dernière difficulté porte sur l'ensemble du projet. Certaines parties n'ont pas pu être terminées dans les temps, il a fallu repousser les échéances. Après 5 mois de stage, le travail n'est pas complet, en effet le temps passé sur la grammaire et sur le parseur a été très important. Les autres parties ont du être légèrement réduite.



La vie en entreprise

Mon stage chez Sodifrance a été très instructif. Au cours de ces 5 mois, j'ai ainsi pu observer le fonctionnement d'une société de service en informatique (SSII et éditrice de logiciel). Les relations humaines entre les différents employés de la société, indépendamment de l'activité de chacun d'eux, m'a appris sur le comportement à avoir en toute circonstance.

Le stage

Cartographier un programme, c'est comprendre comment fonctionne celui-ci, créer l'outil qui va le permettre, c'est comprendre comment le langage fonctionne. Transformer une application d'un langage à un autre permet de suivre les nouvelles directives, mais étudier la transformation permet de comprendre ce qui différencie deux langages concurrents.

A titre de conclusion, il semble intéressant de mettre en évidence l'évolution grandissante du monde *automatique* qui nous entoure. A la préhistoire de l'informatique, ce domaine était limité à quelques machines très imposantes. Depuis les machines se sont multipliées et se sont miniaturisées, et font que chaque pas de l'homme est guidé par un ou plusieurs ordinateurs. Mon stage a permis de comprendre que l'avenir de l'informatique ne se joue plus avec une plateforme particulière, mais avec des changements fréquents de technologie. Comme les vêtements, l'informatique a ses modes. Aujourd'hui, la mode est au Java-J2EE, mais demain la mode se penchera peut-être sur .NET ou C++. Il en résulte un fort attrait pour les techniques de cartographie indépendantes du langage, de la plateforme pour permettre des évolutions rapides quelque soit la nouvelle mode.

Après le stage

Le regain d'intérêt des sociétés de services pour le recrutement d'ingénieurs informaticiens débutants nous permet de voir positivement notre entrée dans le marché du travail. Il est quand même nécessaire de surveiller ce phénomène, certains n'y croient pas et pensent plutôt à des effets d'annonces.

Bibliographie

- [1] Microsoft Corporation. Microsoft visual c++, 1998. <http://msdn2.microsoft.com/fr-fr/visualc/>.
- [2] Justin O. Graver. T-gen, 1994. <http://st-www.cs.uiuc.edu/users/droberts/tgen2.2.1/tgen.html>.
- [3] MIA Software. Mia studio, 2007. <http://www.mia-software.com>.
- [4] Paul B Mann. A translational bnf grammar notation (tbnf). *SIGPLAN Not.*, 41(4) :16–23, 2006.
- [5] OMG. Mof qvt final adopted specification. Specifications 05-11-01, OMG, November 2005.
- [6] ECMA International. *C# Language Specification*, volume ECMA 334. ECMA International, Rue du Rhône 114 - CH 1204 - Geneva, 4th edition, June 2006. <http://www.ecma-international.org/>.
- [7] Miguel de Icaza. Mono. <http://www.mono-project.com>.
- [8] Eliane Consola. La plate-forme de développement mono. *Journal du Net*, July 2007. <http://www.journaldunet.com/developpeur/technos-net/07/csharp/0724-expliquez-moi-mono.shtml>.
- [9] Devin Cook. C# grammar, September 2006. <http://www.devincook.com/GOLDParser/grammars/>.
- [10] K.Rajendra Kumar. C# grammar file, October 2003. <http://wwwantlr.org/grammar/list/>.
- [11] Kunle Odutola. kcsparse - an ecma-334 c# grammar sample for antlr v2.7.6, December 2005. <http://wwwantlr.org/grammar/list/>.
- [12] Lutz Roeder. Reflector for .net, 2007. <http://www.aisto.com/roeder/dotnet/>.
- [13] Fabrice MARGUERIE. Madgeek Sharp Toolbox, 2007. <http://sharptoolbox.com/>.
- [14] Microsoft. ILDasm.
- [15] Paolo Molaro Miguel de Icaza and Dietmar Maurer. Monodis.
- [16] Saurik. Anakrino, 2004. <http://www.saurik.com/net/exemplar/>.
- [17] Dr. Huihong Luo. Salamander, 2006. <http://www.remotesoft.com/salamander/>.
- [18] Benjamin Peterson. Asmex - .NET Assembly Examiner, 2003. <http://www.jbrowse.com/products/asmex/>.
- [19] Remotesoft. Remotesoft .NET Explorer, 2004. <http://www.remotesoft.com/dotexplorer/>.
- [20] 9rays. Spices.Net Suite. <http://www.9rays.net/Products/Spices.Net/>.
- [21] NETdecompiler.com. Dis#, 2006. <http://www.netdecompiler.com/>.
- [22] Wikipédia. Comparison of c sharp and java. *Wikipédia*, 2007. http://en.wikipedia.org/wiki/Comparison_of_C_Sharp_and_Java.
- [23] Anand Narayanaswamy. Java and c-sharp compared. *C# Help*. <http://www.csharpshelp.com/archives/archive96.html>.



- [24] Raffi Krikorian. Contrasting c# and java syntax. *O'Reilly WindowsDevCenter.com*, 2001. http://www.ondotnet.com/pub/a/dotnet/2001/06/14/csharp_4_java.html.
- [25] Frank McCown. Java (j2se 5.0) and c# comparison. *Harding.edu*, 2006. http://www.harding.edu/fmccown/java1_5_csharp_comparison.html.
- [26] Jae-Hoon CHOI. Converting c# to java. *kaistizen.net*, 2006. http://kaistizen.net/Project/CSharpJava/csharp_java.htm.
- [27] Dare Obasanjo. A comparison of microsoft's c# programming language to sun microsystems' java programming language. *25hoursaday.com*, 2007. <http://www.25hoursaday.com/>.
- [28] Larry Page et Sergey Brin. Google. <http://www.google.fr>.
- [29] Microsoft. *Programmation en C#*. Microsoft Corporation, 2002. Microsoft Official Course 2132A.
- [30] Jean-François Bobier. Microsoft .net : Architecture et services. Mémoire, École Nationale Supérieure des Télécommunications, July 2001.
- [31] Craig Utley. .net : dix arguments pour se décider. *JDN Développeurs*, January 2003. <http://www.zdnet.fr/actualites/informatique/0,39040745,2127705,00.htm>.
- [32] WikiPedia, a web-based, free-content encyclopedia. <http://www.wikipedia.org>.
- [33] Steve Lewis and Wilhelm Fitzpatrick. A java programmer looks at c# delegates. *OnJava.com*, 2003. www.onjava.com/pub/a/onjava/2003/05/21/delegates.html.
- [34] OMG. Unified modeling language : Superstructure. Specifications 07-02-05, OMG, February 2007.
- [35] OMG. Unified modeling language (uml) specification : Infrastructure. Specifications 04-10-14, OMG, October 2004.
- [36] ECMA International. *Common Language Infrastructure (CLI)*, volume ECMA 335. ECMA International, Rue du Rhône 114 - CH 1204 - Geneva, 4th edition, June 2006. <http://www.ecma-international.org/>.
- [37] Leslie Lamport. L^AT_EX. <http://www.latex-project.org>.



Index

- Symboles -

.Net 7

- A -

ADO.NET 46
ANTLR 11, 46
ASP.NET 46
ATL 12, 46
Attribut C# 21, 49, 108

- B -

BNF 46

- C -

C# 54
CLR 18, 46

- D -

DSL 13, 46

- E -

ECMA 19, 20, 46
EPL 12, 47
ESSOR 25
Essor 7

- G -

GLR 12, 47
GNU 11, 47
GNU GPL 47
Grammaire 11, 18, 22

- H -

HJ 9, 47

- I -

Implémentation des règles 9

- J -

J2EE 19, 47

- L -

LALR 12, 19, 39, 47
LINA 12
LIP6 12
LL 11, 12, 47
LR 11, 47

- M -

Méta modélisation et grammaire C# 9
MDA 4, 47
MIA 8, 13, 47
MIA DSL 13
MIA Génération 12, 13
MIA Studio 13
MIA Transformation 13
MIATL 12, 47
MSIL 19, 47, 114

- N -

NFP 48

- O -

OMG 12, 48

- P -

Planning 9
Prise en main des outils 8

- Q -

QVT 12, 48

- R -

Rapport 9
RL-TL 12



- S -

SDK	48
SLR	12
Sodifrance	4-7
Solution de migration C# vers Java	9
Soutenance	9
SSII	48

- T -

T-gen	11
Transformation	12

- U -

UML	48
-----------	----



Table des figures

1.1	Répartition du CA 2006 par métier et par secteur d'activité.	4
1.2	Répartition des effectifs 2006 en France et en Belgique.	6
1.3	Schéma synthétique des différents composants et leurs échanges de données.	7
1.4	Découpage du temps de travail.	8
1.5	Durée établie pour chacune des tâches.	9
1.6	Répartition des tâches en jours par semaine.	10
2.1	TGEN : l'outil.	14
2.2	MIA-Transformation : les règles de transformation.	15
2.3	MIA-Transformation : la lecture de modèles.	16
3.1	Extrait du méta modèle C#.	24
5.1	Schéma de la transformation.	35
5.2	Schéma de l'exemple de modèle C#.	36
5.3	Schéma d'un extrait de l'exemple de modèle Java.	37
5.4	Schéma de l'exemple de modèle Java.	38
C.1	Vue d'ensemble du CLI.	50
C.2	Composition du CLI.	51
D.1	Une application .NET (écrite en C#).	52
D.2	Modélisation des composants d'une application .NET à l'aide de l'outil Reflector for .NET et un plugin (Peli's Assembly Graph).	52
D.3	Une application .NET (écrite en C#) désassemblée par l'outil Reflector for .NET.	53
D.4	Métrique sur le code des méthodes d'une application .NET à l'aide de l'outil Reflector for .NET et un plugin (CodeMetrics).	53

A Abréviations

ADO (ActiveX Data Object), Technologie Microsoft fournissant une interface d'accès aux données dans l'environnement Windows. Elle permet aux programmes clients d'accéder aux données, et de les manipuler, dans un fichier ou un serveur de base de données.

ADO.NET est l'évolution XML du modèle d'accès aux données ADO. Il est construit à partir de et pour des applications à faible couplage. ADO.NET garantit l'indépendance des développements par rapport aux systèmes de gestion de données

ANTLR (ANother Tool for Language Recognition) a été créé par Terence Parr à l'Université de San Francisco. C'est un framework de construction de compilateurs, traducteurs basé sur une description grammaticale contenant des actions Java, C#, C++, ou Python.

ASP.NET est un ensemble de composants et une infrastructure simplifiant le développement d'applications Web et de services Web XML. Le programmeur peut développer dans n'importe quel langage des éléments de site Web qui seront compilés sur le serveur. Le résultat est accessible à partir de n'importe quel terminal.

ATL (ATLAS Transformation Language) est un langage de transformation de modèles implémentant le standard QVT de l'OMG.

ATLAS est un projet de l'INRIA créé en janvier 2004 entre l'INRIA-Rennes et l'Université de Nantes pour travailler sur la gestion des données complexes dans les systèmes distribués.

BNF (Forme de Backus-Naur) est une notation permettant de décrire les règles syntaxiques des langages de programmation. C'est donc un métalangage.

CA est le chiffre d'affaire d'une entreprise sur un exercice comptable.

CIL (Common Intermediate Language) est le langage de programmation le plus bas niveau lisible par un humain dans le CLI.

CLI (Common Language Infrastructure) est une spécification ouverte développée par Microsoft qui décrit l'environnement d'exécution formant le noyau du Framework .NET de Microsoft. La spécification définit un environnement qui permet d'utiliser de nombreux langages de haut niveau sur différentes plateformes sans nécessité de réécrire le code pour des architectures spécifiques (cf. Annexe C).

CLR (Common Language Runtime) est le moteur de compilation et d'exécution des langages. Il permet ainsi l'intégration complète des composants et des services Web XML, quel que soit le langage de programmation avec lequel ils ont été créés. Actuellement, il est possible de créer des applications .NET dans plus de 20 langages différents, parmi lesquels les langages C++, Microsoft® Visual Basic .NET, JScript®, J# sans oublier C#. Nombre de fournisseurs, grâce à CLI ont pu rendre disponible leurs langages pour le CLR : COBOL, Eiffel, Perl, Python, Smalltalk et d'autres encore.

DSL (Domain Specific Languages) est un langage dédié qui peut être défini par une grammaire ou par un métamodèle. Un langage dédié est créé pour résoudre certains problèmes spécifiques dans un domaine particulier, et n'a en principe pas vocation à résoudre des problèmes en dehors de ce contexte.



- ECMA** International - European association for standardizing information and communication systems, connu jusqu'en 1994 comme ECMA - European Computer Manufacturers Association, est une organisation de standardisation active dans le domaine de l'informatique.
- EPL** (Eclipse Public License) est une licence libre utilisée par le logiciel Eclipse.
- GLR** (Generalized LR) est une analyse LR généralisée.
- GNU** (GNU's Not Unix) est un système d'exploitation composé exclusivement de logiciels libres.
- GNU GPL** (General Public License) est une licence qui fixe les conditions légales de distribution des logiciels libres du projet GNU.
- HJ** est une unité "homme/jour" représente la quantité de travail fourni par une personne une journée.
- IDE** (Integrated Development Environment) est un programme regroupant un éditeur de texte, un compilateur, des outils automatiques de fabrication, et souvent un débogueur.
- INRIA** (Institut National de Recherche en Informatique et en Automatique) est un organisme public civil de recherche français créé le 3 janvier 1967 suite au lancement du Plan Calcul.
- ISIN** (International Securities Identification Number) est un code identifiant internationalement une valeur mobilière. Il est utilisé pour identifier les actions, les obligations, les bons, les warrants et les trackers.
- J#** est un dérivé de Java créé par Microsoft et permettant de créer des applications pour l'environnement d'exécution .Net.
- J2EE** (Java 2 Enterprise Edition) est le framework spécialisé aux applications d'entreprise ajoutant une série de bibliothèques plus professionnelles.
- J2ME** (Java 2 Micro Edition) ou Java ME est le framework Java spécialisé dans les applications mobiles. Des plate-formes Java compatibles avec J2ME sont embarquées dans de nombreux téléphones portables et PDA.
- J2SE** (Java 2 Standard Edition) est le framework de base pour le développement rapide d'applications simples. Il est généralement le premier framework pour un développeur débutant en Java.
- LALR** (Look-Ahead Left Recursive) permet d'améliorer la selectivité du parser LR.
- LINA** (Laboratoire d'Informatique Nantes Atlantique) est le laboratoire d'informatique de l'Université de Nantes.
- LIP6** (Laboratoire d'Informatique de Paris 6) est un laboratoire de recherche sous tutelle de l'Université Pierre et Marie Curie et du CNRS.
- LL** le terme "LL(1)" signifie que l'on parcourt l'entrée de gauche à droite (L pour Left-to-right scanning), que l'on utilise les dérivations gauches (L pour Left-most derivation), et qu'un seul symbole de pré-vision est nécessaire à chaque étape nécessitant la prise d'une décision d'action d'analyse (1).
- LR** les grammaires LR (Left-to-right parse, Right-most derivation) sont plus générales : une grammaire LR(1) est aussi une grammaire LL(1) donc la classe des langages reconnus par une grammaire LR(1) contient celle des langages LL(1).
- MDA** (Model Driven Architecture) ou architecture dirigée par les modèles est une démarche de réalisation de logiciel, proposée et soutenue par l'OMG. C'est une variante particulière de l'ingénierie dirigée par les modèles.



