HAL
open science

Étude d'une solution de génération d'IHM pour la mise à jour des données de référence utilisant l'approche ORM

Aurélien Tournier

To cite this version:

Aurélien Tournier. Étude d'une solution de génération d'IHM pour la mise à jour des données de référence utilisant l'approche ORM. Système d'exploitation [cs.OS]. 2012. dumas-01081288

HAL Id: dumas-01081288

https://dumas.ccsd.cnrs.fr/dumas-01081288v1

Submitted on 7 Nov 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

MEMOIRE

présenté en vue d'obtenir

le DIPLOME D'INGENIEUR CNAM

SPECIALITE : INFORMATIQUE

OPTION : INFORMATIQUE, SYSTÈMES D'INFORMATION

par

Aurélien TOURNIER

Etude d'une solution de génération d'IHM pour la mise à jour des données de référence utilisant l'approche ORM

Soutenu le 13 Juin 2012

JURY

PRESIDENT : Monsieur Christophe PICOULEAU (Cnam Paris)

MEMBRES : Monsieur Bertrand DAVID (Cnam Lyon)
Monsieur Claude GENIER (Cnam Lyon)
Monsieur Christophe CARAT (Algorys)
Monsieur David SARTRE (Algorys)

Remerciements

Je tiens à remercier Bertrand DAVID mon responsable de mémoire et Professeur des Universités en Informatique à l'Ecole Centrale de Lyon. Ses recommandations lors de la réalisation de ce mémoire ont été d'une grande aide.

Je désire également adresser mes remerciements à Eléonore GONDEAU ma responsable pédagogique pour son soutien tout au long de la réalisation de ce mémoire.

Je remercie également Christophe CARAT directeur de la société Algorys de m'avoir permis de réaliser ce mémoire dans les conditions les plus optimum et pour la confiance qu'il m'a portée durant toute cette période. Je remercie aussi David SARTRE ingénieur d'études de la société Algorys pour ces conseils techniques lors du développement de l'outil.

Glossaire des termes techniques

Design pattern (Patron de conception) : Le patron de conception est un modèle de solution générique à des problèmes de conceptions récurrents en programmation.

Source : http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

Framework : Ensemble de composants logiciels servant à créer les fondations ainsi que les grandes lignes de tout ou partie d'un logiciel.

Source : http://w3.olf.gouv.qc.ca/terminologie/fiches/8872480.htm

Espace de nom (namespace) : Les espaces de noms sont des conteneurs d'éléments servant à fournir un contexte pour les types (classes, interface, structure ...) et donc d'organiser le code. Cela permet d'éviter la collision des noms.

Source : http://msdn.microsoft.com/en-us/library/z2kcy19k(v=vs.80).aspx

GNU LGPL : GNU Lesser General Public License : La licence publique générale limitée GNU, en français, est utilisée par les logiciels libres permettant leur réutilisation dans le cadre de développement de logiciel propriétaire.

Source : http://www.gnu.org/licenses/lgpl.html

IHM : L'interface homme-machine, interaction humain-machine (IHM), intégration homme-système (IHS) ou interface personne-machine (IPM) définit, les moyens et outils mis en œuvre, afin qu'un humain puisse contrôler et communiquer avec une machine par exemple un ordinateur.

Source : http://old.sigchi.org/cdg/cdg2.html

Lazy loading (Chargement tardif) : Design pattern permettant de différer le chargement des données jusqu'au moment nécessaire.

Source : http://medlibrary.org/medwiki/Lazy_loading

Bibliothèque logicielle: Collection de ressources utilisées pour développeur des logiciels. Elle peut contenir des méthodes, classes, valeurs ou des spécifications de type.

Source : http://en.wikipedia.org/wiki/Library_(computing)

ORM (Correspondance objet-relationnel) : L' « object-relational mapping » (ORM) est un mécanisme qui permet d'accéder et de manipuler des objets sans avoir la connaissance du lien entre les objets et leurs sources de données.

Source : http://searchwindevelopment.techtarget.com/definition/object-relational-mapping

SGBD (Système de gestion de base de données) : Ensemble de logiciels utilisé pour la gestion d'une de base de données.

Source : http://www.ird.fr/informatique-scientifique/documents/sgbd/sql/formation_SGBD_cours_SGBD.pdf

TMA (Tierce Maintenance Applicative) : la maintenance des applications, c'est-à-dire confier tout ou partie de la maintenance des applications à un prestataire informatique tiers contre rémunération en définissant des niveaux de disponibilité, de délais et de qualité.

Source : http://www.commentcamarche.net/contents/entreprise/tma-tierce-maintenance-applicative.php3

Table des matières

Remerciements 1
Glossaire des termes techniques 2
Table des matières 4
Introduction 6
1. Cadrage du projet 11
1.1. Initialisation du projet 11
1.1.1. Nature et objectif du projet 11
1.1.2. Livrables 11
1.1.3. Indicateurs de réussite 11
1.1.4. Acteurs 12
1.1.5. Planning 12
1.2. Gestion de projet 12
1.2.1. Rappel de l'objectif 12
1.2.2. Cahier des charges 13
1.2.3. Gestion de la qualité 13
2. Recherche de solutions 18
2.1. Object-relational mapping 18
2.1.1. Approche n-uplet 18
2.1.2. Approche par entité 19
2.1.3. Approche par modèle de domaine 20
2.1.4. LINQ To SQL 21
2.1.5. Entity Framework 22
2.1.6. NHibernate 25
2.1.7. Subsonic 28
2.1.8. Comparaison 29
2.2. Génération de code source 30
2.2.1. Moteur de génération personnalisé 30
2.2.2. Text Templates Transformation Toolkit 30
2.2.3. XML & XSLT 32
2.2.4. CodeDOM 34
2.2.5. Comparaison 39
2.3. Conception d'IHM 40
2.3.1. Modèle Présentation Abstraction Contrôle 40
2.3.2. Monolithique 41
2.3.3. Avec modèle 41
2.3.4. Génération à partir de métadonnées 42
2.3.5. Design Patterns : IHM 43
2.3.6. Solution retenue 44
2.4. Bilan de la recherche 48
3. Réalisation et tests 50
3.1. Architecture logique de l'outil 50
3.2. Paramétrage de l'outil 52
3.3. IHM de l'outil 53
3.4. Configuration d'une application ASP.NET 53

3.5. Génération ORM 54
3.5.1. Interrogation du dictionnaire de données 54
3.5.2. Modélisation des métadonnées 55
3.5.3. Modélisation du code source (CodeDOM) 57
3.5.4. Description des classes générées 58
3.6. Génération IHM 60
3.6.1. Composants utilisés 60
3.6.2. Architecture 61
3.6.3. Principe de génération et de mise en œuvre 62
3.6.4. Fonctionnement du contrôle personnalisé 63
3.6.5. Génération des fichiers 65
3.6.6. UIGenerator 65
3.6.7. MetaDataGenerator 67
3.7. Tests 68
Conclusion 71
Bibliographies 72
Ouvrages 72
Sites internet 72
Table des annexes 75
Table des figures 83
Liste des tableaux 85
Résumé 87

Introduction

Le but de ce document est de présenter le travail que j'ai effectué dans le cadre du mémoire de fin d'études du Conservatoire National des Arts & Métiers de Lyon. Ce travail a porté sur la conception d'une solution de génération d'IHM pour la mise à jour des données de référence basé sur l'approche «ORM».

Le travail de mémoire a été effectué au sein d'ALGORYS, société de services en ingénierie informatique spécialisée dans le conseil et l'ingénierie des systèmes d'informations. Fondée en 1998 par Christophe CARAT et Louis-Manuel NAVENANT, elle compte 15 employés (voir Figure 1) et a réalisé un chiffre d'affaires de 1 430 K€ en 2011 (voir Figure 2).

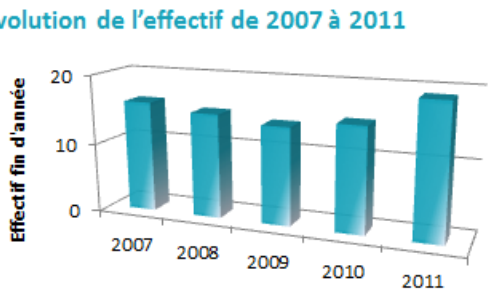

Figure 1 - Evolution des effectifs d'Algorys

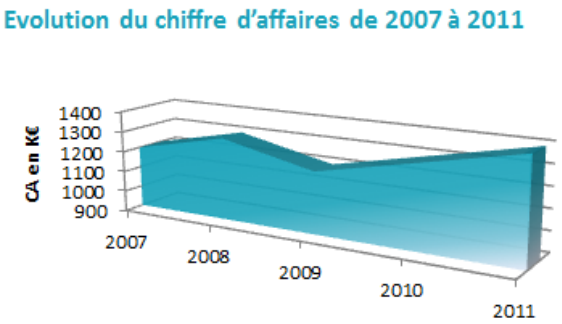

Figure 2 - Evolution du chiffre d'affaire d'Algorys

Le périmètre géographique d'ALGORYS s'étend sur les départements du Rhône, de la Loire ainsi que de l'Isère. Son activité commerciale et technique est structurée en pôles de spécialisation (voir Figure 3) :

- Pôle Audit & Consulting
- Pôle Sécurité des systèmes d'information & Administration des systèmes et réseaux
- Pôle Etudes et Développement

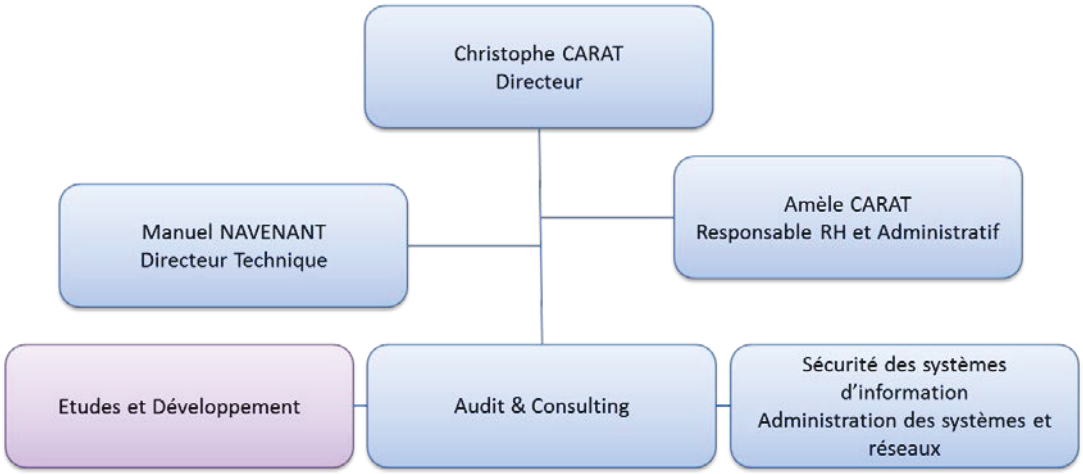

Figure 3 - Organigramme général Algorys 2011

L'étude s'est déroulée au sein du pôle *Etudes et Développement* (voir Figure 3) dont les missions sont multiples :

- Développement d'applications de gestion de l'entreprise : suivi de commandes, gestion de stocks, gestion commerciale, gestion des temps, gestion des ressources, …
- Mise en œuvre d'Intranet et de flux de travail
- Intégration des systèmes d'information : développement d'interfaces, mise en œuvre de solutions d'ETL/EAI, XML, ...
- Portage des applications sur un nouvel environnement
- Tierce Maintenance Applicative

ALGORYS est certifiée Microsoft Gold Partenaire depuis 2000. Ce partenariat avec l'éditeur est important puisqu'il influence les choix technologiques de la société et par conséquent ceux de ce projet. Pour ALGORYS, comme pour toute entreprise, la notion de gain de productivité est un enjeu majeur. Notre travail avait donc pour finalité d'optimiser le processus de développement d'applications, principale activité du pôle *Etudes & Développement*.

Ainsi ce mémoire présente l'étude et la réalisation d'un outil d'aide au développement dont l'objectif était de répondre à la problématique suivante : « *Comment améliorer la productivité du développeur dans la réalisation de programmes et notamment le sous ensemble associé à la mise à jour des données de référence* ». Par « mise à jour » s'entend la récupération, l'ajout, la modification et la suppression des données.

Dans l'utilisation normale d'une application de gestion, on distingue souvent deux catégories ou groupes de personnes ayant des droits d'accès différents.

- **L'utilisateur classique**
  Il a accès aux écrans de saisie et alimente les données principales de l'application (exemple : les commandes dans un logiciel de gestion des achats).
- **L'utilisateur avancé**
  En plus de l'accès aux mêmes formulaires que l'utilisateur classique, l'utilisateur avancé a la possibilité de modifier les données de référence (exemple : les taux de TVA dans un logiciel de gestion des achats).

Le périmètre fonctionnel du projet est limité aux données qui évoluent peu dans le processus classique de l'utilisation d'un logiciel de gestion. Ces dernières sont nommées « données de référence ». Elles sont utilisées pour structurer les données opérationnelles. Seules les personnes disposant d'un niveau élevé de droit dans les applications ont accès à ces données (voir Figure 4).

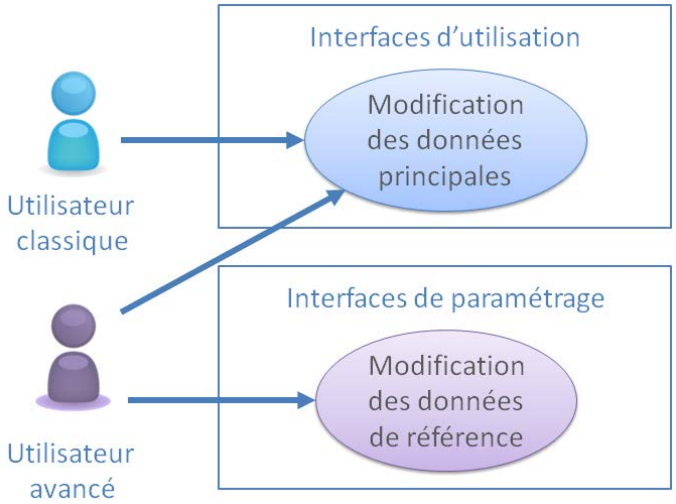

Figure 4 - Diagramme des cas d'utilisation des applications de gestion

Dans les projets de développement d'applications, et plus spécifiquement dans la réalisation du code, deux aspects peuvent être distingués :

- La réalisation du code lié aux formulaires du cœur de l'application, que l'on appelle ici Formulaire « d'utilisation » (exemple : Saisie d'une commande).
- La réalisation du code lié aux formulaires de mise à jour des données de référence, que l'on appelle ici Formulaire « de paramétrage » (exemple : Gestion des taux de TVA).

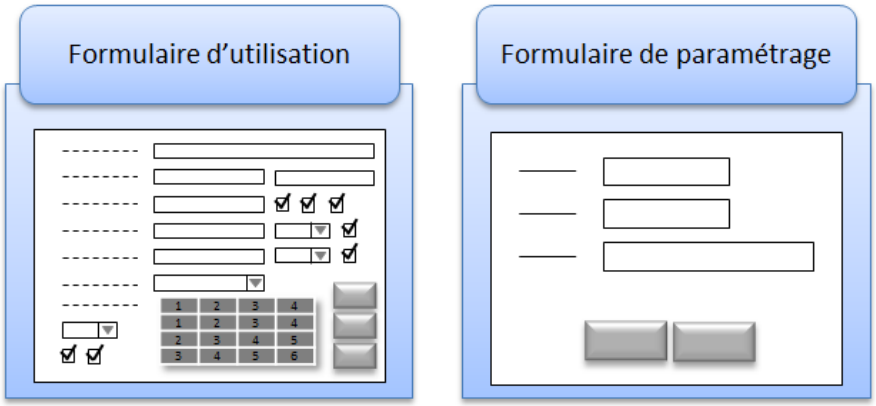

Figure 5 - Exemple de formulaire d'utilisation et de paramétrage

Le choix du périmètre de la mise à jour des données de référence se justifie par le fait que les formulaires de paramétrages ont, par leur nature, une possibilité de factorisation plus élevée que les formulaires d'utilisations.

Dans l'étude de la solution deux concepts importants sont pris en compte :

- Architecture n-tiers
- ORM : Object-Relational mapping

Architecture n-tiers

L'architecture n-tiers est un modèle logique d'architecture applicative dont le principe est de séparer en plusieurs niveaux ou couches l'organisation d'une application ou d'un système. Comme le montre la Figure 6, l'architecture 3-tiers est la mise en œuvre la plus répandue du modèle d'architecture n-tiers. Dans ce cas les 3 niveaux sont les suivants :

- Données : Accès et stockage des données.
- Application : Traitement des règles de gestion et de la logique applicative.
- Présentation : Gestion de l'interface homme-machine (IHM).

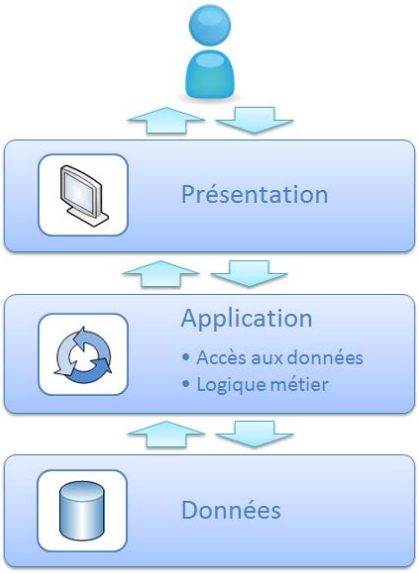

Figure 6 - Architecture 3-tiers

L'avantage d'un tel modèle est l'indépendance technologique de chacun des tiers, ce qui amène une amélioration de la modularité ainsi qu'une simplification de la maintenance et de l'évolution.

La gestion de la persistance des données étant généralement effectuée par la base de données, nous nous préoccupons uniquement des niveaux Application et IHM.

La couche Application peut être divisée en deux sous-ensembles :

- La couche métier (BLL, Business Logic Layer) : comprenant les objets métier (Business Objects) ainsi que leurs règles de gestion.
- La couche d'accès aux données (DAL, Data Access Layer) : contenant les méthodes d'accès et de modification des données persistantes.