MIA est l'abréviation de "Model In Action".

MIA-TL (Model In Action Transformation Language) est le langage de transformation de la suite Mia-Studio.

MSIL est un sigle qui signifie dans le domaine informatique MicroSoft Intermediate Language

NFP (not for profit) est l'abréviation anglaise de OSBL (Organisme Sans But Lucratif).

OMG (Object Management Group) est une association américaine à but non-lucratif créée en 1989 dont l'objectif est de standardiser et promouvoir le modèle objet sous toutes ses formes.

OSBL est l'abréviation d'"organisme à but non lucratif".

R&D les activités de recherche et développement servent la stratégie de développement de l'organisation.

SA (Société Anonyme) est une forme juridique de société commerciale.

SDK (Software Development Kit) est un ensemble d'outils permettant de développer des applications.

SSII (Société de Services en Ingénierie Informatique) est une société de services spécialisée en informatique.

QVT (Query/View/Transformation) est un standard défini par l'OMG. Il s'agit d'un langage standardisé pour exprimer ces transformations de modèles.

UML (Unified Modeling Language) est un langage de modélisation des données et des traitements. C'est une formalisation très aboutie et non-propriétaire de la modélisation objet utilisée en génie logiciel.

VB.NET (Visual Basic .NET) est un langage de programmation objet issu du Visual Basic classique mais permet des développements pour l'environnement d'exécution .Net.

B Glossaire

ARGUMENT est une liste d'expressions séparées par des virgules dans l'appel d'une méthode ou d'un constructeur ou lors de l'accès dans un élément à l'aide de crochet.

APPLICATION WEB (parfois appelée "Web App") est une application livrée aux utilisateurs à partir d'un serveur Web par un réseau tel que l'Internet ou l'Intranet.

ATTRIBUT est une entité qui définit les propriétés d'objets, d'éléments, ou de fichiers. Un attribut est habituellement composé d'un identificateur (ou nom ou clé) et d'une valeur.

ATTRIBUT C# permet d'injecter de manière déclarative des méta données dans des types tels que les assemblies, les classes, les interfaces, les méthodes, etc.

GARBAGE COLLECTOR (en anglais garbage collector) est un sous-système informatique de gestion automatique de la mémoire. Il est responsable du recyclage de la mémoire préalablement allouée puis inutilisée.

LEGACY MODERNIZATION est la modernisation du patrimoine applicatif des entreprises. Dans une majorité des cas, elle est effectuée par une transformation industrielle des systèmes d'information.

MACHINE VIRTUELLE désigne un logiciel ou interpréteur qui isole l'application utilisée par l'utilisateur des spécificités de l'ordinateur, c'est-à-dire de celles de son architecture ou de son système d'exploitation.

P-CODE est un raccourci de pseudo-code.

REVERSE ENGINEERING (en français Rétro-ingénierie ou Rétro-conception), est l'activité qui consiste à étudier un objet pour en déterminer le fonctionnement interne.

SERVICE WEB est un programme informatique permettant la communication et l'échange de données entre applications et systèmes hétérogènes dans des environnements distribués. Il s'agit donc d'un ensemble de fonctionnalités exposées sur Internet ou sur un Intranet, par et pour des applications ou machines, sans intervention humaine, et en temps réel.

WINDOWS FORMS Windows Forms, framework .Net, permet, quelque soit le langage de programmation choisi, de créer des applications Windows.

C Fonctionnement du Common Language Infrastructure

La Common Language Infrastructure (CLI) est une spécification ouverte développée par Microsoft. Elle décrit le code exécutable et l'environnement d'exécution qui forment le cœur du framework .NET. La spécification définit un environnement qui autorise de multiples langages haut niveau pour être utilisés sur différentes plateformes sans être réécrit pour les architectures spécifiques.

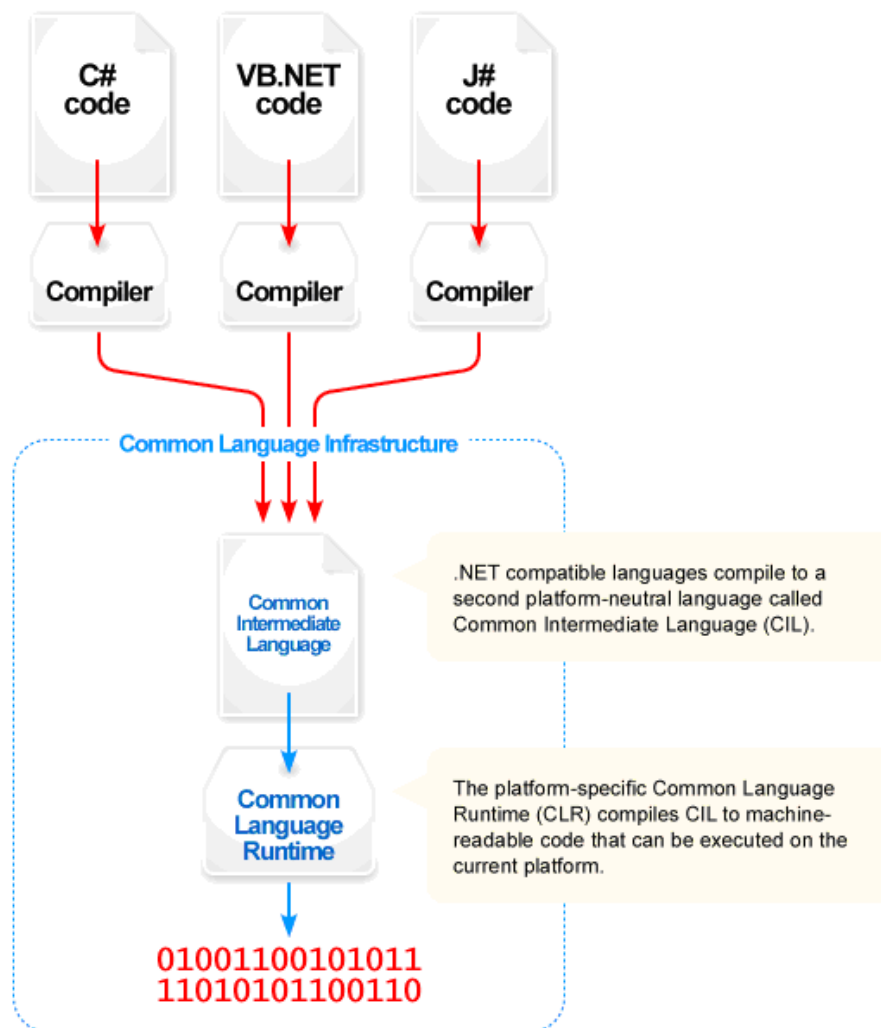


FIG. C.1 – Vue d'ensemble du CLI.

La CLI est une spécification, pas une implémentation, et est souvent confondus avec le Common Language Runtime (CLR). CLR inclut la CLI et fournis l'environnement d'exécution pour les applications .NET.

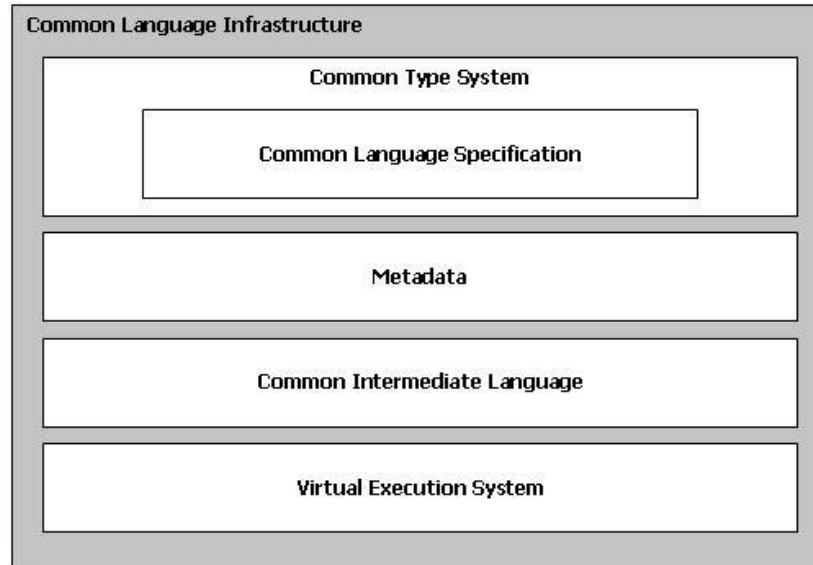


FIG. C.2 – Composition du CLI.

Cette spécification a été ratifié par ECMA sous la référence ECMA-335 [36].

On trouve dans la spécification CLI :

- Le **Common Type System** (CTS) : c'est une spécification déterminant la façon dont le Common Language Runtime définit, utilise et gère les types.
- La **Common Language Specification** (CLS) : c'est un sous-ensemble de fonctionnalités de langage prises en charge par le Common Language Runtime, comprenant des fonctionnalités communes à plusieurs langages de programmation orientés objet. Les composants et outils conformes CLS sont ainsi assurés de fonctionner avec les autres composants et outils conformes CLS.
- **Metadata** : ce sont les informations décrivant chaque élément managé par le Common Language Runtime : assembly, fichier chargeable, type, méthode, etc. Elles peuvent inclure des informations requises pour le débogage et l'opération garbage collection, de même que des attributs de sécurité, des données de marshaling, des définitions de membres et de classes étendues, la liaison de versions et d'autres informations demandées par le runtime.
- Le **Common Intermediate Language** (CIL) : c'est le langage de programmation le plus bas niveau lisible par un humain dans le CLI.
- Le **Virtual Execution System** (VES) : C'est la machine d'exécution.

D Un exemple d'utilisation de Reflector for .NET



FIG. D.1 – Une application .NET (écrite en C#).

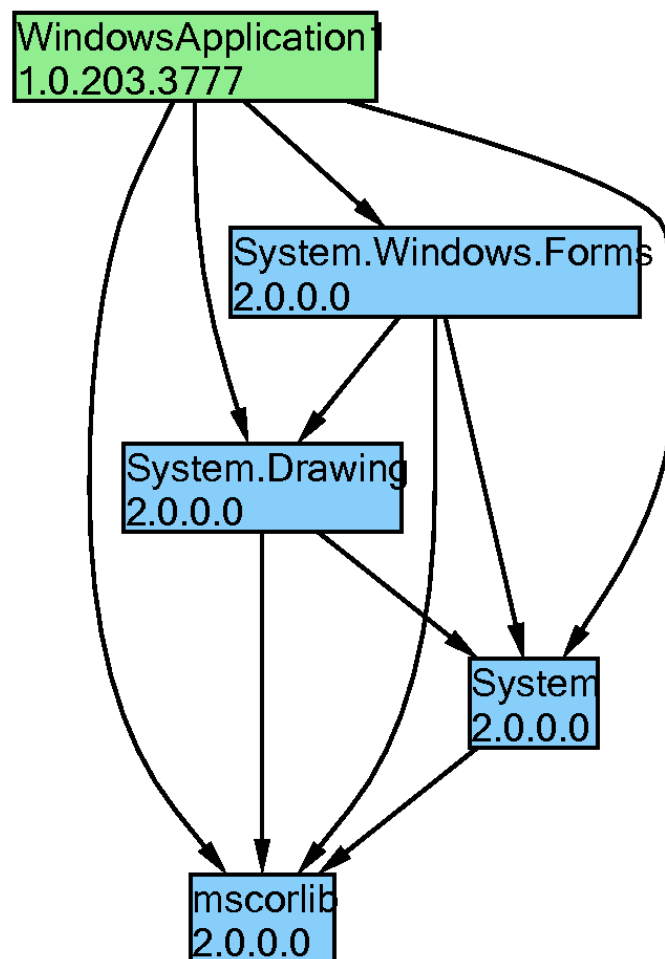


FIG. D.2 – Modélisation des composants d'une application .NET à l'aide de l'outil Reflector for .NET et un plugin (Peli's Assembly Graph).