Object-relational mapping

Une des techniques pour la réalisation d'une couche d'accès aux données est la correspondance objet-relationnel (ORM, object-relational mapping). Elle consiste à effectuer la conversion entre les types utilisés dans la base de données et ceux du modèle objet. La base de données est alors vue comme une collection d'objets à travers la couche ORM. Cette technique est mise en œuvre au travers d'outils générant automatiquement cette couche en se basant sur la base de données et sur des schémas de modélisation des objets.

Ce projet consistait donc à fournir un outil capable de créer des formulaires pour le périmètre de la gestion des données de référence en s'appuyant sur la technique d'ORM.

Mon rôle dans ce projet était d'organiser et de planifier sa réalisation et son bon déroulement. Une fois la gestion de projet initialisée, j'ai effectué une étude sur les techniques d'ORM et le modèle de conception d'IHM. Puis, l'étude réalisée, j'ai déterminé l'architecture à générer et l'ai conceptualisée. Ensuite j'ai conceptualisé l'architecture du programme de génération et défini les procédures de tests et de recettes. Après avoir réalisé le programme, j'ai supervisé le déploiement et les tests.

Le premier chapitre « Cadrage du projet » de ce mémoire est consacré à la définition du projet. Tout d'abord 4 éléments d'initialisation sont présentés dans un sous chapitre. La nature et les objectifs du projet sont clairement définis. Ensuite les livrables sont présentés puis les indicateurs de réussites sont déterminés et enfin les acteurs du projet sont identifiés. Le sous chapitre suivant est dédié à la gestion de projet. Plusieurs éléments sont mis en lumière notamment le rappel de l'objectif détaillé, le cahier des charges et la gestion de la qualité qui passe par la planification des tâches, la gestion des codes sources et la gestion des tests. Le deuxième chapitre intitulé « Recherche de solutions» présente l'étude et l'analyse des trois grands thèmes suivants, la correspondance objet relationnelle, la génération de codes source et la conception d'IHM. Pour chaque thème les principaux concepts sont passés en revue et les différentes techniques ou produits sont comparés. Le troisième chapitre intitulé « **Erreur ! Source du renvoi introuvable.** » détaille la présentation générale de l'outil à travers son architecture, son utilisation et sa configuration. Puis il précise les deux aspects de génération d'ORM et d'IHM en passant par la présentation de l'architecture mise en place et son fonctionnement. Le quatrième et dernier chapitre dresse le bilan de ce projet de mémoire.

1. Cadrage du projet

1.1. Initialisation du projet

Toute démarche visant à organiser le bon déroulement d'un projet commence par une phase d'initialisation. Durant cette phase, plusieurs réunions ont été effectuées afin de préciser le besoin et le cadre technique du projet. Ce processus a déterminé un certain nombre d'éléments :

- Le cahier des charges
  - la nature et les objectifs du projet
  - les livrables du projet
- les indicateurs qui permettent d'établir si le projet atteint ses objectifs
- les acteurs du projet
- le planning

1.1.1. Nature et objectif du projet

Le projet était de type création d'un outil d'aide au développement. Il s'inscrivait dans le cadre d'un projet de type « recherche et développement » au sein de l'entreprise. Son but était d'améliorer la productivité d'un développeur dans le cadre d'un développement, en s'occupant de plusieurs parties d'un programme. Ces dernières étaient : la couche « Application » en utilisant l'approche ORM et un sous ensemble de la couche « Présentation » en proposant des formulaires de l'IHM pour la mise à jour des données de référence.

1.1.2. Livrables

Les livrables étaient naturellement l'outil de génération (code source et l'exécutable), mais également les documents suivants :

- les spécifications techniques décrivant les besoins du projet sous forme de critères techniques.
- les spécifications fonctionnelles décrivant les besoins du projet sous forme de critères fonctionnels.
- la documentation utilisateur guidant l'utilisateur du programme.

Ces documents sont importants. En effet, ils établissent les spécificités du projet et autorisent, par la suite son évolution ou sa modification. La livraison du projet a été considérée comme terminée lorsque tous les éléments décrits ci-dessus ont été livrés.

1.1.3. Indicateurs de réussite

Le premier des indicateurs est le dossier de recette établi au début du projet en fonction des spécifications fonctionnelles. Il vise à s'assurer que le résultat de l'exécution des scénarii produit bien les effets désirés en termes fonctionnels.

Une fois la réalisation de l'application terminée, sa mise en œuvre en condition réelle, c'est-à-dire dans le cadre d'un développement d'une application pour un client, apporte ses retours quant aux fonctionnalités manquantes ou posant des problèmes d'utilisation.

Un autre indicateur de réussite est le temps économisé lors du développement d'une application grâce à l'outil par rapport à un développement classique.

1.1.4. Acteurs

Le projet était dirigé par Christophe CARAT Directeur de la solution Algorys. Il a assuré le rôle de directeur technique en déterminant les grands axes du projet ainsi que les grandes orientations techniques. Le rôle d'expert technique était assuré par David SARTRE ingénieur d'études. Mon rôle était multiple dans ce projet. Il a commencé par celui de chef de projet pour l'initialisation du projet (découpage, planification) et la gestion du projet (rapport d'avancement, vérification de l'atteinte des objectifs), puis a continué avec celui d'architecte pour les tâches de conception et d'étude, pour se terminer par celui de développeur pour les phases réalisation, déploiement et tests.

1.1.5. Planning

La création du planning se divisait en plusieurs aspects. Tout d'abord le lotissement, c'est-à-dire le découpage des tâches à effectuer en unités. Ensuite, l'ordonnancement optimisé de ces tâches en prenant en compte le fait que certaines tâches ne pouvaient pas démarrer si d'autres n'ont pas été terminées. Le planning est présenté dans le paragraphe 1.2.3.1. Planification (voir Figure 10).

1.2. Gestion de projet

1.2.1. Rappel de l'objectif

L'objectif premier en termes de livrable était de fournir un outil d'aide au développement pour les applications. Les fonctionnalités principales de l'outil étant de générer d'une part la couche d'accès aux données (appelée Data Access Layer en anglais, DAL) à partir d'une base de données et d'autre part des contrôles personnalisés qui sont situés au niveau IHM pour des utilisateurs avancés pour la mise à jour des données de référence (voir Figure 7).

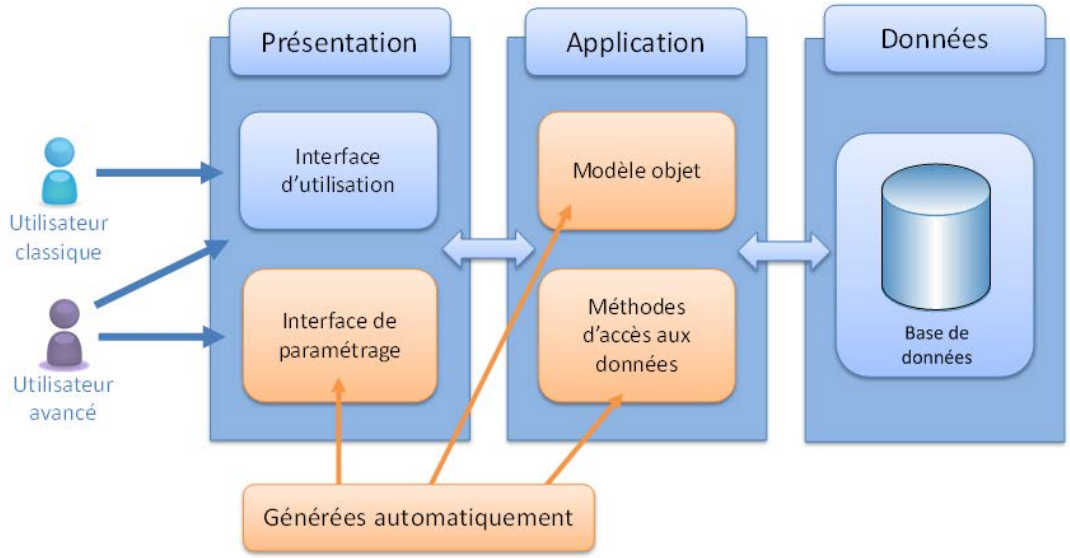

Figure 7 - Définition des modules à générer

1.2.2. Cahier des charges

Le cahier des charges vise à formaliser de manière simple l'objectif du projet ainsi que les livrables. Il sert avant tout de référentiel entre le client et le chef de projet chargé de sa réalisation.

Etant donné la charge du projet initial, il a été défini plusieurs niveaux de fonctionnement correspondant à des niveaux de versions différentes (voir Figure 8). Cela a permis d'effectuer plusieurs itérations et d'avoir des temps de réalisation plus courts. Le bénéfice d'une telle approche est la possibilité du recadrage des objectifs fonctionnels et techniques de l'itération suivante.

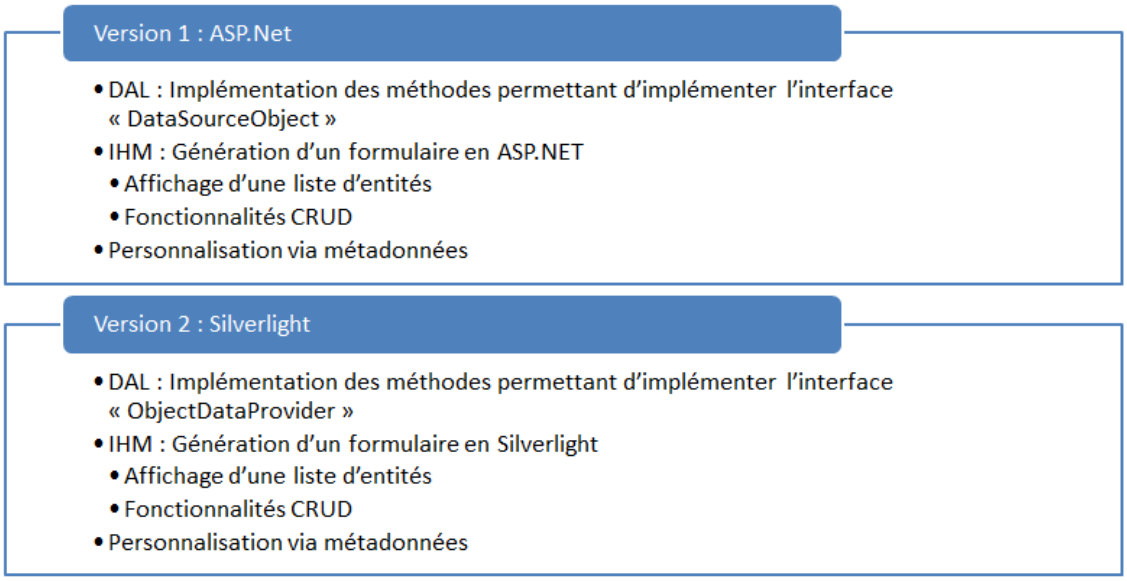

Figure 8 - Description des versions

1.2.3. Gestion de la qualité

La qualité est une appréciation générale d'un logiciel, plusieurs critères rentrent en jeu. L'exhaustivité des fonctionnalités, la précision des résultats, la fiabilité, la tolérance aux pannes, la facilité et la flexibilité de son utilisation, la simplicité, l'extensibilité, la compatibilité et la portabilité, la facilité de correction et d'évolution, la performance, la consistance et l'intégrité des informations qu'il contient sont tous des facteurs de qualité [ZEH90].
Pour maîtriser et atteindre des objectifs de qualité, plusieurs actions ont été mises en place notamment la planification, la gestion des codes source ainsi que la gestion des tests.

1.2.3.1. Planification

La première étape de la planification était le découpage en tâches du projet. Dans ce projet, plusieurs sous-projets ont été distingués :

- Un lot « Rédaction » incluant la rédaction du cahier des charges, des spécifications fonctionnelles, des spécifications techniques.
- Un lot « Etude », incluant l'étude de faisabilité et l'étude technique.
- Un lot « Générateur ORM » comprenant la conception de l'architecture, la réalisation du programme et la réalisation des tests unitaires.

- Un lot concernant le module « Générateur IHM », comprenant la conception de l'architecture, la réalisation du programme et la réalisation des tests unitaires.
- Un lot « Application globale » (IHM de l'outil et contrôle des modules), comprenant la conception de l'architecture, la réalisation du programme et la réalisation des tests unitaires.
- Un lot « Intégration » pour les modules « Générateur ORM » et « Générateur IHM »,
- Un lot « tests » comprenant les tests fonctionnels finaux.

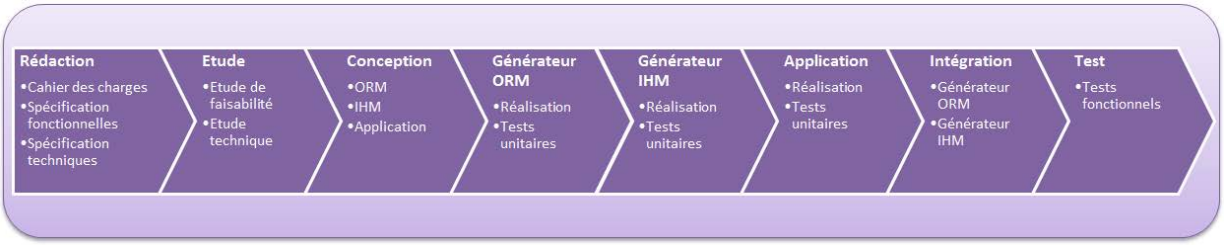

Figure 9 - Ordonnancement des tâches

La deuxième étape de la planification, une fois l'ordonnancement des tâches effectué, était la réalisation du planning. Pour chaque tâche définie ci-dessus, une date de début ainsi qu'une durée en nombre de jours a été définie.

Un point hebdomadaire avec le directeur technique Christophe CARAT a permis de faire le point sur l'état d'avancement du projet et d'ajuster le contenu d'une tâche ou de modifier le plan de charge pour permettre le respect des délais imposés par le planning (voir Figure 10).

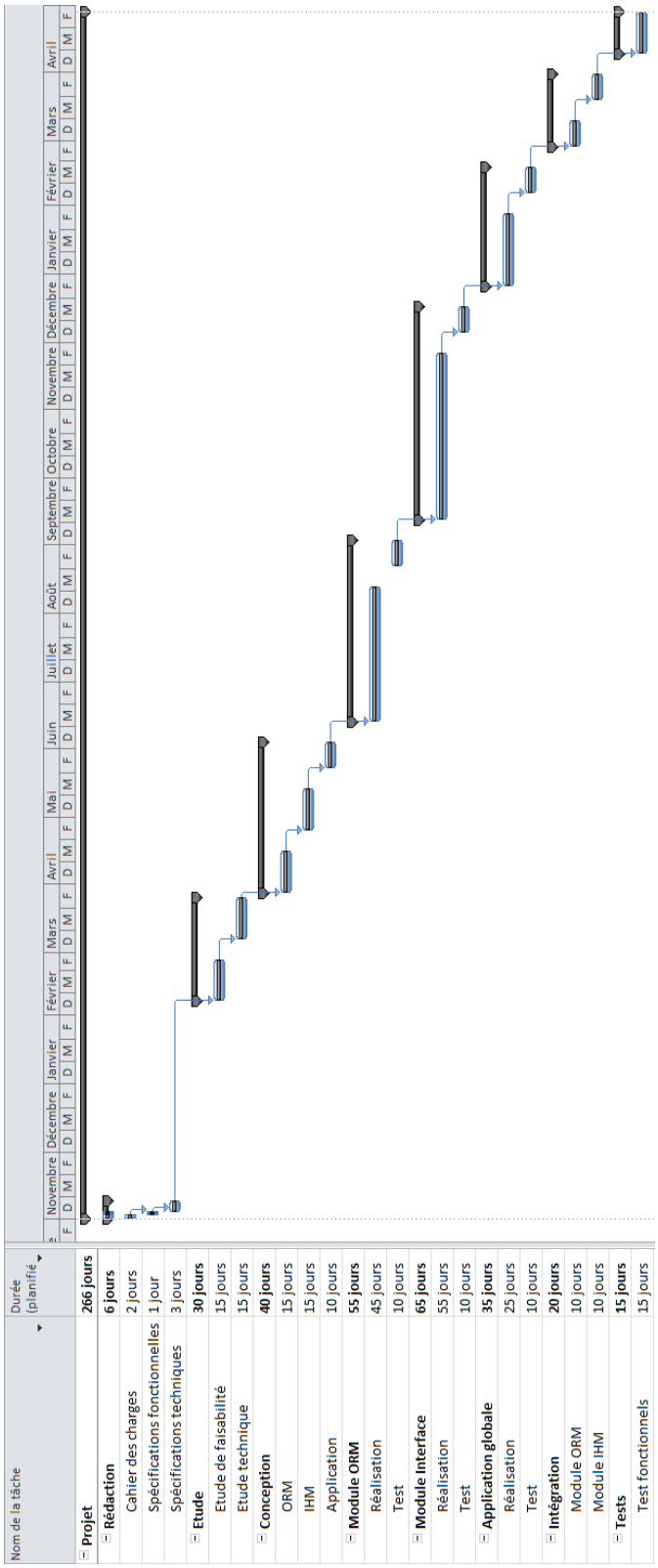

Figure 10 - Planning du projet

1.2.3.2. Gestion des versions

La gestion des versions de codes source, également appelé gestion des versions, consiste à gérer et à conserver les modifications apportées aux documents, programmes et autres informations stockées sous forme de fichiers informatiques.

Les changements sont généralement identifiés par un numéro ou un code lettre, appelé « version » ou « révision ». Chaque version est associée à un horodatage et à la personne qui effectue le changement. Les révisions peuvent être comparées, restaurées, et, avec certains types de fichiers, fusionnées. Ces fonctionnalités apportent une gestion de l'historique et de l'évolution des codes sources, points primordiaux dans la gestion de la qualité.

Il existe plusieurs types de systèmes de gestion de versions, les deux grandes catégories sont :

- les systèmes client-serveur, facile à appréhender et à mettre en place.
    - licence libre : CVS, Subversion (SVN) …
    - licence propriétaire : Microsoft Visual SourceSafe, Microsoft Team Foundation Server …
- les systèmes distribués
    - licence libre : Bazaar (Canonical à l'origine de la distribution Linux Ubuntu), Git (utilisé pour le noyau Linux) et Mercurial …
    - licence propriétaire : BitKeeper …

Etant donné le statut de partenaire privilégié Microsoft le choix du logiciel Team Foundation Server (TFS) s'est avéré évident. TFS permet la gestion des sources (contrôle de version) et la gestion des compilations. De plus, il s'intègre parfaitement avec l'environnement de développement intégré Microsoft Visual Studio.

1.2.3.3. Gestion des tests

L'objectif du test logiciel est de rechercher la présence d'erreurs pour les corriger, et non pas de démontrer qu'il en est exempt. Pour cela il faudrait être capable de tester tous les comportements ce qui est rarement envisageable et possible.

Les tests peuvent être divisés en plusieurs catégories dont les principales sont :

- Le test unitaire qui a pour but de tester un sous-ensemble de taille variable du logiciel.
- Le test d'intégration qui s'assure que le code que l'on souhaite ajouter au projet s'intègre sans effet de bord.
- Le test de validation qui vérifie le respect des exigences de la spécification.

Les tests de fuite mémoire, de régression, de performance et de robustesse pourraient également être cités.

Dans le cadre de la réalisation du projet, le choix d'utiliser le gestionnaire de tests de l'environnement de développement intégré Microsoft Visual Studio s'est avéré être la solution optimale. Pas de multiplication de logiciels pour le développement facilite le portage de l'environnement du développeur. Cela lui permet d'être plus mobile et de recréer son environnement de travail plus rapidement et d'apporter un gain d'ergonomie lié à l'utilisation d'une seule application lors de la programmation. Des tests unitaires ont été utilisés spécifiquement durant la réalisation des modules « ORM» et « IHM ». L'utilisation des tests est plus amplement détaillée dans le paragraphe 3.7 « Tests ».

Dans ce chapitre, ont été présentées, la définition du projet ainsi que les méthodes qui ont été mises en place pour assurer sa gestion. L'objectif du projet était donc d'accélérer la phase de développement d'applications en automatisant la réalisation de plusieurs parties de ces dernières. Le planning indique que les lots « Etude » et « Conception » ont représenté au total plus de 25% du temps de réalisation du projet et que l'ensemble des tests a totalisé plus de 15%. Cela démontre à quel point l'aspect « qualité » a été omniprésent. Cet aspect a

démarré dès l'initialisation du projet et la définition des méthodes de gestion, puis a continué avec la partie des lots « Etude » et « Conception » décrit dans le chapitre suivant.

2. Recherche de solutions

Ce chapitre s'articule autour de plusieurs axes. Tout d'abord la technique de l'object-relational mapping est décrite dans ses différentes approches puis les principaux logiciels ou Frameworks existants sont analysés. Puis une réflexion sur la génération de code source est menée. Et enfin un tour d'horizon sur la conception d'IHM est effectué.

2.1. Object-relational mapping

De nos jours la programmation orientée objet est largement utilisée. Elle a fait ses preuves quant à sa supériorité sur le paradigme de la programmation structurée dans le développement d'applications d'entreprises, grâce à toutes les possibilités qu'elle apporte par rapport au paradigme classique : encapsulation, héritage, polymorphisme, etc. D'un autre côté les applications utilisées en entreprise sont pour la plupart des applications utilisant le stockage en base de données relationnelles où ce dernier est effectué dans des conteneurs différents de ceux utilisés en programmation. Le problème de l'incohérence des types entre programmation orientée objet et base de données relationnelle est alors posé.

En 1990 lors de l'arrivée des premiers langages orientés objet sur le marché, l'idée des bases de données orientées objet a fait son apparition. Seulement, les bases de données relationnelles étaient déjà bien implantées sur le marché. Malgré les efforts du consortium ODMG (Object Data Management Group), une branche du OMG (Object Management Group), créé en 1991, publia des spécifications pour standardiser les bases de données orientées objet. Cette nouvelle façon de gérer la persistance des données ne trouva jamais son marché. Aujourd'hui seuls quelques éditeurs de système de gestion de base de données commercialisent encore des produits orientés objets comme Versant Object Database pour C++ et Java, Fast Objects pour .NET ou encore db4o pour l'informatique embarquée.

La technique « Object-relational mapping » soit en français « correspondance objet-relationnel (ORM) » se veut une réponse au problème posé par la différence entre la nature des objets utilisés dans les programmes et les structures de stockage utilisées pour la persistance des données [ROE11]. Elle consiste donc à mettre en place des mécanismes pour effectuer cette conversion. Dès lors la base de données est alors vue comme un ensemble d'objets à travers la couche ORM. Cette technique est mise en œuvre au travers d'outils permettant de générer automatiquement cette couche en se basant sur la base de données ou sur des schémas de modélisation des objets.

Il existe différentes approches dans la mise en œuvre de cette technique. Il est présenté ici une classification des différentes approches utilisées, puis un aperçu des outils majeurs disponibles à ce jour et compatibles avec la technologie .Net étant donné le partenariat technique de la société Algorys avec l'éditeur Microsoft.

2.1.1. Approche n-uplet

Cette approche est la plus basique. Elle pourrait même remettre en question le fait qu'il s'agisse d'une technique d'ORM. En effet le principe de fonctionnement est de considérer chaque enregistrement d'une table en tant que n-uplet.

En pratique, pour chaque ligne lue dans la base de données relationnelle, les données sont stockées dans un vecteur (tableau à une dimension) indexé soit par le nom de la colonne soit par la position de la colonne dans la table. Quand plusieurs lignes sont sélectionnées un vecteur ou une collection de niveaux supérieurs est créé stockant l'ensemble des données. L'ensemble du jeu de résultat est ainsi manipulable dans un tableau à deux dimensions (une pour les lignes et une pour les colonnes).

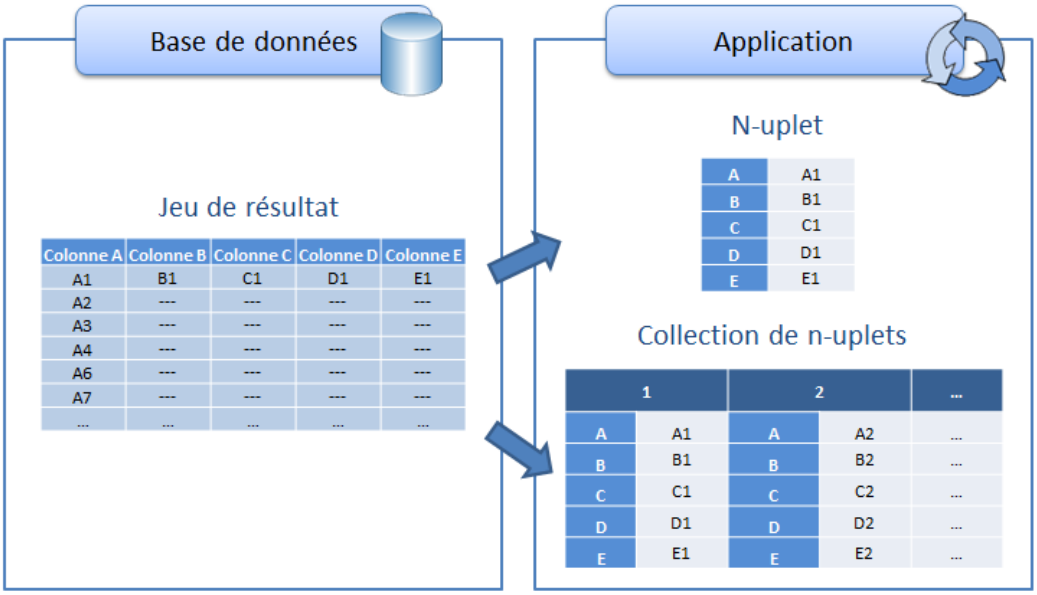

Figure 11 - Schéma de l'approche n-uplet

Ce système est souvent facile à mettre en place car il ne nécessite aucun paramétrage et est très simple d'utilisation. Il est néanmoins déconseillé pour les applications où la couche métier est très liée au format de stockage des données car chaque changement dans la base entraine une modification de la logique métier.

2.1.2. Approche par entité

L'approche par entité, diffère de l'approche par n-uplet car elle utilise des classes pour chaque table, là où l'approche n-uplet se contente d'utiliser des vecteurs ou des tableaux. Deux types de classes sont utilisés :

- Les classes métiers
  Elles contiennent l'ensemble des attributs correspondant aux types des colonnes d'une table
- Les classes d'accès aux données (suffixé par DAL pour Data Access Layer)
  Elles contiennent l'ensemble des méthodes pour effectuer les opérations de base pour accéder et modifier les données et ne contiennent aucun attribut. Elles peuvent gérer les contraintes relationnelles de la base de données ainsi que la mise à jour et suppression en cascade.

Le fonctionnement s'établit de la façon suivante (voir Figure 12) :

- Les classes issues de la couche composant d'accès aux données interrogent la base de données en passant par un Framework d'accès aux données (ici ADO de Microsoft) en utilisant un ordre SQL ou en appelant une procédure stockée ou une fonction.
- Les entités métiers sont ensuite remplies avec les données récupérées.

Les règles métiers doivent alors être déportées dans un troisième type de classement, suffixé ici par « Logic », et aussi appelé classe de gestion.

Ce système permet donc de gérer les contraintes relationnelles de la base de données et autorise également la prise en compte de la mise à jour et de la suppression en cascade via les classes d'accès aux données.

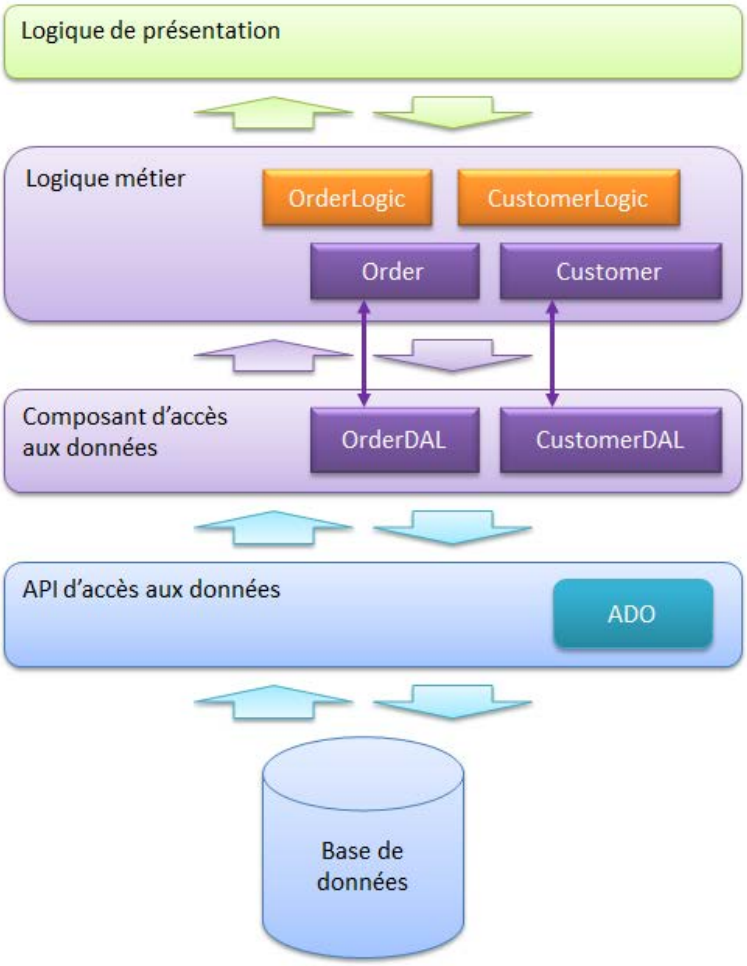

Figure 12 - Architecture de référence

2.1.3. Approche par modèle de domaine

Cette approche reprend et complète les caractéristiques de l'approche par entité. La différence se situe au niveau de l'entité métier qui se veut plus complexe. En effet, dans cette approche l'entité est un véritable objet avec ses comportements et non un conteneur pour le stockage des informations issues de la base de données [FOW02]. L'approche est centrée sur le domaine, qui représente lui-même les objets réels ainsi que leurs comportements.

Le point de départ de cette approche est souvent le modèle conceptuel. La base de données est alors automatiquement générée en fonction de ce dernier. Les entités issues de cette approche sont souvent plus proches des entités métiers à représenter.

Tous les concepts objets tels que l'héritage, le polymorphisme etc. sont alors plus facilement utilisables. La logique métier est également contenue dans cette entité, il n'y a donc plus besoin des classes de gestion. Quand des évolutions sont nécessaires, seul le domaine doit être modifié. Le gain est important au niveau de la facilité de maintenance et de la capacité d'évolution des applications suivant cette approche.

Le principal désavantage de cette approche est qu'elle requiert un paramétrage important au niveau de la correspondance des objets et de la source de données car il faut décrire précisément toutes les correspondances entre le côté base de données relationnelle et le modèle objet.

Au final, il apparait que chaque solution comporte des avantages et des limites. L'avantage de l'approche par n-uplet est sa facilité à être déployée et utilisée, mais les fonctionnalités qu'elle apporte sont limitées. A l'inverse, plus l'approche est complexe, plus la réponse est générale mais difficile à mettre en œuvre.

2.1.4. LINQ To SQL

LINQ to SQL est un composant du .NET Framework qui fournit un ensemble d'outils pour gérer les données relationnelles comme des objets. LINQ to SQL est un dérivé de LINQ qui signifie Language-Integrated Query soit en français « requête intégrée au langage » distribué dans le cadre du Framework .NET 3.5 en 2007. L'objectif de LINQ est de proposer un langage d'interrogation de données intégré aux langages .NET en utilisant une syntaxe proche de celle de SQL. De nouveaux opérateurs sont proposés afin d'effectuer des requêtes, de filtrer et sélectionner des données dans des collections, des structures XML, des bases de données relationnelles et des sources de données tierces. Il existe également d'autres dérivés de LINQ : LINQ to XML, LINQ to OBJECT, etc. LINQ to SQL est donc le premier composant de correspondance objet-relationnel édité par Microsoft pour ses langages .NET. Le principe de fonctionnement est le suivant :

1. Création de la correspondance

A travers un assistant de conception visuelle, un modèle, représentant tout ou partie de la base de données, est construit par l'ajout d'objets de la base de données relationnelle. Le modèle peut contenir des tables, des vues, des procédures stockées ainsi que des fonctions.

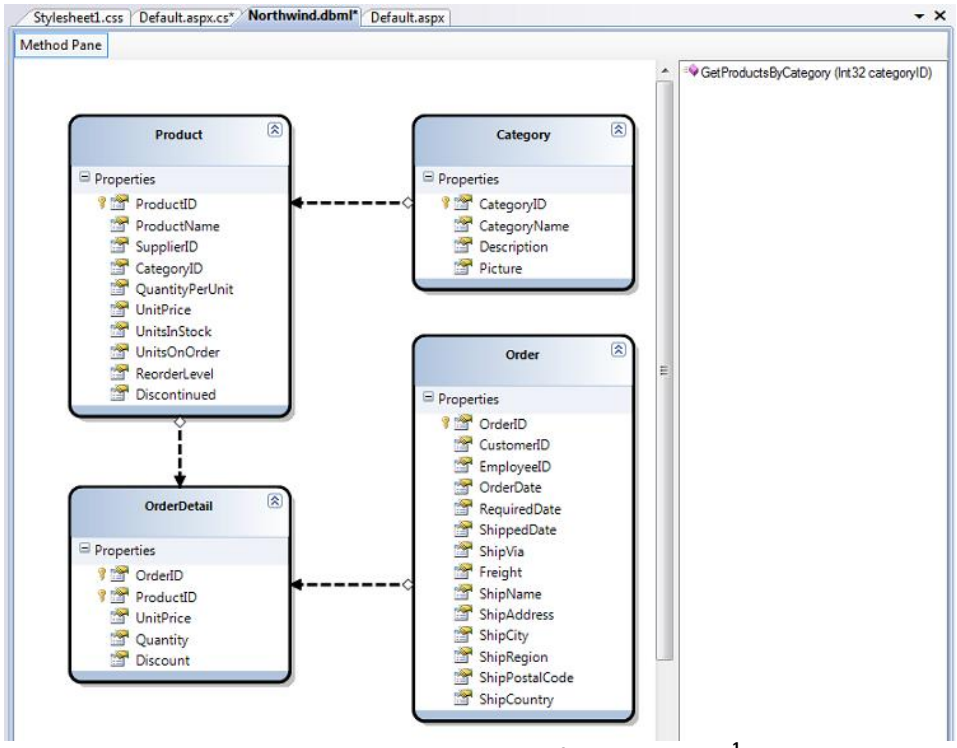

Figure 13 - Concepteur de modèle LINQ to SQL[1]

[1] Source: http://msdn.microsoft.com

2. DataContext

Dès l'enregistrement du modèle dans un fichier d'extension dbml, un outil s'exécute et génère un fichier suffixé DataContext. Ce fichier contient une propriété pour chaque table modélisée ainsi qu'une méthode pour chaque procédure stockée. Les propriétés représentant les tables permettent d'interroger la base et de récupérer le résultat sous forme d'une collection d'objets typés.

3. Opération sur le DataContext

Il est dès lors possible d'effectuer un chargement des données de la base de données relationnelle et de les filtrer en seulement deux lignes de code (voir Figure 14).

```
NorthwindDataContext db = new NorthwindDataContext();

var products = from p in db.Products
               where p.Category.CategoryName == "Beverages"
               select p;
```

Figure 14 - Exemple de requête LINQ to SQL : Chargement[2]

Il également possible de récupérer une ligne dans la base de données, de la mettre à jour et de valider les changements dans la base de données.

```
NorthwindDataContext db = new NorthwindDataContext();

Product product = db.Products.Single(p => p.ProductName == "Toy 1");

product.UnitPrice = 99;
product.UnitsInStock = 5;

db.SubmitChanges();
```

Figure 15 - Exemple de requête LINQ to SQL : Mise à jour[3]

L'insertion dans la base de données, se fait à l'instanciation d'un nouvel objet et en l'ajoutant à la propriété correspondante à la collection de son type dans le DataContext puis à valider le DataContext.

De plus il est possible de spécifier si les entités en relation doivent être chargées dès le départ ou seulement lorsqu'elles sont interrogées (Lazy loading).