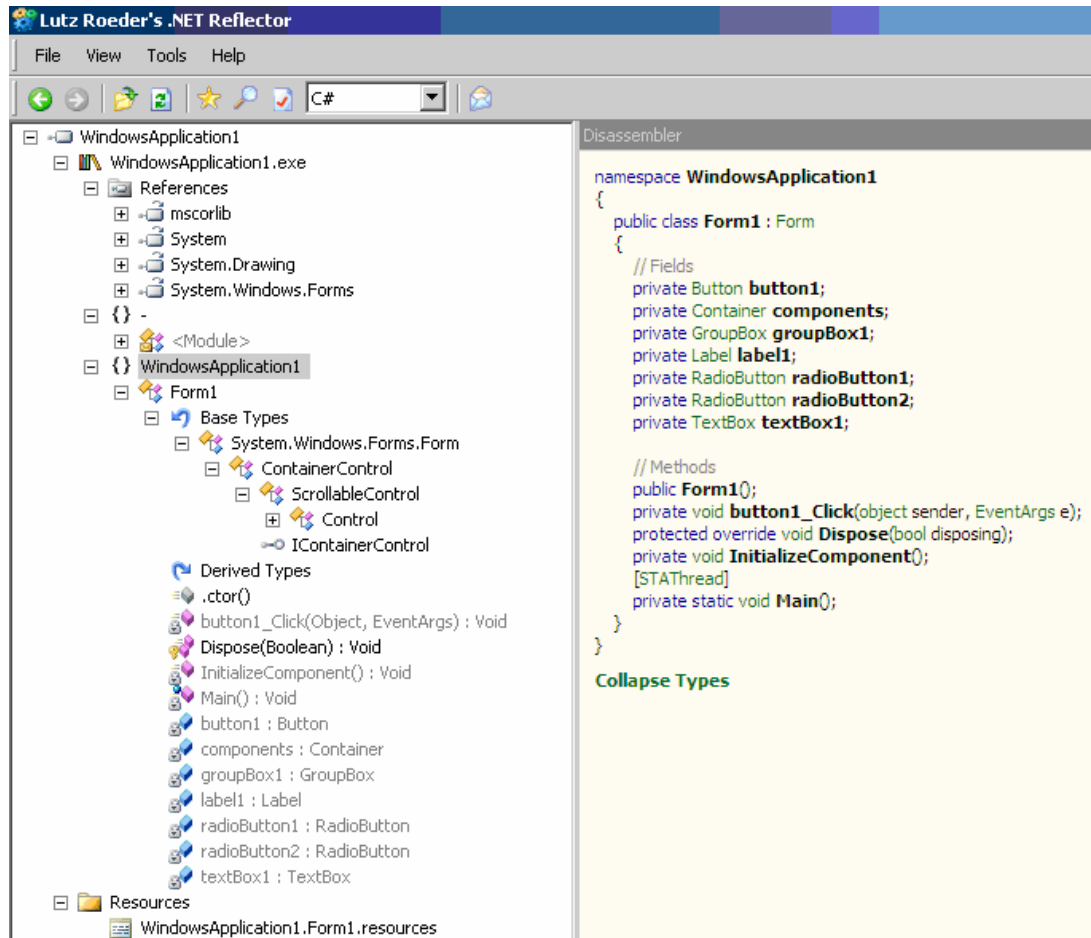


FIG. D.3 – Une application .NET (écrite en C#) désassemblée par l'outil Reflector for .NET.

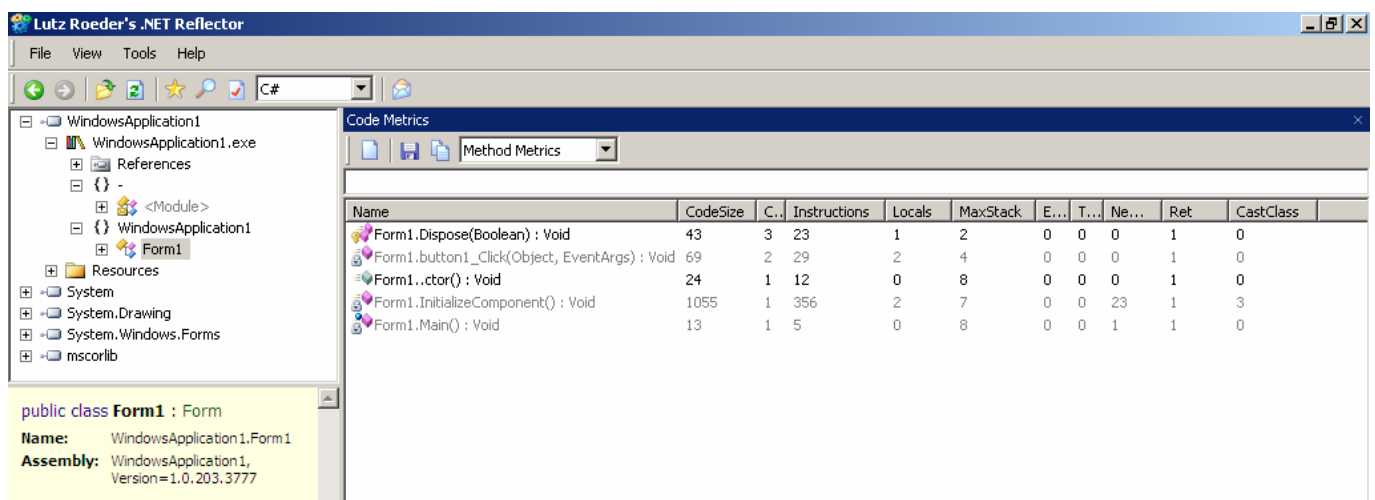


FIG. D.4 – Métrique sur le code des méthodes d'une application .NET à l'aide de l'outil Reflector for .NET et un plugin (CodeMetrics).

E Étude du langage C#

Cette annexe réalise un petit tour rapide du langage C# , quelques exemples montreront les grands principes du langage.

Pour commencer : Hello World

Pour commencer étudions le programme [Hello World](#) :

```
using System;  
class Hello  
{  
    static void Main() {  
        Console.WriteLine("hello, world");  
    }  
}
```

Le code source C# est placé dans des fichiers dont l'extension est `.cs` (pour C Sharp), c'est pourquoi le programme précédent est stocké dans le fichier `hello.cs`. Le nom du fichier est arbitraire, aucune obligation de nommage.

La ligne de commande pour compiler les sources C# est `csc`. Il est également possible d'utiliser un environnement de développement qui permettra la compilation des fichiers C#.

```
csc hello.cs
```

La compilation va produire un exécutable `hello.exe`. A l'exécution le terminal affichera ceci :

```
hello , world
```

Quelques explications sur les différentes parties du programme :

- La première ligne `using System;` fait référence à un namespace appelé `System` appartenant à la librairie du framework .NET. Ce namespace contient une classe nommée `Console`. Le programme [Hello World](#) utilise `Console.WriteLine` comme raccourci de `System.Console.WriteLine`.
- La méthode `Main` appartient à la classe `Hello`. Elle est statique.
- Le premier élément exécuté par une application est toujours une méthode statique nommée `Main`.
- L'affichage `hello, world` est effectué par l'utilisation d'une librairie.

Pour les développeurs C et C++, il est intéressant de noter quelques informations qui n'apparaissent pas dans le programme `hello, world` :

- Le programme n'utilise pas une méthode globale pour le `Main`. Les méthodes et les variables ne sont pas supportés à un haut niveau.
- Le programme n'utilise pas les symboles `::` ou `->`. Le symbole `::` n'est pas un opérateur du tout, et l'opérateur `->` est utilisé seulement dans une petite partie du programme

(utilisé dans le code `unsafe`). Le séparateur `.` est utilisé dans les noms composés comme `Console.WriteLine`.

- Le programme ne contient pas de déclarations avancées. Elles ne sont pas nécessaires, tout comme l'ordre des déclarations n'a pas d'importance.
- Le programme n'utilise pas `#include` pour importer du texte. Les dépendances entre programmes sont plus symboliques que textuelles. Cette approche élimine les barrières entre les applications écrites dans différents langages. Par exemple, la classe `Console` n'a pas besoin d'être écrite en C#.

Les Types

C# supporte 2 sortes de types : les types de valeur et les types de référence.

Les types de valeur incluent :

- les types simples (comme : `char`, `int`, et `float`),
- les types énumérés,
- et les types structurés.

Les types de référence sont

- les types utilisateurs (les classes, les interfaces, les délégués, et les tableaux).

Les types de valeur diffèrent des types références dans l'information qu'ils contiennent. Les types de valeur stockent la valeur de l'objet alors que les types de référence stockent seulement une référence sur l'objet. Avec les types de référence, il est possible pour 2 variables de référencer le même objet, il est possible pour une opération sur une variable d'affecter l'objet référence par une autre variable. Avec les types de valeur, ce n'est pas possible d'affecter une autre variable pour les opérations.

```
using System;
class Class1
{
    public int Value = 0;
}
class Test
{
    static void Main() {
        int val1 = 0;
        int val2 = val1;
        val2 = 123;
        Class1 ref1 = new Class1();
        Class1 ref2 = ref1;
        ref2.Value = 123;
        Console.WriteLine("Values: {0}, {1}", val1, val2);
        Console.WriteLine("Refs: {0}, {1}",
            ref1.Value, ref2.Value);
        Console.WriteLine("Refs: {0}, {1}",
            ref1.Value, ref2.Value);
    }
}
```



Cet exemple montre ces différences. La sortie produit ceci :

```
Values: 0, 123  
Refs: 123, 123
```

L'affectation de la variable locale `val1` n'agit pas sur la variable locale `val2`. En effet, les deux variables locales sont des types de valeur (entier) et chaque variable locale a son propre stockage. A l'inverse, l'affectation `ref2.Value = 123;` modifie les références `ref1` et `ref2`.

```
Console.WriteLine("Values: {0}, {1}", val1, val2);  
Console.WriteLine("Refs: {0}, {1}", ref1.Value, ref2.Value);
```

Les lignes précédentes montrent la formation des chaînes de caractères pour la méthode `Console.WriteLine`. Elle prend en fait un nombre variable d'arguments. Le premier argument est la chaîne de caractères qui peut contenir des pseudos paramètres comme `{0}`. A chaque pseudo paramètre, c'est un argument supplémentaire nécessaire à la méthode. Ainsi `{0}` fait référence au deuxième argument et `{1}` au troisième.

Les développeurs peuvent définir de nouveaux types de valeur à l'aide des types énumérés et les structures. Ils peuvent définir de nouvelles références avec les déclarations de classes, d'interfaces, de délégués.

```
using System;  
public enum Color  
{  
    Red, Blue, Green  
}  
public struct Point  
{  
    public int x, y;  
}  
public interface IBase  
{  
    void F();  
}  
public interface IDerived: IBase  
{  
    void G();  
}  
public class A  
{  
    protected virtual void H() {  
        Console.WriteLine("A.H");  
    }  
}  
public class B: A, IDerived  
{  
    public void F() {  
        Console.WriteLine("B.F, implementation of IDerived.F");  
    }  
    public void G() {  
        Console.WriteLine("B.G, implementation of IDerived.G");  
    }  
}
```

```
    }  
    override protected void H() {  
        Console.WriteLine("B.H, override of A.H");  
    }  
}  
public delegate void EmptyDelegate();
```

On voit par cet exemple chaque sorte de déclaration.

Les types prédéfinis

C# contient un ensemble de types prédéfinis plus important que dans les langages C ou C++. Les types de référence prédéfinis sont `object` et `string`. Le type `object` est le type de base de tous les autres types. Le type `string` est utilisé pour représenter les valeurs des chaînes de caractères Unicode.

Les types de valeur prédéfinis incluent les types d'entiers signés et non signés, les types flottants ainsi que les types : booléen, caractère et décimal. Les types d'entiers signés sont `sbyte`, `short`, `int`, et `long`, les types non signés sont `byte`, `ushort`, `uint`, et `ulong`; et les types flottants sont `float` et `double`.

Le type `bool` est utilisé pour représenter les valeurs booléennes. C'est soit vrai soit faux. Le type `bool` permet plus facilement d'écrire du code auto documenté.

```
int i = ...;  
F(i);  
if (i == 0) // Bug : le test doit être (i == 0)  
    G();
```

Il en résulte une erreur de compilation due à l'expression `i == 0` qui est de type `int`, et la déclaration du `if` demande une expression de type `bool`.

Le type `char` est utilisé pour représenter les caractères Unicode. Une variable de type `char` représente un simple caractère Unicode de 16-bits.

Le type `decimal` est approprié pour les calculs dans lesquels les erreurs d'arrondis sont inacceptables. Un exemple simple est le calcul financier comme le calcul des taxes et les conversions de devises. Le type `decimal` fournit 28 chiffres significatifs.

Le tableau suivant liste les types prédéfinis et montre comment écrire leurs valeurs.

Type	Description	Exemple
object	Le type de base de tous les autres types	object o = null;
string	C'est une séquence de caractères Unicode	string s = "hello";
sbyte	Entier signé sur 8-bits	sbyte val = 12;
short	Entier signé sur 16-bits	short val = 12;
int	Entier signé sur 32-bits	int val = 12;
long	Entier signé sur 64-bits	long val1 = 12; long val2 = 34L;
byte	Entier non signé sur 8-bits	byte val1 = 12;
ushort	Entier non signé sur 16-bits	ushort val1 = 12;
uint	Entier non signé sur 32-bits	uint val1 = 12; uint val2 = 34U;
ulong	Entier non signé sur 64-bits	ulong val1 = 12; ulong val2 = 34U; ulong val3 = 56L; ulong val4 = 78UL;
float	Type simple sur 32-bits à virgule flottante	float val = 1.23F;
double	Type simple sur 64-bits à virgule flottante	double val1 = 1.23; double val2 = 4.56D;
bool	La valeur du type bool est soit vrai soit fausse	bool val1 = true; bool val2 = false;
char	C'est un caractère Unicode	char val = 'h';
decimal	C'est un type décimal précis avec 28 chiffres significatifs	decimal val = 1.23M;

Tous les types prédéfinis sont des raccourcis de type système. Par exemple, le mot clé `int` fait référence à la structure `System.Int32`.

Les types prédéfinis surchargent les opérateurs. Par exemple, la comparaison `==` et `!=` est différente suivant les types prédéfinis :

- Deux expressions de type `int` sont considérées équivalentes si elles représentent la même valeur entière.
- Deux expressions de type `object` sont considérées équivalentes si elles référencent le même objet ou si les deux sont `null`.
- Deux expressions de type `string` sont considérées équivalentes si leur valeur possède la même longueur et la même suite de caractères, ou si les deux sont `null`.

```
using System;
class Test
{
    static void Main() {
        string s = "Test";
        string t = string.Copy(s);
    }
}
```

```
        Console.WriteLine(s == t);  
        Console.WriteLine((object)s == (object)t);  
    }  
}
```

L'exemple produit la sortie suivante :

```
True  
False
```

En effet, la première comparaison s'effectue sur 2 chaînes de caractères et la seconde sur 2 objets. Une recopie a été faite entre les deux. Les chaînes ont bien la même taille et la même suite, mais l'objet n'est pas le même. Ce sont deux objets distincts.

Conversions

Les types prédéfinis possèdent des conversions prédéfinies. Par exemple, des conversions existent entre les types `int` et `long`. C# différencie deux types de conversions : conversions implicites et conversions explicites. Les conversions implicites n'ont pas besoin d'examen approfondi des types. Par exemple, la conversion du type `int` vers `long` est une conversion implicite. Cette conversion réussie tout le temps, et ne résulte jamais d'une perte d'informations. Un petit exemple :

```
using System;  
class Test  
{  
    static void Main() {  
        int intValue = 123;  
        long longValue = intValue;  
        Console.WriteLine("{0}, {1}", intValue, longValue);  
    }  
}
```

A l'inverse, les conversions explicites utilisent un `cast`.

```
using System;  
class Test  
{  
    static void Main() {  
        long longValue = Int64.MaxValue;  
        int intValue = (int)longValue;  
        Console.WriteLine("(int){0} = {1}",  
            longValue, intValue);  
    }  
}
```

Cet exemple utilise une conversion explicite pour convertir un type `long` dans un type `int`. La sortie sera :

```
(int)9223372036854775807 = -1
```

A cause de l'overflow, le résultat est différent. Les `cast` permettent l'utilisation des conversions explicites et implicites.