Les avantages d'un tel fonctionnement sont :

- compilation de la requête pour la validation de la syntaxe
- auto-complétion directement dans l'éditeur
- Augmentation de la maintenabilité et de l'intégration de l'interrogation des données dans le code
- Gestion du « Lazy loading »

2.1.5. Entity Framework

Basé sur ADO.Net (composant d'accès aux données .Net), Entity Framework est un Framework de correspondance objet relationnel qui permet au développeur de travailler avec des données relationnelles en utilisant l'approche par domaine (voir paragraphe 2.1.3. Approche par modèle de domaine).

[2] Source: http://msdn.microsoft.com
[3] Source: http://msdn.microsoft.com

Plusieurs versions sont disponibles :

1. Entity Framework v1.0 – 11/08/2008 - disponible avec Visual Studio 2008 SP1
2. Entity Framework v4.0 – 12/04/2010 - disponible avec Visual Studio 2010.
3. Entity Framework v4.1 – 12/04/2011 (Mise à jour le 24/07/2011).
4. Entity Framework v4.3 – 09/02/2012.

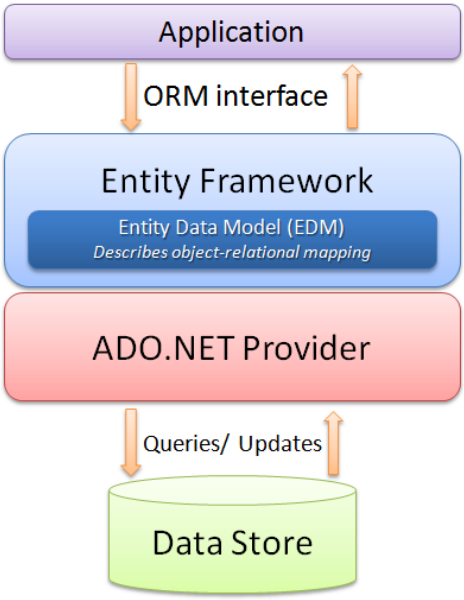

Figure 16 - Schéma d'architecture d'Entity Framework

La correspondance objet-relationnel est basée sur un modèle appelé Entity Data Model (EDM) plutôt que sur les entités. Ce modèle s'appuie sur le modèle Entité-Association décrit par Dr. Peter Chen qui distingue d'un côté les objets et de l'autre les associations.

Figure 17 - Architecture Entity Data Model[4]

L'EDM est constitué d'un fichier de descriptions XML d'extension « edmx » qui contient trois concepts (voir Figure 17) :

- **Le modèle conceptuel** : Description des entités et des associations en se basant sur le modèle Entité-Association. Un concepteur qui permet de visualiser sous forme graphique les objets et associations est disponible dans l'outil Visual Studio (voir Figure 18).
- **Le modèle logique de données** : Description de la base de données.
- **Le schéma de liaison** : Description entre les deux modèles.

[4] Source: http://msdn.microsoft.com

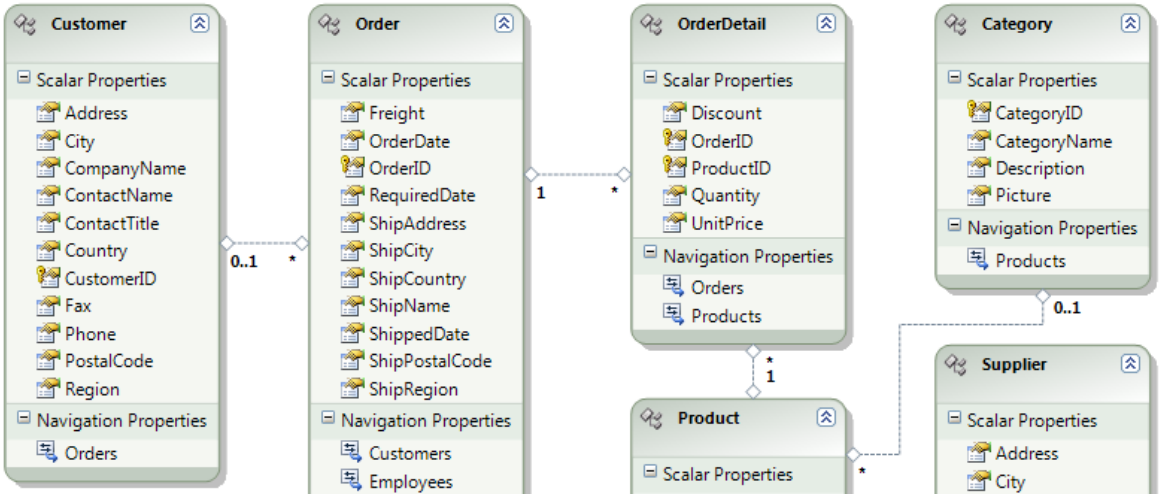

Figure 18 - Exemple de schéma conceptuel sous Visual Studio

Pour la réalisation de la correspondance entre le modèle relationnel de la base de données et le modèle objet, Entity Framework supporte plusieurs approches :

DataBase First : en partant de la base de données

C'est l'approche historique de l'outil. Elle prend comme prérequis l'existence d'une base déjà utilisée ou de la création d'une nouvelle base. En utilisant l'assistant de création d'Entity Data Model, les objets de la base de données à inclure dans le modèle peuvent être sélectionnés et un modèle objet-association est généré automatiquement en fonction des relations existantes dans la base de données. Un deuxième assistant permet de modifier le schéma une fois celui-ci créé. Le concepteur visuel ainsi que les deux assistants ne prennent en charge que la base de données de type Microsoft SQL Server. Cependant il existe des outils tiers qui se substituent pour prendre en charge les bases de données suivantes : DB2, EffiProz, Firebird, Informix, MySQL, Oracle, PostgreSQL, SQLite, Sybase, and VistaDB[5].

Model First : en partant du modèle

Disponible avec la version 4.0 d'Entity Framework, elle convient aux projets de création où il n'y a pas d'existant en termes d'application. La première étape consiste à créer le modèle objet-association dans le concepteur sans la correspondance. Puis un premier assistant crée le schéma logique de la base de données et la correspondance entre les deux modèles. Un second assistant génère le script de création de la base de données. Un dernier assistant permet de modifier le schéma conceptuel et de régénérer les modèles logiques et modèles de liaisons ainsi que les scripts de modifications de la base de données. Dans cette approche, il n'est pas possible de modifier le schéma de liaison ou la structure de la base de données directement sous peine de voir les modifications écrasées par l'assistant de modification. Comme pour l'approche partant de la base de données seul SQL Server est supporté par l'outil. Cependant, des outils tiers permettent de travailler avec les bases de données suivantes : Oracle, MySQL, and PostgreSQL[6].

Code First : en partant du code source

Depuis la version 4.1 d'Entity Framework, il est possible de créer le schéma conceptuel à partir des classes C# ou VB.NET suivant des annotations spécifiques. Ceci permet de générer la base de données ou de travailler sur une base de données existante.

---

[5] *ADO.NET Data Providers for Entity Framework*, MSDN, consulté le 27/11/2011. http://msdn.microsoft.com/en-us/data/dd363565.aspx

[6] *Entity Framework 4 Release Candidate supported*, DEVART, consulté le 27/11/2011. http://www.devart.com/blogs/dotconnect/?p=2062

Quelle que soit l'approche utilisée, le fonctionnement lors de l'exécution est identique. Les requêtes sont générées automatiquement à partir des fichiers de description. Pour l'interrogation, plusieurs possibilités sont offertes (voir Figure 19) :

- utiliser les méthodes générées lors de la compilation par Entity Framework qui retourne les objets sous forme de collections (IEnumerable<T>, T représentant le type Entity).
- utiliser LINQ à travers LINQ to ENTITIES, pour consulter et modifier les données à travers une syntaxe LINQ. (voir 2.1.4. LINQ To SQL).
- Utiliser SQL avec Entity SQL qui est un langage similaire à SQL permettant d'interroger les modèles conceptuels.

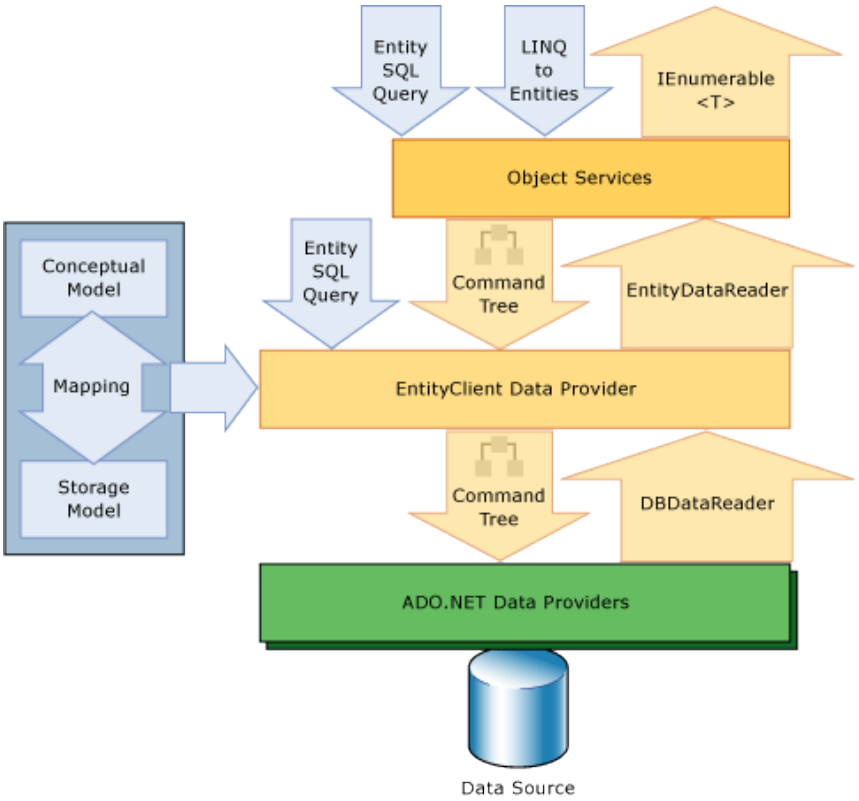

Figure 19 - Schémas d'interrogation des données dans Entity Framework

Entity Framework, est donc une solution complète ORM qui, à travers plusieurs approches possibles, s'adapte à chaque type de projet. De plus il offre un panel de fonctionnalités supplémentaires par rapport aux solutions ORM classiques comme LINQ.

2.1.6. NHibernate

NHibernate (NH) est un outil d'ORM pour la plateforme .Net. Il gère la persistance et l'interrogation des objets en dans la base de données. NH utilise l'approche par domaine. La correspondance objet-relationnel est effectuée à l'aide de fichier XML. Il s'appuie ensuite sur ces classes pour générer les ordres SQL.

Il est issu de l'outil Hibernate pour la plateforme Java. Gavin KING a démarré la création du produit Hibernate en 2001. La version 2 est sortie en 2003. L'entreprise JBOSS a ensuite recruté les principaux développeurs du produit pour proposer un produit disposant d'un vrai support. En 2010 la version 3 était finalisée puis la version 4 fin 2011. NH a démarré sur l'initiative de Tom BARRET puis a été repris plus tard par Mike DOERFLER et Peter SMULOVICS en tant que développement communautaire sous licence GNU LGPL.

L'historique des versions principales :

- NH 1.2.1 : 11/2007 – compatible avec .NET 1.1 / 2.0
- NH 2.0 : 23/08/2008 – compatible avec .NET 2.0
- NH 3.0 : 04/12/2010 – compatible avec .NET 3.5 et LINQ via un fournisseur tiers
- NH 3.2 : 04/2011 – Version actuelle

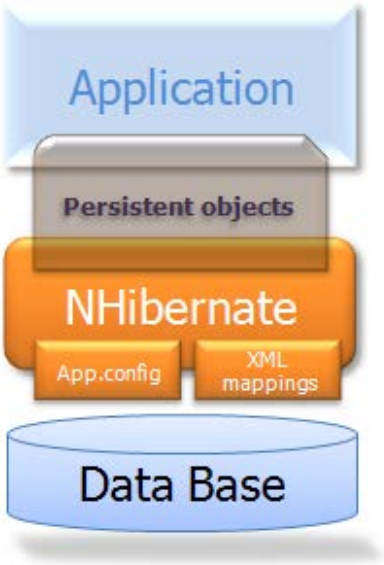

Figure 20 - Schéma de haut niveau de l'architecture de NHibernate[7]

Comme l'illustre la Figure 20, le fonctionnement de NHibernate s'appuie sur une configuration et une correspondance XML. La première étape est le paramétrage de la connexion à la source de données dans le fichier de configuration de l'application (App.config). Il faut ensuite définir le modèle de domaine puis créer les classes .NET implémentant ce modèle. Une fois les classes réalisées, il faut effectuer la correspondance entre les classes et les objets de la base de données. Deux possibilités sont offertes depuis la dernière version :

- La correspondance via un fichier XML
  Un fichier est créé pour chaque classe avec un nom spécifique (nom de la classe avec le suffixe « .hbm.xml »). Dans le fichier chaque propriété correspondant à une colonne de la base doit être définie. Un grand nombre d'attributs et de propriétés sont disponibles afin de réaliser une correspondance précise.

[7] Source : http://nhforge.org/doc/nh/en/index.html#quickstart-mapping

```xml
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
  namespace="QuickStart" assembly="QuickStart">

  <class name="Cat" table="Cat">
    <id name="Id">
      <generator class="identity" />
    </id>

    <property name="Name">
      <column name="Name" length="16" not-null="true" />
    </property>
    <property name="Sex" />
    <many-to-one name="Mate" />
    <bag name="Kittens">
      <key column="mother_id" />
        <one-to-many class="Cat" />
      </bag>
  </class>
</hibernate-mapping>
```

Figure 21 - Exemple de correspondance XML

- La correspondance via le code .NET via le plugin « Fluent mapping »
  Dans cette méthode la correspondance passe par l'écriture de classes de correspondance. L'avantage de cette méthode est l'évaluation du code par le compilateur qui limite les erreurs de correspondance. Lorsqu'une classe ou un attribut est renommé, la correspondance peut automatiquement être mise à jour. La correspondance est plus compacte et donc plus lisible.

```dotnet
public class CatMap : ClassMap<Cat>
{
  public CatMap()
  {
    Id(x => x.Id);
    Map(x => x.Name)
      .Length(16)
      .Not.Nullable();
    Map(x => x.Sex);
    References(x => x.Mate);
    HasMany(x => x.Kittens);
  }
}
```

Figure 22 - Exemple de correspondance "Fluent"

Plusieurs modes d'interrogation des données sont proposés :

- Hibernate Query Language (HQL). Langage d'interrogation qui ressemble au SQL mais qui reste orienté objet. A la différence de LINQ, HQL n'est pas évalué par le compilateur.

```dotnet
var blogs = s.CreateQuery("from Blog b where b.Title = :title and b.Subtitle = :subtitle")
    .SetParameter("title","Ayende @ Rahien")
    .SetParameter("subtitle", "Send me a patch for that")
    .List<Blog>();
```

Figure 23 - Exemple de requête HQL

- Criteria Queries. Méthode qui retourne un objet de type ICriteria représentant la requête. Lors de son instanciation le type de l'objet désiré est spécifié. Puis les critères sont ajoutés via la méthode Add.

```
var blogs = s.CreateCriteria<Blog>()
    .Add(
            Restrictions.Eq("Title", "Ayende @ Rahien") ||
            Restrictions.Eq("Subtitle", "Send me a patch for that")
        )
    .List<Blog>();
```

Figure 24 - Exemple de requête Critera Queries

- LINQ via un fournisseur tiers.

Il est également possible d'outrepasser NH pour écrire directement les requêtes en SQL mais la portabilité du système est alors perdue.

En conclusion, le principal atout de NH est la gestion de la correspondance qui offre la possibilité de définir précisément les liens entre les objets du code et les objets de la base de données. NH est un produit mature offrant un grand nombre de fonctionnalités ainsi que de plugins comme Fluent Hibernate (qui remplace les fichiers de configuration XML par des objets de type fort C#).

2.1.7. Subsonic

Subsonic est un logiciel open source de génération de composants d'accès aux données. Il a été développé depuis août 2006 par une équipe dirigée par Rob Conery qui a travaillé de 2007 à 2009 pour Microsoft. Subsonic fournit plusieurs approches :

Active Record
Très proche de l'approche par entités, ce pattern d'architecture a été décrit par Martin Fowler [FOW02]. Les tables ou les vues de la base de données sont encapsulées dans des classes (une classe par objet de la base de données). Chaque instanciation de classe comporte au minimum les fonctions CRUD et des propriétés correspondant aux colonnes de la table ou de la vue.

Simple Repository
Le pattern Simple Repository peut être utilisé pour des applications où la base de données n'est vue que comme un dépôt de données. Ce mode de fonctionnement se base sur la classe et non sur une correspondance classes / objet de la base de données. C'est l'outil qui a la charge de générer et de modifier automatiquement la base de données en fonction du type des objets utilisés dans le code.

Advanced Template
La dernière approche s'appuie sur les T4 Text Templates (voir 2.2.2. Text Templates) pour générer des classes qui peuvent être manipulées de la même manière que LINQ to SQL.

Subsonic offre une alternative intéressante par rapport aux outils ORM traditionnels de par sa facilité d'installation et d'utilisation. A la différence d'un outil conventionnel, dans Subsonic il n'y a pas réellement de personnalisation de la correspondance possible. De plus le support est limité dans le cadre du logiciel open source.

2.1.8. Comparaison

Voici un récapitulatif de l'étude des principaux outils d'ORM disponibles pour le langage C#.

Tableau 1 - Comparaison des outils d'ORM

	Linq To SQL	Entity Framework	NHibernate	Subsonic
Type de correspondance	Simple (1-1)	Complexe	Complexe	Simple
Lazy loading	✔	✔	✔	✘
Cache	✘	✔	✔	✘
Source de données	SQL server	Multi bases de données (en passant par des produits tiers)	Multi bases de données	Multi bases de données
Génération de la base de données	✔	✔	✔	✘
Performance*	🟡	🟡	🟡	🔴
Point fort	- Rapidité de mise en œuvre	- couplage Entity Framework / .NET et Entity Framework / Visual Studio	-Richesse des fonctionnalités - Nombre d'extensions	- Pas de correspondance à mettre en place
Point faible	- Mono base de données	- Complexité - Boîte noire	- Complexité	- pas de cache - pas de Lazy loading

Légende	✔ Inclus ✘ Exclus
	🟢 Bonne 🟡 Moyenne 🔴 Mauvaise

*Le critère performance est évalué en fonction de l'écart en termes de temps d'exécution des requêtes des outils testés par rapport à des requêtes SQL exécutées via le fournisseur d'ADO.NET. Les tests sont détaillés dans le Tableau 2.

Tableau 2 - Résultat du comparatif des performances

	Linq To SQL		Entity Framework		NHibernate		Subsonic		ADO.NET (Référence)
	Temps (ms)	Ecart (vs référence)	Temps (ms)	Ecart (vs référence)	Temps (ms)	Ecart (vs référence)	Temps (ms)	Ecart (vs référence)	Temps (ms)
SELECT SIMPLE	23	🟢 15%	50	🟡 150%	44	🟡 120%	215	🟡 975%	20
JOINTURE	85	🟡 270%	88	🟡 283%	82	🟡 257%	807	🔴 3409%	23
Lazy loading	102	🟡 209%	140	🟡 324%	155	🟡 370%	945	🔴 2764%	33

Légende	🟢 < à 100% 🟡 < à 1000% 🔴 > à 1000%

Deux catégories se distinguent. D'un côté LINQ To SQL et Subsonic dont la facilité de mise en place (pas de correspondance à réaliser) et d'utilisation s'accordent parfaitement avec les petits et moyens projets. Et d'un autre côté, Entity Framework et NHibernate qui disposent d'une grande richesse de fonctionnalités (gestion de cache, lazy loading, approche par domaine) mais dont le temps de paramétrage et de prise en main nécessitent des projets de taille plus importante pour être rentables.

2.2. Génération de code source

Dans l'optique de la réalisation des modules de génération d'ORM et d'IHM une réflexion sur les différentes possibilités pour générer du code source a été réalisée. Sont détaillées ci-dessous les principales techniques de génération de code source.

2.2.1. Moteur de génération personnalisé

Dans cette optique, le moteur de génération serait conçu et réalisé spécifiquement pour le besoin. Il ne n'utiliserait aucune technique d'aide à la génération de code. Le moteur se connecterait à une base de données pour récupérer les informations. Puis ces dernières serait traité pour générer des fichiers textes au format de code source désiré. La réalisation d'un tel moteur serait longue et fastidieuse impliquant une évolution et maintenance Complexe. De plus il serait construit pour générer du code source d'un langage unique.

2.2.2. Text Templates Transformation Toolkit

Disponible à travers l'outil de développement Visual Studio Microsoft, le Text Templates Transformation Toolkit (T4 Text Templates) est un outil de transformation basé sur des modèles. Ces derniers sont composés de blocs de textes et de logique de contrôle permettant de générer un fichier texte. Les langages C# ou Visual Basic peuvent être utilisés pour écrire la logique de contrôle. Le fichier généré est un fichier texte qui peut donc aussi être une page html, un fichier de code source, etc. Il existe deux types de modèles :

Les modèles dit « Run Time » : la génération de texte est effectuée pendant l'exécution de l'application .Net. Ils ne nécessitent pas que l'application Visual Studio soit installée pour fonctionner. Dans ce type de modèle, le code est composé en très grande majorité de blocs de textes.

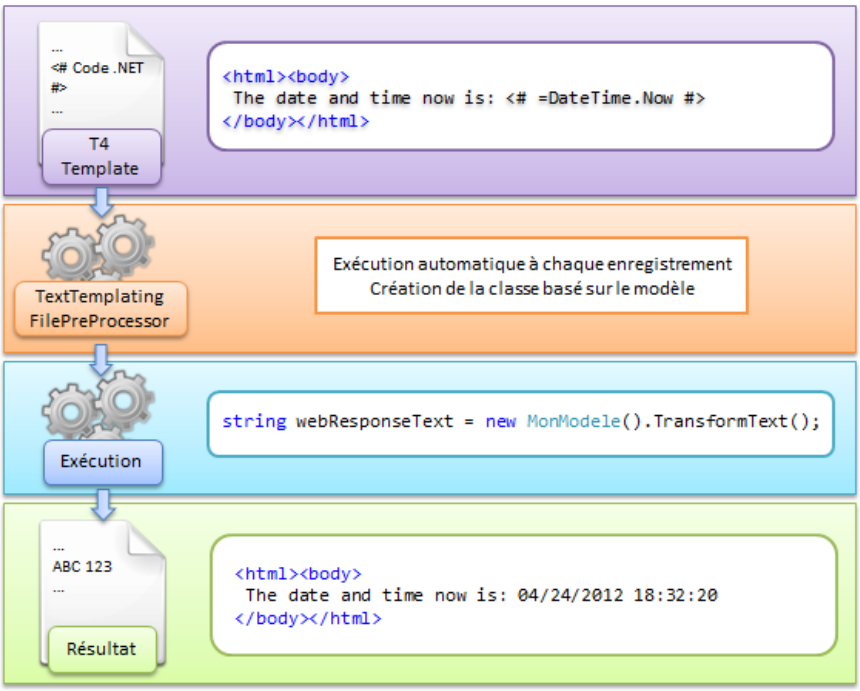

Figure 25 - Processus de génération avec un modèle T4 "Run Time"

La Figure 25 illustre le processus de génération avec un exemple de modèle « Run Time ». L'objectif est de générer une page html avec la date et l'heure mises à jour au moment de l'exécution du modèle. Le processus de génération avec un modèle T4 « Run Time » s'effectue en plusieurs étapes. La première est la création du modèle. Ce dernier est contenu dans un fichier d'extension « .tt ». Dès lors qu'il est enregistré un outil (TextTemplatingFilePreprocessor.exe) s'exécute et génère une classe (voir Figure 26) basée sur le modèle défini précédemment. Cette classe fournit une méthode « TransformText » qui lance la transformation. La deuxième étape consiste à exécuter la méthode « TransformText » lors de l'exécution d'un programme. La méthode retourne une chaîne de caractères contenant les éléments de textes ainsi que le résultat des instructions définies dans le modèle.

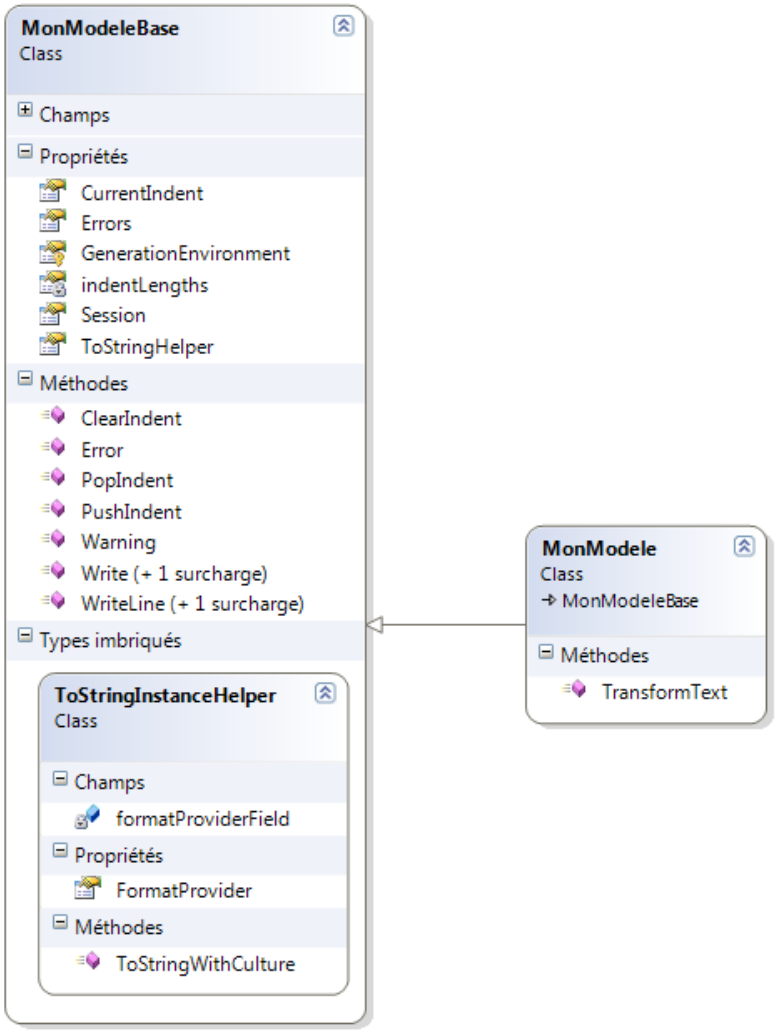

Figure 26 - Classe générée par le modèle T4 "Run Time"

Les modèles dits « Design Time » : La génération de texte s'effectue dans Visual Studio à chaque fois que la source du modèle ou le modèle lui-même est modifié. Ici le modèle contient majoritairement de la logique de contrôle.

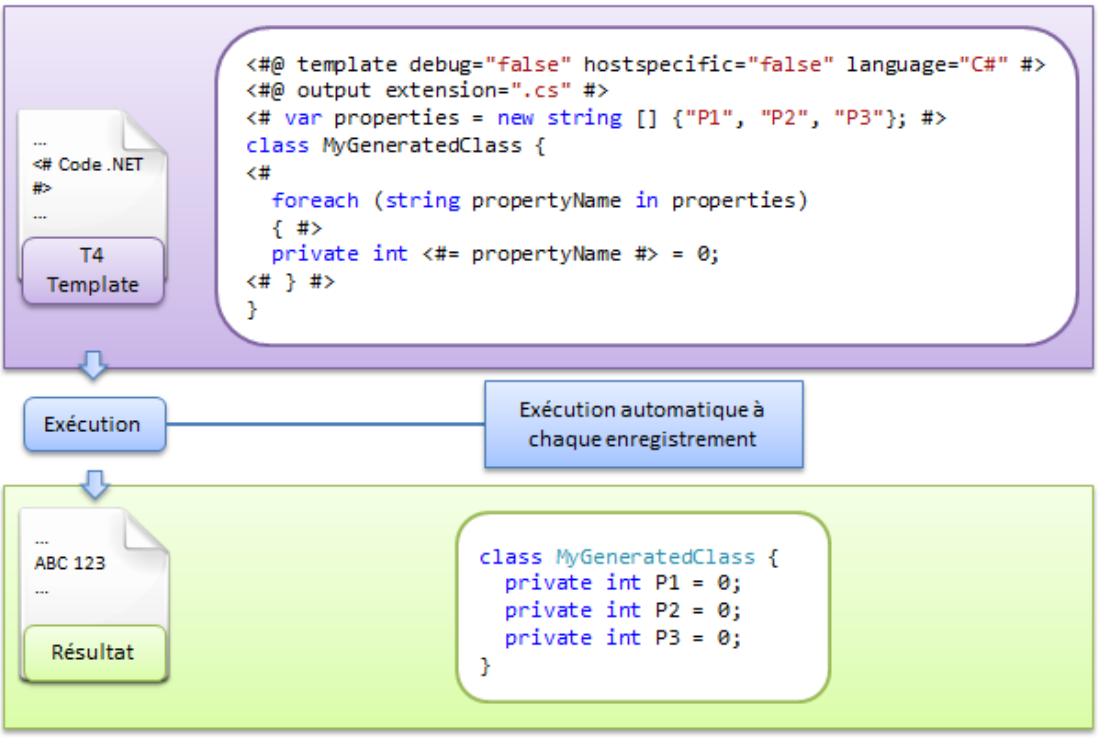

Figure 27 - Processus de génération d'un modèle T4 "Design Time"

Le processus de génération est identique au modèle « Run Time » à la différence que l'exécution de la méthode « TransformText » est d'une part déclenchée automatiquement après la création de la classe du modèle et d'autre part que le résultat de cette exécution est immédiatement stocké sous la forme d'un fichier dont l'extension est définie dans le modèle.

2.2.3. XML & XSLT

Le langage XML (Extensible Markup Language, « langage de balisage extensible ») est un langage de structuration largement utilisé. Le langage est défini sous la forme d'une recommandation du World Wide Web Consortium (W3C). Le XML est utilisé pour hiérarchiser des données et représenter une arborescence dans un langage à de balisage. Le XSLT (eXtensible Stylesheet Language Transforming, « langage de transformation des feuilles de style ») est un langage qui permet de transformer un fichier XML.

Le fonctionnement se base sur un fichier XML qui sert de source de données et d'un modèle de transformation XSLT qui définit la transformation à appliquer. Un processeur XSLT effectue la transformation et génère un nouveau document. Bien qu'à la base XSLT n'ait été prévu que pour générer des documents XML ou dérivés, il est possible de générer tout type de documents texte. XSLT utilise le XPath pour interroger les données du document XML et le mettre en forme. Le processeur va parcourir le document XSLT. A chaque bloc de texte, ce dernier est reporté tel quel dans le document résultat. Pour chaque instruction, une requête XPath est exécutée dans le document XML et les résultats de la requête sont écrits dans le document de sortie (voir Figure 28).

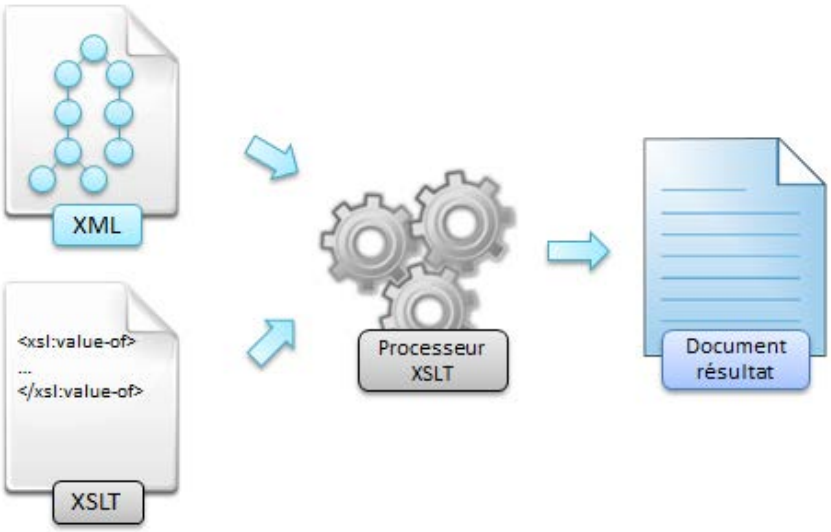

Figure 28 - Schéma de flux d'une transformation XSLT

La Figure 29 représente un exemple d'une transformation XSLT avec un fichier XML contenant une liste de CD. Le document XML est composé d'un nœud racine « catalog » qui est lui-même composé de « cd ». Les « cd » sont constitués de 4 nœuds : « title », « artist », « price », « year ». Le modèle XSLT est, quant à lui, composé d'un premier nœud qui recherche le nœud « / » c'est-à-dire le nœud racine autrement dit le nœud « catalog ». Dès que le nœud est trouvé le premier bloc de texte est écrit dans le document résultat. La deuxième instruction XSL est une déclaration de boucle qui va écrire un bloc de texte comportant la valeur des propriétés « title » et « artist » de chaque nœud cd trouvé.

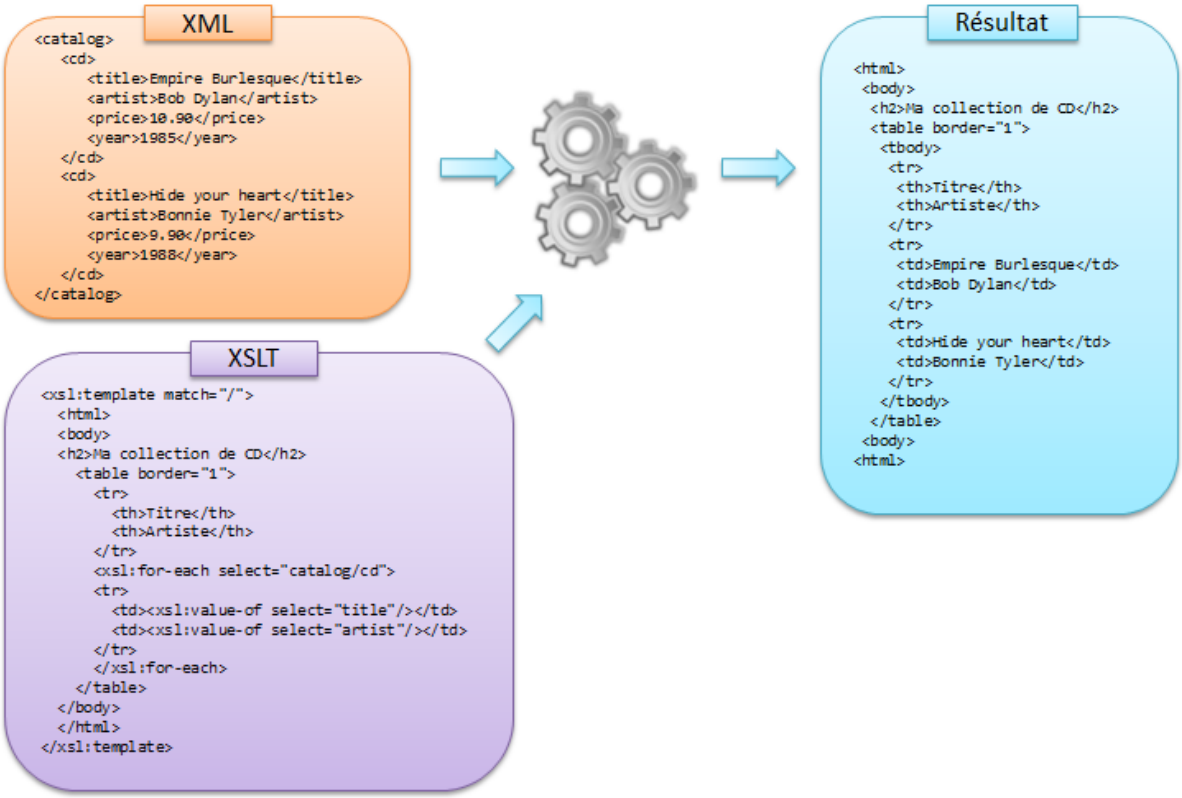

Figure 29 - Exemple d'une transformation XSLT

2.2.4. CodeDOM

Le Kit de développement .NET Framework fournit une bibliothèque logicielle appelée CodeDOM (Code Document Object Model : Modèle de document objet appliqué au code) qui permet aux développeurs de générer un code source au moment de l'exécution. Le code peut être généré en plusieurs langages :