Le type Array

Les tableaux peuvent être de simples tableaux ou des tableaux à plusieurs dimensions.

Les tableaux sont des types simples :

```
using System;
class Test
{
    static void Main() {
        int [] arr = new int [5];
        for (int i = 0; i < arr.Length; i++)
            arr[i] = i * i;
        for (int i = 0; i < arr.Length; i++)
            Console.WriteLine("arr[{0}] = {1}", i, arr[i]);
    }
}
```

Cet exemple crée un tableau à une dimension de type `int`, il est initialisé et est ensuite affiché sur la console. La console affiche ceci :

```
arr[0] = 0
arr[1] = 1
arr[2] = 4
arr[3] = 9
arr[4] = 16
```

Le type `int []` utilisé dans l'exemple précédent est un type de tableau. Les types `array` sont écrit en utilisant un type (qui n'est pas un tableau) suivi un ou plusieurs crochets :

```
class Test
{
    static void Main() {
        int [] a1;           // tableau a une dimension
                             // de type int
        int [,] a2;          // tableau a deux dimensions
                             // de type int
        int [,,] a3;         // tableau a trois dimensions
                             // de type int
        int [][] j2;         // tableau de tableaux d'entier
        int [][][] j3;       // tableau de tableaux
                             // de tableaux d'entier
    }
}
```

Cet exemple montre la variété de déclarations possibles de tableaux d'entiers.

Les tableaux sont des types de référence. Les tableaux sont créés grâce à l'initialisation et à la création de valeurs, comme on peut le voir dans l'exemple suivant :

```
class Test
{
    static void Main() {
```

```
int [] a1 = new int [] {1, 2, 3};  
int [,] a2 = new int [,] {{1, 2, 3}, {4, 5, 6}};  
int [,] a3 = new int [10, 20, 30];  
int [][] j2 = new int [3][];  
j2[0] = new int [] {1, 2, 3};  
j2[1] = new int [] {1, 2, 3, 4, 5, 6};  
j2[2] = new int [] {1, 2, 3, 4, 5, 6, 7, 8, 9};  
}  
}
```

Il montre ainsi la variété de création de tableau. Les variables `a1`, `a2` et `a3` sont des tableaux rectangulaires alors que la variable `j2` est un tableau irrégulier (table de hachage).

La variable `j2` déclare un tableau de tableaux. Plus particulièrement, `j2` annonce un tableau de tableaux ou un tableau a une dimension de type `int []`. Chacune des ces variables peut être initialisé individuellement.

En C++, un tableau déclaré comme ceci : `int x [3][5][7]` est considéré comme un tableau a trois dimension "rectangulaire". En C#, la déclaration : `int [][][]` déclare un tableau "irrégulier".

```
int [,] a3 = new int [10, 20, 30];
```

C'est un tableau de type `int [,]` avec une création : `new int[10, 20, 30]`. Pour les variables locales et les champs, une forme raccourcie est permise. Par exemple :

```
int [] a1 = new int [] {1, 2, 3};
```

Peut être transformé en :

```
int [] a1 = {1, 2, 3};
```

Sans changement sémantique.

Le contexte dans lequel une initialisation de tableau `{1, 2, 3}` est utilisée, détermine le type de tableau qui doit être initialisé.

```
class Test  
{  
    static void Main() {  
        short [] a = {1, 2, 3};  
        int [] b = {1, 2, 3};  
        long [] c = {1, 2, 3};  
    }  
}
```

On remarque avec cet exemple que la même manière d'initialiser peut être utilisée pour plusieurs types de tableaux. On remarque alors que le contexte est nécessaire pour déterminer le type d'initialisation.

Le système d'unification de type C# fournis un système de type unifiés. Tous les types dérivent du type `object`. Il est possible d'appeler ses méthodes sur n'importe quelle valeur.

```
using System;  
class Test  
{
```

```
static void Main() {  
    Console.WriteLine(3.ToString());  
}
```

Ainsi la méthode `ToString` du type `object` est applicable sur un entier "3".

```
class Test  
{  
    static void Main() {  
        int i = 123;  
        object o = i;           // boxing  
        int j = (int) o;        // unboxing  
    }  
}
```

Une valeur `int` peut être convertie en un `object` et être retraduite en `int`. Les notions de boxing et unboxing apparaissent ici. Quand un type de valeur a besoin d'être converti en type référence, un objet box est alloué pour garder la valeur et la valeur est copiée dans la "boîte". Unboxing c'est quand une "boîte" est à nouveau dans son type original, la valeur est copiée hors de la "boîte".

Variables et paramètres

Les variables représentent les lieux de stockage. Toutes les variables ont un type qui détermine les valeurs qui peuvent être stockées dans ces variables. Les variables locales sont des variables déclarées dans les méthodes, les propriétés et les indexeurs. Une variable locale est définie par un type, un nom et une valeur initiale optionnelle.

```
int a;  
int b = 1;
```

Mais il est possible lors d'une déclaration de variable, d'avoir une déclaration multiple. Ainsi, `a` et `b` peuvent être déclarés comme ceci :

```
int a, b = 1;
```

Une variable doit être affectée avant d'être manipulée.

```
class Test  
{  
    static void Main() {  
        int a;  
        int b = 1;  
        int c = a + b; // erreur, a n'a pas été affecté  
        ...  
    }  
}
```

Dans notre exemple, lors de la compilation une erreur surviendra à cause de l'utilisation de la variable `a` avant qu'une valeur lui soit affectée.

Un champ est une variable associée à une classe ou une structure. Un champ déclaré avec le mot clé `static` définit une variable statique, et un champ déclaré sans, définit une variable d'instance. Un champ statique est associé avec un type, et une variable d'instance est associée à une instance.

```
using Personnel.Data;
class Employee
{
    private static DataSet ds;
    public string Name;
    public decimal Salary;
    ...
}
```

La classe `Employee` possède une variable privée statique et deux variables publiques d'instance. La déclaration des paramètres formels est semblable à celle des variables. Il existe quatre sortes de paramètres : les paramètres par valeur, les paramètres par référence, les paramètres de sortie et les tableaux de paramètres.

Un paramètre par valeur est utilisé pour les paramètres d'entrée, dans lequel la valeur de l'argument est passée dans la méthode. Les modifications des paramètres n'auront pas d'impact sur l'argument original. Un paramètre par valeur référence sa propre variable. La variable est initialisé par copie de la valeur de l'argument original.

```
using System;
class Test {
    static void F(int p) {
        Console.WriteLine("p = {0}", p);
        p++;
    }
    static void Main() {
        int a = 1;
        Console.WriteLine("pre: a = {0}", a);
        F(a);
        Console.WriteLine("post: a = {0}", a);
    }
}
```

La méthode `F` possède un paramètre par valeur nommé `p`.

```
pre: a = 1
p = 1
post: a = 1
```

On remarque que seule la valeur de `p` est modifiée.

Un paramètre par référence est utilisé pour les passages par référence, dans lesquels le paramètre remplace l'argument fourni. Un paramètre par référence ne définit pas une variable, mais référence la variable correspondant à l'argument. Les modifications des paramètres agissent sur l'argument correspondant. Un paramètre par référence est déclaré avec le mot clé `ref`.

```
using System;
class Test {
    static void Swap(ref int a, ref int b) {
```

```
        int t = a;
        a = b;
        b = t;
    }
    static void Main() {
        int x = 1;
        int y = 2;

        Console.WriteLine("pre:  x = {0}, y = {1}", x, y);
        Swap(ref x, ref y);
        Console.WriteLine("post: x = {0}, y = {1}", x, y);
    }
}
```

La méthode `Swap` possède deux paramètres par références. L'exécution produit :

```
pre:  x = 1, y = 2
post: x = 2, y = 1
```

Le mot clé `ref` doit être utilisé dans la déclaration des paramètres formels et dans l'utilisation de ceux-ci.

Un paramètre de sortie est similaire à un paramètre de références excepté que la valeur initiale de l'argument est sans importance. On ajoute le mot clé `out` pour déclarer un paramètre de sortie.

```
using System;
class Test {
    static void Divide(int a, int b, out int result,
        out int remainder) {
        result = a / b;
        remainder = a % b;
    }
    static void Main() {
        for (int i = 1; i < 10; i++)
            for (int j = 1; j < 10; j++) {
                int ans, r;
                Divide(i, j, out ans, out r);
                Console.WriteLine("{0} / {1} = {2}r{3}",
                    i, j, ans, r);
            }
    }
}
```

La méthode `Divide` a deux paramètres de sortie (une pour le résultat de la division et l'autre pour le reste).

Pour les paramètres vus précédemment, il existe une correspondance entre l'argument appelé et le paramètre qui le représente. Un tableau de paramètres permet un passage de paramètres plus important, c'est l'utilisation d'une liste d'arguments.

Un tableau de paramètres est déclaré avec le mot clé `params`. C'est un paramètre pour une méthode donnée, et doit toujours être le dernier des paramètres. Le type de ce paramètre est un

tableau à une dimension. A l'appel de méthodes incluant ce paramètre, il existe deux manières de faire. La première consiste à passer en argument un objet correspond au type tableau. La seconde est de passer les éléments que l'on désire.

```
using System;
class Test
{
    static void F(params int[] args) {
        Console.WriteLine("# of arguments: {0}", args.Length);
        for (int i = 0; i < args.Length; i++)
            Console.WriteLine("targs[{0}] = {1}",
                               i, args[i]);
    }
    static void Main() {
        F();
        F(1);
        F(1, 2);
        F(1, 2, 3);
        F(new int[] {1, 2, 3, 4});
    }
}
```

La méthode `F` prend un nombre variable d'argument. On obtient ceci :

```
# of arguments: 0
# of arguments: 1
    args[0] = 1
# of arguments: 2
    args[0] = 1
    args[1] = 2
# of arguments: 3
    args[0] = 1
    args[1] = 2
    args[2] = 3
# of arguments: 4
    args[0] = 1
    args[1] = 2
    args[2] = 3
    args[3] = 4
```

Expressions

Dans le langage C#, on trouve les opérateurs unaires, les opérateurs binaires et un opérateur ternaire. Le tableau suivant résume les différents opérateurs du langage dans l'ordre du plus fort au plus faible.

Catégorie	Opérateurs
Primaire	<code>x.y f(x) a[x] x++ x-- new</code>
typeof	<code>checked unchecked</code>
Unaire	<code>+ - ! ~ ++x --x (T)x</code>
Multiplication	<code>* / %</code>
Addition	<code>+ -</code>
Décalage	<code>>> <<</code>
Relationnel et information de type	<code>> < >= <= is as</code>
Egalité	<code>== !=</code>
ET logique	<code>&</code>
XOR logique	<code>^</code>
OU logique	<code> </code>
ET conditionnel	<code>&&</code>
OU conditionnel	<code> </code>
Conditionnel	<code>?:</code>
Affectation	<code>= *= /= %= += -= <<= >>= &= ^= =</code>

Quand une expression contient plusieurs opérateurs, l'évaluation s'effectue en fonction de leur force. Par exemple, l'expression `x + y * z` est évaluée comme ceci : `x + (y * z)`. En effet, l'opérateur `*` est plus fort que l'opérateur `+`.

Quand deux opérateurs ont la même force, c'est l'associativité des opérateurs qui contrôle l'évaluation :

- A l'exception des opérateurs d'affectation, tous les opérateurs binaires sont associatifs gauches. Par exemple, `x + y + z` est évalué comme ceci : `(x + y) + z`.
- Les opérateurs d'affectation et l'opérateur conditionnel (`?:`) sont associatifs droits. Par exemple, `x = y = z` est évalué comme ceci : `x = (y = z)`.
- L'évaluation des opérateurs peut être contrôlé en utilisant des parenthèses. Par exemple, `x + y * z` sera évalué en commençant par multiplier `y` par `z` puis ensuite avec l'addition de `x` au résultat, mais `(x + y) * z` additionne d'abord `x` et `y` et ensuite multiplie le résultat avec `z`.

Instructions

C# emprunte beaucoup de instructions aux langages C et C++. Il existe quand même des ajouts particuliers. La liste suivante montre les différentes instructions avec un exemple d'utilisation pour chaque.