- C# (Microsoft.CSharp.CSharpCodeProvider)
- VB / VBS / VB.Net (Microsoft.VisualBasic.VBCodeProvider)
- JavaScript (Microsoft.JScript.JScriptCodeProvider)
- J# (Microsoft.VJSharp.VJSharpCodeProvider jusqu'en version 2.0)

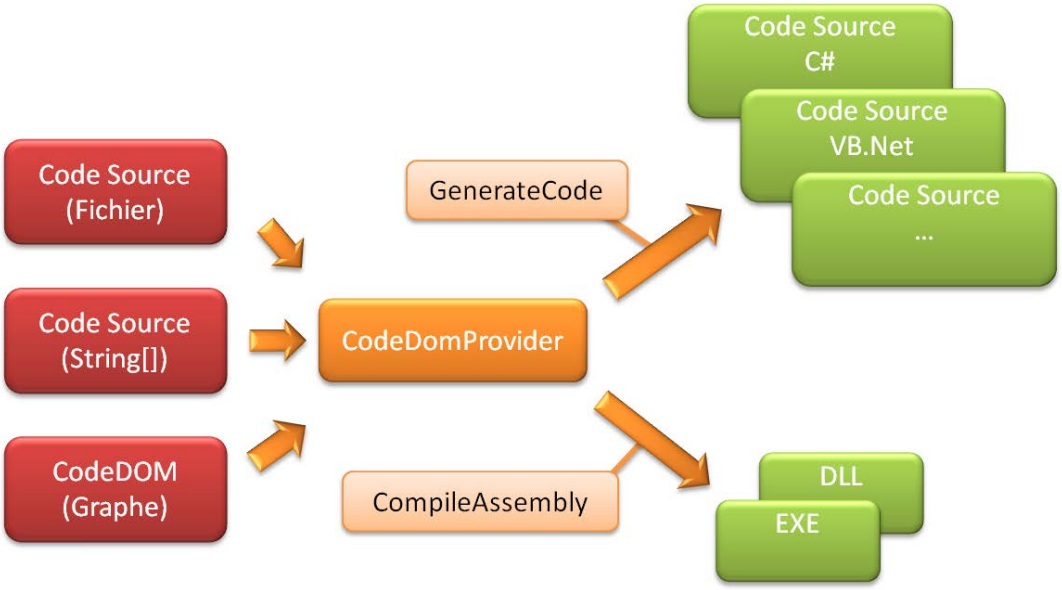

Figure 30 - Schéma de principe CodeDOM

Le composant de Microsoft fournit un ensemble de classes pour représenter un code source sous forme de graphe donc indépendamment du langage cible. Dans ce modèle la classe « CodeDomProvider » fournit un ensemble de services permettant de générer ou de compiler du code (voir Figure 30 - Schéma de principe CodeDOM).
En entrée, elle accepte :

- Du code source directement (sans vérification)
  - Sous forme de fichier (emplacement du fichier sur le système)
  - Sous forme de chaîne de caractères (contenu dans une variable de type chaîne de caractères)
- Graphe CodeDOM de type CodeCompteUnit (voir Figure 32 - Exemple d'un arbre dans CodeDOM).

CodeDomProvider possède deux méthodes suivant la forme de l'objet de sortie désiré :

- GenerateCode : Génère du code source (C#/ VB / JS / J# / …).
- CompileAssembly : Génère un exécutable ou une bibliothèque logicielle.

Les classes disponibles dans cette bibliothèque logicielle permettent de représenter chaque sous-ensemble d'un code source sous forme d'un graphe. Voici les principaux types (CodeDOM) et les objets qu'ils représentent :

- CodeCompileUnit : Classe - Conteneur de base pour un graphe CodeDOM.
- CodeNamespace : Classe - Représente une déclaration d'espace de nom.
- CodeType : Interface – Est implémenté par les classes suivantes.

    - CodeClass : Classe – Représente la déclaration d'une classe.
    - CodeStructure : Classe – Représente la déclaration d'une structure.
    - CodeInterface : Classe – Représente la déclaration d'une interface.
    - CodeEnum : Classe - Représente une déclaration d'énumérations.
    - CodeDelegate : Classe – Représente une déclaration d'un délégué.
- CodeStatement : Classe abstraite : base à partir de laquelle toutes les instructions de code dérivent (par exemple : CodeAssignStatement, CodeVariableDeclarationStatement, CodeConditionStatement …).
- CodeExpression : Classe abstraite : base à partir de laquelle toutes les expressions de code dérivent (par exemple : CodeMethodInvokeExpression, CodePropertyReferenceExpression, CodeTypeReferenceExpression …)

L'avantage de CodeDOM est une validation au niveau hiérarchique d'un code généré et également d'une compilation. La principale contrainte est la complexité du processus de création (voir Figure 32) d'un graphe CodeDOM étant donné le nombre d'objets disponibles, plus de 80 objets (voir annexe 3) et également le niveau d'abstraction supplémentaire venant du fait qu'on modélise un ensemble d'objets qui est lui-même un modèle.

```
private void GenerateHelloWordClass()
{
    //Déclaration du conteneur de base
    CodeCompileUnit compileUnit = new CodeCompileUnit();

    //Déclaration de l'espace de nom "EspaceDeNom1"
    CodeNamespace namespace1 = new CodeNamespace("EspaceDeNom1");

    //Ajout d'une référence dans l'espace de nom
    namespace1.Imports.Add(new CodeNamespaceImport("System"));

    //Ajout de l'espace de nom au conteneur de base
    compileUnit.Namespaces.Add(namespace1);

    //Déclaretion d'une classe "Classe1"
    CodeTypeDeclaration classe1 = new CodeTypeDeclaration("Classe1");

    //Ajout de la classe "Classe1" à l'espace de nom "EspaceDeNom1"
    namespace1.Types.Add(classe1);

    //Déclaration d'une methode de point d'entrée de code pour un exécutable
    CodeEntryPointMethod start = new CodeEntryPointMethod();

    //Déclaration d'une référence a un Type
    CodeTypeReferenceExpression typeReferenceExpression =
        new CodeTypeReferenceExpression(typeof(System.Console));

    //Déclaration d'une expresssion primitive
    CodePrimitiveExpression primitiveExpression =
        new CodePrimitiveExpression("Hello World!");

    //Déclaration d'une expression, qui exécutera la méthode WriteLine
    //du type "System.Console" avec comme argument "Hello World!"
    CodeMethodInvokeExpression cs1 =
        new CodeMethodInvokeExpression(typeReferenceExpression,
                                       "WriteLine",
                                       primitiveExpression);
    //Ajout de la méthode
    start.Statements.Add(cs1);

    //Ajout du point d'entrée
    classe1.Members.Add(start);
}
```

Figure 31 - Exemple de code CodeDOM pour la génération d'une classe

Description de la construction de l'arbre CodeDOM de la Figure 31.

Création d'un conteneur racine

La première étape pour la création d'un arbre de type CodeDOM est la déclaration du conteneur de base de type CodeCompileUnit.

Création d'un espace de nom

Ensuite il faut déclarer un espace de nom via une référence vers un objet de type CodeNamespace instancié avec un paramètre contenant son nom. Les références sont ensuite ajoutées dans l'espace de nom via la méthode Add de la collection Imports.

Ajout d'une classe

L'étape suivante consiste à déclarer une référence vers un objet de type CodeTypeDeclaration qui représente un type (classe, structure, etc.) puis de l'ajouter à la collection des types de l'espace de nom.

Création du point d'entrée du programme
Il faut ensuite déclarer un point d'entrée à travers une référence vers un objet de CodeEntryPointMethod.

Ajout de l'appel d'une méthode d'écriture dans la console
Pour cela il faut tout d'abord déclarer une référence vers un type « CodeTypeReferenceExpression » qui représente lui-même une référence vers un type. L'instanciation se fait avec un paramètre qui est une référence vers le type de l'objet possédant la méthode. La référence vers le type de l'objet est effectuée à l'aide de l'opérateur « typeof ». Il faut ensuite créer une référence vers un objet de type « CodePrimitiveExpression » qui contient la chaîne de caractères à passer en paramètre de la méthode d'écriture dans la console. La dernière étape consiste à déclarer une référence vers un objet de type CodeMethodInvokeExpression qui représente l'appel d'une méthode. L'instanciation se fait avec 3 paramètres :

- La référence vers le type de l'objet qui contient la méthode à appeler (typeReferenceExpression)
- Le nom de la méthode à appeler (« WriteLine »)
- Une collection d'objets contenant la liste des arguments à fournir à la méthode appelée (primitiveExpression)

Il ne reste alors qu'à ajouter l'appel de la méthode (cs1) à la collection de déclaration du type (Statements). La Figure 32 montre l'arbre CodeDOM de l'exemple décrit ci-dessus.

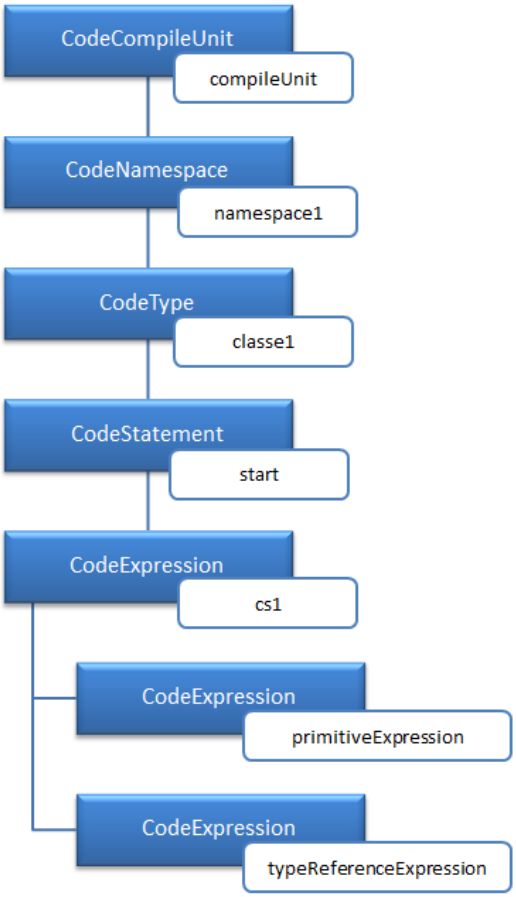

Figure 32 - Exemple d'un arbre dans CodeDOM

Une fois compilé en langage C#, l'arbre CodeDOM de la Figure 32 donne le résultat suivant :

```
using System;

namespace ExempleDeNom1
{
    class Classe1
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello Word!");
        }
    }
}
```

Figure 33 - Résultat de la génération de source de l'exemple de l'arbre CodeDOM

2.2.5. Comparaison

Voici un récapitulatif de l'étude des principales technique de génération de code source.

Tableau 3 - Comparaison des méthodes de génération de code source

	Moteur personnalisé	XML & XSL	T4 Text Template	CodeDOM
Mise en œuvre	Difficile	Moyenne	Moyenne	Difficile
Indépendance du modèle	✗	✗	✗	✓
Langage supporté nativement	✗	✗	✗	C# et VB.Net
Maintenance	Difficile	Moyenne	Moyenne	Moyenne
Point fort	- Ne nécessite aucune connaissance particulière	- Facilité de mise en place	- Facilité de mise en place	-Indépendance du modèle - Langages C# et VB.Net supportés nativement
Point faible	- A réaliser entièrement -Maintenance difficile - Complexité de code à réaliser - Nécessité de développer tous les modèles en double pour supporter C# et VB.Net	- Pas d'indépendance du modèle - Nécessité de développer tous les modèles en double pour supporter C# et VB.Net	- Pas d'indépendance du modèle - Nécessité de développer tous les modèles en double pour supporter C# et VB.Net	- Complexité - Certaines expressions ne sont pas prises en charge dans le modèle

Légende	✓ Inclus ✗ Exclus

La technique du moteur de génération personnalisé, bien que ne nécessitant peu de connaissances, est écartée au vu de la charge de travail pour la réalisation du modèle de génération de codes sources.
Les modèles de textes T4 et CodeDOM sont les deux solutions les plus envisageables. La technologie CodeDOM a été préférée grâce à ses capacités de générer nativement le C# et le VB.Net.

2.3. Conception d'IHM

L'interaction homme-machine (IHM) est concernée par l'étude, la gestion et la conception des interactions entre les humains (utilisateurs) et les machines (ordinateurs). Ce domaine constitue un des aspects majeurs de la réalisation d'un programme informatique [DAV10]. Les conséquences d'une mauvaise conception d'IHM peuvent être importantes, preuve en est l'accident de « Three Mile Island » dont l'interprétation erronée d'un signal par les opérateurs a provoqué un accident d'une grande envergure.

L'objectif de ce paragraphe est de présenter une étude portant sur la conception des composants visuels nécessaires pour la création des formulaires de manipulation des données de référence. Tout d'abord le modèle Présentation Abstraction Contrôle (PAC), un des modèles d'architecture les plus reconnus, est présenté. Puis une analyse des différentes techniques d'implémentation de ce modèle est conduite. Les points suivants sont abordés : monolithique, héritage, et génération à partir de métadonnées.

2.3.1. Modèle Présentation Abstraction Contrôle

Le modèle d'architecture logicielle Présentation Abstraction Contrôle (PAC) pour les IHM a été introduit par la chercheuse en informatique grenobloise Joëlle COUTAZ en 1987. Ce dernier repose sur l'organisation des IHM en hiérarchie de composants (voir Figure 34).

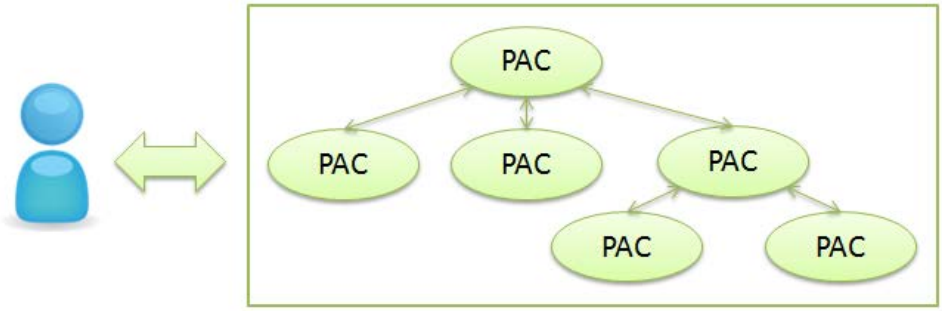

Figure 34 - Organisation hiérarchique du modèle PAC

Chaque composant est lui-même organisé en 3 facettes (voir Figure 35) :

La Présentation est chargée de l'interaction avec l'utilisateur aussi bien au niveau des entrées (clavier, souris, …) que des sorties (gestion de l'affichage, sons …). L'ensemble des facettes de Présentation constitue l'IHM d'un programme.

L'Abstraction gère les données à représenter. Il a pour fonction la manipulation et le traitement de ces données. Il contient donc la logique fonctionnelle.

Le Contrôle gère la correspondance entre les deux autres facettes : cohérence des représentations avec les données internes, conversion des actions de l'utilisateur en opérations du noyau fonctionnel. Les facettes de contrôle servent aussi à créer une hiérarchie de composants logiciels pour organiser le programme : la facette de contrôle du composant parent communique avec celle du composant fils.

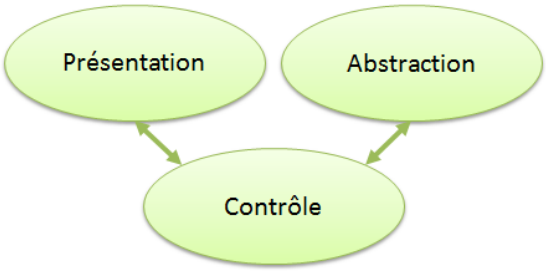

Figure 35 - Composition modèle PAC

Le patron de conception Modèle-Vue-Contrôleur (MVC) introduit en 1979 autour du langage Smalltalk est souvent confondu avec PAC. Or ce dernier est beaucoup plus orienté vers la séparation entre IHM et noyau fonctionnel. Dans le modèle MVC l'accent est mis sur l'organisation des entrées et des sorties dans les composants logiciels. D'ailleurs la notion de « Contrôle » est différente dans les 2 modèles. Dans PAC, il a pour rôle de contrôler la cohérence des facettes logicielles entre elles. Dans MVC, il doit gérer les entrées de l'utilisateur et contrôler l'exécution du programme.

2.3.2. Monolithique

Le cas de figure le plus basique est le cas où une classe ou un conteneur quelconque encapsule les notions de présentation et de contrôle. Ceci implique que pour chaque formulaire un conteneur est nécessaire. Il n'y a donc aucun mécanisme de réutilisabilité (voir Figure 36).

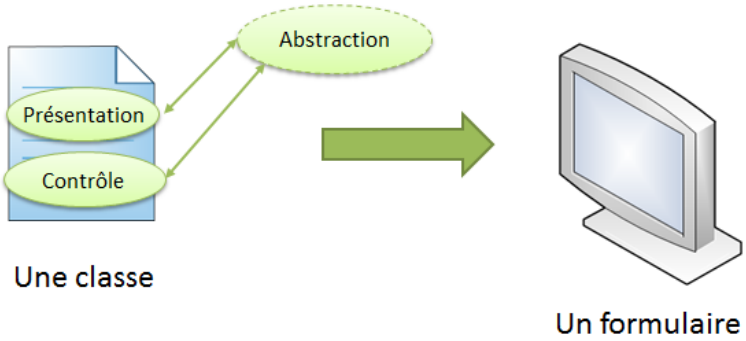

Figure 36 - Technique de conception « Monolithique »

La facette Abstraction est déportée dans le schéma car elle est utilisée en tant que référence. Il est tout à fait possible de séparer la notion de présentation et de contrôle dans des classes différentes.

L'inconvénient majeur est notamment de multiplier le code pour des formulaires peu différents. Aucune notion de réutilisabilité est mise en œuvre dans cette approche. Le code est peu évolutif.

2.3.3. Avec modèle

Ce type de technique d'IHM est dérivé du concept d'héritage de la programmation par objet. Le but est de rassembler tous les éléments communs de plusieurs formulaires en un composant (modèle). Ce dernier est ensuite personnalisé pour chaque formulaire avec une partie du code, et donc du comportement, est commune (voir Figure 37).

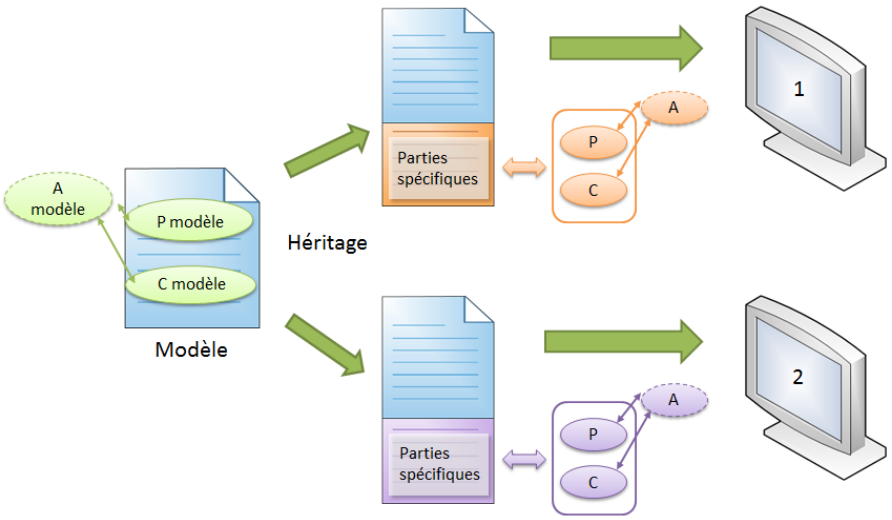

Figure 37 - Modèle de programmation avec modèle

Comme pour la technique précédente, le choix de représenter dans une seule classe les facettes Présentation et Contrôle n'est pas une obligation. Cette technique met en commun des comportements et des représentations visuelles identiques.

2.3.4. Génération à partir de métadonnées

Une des méthodes de gestion évoluée des IHM est la génération d'IHM à partir de métadonnées. Le fonctionnement de la génération s'appuie d'une part, sur une description des IHM appelées métadonnées et d'autre part, sur un processus de création qui prend en compte les règles à appliquer pour la création des formulaires en fonction des contextes rencontrés (voir Figure 38).

Les métadonnées peuvent décrire directement les aspects de personnalisation visuelle de l'IHM. Dans un niveau d'abstraction supérieur, les métadonnées peuvent décrire directement les objets métiers de l'application. Dans ce dernier cas, non seulement les aspects d'ergonomie et de positionnement mais également la nature même des contrôles visuels mis en jeux sont fixés par les règles de création d'IHM.

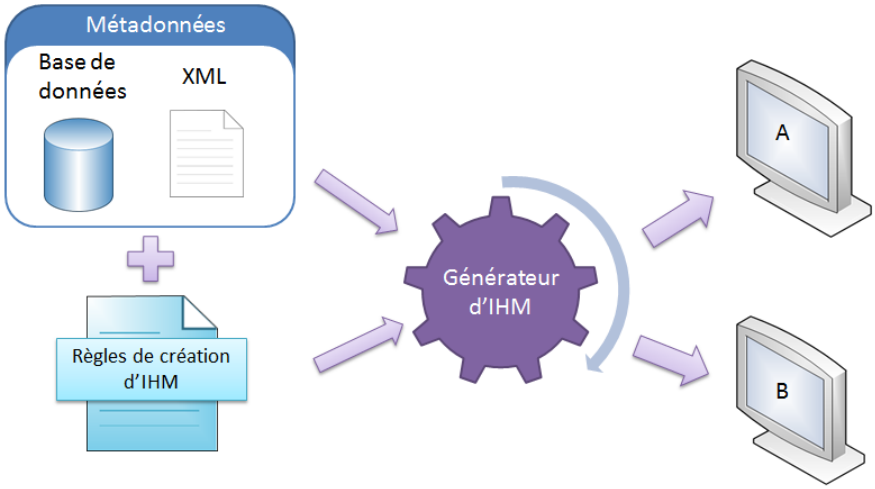

Figure 38 - Technique de génération de formulaire à l'aide de métadonnées

Sur la Figure 38, les métadonnées sont des données stockées soit dans une base de données, soit dans un fichier XML. Elles peuvent prendre toutes formes à la condition qu'elles soient lisibles par le générateur d'IHM. Le processus prend, en entrée, les métadonnées d'un côté et de l'autre les règles de création de l'IHM, qui peuvent être définies sous la forme d'un programme.

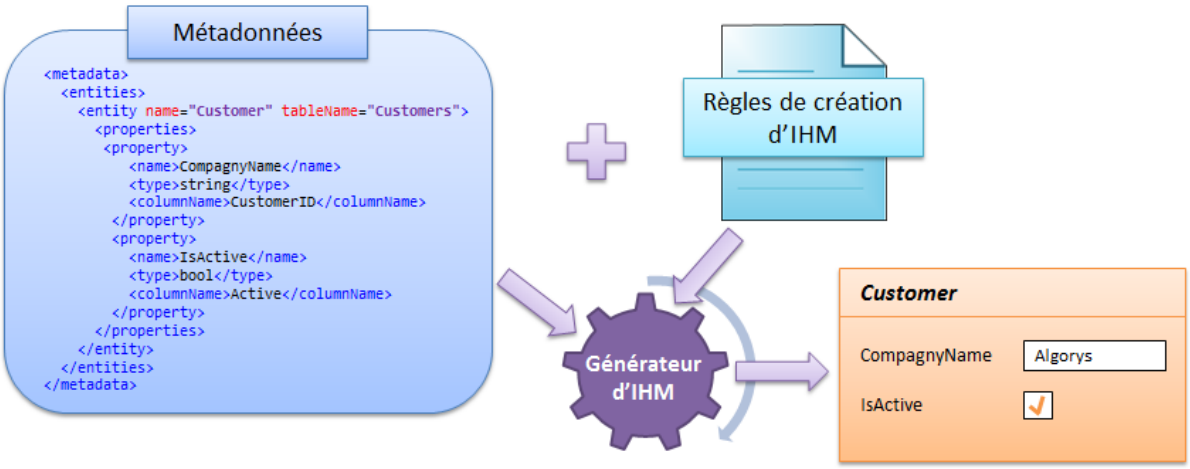

Figure 39 - Exemple de génération d'IHM

Dans l'exemple de la Figure 39, les métadonnées sont stockées sous la forme d'un fichier XML. Ce fichier contient une entité « Customer » avec une liste de propriétés. Ces métadonnées, associées aux règles de création de l'IHM, représentent les entrées du processus prises en charge par le générateur d'IHM qui va générer un formulaire de traitement personnalisé pour l'entité « Customer » avec un champ de saisie en fonction du type de données.

2.3.5. Design Patterns : IHM

La notion de design pattern (patron de conception) a été introduite dans les années 1970 par l'architecte Christopher Alexander. L'idée était de capitaliser les solutions efficaces trouvées face aux problèmes récurrents dans l'architecture urbaine. Cette notion réapparait ensuite en 1995 dans le domaine de la conception logicielle dans le livre intitulé « Design Patterns - Elements of Reusable Object-Oriented Software » écrit par « the gang of Four » [GOF94].

Dans le contexte de la présentation de données de référence, le but est de rechercher des design patterns d'IHM permettant d'offrir une ergonomie appropriée pour effectuer les fonctionnalités de base (récupération, création, modification, suppression) pour un utilisateur et pour le maximum de types de données. Dans ce cadre nous nous intéressons tout particulièrement aux deux patterns suivants :

Data grid pattern

Problème : Affichage et manipulation d'une série d'informations de même nature.

Solution : Utilisation d'une grille de données pour l'affichage et quelques interactions.

Avantage : Quantité d'informations affichées maximisée, possibilité de combiner avec des patterns de type tri sur colonne, pagination, cache …

Inconvénients : Mise en page complexe suivant le type de données utilisées, notamment au moment de la mise à jour ou de la création. Si le nombre de propriétés des objets est très grand, l'ergonomie devient inappropriée.

		picture	categoryname	description
Modifier	Supprimer	17	zerzer	sdfffffffffffffffffffff
Modifier	Supprimer	9	TEST2	test
Modifier	Supprimer	11	test	test2
Modifier	Supprimer	10	test	test2
Modifier	Supprimer	8	Seafood	Seaweed and fish
Modifier	Supprimer	7	Produce	Dried fruit and bean curd
Modifier	Supprimer	6	Meat/Poultry	Prepared meats
Modifier	Supprimer	5	Grains/Cereals	Breads, crackers, pasta, and cereal
Modifier	Supprimer	4	Dairy Products	Cheeses
Modifier	Supprimer	3	Confections	Desserts, candies, and sweet breads

1 2

Figure 40 - Exemple de mise en œuvre du Data grid pattern

Form pattern

Problème : Affichage et manipulation d'une série d'informations

Solution : Utilisation d'une grille de données pour l'affichage et quelques interactions.

Avantage : Plus adapté à la saisie d'information, car l'affichage est plus clair et permet d'utiliser un plus grand nombre de composants graphiques.

Inconvénients : La manipulation de données se limite à un seul objet à la fois.

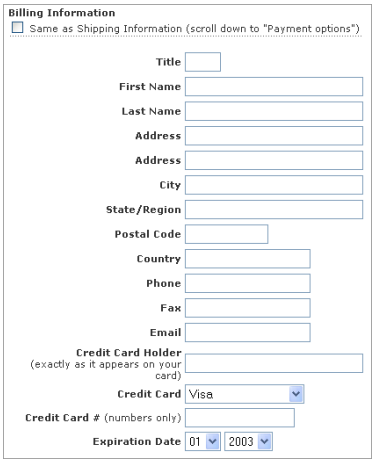

Figure 41 - Exemple d'une mise en œuvre du « Form pattern »

2.3.6. Solution retenue

Au cours de cette étude sur la conception d'IHM, il apparaît que l'approche « monolithique » est à proscrire, en raison de la duplicité du code inhérent au fait de la similitude des formulaires. Les formulaires de manipulation des données de référence étant relativement simples et, étant donné l'objectif de simplifier la réalisation d'écrans, l'approche par modèle apparaît la plus adaptée. L'approche par métadonnées est plus efficace en termes de réduction du code et de souplesse d'utilisation malgré la complexité de sa mise en œuvre. Une solution hybride consiste à combiner l'approche par métadonnées avec une approche classique en s'appuyant sur des design patterns simples (Data Grid et Form).

L'objectif de l'étude est de trouver la solution optimale pour générer des formulaires fonctionnels (directement utilisables pour les utilisateurs), ergonomiques et personnalisables (pour adapter visuellement les formulaires aux programmes dans lesquels ils sont utilisés). Précédemment, plusieurs approches ont été

étudiées en termes de conception d'IHM. En conclusion, il est apparu comme optimale une solution hybride entre une technique de conception de formulaires à l'aide de modèles et de métadonnées. Cette décision a engendré deux problèmes détaillés ci-dessous.

Le premier problème est le manque des informations pour que les formulaires générés soient directement utilisables. La principale information est le libellé utilisateur de chaque nom de colonne. Ces libellés sont en général des codes et ne sont pas directement utilisables dans le formulaire.

Le deuxième problème est d'ordre purement technique. Les deux types de formulaires sont prévus pour des applications « Web » avec des contraintes techniques au niveau des composants Microsoft utilisés. Le fonctionnement des pages ASPX étant plutôt strict, on ne peut pas utiliser tous les composants dans toutes les conditions.

Pour résoudre le problème des libellés, une gestion des métadonnées en XML a été réalisée. Pour chaque composant graphique à générer, un fichier XML de configuration est également produit. Les données contenues dans ce fichier viennent surcharger, au moment de l'exécution, les valeurs définies par défaut par l'outil.

Les métadonnées sont composées d'un élément racine, appelé MetaDataView qui regroupe les informations sur la vue (Listes des clés primaires, le nom de l'objet en base de données, le type de l'objet (table/vue), son identifiant, et la liste de ses propriétés) (voir Figure 42).

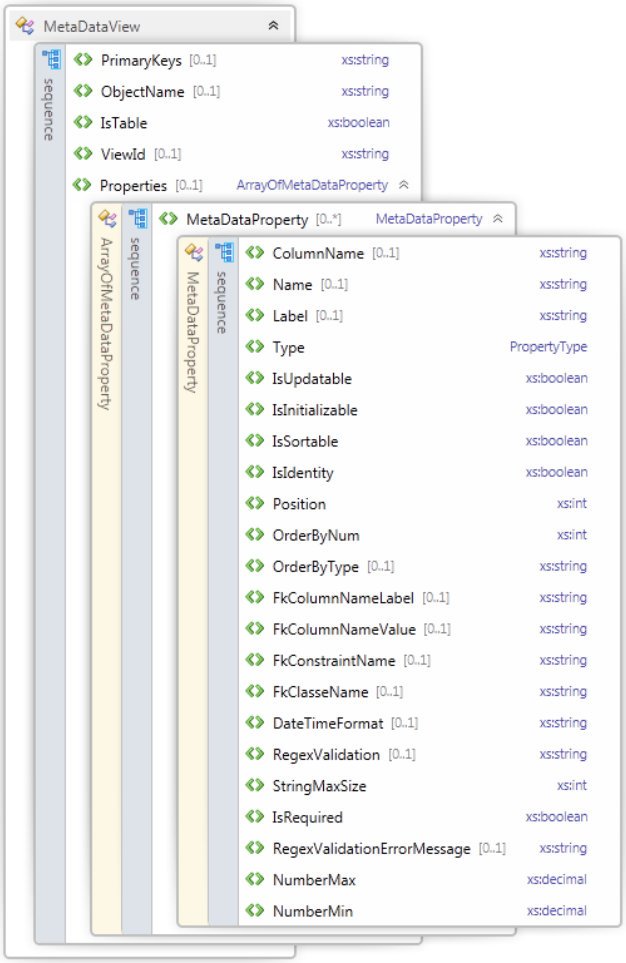

Figure 42 - Détail du schéma XML des métadonnées – MetaDataColumn

Voici la description des propriétés de l'objet MetaDataView

Propriétés de base

- ColumnName : nom de la colonne dans la base de données. Utilisé en tant que clé.
- Label : Libellé affiché dans l'interface utilisateur.

Personnalisation du fonctionnement

- IsUpdatable : Autorisation de modification de la donnée (UPDATE)
- IsInitializable : Autorisation de l'initialisation de la donnée (INSERT)
- IsSortable : Autorisation d'effectuer un tri sur la propriété
- IsIdentity : Compteur automatique
- OrderByNum : Ordre du tri (si tri sur plusieurs propriétés)
- OrderByType : Type du tri (croissant ou décroissant)
- IsRequired : Valeur obligatoire

Gestion des associations

- FkColumnNameLabel : Propriété à afficher dans la liste déroulante
- FkColumnNameValue : Propriété pour l'identifiant dans la liste déroulante
- FkConstraintName : Nom de la contrainte
- FkClasseName : Nom de la classe pour la gestion de la contrainte

Personnalisation de l'affichage

- Position : Position à l'affichage
- Type : Type (exemple : date, heure, entier, décimal, chaîne de caractères)
- DateTimeFormat : Format d'affichage

Validation des données

- RegexValidation : Expression régulière à valider
- RegexValidationErrorMessage : Message d'erreur lors de l'échec de la validation de l'expression régulière
- StringMaxSize : Taille maximum de la chaîne de caractère
- NumberMax : Nombre maximum
- NumberMin : Nombre minimum

Les métadonnées ne concernent volontairement qu'un nombre restreint de propriétés. Le but est de permettre facilement la personnalisation des propriétés principales, couvrant 80% des besoins, via les métadonnées. Pour le reste de la personnalisation des formulaires, la solution envisagée a été de s'appuyer sur le concept d'héritage. Ce concept s'applique sans aucun problème à des classes purement objet mais, dans le cas de composants visuels et surtout dans l'environnement « Web » des composants Microsoft, des contraintes sont apparues.

Pour bien les comprendre, il faut avoir connaissance des bases de l'architecture technologique « Web » de Microsoft. Le kit de développement ASP.NET est donc détaillé ci-dessous. En ASP.net on parle de « WebForms » par opposition aux « WinForms » dédiées à un formulaire en client lourd de type Windows.

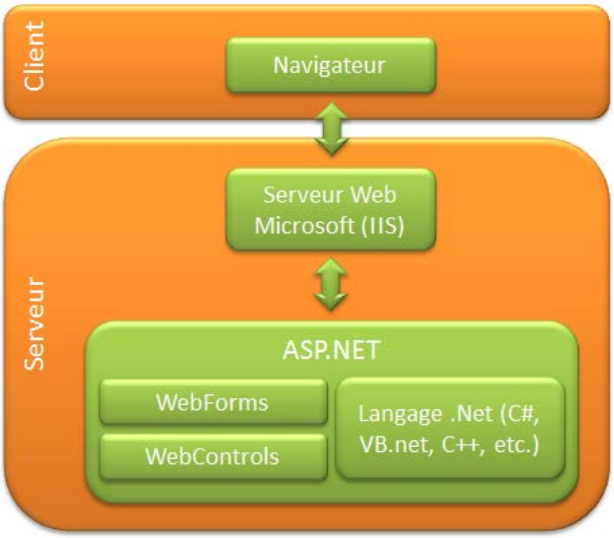

Figure 43 - Architecture globale ASP.NET

L'architecture ASP.Net est équivalente à celle d'un langage orienté serveur comme PHP. Elle se compose de deux aspects distincts : un aspect de déclaration des éléments web (WebForms et WebControls) contenus dans un fichier aspx ou ascx et un autre de logique (au sens programmation) de page avec un des langages .Net contenu dans un fichier de code source associé au premier appelé Code-Behind (voir Figure 43).