Les listes des instructions et les blocks

```
static void Main() {
    F();
    G();
    {
        H();
        I();
    }
}
```

```
}  
}
```

Les instructions avec étiquettes et les instructions de saut comme `goto`

```
static void Main(string[] args) {  
    if (args.Length == 0)  
        goto done;  
    Console.WriteLine(args.Length);  
done:  
    Console.WriteLine("Done");  
}
```

Les déclarations locales de constantes

```
static void Main() {  
    const float pi = 3.14f;  
    const int r = 123;  
    Console.WriteLine(pi * r * r);  
}
```

Les déclarations locales de variables

```
static void Main() {  
    int a;  
    int b = 2, c = 3;  
    a = 1;  
    Console.WriteLine(a + b + c);  
}
```

Les déclarations d'expression

```
static int F(int a, int b) {  
    return a + b;  
}  
static void Main() {  
    F(1, 2); // Expression statement  
}
```

L'instruction `if`

```
static void Main(string[] args) {  
    if (args.Length == 0)  
        Console.WriteLine("No args");  
    else  
        Console.WriteLine("Args");  
}
```

L'instruction `switch`

```
static void Main(string[] args) {  
    switch (args.Length) {  
        case 0:  
            Console.WriteLine("No args");  
            break;  
        case 1:  
            Console.WriteLine("One arg ");  
            break;  
        default:  
            int n = args.Length;  
            Console.WriteLine("{0} args", n);  
            break;  
    }  
}
```

L'instruction while

```
static void Main(string[] args) {  
    int i = 0;  
    while (i < args.Length) {  
        Console.WriteLine(args[i]);  
        i++;  
    }  
}
```

L'instruction do

```
static void Main() {  
    string s;  
    do { s = Console.ReadLine(); }  
    while (s != "Exit");  
}
```

L'instruction for

```
static void Main(string[] args) {  
    for (int i = 0; i < args.Length; i++)  
        Console.WriteLine(args[i]);  
}
```

L'instruction foreach

```
static void Main(string[] args) {  
    foreach (string s in args)  
        Console.WriteLine(s);  
}
```

L'instruction break

```
static void Main(string[] args) {
```

```
int i = 0;
while (true) {
    if (i == args.Length)
        break;
    Console.WriteLine(args[i++]);
}
```

L'instruction continue

```
static void Main(string[] args) {
    int i = 0;
    while (true) {
        Console.WriteLine(args[i++]);
        if (i < args.Length)
            continue;
        break;
    }
}
```

L'instruction return

```
static int F(int a, int b) {
    return a + b;
}
static void Main() {
    Console.WriteLine(F(1, 2));
    return;
}
```

Les instructions throw et try

```
static int F(int a, int b) {
    if (b == 0)
        throw new Exception("Divide by zero");
    return a / b;
}
static void Main() {
    try {
        Console.WriteLine(F(5, 0));
    }
    catch (Exception e) {
        Console.WriteLine("Error");
    }
}
```

Les instructions checked et unchecked

```
static void Main() {
    int x = Int32.MaxValue;
```

```
Console.WriteLine(x + 1);          // Overflow

checked {
    Console.WriteLine(x + 1);      // Exception
}

unchecked {
    Console.WriteLine(x + 1);      // Overflow
}
}
```

L'instruction lock

```
static void Main() {
    A a = ...;
    lock(a) {
        a.P = a.P + 1;
    }
}
```

L'instruction using

```
static void Main() {
    using (Resource r = new Resource()) {
        r.F();
    }
}
```

Classes

Méthodes

Une méthode est un membre qui implémente un calcul ou une action qui peut être réalisée par un objet ou une classe. Les méthodes possèdent une liste de paramètres formels, une valeur de retour (ou `void`), et elle peut être statique. Les méthodes statiques sont accédées à travers la classe. Les méthodes non statiques, sont accédées par les instances de la classe.

```
using System;
public class Stack
{
    public static Stack Clone(Stack s) {...}
    public static Stack Flip(Stack s) {...}
    public object Pop() {...}
    public void Push(object o) {...}
    public override string ToString() {...}
    ...
}
class Test
```

```
{
    static void Main() {
        Stack s = new Stack();
        for (int i = 1; i < 10; i++)
            s.Push(i);
        Stack flipped = Stack.Flip(s);
        Stack cloned = Stack.Clone(s);
        Console.WriteLine("Original stack: " + s.ToString());
        Console.WriteLine("Flipped stack: "
            + flipped.ToString());
        Console.WriteLine("Cloned stack: " + cloned.ToString());
    }
}
```

L'exemple montre une classe `Stack` qui possède plusieurs méthodes statiques (`Clone` et `Flip`) et plusieurs méthodes d'instances (`Push`, `Pop`, et `ToString`).

Les méthodes peuvent être surchargées, cela signifie qu'il peut exister plusieurs méthodes qui ont la même signature. La signature de la méthode, c'est le nom de la méthode et le nombre d'arguments, les modifieurs, et les types de ses paramètres formels. La signature de la méthode ne prend pas le type de retour.

```
using System;
class Test
{
    static void F() {
        Console.WriteLine("F()");
    }
    static void F(object o) {
        Console.WriteLine("F(object)");
    }
    static void F(int value) {
        Console.WriteLine("F(int)");
    }
    static void F(ref int value) {
        Console.WriteLine("F(ref int)");
    }
    static void F(int a, int b) {
        Console.WriteLine("F(int, int)");
    }
    static void F(int[] values) {
        Console.WriteLine("F(int[])");
    }
    static void Main() {
        F();
        F(1);
        int i = 10;
        F(ref i);
        F((object)1);
        F(1, 2);
    }
}
```

```
        F(new int [] {1, 2, 3});  
    }  
}
```

Cet exemple montre une classe avec un nombre de méthodes appelées `F`, en sortie on a :

```
F()  
F(int)  
F(ref int)  
F(object)  
F(int, int)  
F(int [])
```

Propriétés

Une propriété est un membre qui fournit les accès à un attribut d'une classe ou d'un objet. On peut trouver comme exemple de propriétés : la longueur d'une string, la taille d'une police, la légende d'une fenêtre, le nom d'un client. Les propriétés sont une extension naturelle des champs. Les deux sont des membres nommés avec des types associés et la syntaxe pour accéder aux champs et aux propriétés sont les mêmes.

La première partie de la déclaration est similaire aux champs. La seconde partie inclut un accesseur `get` et/ou un accesseur `set`. Dans l'exemple qui suit, la classe `Button` définit une propriété `Caption`.

```
public class Button  
{  
    private string caption;  
    public string Caption {  
        get {  
            return caption;  
        }  
        set {  
            caption = value;  
            Repaint();  
        }  
    }  
}
```

Les propriétés peuvent être soit lues soit écrites comme `Caption`, qui inclut les accesseurs nécessaires. L'accesseur `get` est appelé quand la valeur est lue, le `set` quand elle est écrite. La déclaration de la propriété est relativement intuitive, mais le réel avantage des propriétés est lors de leur utilisation. Par exemple, la propriété `Caption` peut être lue et écrite de la même façon que les champs.

```
Button b = new Button();  
b.Caption = "ABC";           // set  
string s = b.Caption;       // get  
b.Caption += "DEF";          // get & set
```

Évènements

Un évènement (*event*) est un membre qui permet à un objet ou une classe de fournir des notifications. Une classe définit un évènement en fournissant une déclaration d'évènement (qui ressemble aux déclarations de champ avec l'ajout du mot *event*) ainsi qu'un ensemble d'accesseurs. Le type de cette déclaration doit être un *delegate*.

Une instance de *delegate* regroupe plusieurs entités appelables. Pour les méthodes d'instance une entité appelante consiste en une instance et une méthode sur cette instance. Pour les méthodes statiques, une entité appelante consiste juste en une méthode.

```
public delegate void EventHandler(object sender, EventArgs e);
public class Button
{
    public event EventHandler Click;
    public void Reset() {
        Click = null;
    }
}
```

La classe *Button* définit un évènement *Click* de type *EventHandler*. A l'intérieur de la classe *Button*, le membre *Click* est exactement comme un champ privé de type *EventHandler*. A l'extérieur de la classe *Button*, le membre *Click* peut seulement être utilisé du côté gauche des opérateurs *+=* et *-=*. L'opérateur *+=* ajoute un *handler* pour l'évènement, et l'opérateur *-=* le supprime.

```
using System;
public class Form1
{
    public Form1() {
        // Add Button1_Click as an event handler
        // for Button1's Click event
        Button1.Click += new EventHandler(Button1_Click);
    }
    Button Button1 = new Button();
    void Button1_Click(object sender, EventArgs e) {
        Console.WriteLine("Button1 was clicked!");
    }
    public void Disconnect() {
        Button1.Click -= new EventHandler(Button1_Click);
    }
}
```

La classe *Form1* ajoute *Button1_Click* comme un évènement *handler* pour le *Click* du *Button1*. Dans la méthode *Disconnect*, l'évènement est supprimé.

Une déclaration simple d'un évènement :

```
public event EventHandler Click;
```

Le compilateur fourni automatiquement l'implémentation implicite de *+=* et *-=*.

Un programmeur qui veut contrôler les actions d'ajout de suppression pourra explicitement créer les accesseurs.

```
public class Button
{
    private EventHandler handler;
    public event EventHandler Click {
        add { handler += value; }
        remove { handler -= value; }
    }
}
```

Opérateurs

Un opérateur (operator) est un membre qui définit la façon dont l'opérateur peut agir sur les instances de la classe. Il existe trois types d'opérateurs : les opérateurs unaires, les opérateurs binaires et les opérateurs de conversion. Tous les opérateurs doivent être publics et statiques.

L'exemple suivant définit un type `Digit` qui représente les chiffres entre 0 et 9.

```
using System;
public struct Digit
{
    byte value;
    public Digit(byte value) {
        if (value < 0 || value > 9)
            throw new ArgumentException();
        this.value = value;
    }
    public Digit(int value): this((byte) value) {}
    public static implicit operator byte(Digit d) {
        return d.value;
    }
    public static explicit operator Digit(byte b) {
        return new Digit(b);
    }
    public static Digit operator+(Digit a, Digit b) {
        return new Digit(a.value + b.value);
    }
    public static Digit operator-(Digit a, Digit b) {
        return new Digit(a.value - b.value);
    }
    public static bool operator==(Digit a, Digit b) {
        return a.value == b.value;
    }
    public static bool operator!=(Digit a, Digit b) {
        return a.value != b.value;
    }
    public override bool Equals(object value) {
        if (value == null) return false;
        if (GetType() == value.GetType())
            return this == (Digit)value;
    }
}
```



```
        return false;
    }
    public override int GetHashCode() {
        return value.GetHashCode();
    }
    public override string ToString() {
        return value.ToString();
    }
}
class Test
{
    static void Main() {
        Digit a = (Digit) 5;
        Digit b = (Digit) 3;
        Digit plus = a + b;
        Digit minus = a - b;
        bool equals = (a == b);
        Console.WriteLine("{0} + {1} = {2}", a, b, plus);
        Console.WriteLine("{0} - {1} = {2}", a, b, minus);
        Console.WriteLine("{0} == {1} = {2}", a, b, equals);
    }
}
```

Le type `Digit` définit les opérateurs suivants :

- Une conversion implicite de `Digit` vers `byte`.
- Une conversion explicite de `byte` vers `Digit`.
- Un opérateur d'addition entre deux `Digit`, il retourne un `Digit`.
- Un opérateur de soustraction entre deux `Digit`, il retourne un `Digit`.
- L'égalité (`==`) et l'inégalité (`!=`), qui compare deux `Digit`.

Indexeurs

Un indexeur est un membre qui permet aux objets d'être indexés de la même manière qu'un tableau.

Les déclarations d'indexeurs sont similaires aux déclarations de propriété, avec la différence principale que les indexeurs sont anonymes et les indexeurs incluent des paramètres. Les paramètres sont fournis entre crochets.

```
using System;
public class Stack
{
    private Node GetNode(int index) {
        Node temp = first;
        while (index > 0 && temp != null) {
            temp = temp.Next;
            index--;
        }
        if (index < 0 || temp == null)
```

```
        throw new Exception("Index out of range.");
        return temp;
    }
    public object this[int index] {
        get {
            return GetNode(index).Value;
        }
        set {
            GetNode(index).Value = value;
        }
    }
    ...
}
class Test
{
    static void Main() {
        Stack s = new Stack();
        s.Push(1);
        s.Push(2);
        s.Push(3);
        s[0] = 33;           // Changes the top item from 3 to 33
        s[1] = 22;           // Changes the middle item from 2 to 22
        s[2] = 11;           // Changes the bottom item from 1 to 11
    }
}
```

Cet exemple montre un indexer pour la classe `Stack`.

Constructeurs

Un constructeur est un membre qui représente les actions nécessaires pour initialiser une classe.

```
using System;
class Point
{
    public double x, y;
    public Point() {
        this.x = 0;
        this.y = 0;
    }
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public static double Distance(Point a, Point b) {
        double xdiff = a.x - b.x;
        double ydiff = a.y - b.y;
        return Math.Sqrt(xdiff * xdiff + ydiff * ydiff);
    }
}
```

```
    }  
    public override string ToString() {  
        return string.Format("{0}, {1}", x, y);  
    }  
}  
class Test  
{  
    static void Main() {  
        Point a = new Point();  
        Point b = new Point(3, 4);  
        double d = Point.Distance(a, b);  
        Console.WriteLine("Distance from {0} to {1} is {2}",  
            a, b, d);  
    }  
}
```

La classe `Point` fournit deux constructeurs publics, un qui ne prend pas d'argument, et l'autre qui prend deux doubles en arguments.

Si le constructeur n'est pas présent dans la classe, un constructeur vide est fourni automatiquement.

Destructeurs

Un destructeur est un membre qui implémente les actions nécessaires pour détruire l'instance d'une classe. Les destructeurs ne peuvent pas avoir de paramètres et ne peuvent pas être appelé explicitement. Le destructeur d'une instance est appelé automatiquement durant le ramasse miette.

```
using System;  
class Point  
{  
    public double x, y;  
    public Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
    ~Point() {  
        Console.WriteLine("Destructed {0}", this);  
    }  
    public override string ToString() {  
        return string.Format("{0}, {1}", x, y);  
    }  
}
```

Cet exemple montre une classe `Point` avec un destructeur.

Constructeurs statiques

Un constructeur statique est un membre qui implémente les actions nécessaires pour initialiser une classe. Ils ne peuvent pas avoir de paramètres et ne peuvent être appelé explicitement. Le constructeur statique est appelé automatiquement.

```
using Personnel.Data;
class Employee
{
    private static DataSet ds;
    static Employee() {
        ds = new DataSet (...);
    }
    public string Name;
    public decimal Salary;
    ...
}
```

Cet exemple montre une classe `Employee` avec un constructeur statique qui initialise un champ statique.

L'héritage

Les classes supportent l'héritage simple, et l'objet de base pour toutes les classes est le type `object`.

Les classes dérive implicitement d'`object`.

```
using System;
class A
{
    public void F() { Console.WriteLine("A.F"); }
}
```

La classe `A` dérive implicitement d'`object`.

```
using System;
class B: A
{
    public void G() { Console.WriteLine("B.G"); }
}
class Test
{
    static void Main() {
        B b = new B();
        b.F();           // Inherited from A
        b.G();           // Introduced in B
        A a = b;         // Treat a B as an A
        a.F();
    }
}
```

La classe **B** dérive de **A**. la classe **B** hérite de la méthode **F**, et introduit la méthode **G**.

Les méthodes, les propriétés et les indexeurs peuvent être virtuels, ce qui signifie que leur implémentation se trouvera dans les classes dérivées.

```
using System;
class A
{
    public virtual void F() { Console.WriteLine("A.F"); }
}
class B: A
{
    public override void F() {
        base.F();
        Console.WriteLine("B.F");
    }
}
class Test
{
    static void Main() {
        B b = new B();
        b.F();
        A a = b;
        a.F();
    }
}
```

La classe **A** a une méthode virtuelle **F**, et la classe **B** surcharge **F**. la méthode surchargée dans **B** contient un appel a **base.F()**, qui appelle la méthode dans **A**.

Une classe peut indiquer qu'elle est incomplète et inutilisable sans dérivation. C'est une classe abstraite (**abstract class**). Une classe abstraite peut spécifier des membres abstraits (**abstract members**).

```
using System;
abstract class A
{
    public abstract void F();
}
class B: A
{
    public override void F() { Console.WriteLine("B.F"); }
}
class Test
{
    static void Main() {
        B b = new B();
        b.F();
        A a = b;
        a.F();
    }
}
```

On a la méthode abstraite **F** dans la classe abstraite **A**. la classe non abstraite **B** fournit une implémentation pour cette méthode.

Structures

Les points communs entre les classes et les structures sont nombreux. Les structures peuvent implémenter des interfaces et peuvent avoir les mêmes membres qu'une classe. Les différences entre les classes et les structures sont importantes. Les structures ne supportent pas l'héritage, et sont plus des types de valeur que types de référence.

Par exemple, l'utilisation de structure au lieu d'une classe pour un **Point** peut faire une différence importante en terme de mémoires. Le programme qui veut créer et initialiser un tableau de 100 points. Si le point est une classe 101 objets seront instanciés.

```
class Point
{
    public int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
class Test
{
    static void Main() {
        Point[] points = new Point[100];
        for (int i = 0; i < 100; i++)
            points[i] = new Point(i, i*i);
    }
}
```

Si **Point** est implémenté par une structure seulement un objet est instancier.

```
struct Point
{
    public int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Les instances du **Point** sont localisées dans le tableau. Cette optimisation peut être mauvaise. L'utilisation des structures au lieu des classes peut ralentir l'application, ou prendre plus de mémoire dans le cas d'un passage par valeur (une copie devra être créée).

Interfaces

Une interface définit un contrat. Une classe ou une structure qui implémente une interface doit adhérer au contrat. Les interfaces peuvent contenir des méthodes, des propriétés, des événements et des indexeurs comme membre.

L'exemple :

```
interface IExample
{
    string this[int index] { get; set; }
    event EventHandler E;
    void F(int value);
    string P { get; set; }
}
public delegate void EventHandler(object sender, EventArgs e);
```

C'est une interface qui contient un indexeur, un événement `E`, une méthode `F`, et une propriété `P`. Les interfaces supportent l'héritage multiple.

```
interface IControl
{
    void Paint();
}
interface ITextBox: IControl
{
    void SetText(string text);
}
interface IListBox: IControl
{
    void SetItems(string[] items);
}
interface IComboBox: ITextBox, IListBox {}
```

Dans cet exemple, l'interface `IComboBox` hérite de deux autres interfaces : `ITextBox` et `IListBox`.

Les classes et les structures peuvent implémentés plusieurs interfaces.

```
interface IDataBound
{
    void Bind(Binder b);
}
public class EditBox: Control, IControl, IDataBound
{
    public void Paint() {...}
    public void Bind(Binder b) {...}
}
```

La classe `EditBox` dérive de la classe `Control` et implémente deux interfaces : `IControl` et `IDataBound`.

Dans l'exemple précédent, la méthode `Paint` de l'interface `IControl` et la méthode `Bind` de

l'interface `IDataBound` sont implémentées en utilisant la classes `EditBox`. C# fournit une alternative d'implémentation de ces méthodes. En effet, il permet aux membres de la classe d'implémentation d'être public. Par exemple, la classe `EditBox` peut être implémentée avec les méthodes : `IControl.Paint` et `IDataBound.Bind`.

```
public class EditBox: IControl , IDataBound
{
    void IControl.Paint() {...}
    void IDataBound.Bind(Binder b) {...}
}
```

Les membres implémentés de cette façon sont appelés explicitement car chaque membre explicite désigne le membre d'interface qui doit être implémenté. Ces membres peuvent être seulement appelés par l'interface. Par exemple, l'implémentation de la méthode `Paint` de `EditBox` peut être appelée seulement par le cast en `IControl`.

```
class Test
{
    static void Main() {
        EditBox editbox = new EditBox();
        editbox.Paint(); // erreur: no such method
        IControl control = editbox;
        control.Paint(); // calls EditBox's Paint
                        // implementation
    }
}
```

Delegates

Les délégués sont comme dans d'autres langages des pointeurs de fonctions. Contrairement aux pointeurs de fonctions, les délégués sont des objets orientés.

La déclaration d'un délégué définit une classe qui est dérivé de la classe `System.Delegate`. Un délégué encapsule une liste d'invocations, qui est une liste d'une ou plusieurs méthodes, chacune faisant référence à une entité appelante. Pour la méthode d'instance, une entité appellable contient une instance et une méthode sur cette instance. Pour les méthodes statiques, une entité appellable contient juste une méthode. En invoquant un délégué avec un ensemble approprié d'arguments, chaque entité appellable est invoquée avec l'ensemble d'arguments donné.

Il existe trois étapes pour définir et utiliser les délégués : déclaration, instanciation et invocation. Le délégué est déclaré comme ceci :

```
delegate void SimpleDelegate();
```

C'est un délégué nommé `SimpleDelegate` qui ne prends pas d'arguments et ne retourne pas de résultat.

```
class Test
{
    static void F() {
```

```
        System.Console.WriteLine("Test.F");  
    }  
    static void Main() {  
        SimpleDelegate d = new SimpleDelegate(F);  
        d();  
    }  
}
```

Cet exemple crée un SimpleDelegate et l'appelle immédiatement.

Il n'existe peu de cas où l'on instancie un délégué puis on l'appelle immédiatement, il est plus simple de l'appeler directement. Les délégués montrent réellement leur utilité quand leur anonymat est utilisé.

```
void MultiCall(SimpleDelegate d, int count) {  
    for (int i = 0; i < count; i++) {  
        d();  
    }  
}
```

Cet exemple montre une méthode MultiCall qui répète l'appel de SimpleDelegate. La méthode MultiCall ne connaît pas ou ne s'intéresse pas sur le type de retour de la méthode de SimpleDelegate. Tout ce qu'on sait est que la méthode cible est compatible SimpleDelegate.

Énumérations

Un type énuméré définit un type pour un groupe de constantes symboliques. Les types énumérés sont utilisés pour les choix multiples. Dans chaque décision, un nombre de choix fixes est possible. Ces choix sont connus dès la compilation.

L'exemple

```
enum Color  
{  
    Red,  
    Blue,  
    Green  
}  
class Shape  
{  
    public void Fill(Color color) {  
        switch(color) {  
            case Color.Red:  
                ...  
                break;  
            case Color.Blue:  
                ...  
                break;  
            case Color.Green:  
                ...  
            }  
    }  
}
```

```
                break;
            default:
                break;
        }
    }
}
```

Il montre une énumération appelée Color et une méthode qui utilise cette énumération. La signature de la méthode Fill remplit avec une des couleurs données la forme.

L'utilisation d'énumérations étendue rend le code plus lisible et mieux documenté. L'auto documentation est possible avec les outils de développement.

Attributs

C# est un langage impératif, mais comme tous les langages impératifs, il possèdent des éléments déclaratifs. Par exemple, l'accès d'une méthode dans une classe est spécifiée par `public`, `protected`, `internal`, `protected internal`, ou `private`. C# généralise cette capacité, les programmeurs peuvent inventer de nouvelles sortes d'informations déclaratives, attacher cette information aux entités du programme, et récupérer cette information à l'exécution. Les programmes spécifient cette information déclarative additionnelle par définition et en utilisant les attributs.

Par exemple, un framework doit définir un attribut `HelpAttribute` qui peut être placé sur les éléments du programme comme des classes et méthodes, fournissant un outil pour les développeurs afin de documenter les éléments du programme.

```
using System;
[AttributeUsage(AttributeTargets.All)]
public class HelpAttribute : Attribute
{
    public HelpAttribute(string url) {
        this.url = url;
    }
    public string Topic = null;
    private string url;
    public string Url {
        get { return url; }
    }
}
```

Cet exemple définit un attribut nommé `HelpAttribute` qui possède un paramètre positionnel (`string url`) et un paramètre nommé (`string Topic`). Cet attribut doit être référencé par son nom complet, `HelpAttribute`, ou par son surnom implicite, `Help`. Les paramètres positionnels sont définis par les paramètres formels pour les constructeurs de l'attribut, et les paramètres nommés sont définis par les champs de la classe.

```
[Help("http://www.microsoft.com/.../Class1.htm")]
public class Class1
{
    [Help("http://www.microsoft.com/.../Class1.htm", Topic = "F")]
```

```
}  
    public void F() {}  
}
```

Cet exemple montre plusieurs utilisations de l'attribut `Help`.

L'information d'un attribut pour un élément donné peut être récupéré à l'exécution en utilisation la réflexion.

```
using System;  
class Test  
{  
    static void Main() {  
        Type type = typeof(Class1);  
        object[] arr = type.GetCustomAttributes(  
            typeof(HelpAttribute), true);  
        if (arr.Length == 0)  
            Console.WriteLine(  
                "Class1 has no Help attribute.");  
        else {  
            HelpAttribute ha = (HelpAttribute) arr[0];  
            Console.WriteLine("Url = {0}, Topic = {1}",  
                ha.Url, ha.Topic);  
        }  
    }  
}
```

Cet exemple vérifie si `Class1` a un attribut `Help`, et affiche le `Topic` et l'`Url` si l'attribut est présent.

F Une comparaison C# vs Java

Cette annexe montre à l'aide d'extrait de codes, les différences entre les deux langages. Cette traduction a été faite manuellement, et n'est pas unique. Ces exemples ne peuvent pas être généralisés pour toutes les transformations C# vers JAVA.

C#	JAVA
le programme <i>Hello World</i>	
<pre>class A{ public static void Main(String [] args) { Console.WriteLine("Hello World"); } }</pre>	<pre>class B{ public static void main(String [] args) { System.out.println("Hello World"); } }</pre>
les commentaires	
<pre>// Single line /* Multiple line */ /// XML comments on a single line /** XML comments on multiple lines */</pre>	<pre>// Single line /* Multiple line */ /** Javadoc documentation comments */</pre>
Les types les types primitifs	
<pre>bool byte , sbyte char short , ushort , int , uint , long , ulong float , double , decimal // et structures , énumérations</pre>	<pre>boolean byte char short , int , long float , double</pre>
les types références	
<pre>object string // et tableaux , classes , interfaces , // délégués</pre>	<pre>Object String // et tableaux , classes , interfaces</pre>



C#	JAVA
les conversions	
<pre>// int to string int x = 123; string y = x.ToString(); // string to int y = "456"; x = int.Parse(y); // double to int double z = 3.5; x = (int)z;</pre>	<pre>// int to String int x = 123; String y = Integer.toString(x); // String to int y = "456"; x = Integer.parseInt(y); // double to int double z = 3.5; x = (int)z;</pre>
les tableaux	
<pre>int [] nums = {1, 2, 3}; for (int i = 0; i < nums.Length; i++) Console.WriteLine(nums[i]); string [] names = new string [5]; names[0] = "David"; float [,] twoD = new float [rows, cols]; twoD[2,0] = 4.5f; int [][] jagged = new int [3][] { new int [5], new int [2], new int [3] }; jagged[0][4] = 5;</pre>	<pre>int [] nums = {1, 2, 3}; // ou int nums[] for (int i = 0; i < nums.length; i++) System.out.println(nums[i]); String [] names = new String [5]; names[0] = "David"; float [][] twoD = new float [rows][cols]; twoD[2][0] = 4.5; int [][] jagged = new int [3][]; jagged[0] = new int [5]; jagged[1] = new int [2]; jagged[2] = new int [3]; jagged[0][4] = 5;</pre>



C#	JAVA
Variables et paramètres	
<pre>void TestFunc(int x, ref int y, out int z) { x++; y++; z = 5; } int a = 1, b = 1, c; TestFunc(a, ref b, out c); // a = 1, b = 2, c = 5 int Sum(params int[] nums) { int sum = 0; foreach(int i in nums) sum += i; return sum; }</pre>	<pre>void TestFunc(int x, int y, int z) { x++; y++; z = 5; } int a = 1, b = 1, c = 0; TestFunc(a, b, c); // a = 1, b = 1, c = 0 int Sum(int ... nums) { int sum = 0; foreach(int i : nums) sum += i; return sum; }</pre>
Expressions opérateurs	
<pre>{ } [] () . , : ; + - * / % & ^ ! ~ = < > ? ++ -- && << >> == != <= >= + = -= *= /= %= &= = ^= <<= >>= -> // Pre-processing directives : #</pre>	<pre>{ } [] () . , : ; + - * / % & ^ ! ~ = < > ? ++ -- && << >> == != <= >= + = -= *= /= %= &= = ^= <<= >>= // Java doesn't have operator -> // Pre-processing directives : // None</pre>



C#	JAVA
Instructions l'instruction <i>goto</i>	
<pre>retry: try{ num_tries++; Console.WriteLine("Number of tries =" + num_tries); throw new SocketException(); } catch (SocketException){ if(num_tries < 5) goto retry; }</pre>	<pre>boolean retry = true; retry: while(retry) { retry = false; try{ num_tries++; System.out.println("Nb of tries =" + num_tries); throw new SocketException(); } catch (SocketException e){ if(num_tries < 5) retry = true; } }</pre>
les constantes	
<pre>const double PI = 3.14;</pre>	<pre>final double PI = 3.</pre>
les variables locales	
<pre>static void Main() { int a; int b = 2 ,c = 3; a = 1; Console.WriteLine(a + b + c) ; }</pre>	<pre>static void Main() { int a; int b = 2 ,c = 3; a = 1; System.out.println(a + b + c); }</pre>
les expressions	
<pre>static int F(int a, int b) { return a + b; } static void Main() { F(1, 2); // Expression statement }</pre>	<pre>static int F(int a, int b) { return a + b; } static void Main() { F(1, 2); // Expression statement }</pre>



C#	JAVA
l'instruction <i>if</i>	
<pre>greeting = age < 20 ? "What's up?" : "Hello"; if (x < y) System.out.println("greater"); if (x != 100) { x *= 5; y *= 2; } else z *= 6;</pre>	<pre>greeting = age < 20 ? "What's up?" : "Hello"; if (x < y) System.out.println("greater"); if (x != 100) { x *= 5; y *= 2; } else z *= 6;</pre>
l'instruction <i>switch</i>	
<pre>string color = "red"; switch(color) { case "red" : r++; break; case "blue" : b++; break; case "green" : g++; break; default: other++; break; }</pre>	<pre>int selection = 2; switch(selection) { case 1: x++; break; case 2: y++; break; case 3: z++; break; default: other++; }</pre>
l'instruction <i>while</i>	
<pre>while (i < 20) i++;</pre>	<pre>while (i < 20) i++;</pre>
l'instruction <i>do</i>	
<pre>do i++; while(i < 20);</pre>	<pre>do i++; while(i < 20);</pre>
l'instruction <i>for</i>	
<pre>for (int i = 2; i <= 10; i += 2) Console.WriteLine(i);</pre>	<pre>for (int i = 2; i <= 10; i += 2) System.out.println(i);</pre>



C#	JAVA
l'instruction <i>foreach</i>	
<pre>foreach (int i in numArray) sum += i;</pre>	<pre>for (int i : numArray) sum += i;</pre>
l'instruction <i>continue</i>	
<pre>for (int i = 1; i <= 10; i++) { if (i < 9) { continue; } Console.WriteLine(i); } // output : 9 10</pre>	<pre>for (int i = 1; i <= 10; i++) { if (i < 9) { continue; } Console.WriteLine(i); } // output : 9 10</pre>
l'instruction <i>return</i>	
<pre>static double CalculateArea(int r) { double area = r * r * Math.PI; return area; }</pre>	<pre>static double CalculateArea(int r) { double area = r * r * Math.PI; return area; }</pre>



C#	JAVA
les instructions <i>throw</i> et <i>try</i>	
<pre>string s = null; if (s == null) { throw new ArgumentNullException(); } try { Console.WriteLine("Try statement."); throw new NullReferenceException(); } catch (NullReferenceException e) { Console.WriteLine("{0} Except 1",e); } catch { Console.WriteLine("Except 2"); } finally { Console.WriteLine("Finally."); }</pre>	<pre>String s = null; if (s == null) { throw new NullPointerException(); } try { System.out.println("Try statement."); throw new NullPointerException(); } catch (NullPointerException e) { System.out.println(e+" Except 1"); } catch (Exception e) { System.out.println("Exception 2"); } finally { System.out.println("Finally."); }</pre>



C#	JAVA
les instructions <i>checked</i> et <i>unchecked</i>	
<pre>static int CheckedMethod() { int z = 0; try { z = checked((short)(x + y)); } catch (System.OverflowException e) { Console.WriteLine(e.ToString()); } return z; } const int x = 2147483647; // Max int const int y = 2; static void Main() { int z; unchecked { z = x * y; } Console.WriteLine("Unchecked output" + " value: {0}", z); }</pre>	<pre>static int CheckedMethod() { int z = 0; try { z = ((short)(x + y)); } catch (OutOfMemoryError e) { System.out.println(e.toString()); } return z; } // No exist unchecked in java</pre>
l'instruction <i>lock</i>	
<pre>class Foo { private volatile Helper helper = null; public Helper getHelper() { if (helper == null) { lock(this) { if (helper == null) helper = new Helper(); } } return helper; } }</pre>	<pre>class Foo { private volatile Helper helper = null; public Helper getHelper() { if (helper == null) { synchronized(this) { if (helper == null) helper = new Helper(); } } return helper; } }</pre>



C#	JAVA
l'instruction <i>using</i>	
<pre>using Com.Sodifrance.Software; using (MyClass mc = new MyClass("A MyClass Object")){ for(int i = 0; i < 10; i++){ Console.WriteLine(mc.ShowName()); } //for }</pre>	<pre>import com.sodifrance.software.*; { MyClass mc; mc = new MyClass("A MyClass Object"); for(int i = 0; i < 10; i++){ System.out.println(mc.ShowName()); } //for }</pre>



C#	JAVA
Classes	
<pre>// Accessibility keywords public private internal protected protected internal static City myCity = new City(); myCity.name = "Nantes"; myCity.citizens = 600 000; myCity.Add("Zénith"); City.Count(); // static method myCity = null; if (myCity == null) myCity = new City(); Object obj = new City(); Console.WriteLine("object'type: "); Console.WriteLine(obj.GetType(). ToString()); if (obj is City) Console.WriteLine("Is a City.");</pre>	<pre>// Accessibility keywords public private protected static City myCity = new City(); myCity.setName("Nantes"); myCity.setCitizens(600 000); myCity.Add("Zénith"); City.Count(); // static method myCity = null; if (myCity == null) myCity = new City(); Object obj = new City(); System.out.println("object'type: "); System.out.println(obj.getClass(). toString()); if (obj instanceof City) System.out.println("Is a City.");</pre>
Méthodes	
<pre>int Add(int x, int y) { return x + y; } int sum = Add(2,3); void PrintSum(int x, int y) { Console.WriteLine(x + y); } PrintSum(2,3);</pre>	<pre>int add(int x, int y) { return x + y; } int sum = add(2,3); void printSum(int x, int y) { System.out.println(x + y); } printSum(2,3);</pre>

C#	JAVA
Propriétés	
Les propriétés n'existent pas en java. Pour traduire le concept, deux étapes seront nécessaires. La première consistera à récupérer les méthodes contenues par la propriété pour modifier leur nom et les remonter d'un niveau. La seconde consiste à modifier l'appel de ces méthodes. En C#, l'appel est implicite. Après la modification effectuée nous serons obligés de rendre l'appel explicite.	
<pre>private int mSize; public int Size { get {return mSize;} set {if (value < 0) mSize = 0; else mSize = value; } } shoe.Size++;</pre>	<pre>private int size; public int getSize() {return size;} public void setSize(int value) { if (value < 0) size = 0; else size = value; } shoe.setSize(shoe.getSize()+1);</pre>
Évènement	
<pre>using System; class EvenNumberEvent: EventArgs{ internal int number; public EvenNumberEvent(int number) :base(){this.number = number;} }</pre>	<pre>import java.util.*; class EvenNumberEvent extends EventObject{ public int number; public EvenNumberEvent(Object source , int number){ super(source); this.number = number; } } interface EvenNumberSeenListener{ void evenNumberSeen(EvenNumberEvent ene); }</pre>



C#	JAVA
<pre>class Publisher{ public delegate void EvenNumberSeenHandler(object sender , EventArgs e); public event EvenNumberSeenHandler EvenNumHandler; protected void OnEvenNumberSeen(int num){ if(EvenNumHandler!= null) EvenNumHandler(this , new EvenNumberEvent(num)); } public void RunNumbers(){ Random r = new Random((int) DateTime.Now.Ticks); for(int i=0; i < 20; i++){ int current = (int) r.Next(20); Console.WriteLine("Current number is:" + current); if((current % 2) == 0) OnEvenNumberSeen(current); }//for } } //Publisher</pre>	<pre>class Publisher{ Vector subscribers = new Vector(); private void OnEvenNumberSeen(int num){ for(int i=0,size=subscribers.size(); i < size; i++) ((EvenNumberSeenListener)subscribers. get(i)).evenNumberSeen(new EvenNumberEvent(this , num)); } public void addEvenNumberEventListener(EvenNumberSeenListener ensl){ subscribers.add(ensl); } public void removeEvenNumberEventListener(EvenNumberSeenListener ensl){ subscribers.remove(ensl); } public void RunNumbers(){ Random r = new Random(System.currentTimeMillis()); for(int i=0; i < 20; i++){ int current = (int) r.nextInt() % 20; System.out.println("Current number is:" + current); if((current % 2) == 0) OnEvenNumberSeen(current); }//for } } //Publisher</pre>



C#	JAVA
<pre>public class EventTest{ public static void EventHandler(object sender , EventArgs e){ Console.WriteLine("Even Number Seen:" + ((EvenNumberEvent)e).number); } public static void Main(string[] args){ Publisher pub = new Publisher(); //register the callback/subscriber pub.EventNumHandler += new Publisher. EvenNumberSeenHandler(EventHandler); pub.RunNumbers(); //unregister the callback/subscriber pub.EventNumHandler -= new Publisher. EvenNumberSeenHandler(EventHandler); } }</pre>	<pre>public class EventTest implements EvenNumberSeenListener{ public void evenNumberSeen(EvenNumberEvent e){ System.out.println("Even Number Seen:" + ((EvenNumberEvent)e).number); } public static void main(String[] args){ EventTest et = new EventTest(); Publisher pub = new Publisher(); //register the callback/subscriber pub.addEvenNumberEventListener(et); pub.RunNumbers(); //unregister the callback/subscriber pub.removeEvenNumberEventListener(et); } }</pre>

C#	JAVA
Opérateurs	
La surcharge des opérateurs est un concept qui vient du langage C++. En java, il n'existe pas d'équivalence. Le choix est donc de recréer des méthodes avec le même comportement mais avec un appel explicite. Le nom de cette nouvelle méthode est à choisir.	
<pre>using System; class OverloadedNumber{ private int value; public OverloadedNumber(int value){ this.value = value; } public override string ToString(){ return value.ToString(); } public static OverloadedNumber operator -(OverloadedNumber number){ return new OverloadedNumber(-number.value); } public static OverloadedNumber operator +(OverloadedNumber number1, OverloadedNumber number2){ return new OverloadedNumber(number1.value + number2.value); } public static OverloadedNumber operator ++(OverloadedNumber number){ return new OverloadedNumber(number.value + 1); } }</pre>	<pre>class OverloadedNumber{ private int value; public OverloadedNumber(int value){ this.value = value; } public String toString(){ return "" + value; } public static OverloadedNumber minus(OverloadedNumber number){ return new OverloadedNumber(-number.value); } public static OverloadedNumber plus(OverloadedNumber number1, OverloadedNumber number2){ return new OverloadedNumber(number1.value + number2.value); } public static OverloadedNumber inc(OverloadedNumber number){ return new OverloadedNumber(number.value + 1); } }</pre>



C#	JAVA
<pre>public class OperatorOverloadingTest { public static void Main(string [] args){ OverloadedNumber number1; number1 = new OverloadedNumber(12); OverloadedNumber number2; number2 = new OverloadedNumber(125); Console.WriteLine("Inc:{0}", ++number1); Console.WriteLine("Add:{0}", number1 + number2); } } // OperatorOverloadingTest</pre>	<pre>public static void main(String [] args){ OverloadedNumber number1; number1 = new OverloadedNumber(12); OverloadedNumber number2; number2 = new OverloadedNumber(125); System.out.println("Inc: {0}" + inc(number1)); System.out.println("Add: {0}" + plus(number1, number2)); } } // OperatorOverloadingTest</pre>



C#	JAVA
Indexeurs	
Les indexeurs sont des méthodes permettant de créer un type ensembliste. Ces méthodes redéfinissent les crochets. La traduction en java effectue une modification de déclaration des ces objets mais aussi l'appel des fonctions.	
<pre>using System; using System.Collections; public class IndexerTest: IEnumerable, IEnumerator { private Hashtable list; public IndexerTest () { index = -1; list = new Hashtable(); } //indexer that indexes by number public object this[int column]{ get{return list[column];} set{list[column] = value;} } /* indexer that indexes by name */ public object this[string name]{ get{return this[ConvertToInt(name)];} set{this[ConvertToInt(name)]=value;} } }</pre>	<pre>import java.util.Hashtable; public class IndexerTest { private Hashtable<Integer, Object> list; public IndexerTest() { index = -1; list= new Hashtable<Integer, Object>(); } // indexer that indexes by number public Object getObject(int column) { return list.get(column);} public void setObject(int column, Object value) { list.put(column, value);} /* indexer that indexes by name */ public Object getObject(String name) { return getObject(ConvertToInt(name));} public void setObject(String name, Object value) { setObject(ConvertToInt(name), value);} }</pre>



C#	JAVA
<pre>/* Convert strings to integer */ private int ConvertToInt(string value){ string loVal = value.ToLower(); switch(loVal){ case "zero": return 0; case "one": return 1; case "two": return 2; case "three": return 3; case "four": return 4; case "five": return 5; default: return 0; } return 0; } /** Needed to implement * IEnumerable interface. */ public IEnumerator GetEnumerator() { return (IEnumerator) this; } /** Needed for IEnumerator. */ private int index; /** Needed for IEnumerator. */ public bool MoveNext(){ index++; if(index >= list.Count) return false; else return true; } /** Needed for IEnumerator. */ public void Reset(){index = -1;} /** Needed for IEnumerator. */ public object Current{ get{return list[index];} }</pre>	<pre>/* Convert strings to integer */ private int ConvertToInt(String value){ String val = value.toLowerCase(); if (val.equals("zero"))return 0; else if (val.equals("one"))return 1; else if (val.equals("two"))return 2; else if (val.equals("thre"))return 3; else if (val.equals("four"))return 4; else if (val.equals("five"))return 5; else return 0; } private int index; public boolean MoveNext() { index++; if (index >= list.size()) return false; else return true; } public void Reset() {index = -1;} public Object getCurrent() { return list.get(index); }</pre>



C#	JAVA
<pre>public static void Main(string[] args){ IndexerTest it = new IndexerTest(); it[0] = "A"; it[1] = "B"; it[2] = "C"; it[3] = "D"; it[4] = "E"; Console.WriteLine("it[0]=" + it[0]); Console.WriteLine("it[\"Three\"] = " + it["Three"]); foreach(string str in it){ Console.WriteLine(str); } } } // IndexerTes</pre>	<pre>public static void main(String[] args){ IndexerTest it = new IndexerTest(); it.setObject(0, "A"); it.setObject(1, "B"); it.setObject(2, "C"); it.setObject(3, "D"); it.setObject(4, "E"); System.out.println("it[0]=" + it.getObject(0)); System.out.println("it[\"Three\"]=" + it.getObject("Three")); for(int i=0;i < it.list.size();i++) { System.out.println(it.getObject(i)); } } } // IndexerTes</pre>
Constructeurs	
<pre>class City { private int citizens; public City() { citizens = 0; } public City(int nbCitizens) { this.citizens = nbCitizens; } }</pre>	<pre>class City { private int citizens; public City() { citizens = 0; } public City(int nbCitizens) { this.citizens = nbCitizens; } }</pre>
Destructeurs	
<pre>class City { private int citizens; ~City() { } }</pre>	<pre>class City { protected void finalize() throws Throwable { super.finalize(); } }</pre>



C#	JAVA
Constructeur statique	
<pre>using Personnel.Data ; class Employee { private static DataSet ds; static Employee() { ds = new DataSet (...); } public string Name; public decimal Salary; . . . }</pre>	<pre>import Personnel.Data.*; class Employee { private static DataSet ds; Employee() { ds = new DataSet (); } public String Name; public double Salary; ... }</pre>
l'héritage	
<pre>class TransformingRobot : IRobot, IVehicle{ string model; string make; short year; string name; }</pre>	<pre>class TransformingRobot implements Robot, Vehicle { String model; String make; short year; String name; }</pre>



C#	JAVA
Structures	
<pre>struct StudentRecord { public string name; public float gpa; public StudentRecord(string name, float gpa) { this.name = name; this.gpa = gpa; } }</pre> <pre>StudentRecord stu; stu = new StudentRecord("Bob",3.5f); StudentRecord stu2 = stu; stu2.name = "Sue"; Console.WriteLine(stu.name);</pre>	<pre>class StudentRecord { public string name; public float gpa; public StudentRecord(string name, float gpa) { this.name = name; this.gpa = gpa; } }</pre> <pre>StudentRecord stu; stu = new StudentRecord("Bob",3.5f); StudentRecord stu2 = stu; stu2.name = "Sue"; System.out.println(stu.name);</pre>
Interfaces	
<pre>interface IAlarmClock { ... } interface IAlarmClock : IClock { ... }</pre>	<pre>interface IAlarmClock { ... } interface IAlarmClock extends IClock { ... }</pre>

C#	JAVA
Délégués	
Les délégués sont en C# des pointeurs de fonction. En java, ce concept n'existe pas. Comme le pointeur de fonction n'existe pas, une solution est de remplacer directement le pointeur par l'appel de la méthode elle-même. Cela pose un problème, lorsque ce pointeur est un paramètre d'une méthode. C'est pourquoi, on utilise deux objets java. Une interface Delegate et une classe Delegator.	
<pre>using System; public class Class1 { public void show(string s) { Console.WriteLine(s); } } public class Class2 { public void display(string s) { Console.WriteLine(s); } } public class Class3 { public static void staticDisplay(string s) { Console.WriteLine(s); } }</pre>	<pre>class Class1 { public void show(String s) { System.out.println(s); } } class Class2 { public void display(String s) { System.out.println(s); } } class Class3 { public static void staticDisplay(String s) { System.out.println(s); } }</pre>



C#	JAVA
<pre> public class TestDelegate { // define a type as a method taking // a string returning void public delegate void doShow(string s); public static void Main(string[] args) { // make an array of these methods doShow[] items = new doShow[3]; items[0] = new doShow(new Class1(). show); items[1] = new doShow(new Class2(). display); items[2] = new doShow(Class3. staticDisplay); // call all items the same way for(int i = 0; i < items.Length; i++){ items[i]("Hello World"); } } } </pre>	<pre> public class TestDelegate { public static final Class[] OutputARG; OUTPUT_ARG = { String.class }; public static final Class OutputRET; OUTPUT_RETURN = Void.TYPE; public final static Delegator DO_SHOW = new Delegator(OUTPUTARG,OUTPUTRET); public static void main(String[] args){ Delegate[] items = new Delegate[3]; items[0] = DO_SHOW.build(new Class1(), "show"); items[1] = DO_SHOW.build(new Class2(), "display"); items[2] = DO_SHOW.build(Class3.class, "staticDisplay"); for(int i = 0; i < items.length; i++){ items[i].invoke("Hello World"); } } } </pre>
Énumération	
<pre> enum Action{Start,Stop,Rewind,Forward}; enum Status { Flunk = 50,Pass = 70,Excel = 90 }; Action a = Action.Stop; if (a != Action.Start) Console.WriteLine(a); Status s = Status.Pass; Console.WriteLine((int)s); </pre>	<pre> enum Action{Start,Stop,Rewind,Forward}; enum Status { Flunk(50), Pass(70), Excel(90); private final int value; Status(int value) {this.value = value;} public int value() {return value;} }; Action a = Action.Stop; if (a != Action.Start) System.out.println(a); Status s = Status.Pass; Console.WriteLine(s.value()); </pre>



C#	JAVA
Attributs C#	
<pre>using System; using System.Reflection; [AttributeUsage(AttributeTargets.Class)] public class AuthorInfoAttribute : System.Attribute{ string author; // with get string email; // with get string version; // with get et set public AuthorInfoAttribute(string author, string email){ this.author = author; this.email = email; } }</pre>	<pre>import java.lang.annotation.*; import java.lang.reflect.*; @Documented // we want the annotation to show up // in the Javadocs @Retention(RetentionPolicy.RUNTIME) // we want annotation metadata to be // exposed at runtime @interface AuthorInfo{ String author(); String email(); String version() default "1.0"; }</pre>



C#	JAVA
<pre>[AuthorInfo("Dare Obasanjo", "kpako@yahoo.com", Version="1.0")] class HelloWorld{ } class AttributeTest{ public static void Main(string[] args) { /* Get Type object of this class */ Type t = typeof(HelloWorld); foreach(AuthorInfoAttribute att in t.GetCustomAttributes(typeof(AuthorInfoAttribute), false)){ Console.WriteLine("Author: "); Console.WriteLine(att.Author); Console.WriteLine("Email: "); Console.WriteLine(att.Email); Console.WriteLine("Version:"); Console.WriteLine(att.Version); } //foreach } //Main }</pre>	<pre>@AuthorInfo(author="Dare Obasanjo", email="kpako@yahoo.com") class HelloWorld{ } public class Test{ public static void main(String[] args) throws Exception{ /* Get Class object of this class */ Class c= Class.forName("HelloWorld"); AuthorInfo a = (AuthorInfo) c.getAnnotation(AuthorInfo.class); System.out.println("Author: "); System.out.println(a.author()); System.out.println("Email: "); System.out.println(a.email()); System.out.println("Version: "); System.out.println(a.version()); } }</pre>



C#	JAVA
Quelques informations pratiques la manipulation des strings	
<pre>// String concatenation string school = "Harding "; // school is "Harding University" school = school + "University"; // String comparison string mascot = "Bisons"; if (mascot == "Bisons") // true if (mascot.Equals("Bisons")) // true if (mascot.ToUpper().Equals("BISONS")) // true if (mascot.CompareTo("Bisons") == 0) // true Console.WriteLine(// Prints "son" mascot.Substring(2,3)); // My birthday: Oct 12, 1973 DateTime dt = new DateTime(1973,10,12); string s = "My birthday: " + dt.ToString("MMM dd, yyyy"); // Mutable string System.Text.StringBuilder buffer = new System.Text.StringBuilder("two "); buffer.Append("three "); buffer.Insert(0, "one "); buffer.Replace("two","TWO"); // Prints "one TWO three" Console.WriteLine(buffer);</pre>	<pre>// String concatenation String school = "Harding "; // school is "Harding University" school = school + "University"; // String comparison String mascot = "Bisons"; // Not the correct way if (mascot == "Bisons") // to do string comparisons if (mascot.equals("Bisons")) // true if (mascot.equalsIgnoreCase("BISONS")) // true if (mascot.compareTo("Bisons") == 0) // true System.out.println(// Prints "son" mascot.substring(2, 5)); // My birthday: Oct 12, 1973 java.util.Calendar c = new java. util.GregorianCalendar(1973,10,12); String s = String.format("My birthday: %1\$tb %1\$te, %1\$tY", c); // Mutable string StringBuffer buffer = new StringBuffer("two"); buffer.append("three "); buffer.insert(0, "one "); buffer.replace(4, 7, "TWO"); // Prints "one TWO three" System.out.println(buffer);</pre>



C#	JAVA
les consoles d'entrée - sortie	
<pre>Console.Write("What's your name? "); string name = Console.ReadLine(); Console.Write("How old are you? "); int age = Convert.ToInt32(Console.ReadLine()); Console.WriteLine("{0} is {1} years old.", name, age); // or name + " is " + age + " years old."); // Read single char int c = Console.Read(); // Prints 65 if user enters "A" Console.WriteLine(c); // Studio costs 499.00 for 3months. Console.WriteLine("The {0}costs {1:C} for {2} onths.", "studio", 499.0, 3); // Today is 06/25/2004 Console.WriteLine("Today is " + DateTime.Now.ToShortDateString());</pre>	<pre>java.io.DataInput in = new java.io.DataInputStream(System.in); System.out.print("What is your name?"); String name = in.readLine(); System.out.print("How old are you? "); int age = Integer.parseInt(in.readLine()); System.out.println(name + " is " + age + " years old."); // Read single char int c = System.in.read(); // Prints 65 if user enters "A" System.out.println(c); // Studio costs 499.00 for 3months. System.out.printf("The %s costs \$%.2f for %d months.%n", "studio", 499.0, 3); // Today is 06/25/04 System.out.printf("Today is %tD\n", new java.util.Date());</pre>



C#	JAVA
lecture et écriture de fichiers	
<pre>using System.IO; // Character stream writing StreamWriter writer = File.CreateText("c:\\myfile.txt"); writer.WriteLine("Out to file."); writer.Close(); // Character stream reading StreamReader reader = File.OpenText("c:\\myfile.txt"); string line = reader.ReadLine(); while (line != null) { Console.WriteLine(line); line = reader.ReadLine(); } reader.Close(); // Binary stream writing BinaryWriter out = new BinaryWriter(File.OpenWrite("c:\\myfile.dat")); out.Write("Text data"); out.Write(123); out.Close(); // Binary stream reading BinaryReader in = new BinaryReader(File.OpenRead("c:\\myfile.dat")); string s = in.ReadString(); int num = in.ReadInt32(); in.Close();</pre>	<pre>import java.io.*; // Character stream writing FileWriter writer = new FileWriter("c:\\myfile.txt"); writer.write("Out to file.\n"); writer.close(); // Character stream reading FileReader reader = new FileReader("c:\\myfile.txt"); BufferedReader br = new BufferedReader(reader); String line = br.readLine(); while (line != null) { System.out.println(line); line = br.readLine(); } reader.close(); // Binary stream writing FileOutputStream out = new FileOutputStream("c:\\myfile.dat"); out.write("Text data".getBytes()); out.write(123); out.close(); // Binary stream reading FileInputStream in = new FileInputStream("c:\\myfile.dat"); byte buff[] = new byte[9]; // Read first 9 bytes into buff in.read(buff, 0, 9); String s = new String(buff); // Next is 123 int num = in.read(); in.close();</pre>

G C# vs Java : Les mots clés

C#	Java	C#	Java
abstract break byte case catch char class const continue default do double else false finally float for goto if	abstract break case catch char class const continue default do double else false finally float for goto if	int interface long new null private protected public return short static switch this throw true try void volatile while	int interface long new null private public return short static switch this throw true try void volatile while
as base bool checked decimal delegate enum event explicit extern fixed foreach implicit int internal is lock namespace object operator	 super boolean native for protected instanceof synchronised package // Java : Object	out override params readonly ref sbyte sealed sizeof stackalloc string struct typeof uint ulong unchecked unsafe ushort using virtual	 byte final // Java : String import

H Un résumé des plateformes .NET & J2EE

Les plateformes .Net et J2EE sont en concurrence, le tableau qui suit en résume les principales caractéristiques.

	.Net	J2EE
PROPRIÉTAIRES	Microsoft	Sun
DATE DE CRÉATION	2001	1998
STATUT	Produit	Spécification (de nombreuses implémentation)
LANGAGE PRINCIPAL	C# (mais il en existe d'autres)	Java (le seul)
P-CODE	MSIL	bytecode
MACHINE VIRTUELLE ¹	CLR	JVM
Classes de base	Base Class Library	Java 2 Platform API
APPLICATIONS WEB ²	Active Server Pages (ASP.NET) WebForms	Java Server Pages (JSP)
SERVICES WEB ³	Services Web ADO.NET	Enterprise JavaBeans JDBC (Apache-SOAP)
GRAPHIQUE ⁴	Windows.Forms, Web.Forms	Java Swing, AWT
IDE	Visual Studio.NET	Eclipse
ARCHITECTURES SUPPORTÉES ⁵	Windows	Nombreuses (Windows, Solaris, AIX, Linux, Amiga ...)
SUPPORT DES TERMINAUX MOBILES	.NET Mobile SDK limité à Windows CE pour l'instant.	Java 2 Micro Edition (J2ME).
DÉVELOPPEMENT MULTI-LANGAGE	Plus de 20 langages annoncés.	Langage Java uniquement

¹La JVM est un interpréteur de bytecode, le CLR compile le code IL systématiquement en code natif.

²Les WebForms apportent une couche d'abstraction par rapport au langage cible de présentation (HTML, WML ...) dont JSP ne dispose pas.

³L'architecture J2EE est moins ouverte de par sa dépendance envers Java (RMI, messagerie Java).

⁴Swing fonctionne de manière presque similaire sur toutes les plates-formes cibles. WinForms est pour l'instant restreint à Windows uniquement.

⁵Les soumissions à l'ECMA permettent au framework .NET un portage par des tiers, exemple le projet Mono.

Les dix arguments pour décider de choisir .NET [31] :

1. Intégration de standards : XML, SOAP et d'autres
2. Facilité de déploiement
3. Support des services web
4. Jeu d'outils standard pour tout langage .NET
5. Support des appareils mobiles
6. Codé géré
7. Indépendance de la plate-forme
8. Abondance des ressources d'apprentissage
9. Langages modernisés
10. Types de base standard entre les langages

Les arguments concernant le choix de Java sont nombreux aussi. Pour résumer les pro-Java, les avantages qu'offre Java sur la plateforme .NET sont

1. les machines virtuelles non limitées à un système d'exploitation.
2. J2EE est élaborée par le Java Community Process (un organisme dirigé par Sun qui regroupe plus d'une centaine d'éditeurs dont IBM, BEA ou encore Oracle)
3. Java est le langage préféré de l'open source

Aucune de ces plateformes gagnent. Chacune a ses adeptes et ses avantages. C'est l'existant qui décidera du meilleur des deux. Un système informatique qui repose sur un système Microsoft sera intégrable beaucoup plus facilement dans un environnement J2EE. Et vice versa.

J2EE et .NET proposent des mécanismes très similaires. Il convient de choisir la meilleure technologie par rapport à un ensemble d'exigences comme la portabilité, la culture d'entreprise, l'existant ...

Résumé :

L'informatique d'aujourd'hui évolue très vite. Les entreprises qui développent des logiciels ne veulent plus se laisser "emprisonner" par un seul langage. Elles veulent créer et pouvoir transformer leur logiciel dans le nouveau langage à la mode.

Pour arriver à de telles applications, il est nécessaire de scanner les fichiers sources, de les analyser puis de les transformer. Ainsi, mon travail durant ce stage a été de scanner les sources C# puis de les charger dans un référentiel. A partir de ce dernier, j'ai travaillé sur une solution de migration du langage C# vers le langage Java.

Une partie importante de mon travail fut l'étude de ce nouveau langage.

Mots clefs :

Théorie des langages, Rétro ingénierie, C#, .Net, Parseur, Grammaire