La partie Web (« Page Web Forms » sur la Figure 44) contient du code statique HTML ainsi que du code ASP.NET dit « in line » car il s'exécute au moment de la compilation à la manière d'une page ASP classique. Toutes les balises HTML peuvent être utilisées et associées à des balises ASP.NET (exemple : <asp:textbox> </asp:textbox>). A la différence des balises HTML qui sont interprétées par le navigateur (donc côté client), les balises ASP.NET sont interprétées par le serveur Web de Microsoft (Internet Information Service, IIS), qui renvoie le résultat sous forme HTML au navigateur. La sauvegarde de l'état des éléments composants une page entre les différents appels d'une même page (POSTBACK) est effectuée par un élément nommé ViewState. Il permet, par exemple lors d'un rafraîchissement d'une page, de sauvegarder un formulaire qui contient des données. Grâce à l'utilisation du ViewState, il est donc possible de manipuler les données côté serveur (avant l'envoi au client, ou après la validation d'un formulaire). Ces manipulations ont lieu dans le composant « Code-Behind » (voir Figure 44). Ce dernier est un fichier qui contient uniquement des instructions .NET. Il possède un gestionnaire d'évènements qui offre la possibilité d'exécuter du code à différents moment du cycle de vie d'une page (Initialisation, chargement, rendu, etc.). Il est également possible de définir des gestionnaires d'évènements sur les éléments ASP.NET de la « Page Web Forms ». Par exemple, un champ de saisie texte peut être initialisé avec la date du jour. Il est également possible de définir une méthode qui s'exécutera à chaque changement de la valeur sélectionnée dans une liste déroulante ASP.NET (Cela aura pour effet de renvoyer la page à chaque modification de la valeur de la liste déroulante).

Un exemple de page « web forms » et de « code-behind » est illustré sur la Figure 44. Dans le fichier de déclaration des balises HTML « body » et « h1 » sont associés aux balises ASP.NET déclarant deux champs de saisie (<asp:TextBox>). Le fichier « code-behind » est utilisé pour décrire la logique de contrôle de la page. Par exemple la méthode à exécuter, lorsque l'évènement « clic » du bouton se produit, est définie dans le « code-behind » et des références seront créés automatiquement vers les éléments serveur de la page « web forms », en l'occurrence les deux champs textes.

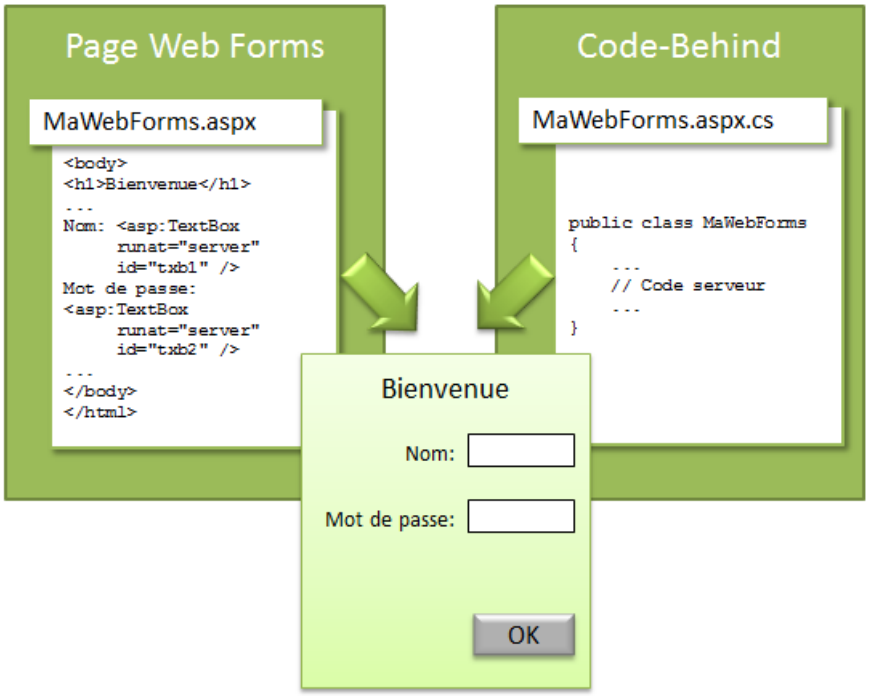

Figure 44 - Web Forms : Structure des fichiers

Pour en revenir au problème de personnalisation des propriétés et éléments non pris en charge par le système de métadonnées, la solution envisagée est de pouvoir hériter des composants personnalisés pour surcharger les propriétés visuelles et, de cette manière, résoudre notre problème de personnalisation.

La contrainte technique vient d'une restriction sur l'héritage des composants personnalisés, qui limite l'héritage aux composants dont l'aspect déclaration est vide. Tous les contenus de la partie visuelle habituellement décrits dans la page Web Forms doivent donc être transférés dans le Code-Behind et instanciés de manière dynamique.

2.4. Bilan de la recherche

Au niveau ORM, grâce aux informations provenant de la recherche de solutions, il a été décidé de réaliser notre propre générateur d'ORM plutôt que de se baser sur un outil existant. En effet, les limitations en termes de versions de Framework supportées étaient incompatibles avec les futurs projets de la société Algorys. Dans la plupart des cas, l'environnement logiciel d'un client va du très récent, à des systèmes qui peuvent atteindre plus d'une dizaine d'années. Dans ce contexte il était difficile d'imposer la limitation d'une technologie qui a moins de 3 ans (Framework .Net 4.0). Un développement personnalisé a donc été préféré à l'utilisation des technologies de Microsoft (Linq To SQL et Entity Framework). Nhibernate et Subsonic ont été écartés respectivement en raison de la complexité de mise en œuvre et des faibles performances des requêtes au niveau du temps d'exécution.

Les contraintes du projet concernant l'approche de l'ORM sont doubles. Premièrement, la mise en œuvre et l'utilisation doivent être facilitées. La cible étant de petits projets, une solution trop complexe à déployer n'est rentable ni pour la société ni pour le client. Deuxièmement, il faut que cette approche amène suffisamment d'abstraction par rapport au modèle physique afin de faciliter la programmation de la logique métier. L'approche répondant au mieux à ces contraintes est l'approche par entité. C'est-à-dire que chaque table a :

- une classe « état » contenant une propriété par colonne

- une classe « méthode » permettant d'effectuer les actions « CRUD » dans la base de données.

Via l'héritage, les classes peuvent être étendues sans modifier les fichiers générés. Par cette technique, il est possible aux développeurs de personnaliser le code généré en ajoutant des propriétés et des méthodes métiers aux classes générées et donc d'implémenter la logique métier (règles fonctionnelles supplémentaires). Ceci permet de ne pas modifier les fichiers générés et évite le risque de voir la personnalisation des classes perdue lors d'une nouvelle génération de la couche ORM.

A propos de la technique de génération de codes sources, le choix de passer par la technique du CodeDOM a été retenu. Elle est la seule à proposer un modèle indépendant, arbre CodeDOM, pour représenter le code et un générateur multi-langages (.NET).

En ce qui concerne la conception IHM, la solution retenue est de passer par la création de contrôle personnalisé. Un contrôle personnalisé est un ensemble d'éléments visuels (champs de saisies, grilles, boutons, etc.) disposé sur un conteneur associé à son code de gestion. Ce conteneur devient alors lui-même un élément visuel utilisable comme tel dans une IHM.

Cette solution nous permet de mettre plusieurs design patterns d'IHM en œuvre, à savoir :

- Form pattern, affichage du détail des données, modification et création de données.
- Data grid pattern, affichage d'une collection de données, suppression d'une donnée, sélection d'une donnée.
- Pagination pattern, gestion de l'affichage d'une grande quantité de données, optimisation du temps de chargement des données, amélioration de l'ergonomie.

Dans ce chapitre, une étude sur les thèmes de l'ORM, la génération de code source et la conception de l'IHM ont été présentées. Une solution personnalisée pour les deux premiers thèmes a été préférée. La génération de la couche ORM utilise une approche par entité en s'appuyant sur le fournisseur standard de données de Microsoft (ADO.NET), solution la plus performante. Le code est généré avec la bibliothèque CodeDOM qui fournit une abstraction entre la structure de la base de données et le modèle objet. La conception IHM s'appuiera sur plusieurs techniques de génération d'IHM avec métadonnées, composant d'IHM modèle et les patterns (form, data grid, pagination). La mise en œuvre de ces solutions est décrite dans le chapitre suivant.

3. Réalisation et tests

Dans ce chapitre, la réalisation de l'outil est détaillée. Dans un premier temps, Nous décrivons l'architecture, le paramétrage, l'IHM de l'outil et la configuration d'une application ASP.NET utilisant l'outil. Puis nous passons la revue de la réalisation de l'aspect ORM, puis la réalisation de l'aspect génération de composant IHM puis la gestion des tests.

3.1. Architecture logique de l'outil

Comme tout projet informatique, la conception de l'outil AlgorysORM a fait l'objet d'une étude conceptuelle au niveau de l'architecture des classes. Une réflexion a également été menée concernant le découpage en bibliothèque dans la solution. Lors de la conception, les deux contraintes les plus importantes étaient :

- de constituer un découpage logique de chaque bibliothèque logicielle afin de faciliter les évolutions et la maintenance de l'outil (aspect présentation, aspect ressources, et bibliothèque logicielle principale de génération et de configuration appelée Core).
- de faire en sorte que la configuration d'une application cible (c'est-à-dire un programme qui utilise la couche ORM et les formulaires générés) ne nécessite l'ajout que d'une référence à une bibliothèque logicielle (AlgorysORM.Reference) de manière à en simplifier le déploiement et la mise en route.

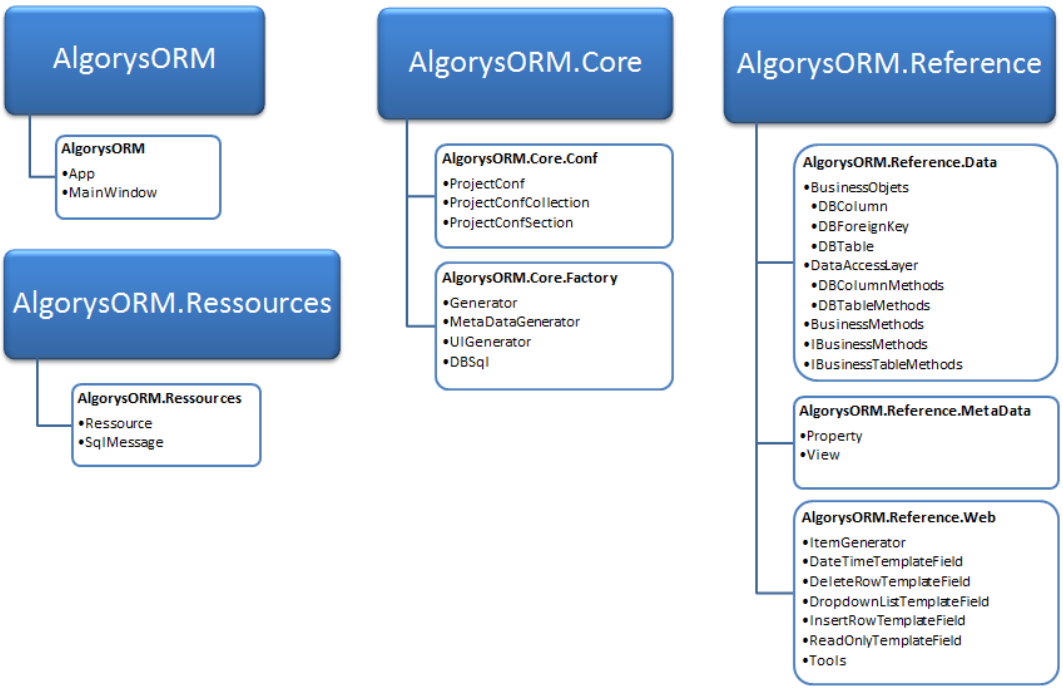

Figure 45 - Architecture logique des modules AlgorysORM

L'outil AlgorysORM se compose donc de 4 modules (voir Figure 45):
AlgorysORM – contenant l'interface graphique de l'outil et sa logique
AlgorysORM.Core – constitué des deux espaces de noms (voir glossaire des termes techniques) :

- **AlgorysORM.Core.Conf** – gestion du paramétrage d'une configuration de génération
- **AlgorysORM.Core.Factory** – regroupement des objets de génération de codes sources

AlgorysORM.Ressources – ensemble des ressources de type texte.

AlgorysORM.Reference – ensemble des espaces de noms (voir glossaire des termes techniques) nécessaire lors de l'exécution du code généré (ORM ou IHM)

- **AlgorysORM.Reference.Data** – référence pour les objets entités ou méthodes de la couche ORM générée
- **AlgorysORM.Reference.MetaData** – référence pour la gestion des métadonnées des contrôles dans l'IHM
- **AlgorysORM.Reference.Web** – fournit les classes de base pour la gestion des modèles de colonnes (suffixées par « TemplateField »), la classe d'aide dans les contrôles personnalisés (Tools) ainsi que la classe qui fournit les méthodes d'initialisations des éléments visuels du contrôle personnalisé (ItemGenerator)

Le schéma de dépendance des bibliothèques (voir Figure 46) montre que la bibliothèque « AlgorysORM.Reference » fait elle-même référence à la bibliothèque « AlgorysORM.Ressource ». Cette référence est en contradiction avec le fait que l'application client ne fasse référence qu'à une seule bibliothèque logicielle. Cette logique a été choisie car elle permet de facilement créer un mécanisme de sélection de ressources en fonction de la région.

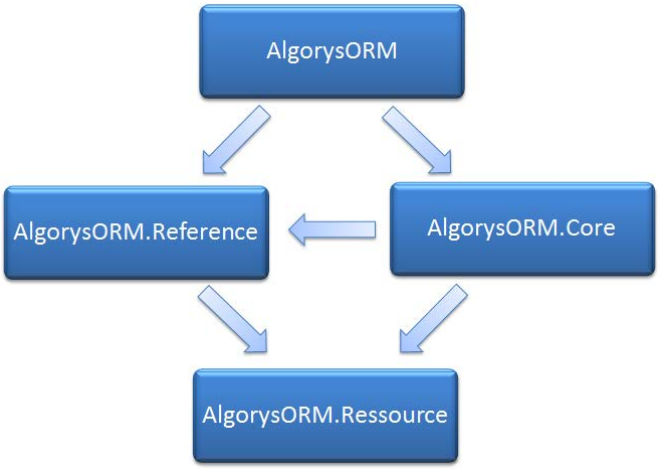

Figure 46 - Dépendances des bibliothèques AlgorysORM

Le schéma suivant illustre l'utilisation des méthodes de la bibliothèque AlgorysORM.Reference par l'application utilisatrice du générateur.

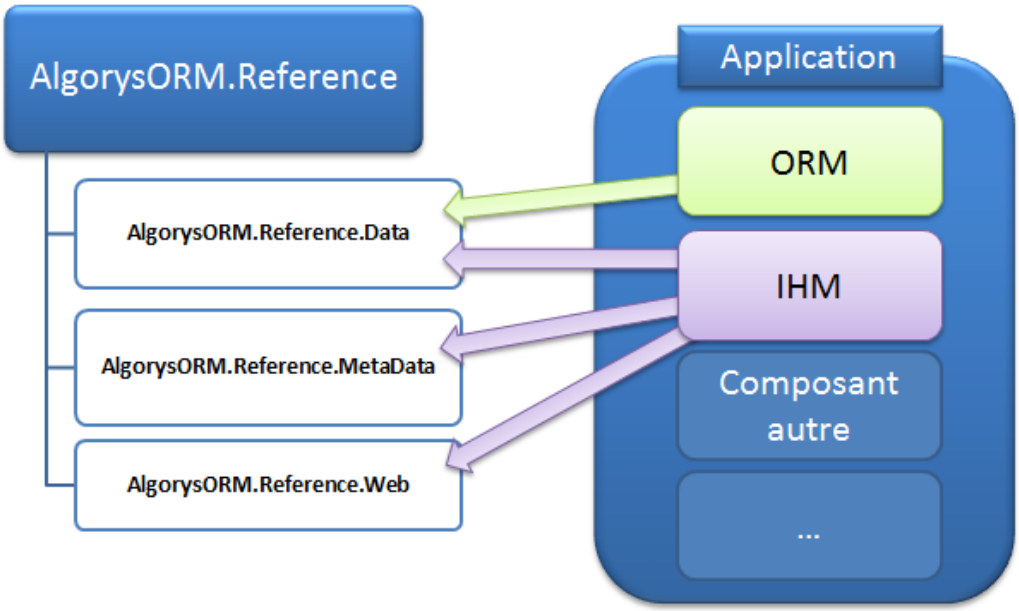

Figure 47 - Schéma de référencement entre l'application et la bibliothèque logicielle AlgorysORM.Reference

3.2. Paramétrage de l'outil

Avant le lancement de l'application il faut tout d'abord créer une configuration. Celle-ci doit contenir les paramètres nécessaires à l'outil AlgorysORM pour la génération du code source et notamment les informations pour se connecter à la base de données.

Les paramètres sont :

- Name : Nom de la configuration
- ConnectionString : Chaîne de connexion à la base de données
- ProviderName : Nom de fournisseur de données, utilisé pour la connexion à la base de données.
- SourcesGenerationPath : Répertoire de sortie pour la génération des fichiers ORM.
- GUIGenerationPath : Répertoire de sortie pour la génération de composants IHM.
- MetaDataGenerationPath : Répertoire de sortie pour la génération des métadonnées.
- SourcesNamespace : Espace de nom pour le code source ORM.
- GUINamespace : Espace de nom pour le code source IHM.
- DefaultORMValue : Valeur par défaut pour l'option génération du code source ORM.
- DefaultUIValue : Valeur par défaut pour l'option génération du code source IHM.
- DefaultMetaDataValue : Valeur par défaut pour l'option génération des métadonnées.

Les informations de configuration sont stockées sous forme de texte dans le fichier « app.config » dans la bibliothèque AlgorysORM. Il est possible de définir plusieurs configurations qui seront proposées dans l'interface de l'outil au moment de la génération.

3.3. IHM de l'outil

La présentation de l'outil pour le développeur se découpe en 4 zones :

1. Configuration
   Choix de la configuration préalablement créée via la liste déroulante. Puis chargement de cette dernière via le bouton « Charger la configuration ».

2. Objets
   Affichage, sous forme d'une liste, de tous les objets de type table ou vue de la base de données. C'est également sur cette liste que les objets peuvent être sélectionnés ou désélectionnés pour la génération.

3. Options
   Choix des objets à générer.

4. Log
   Affichage du log de génération.

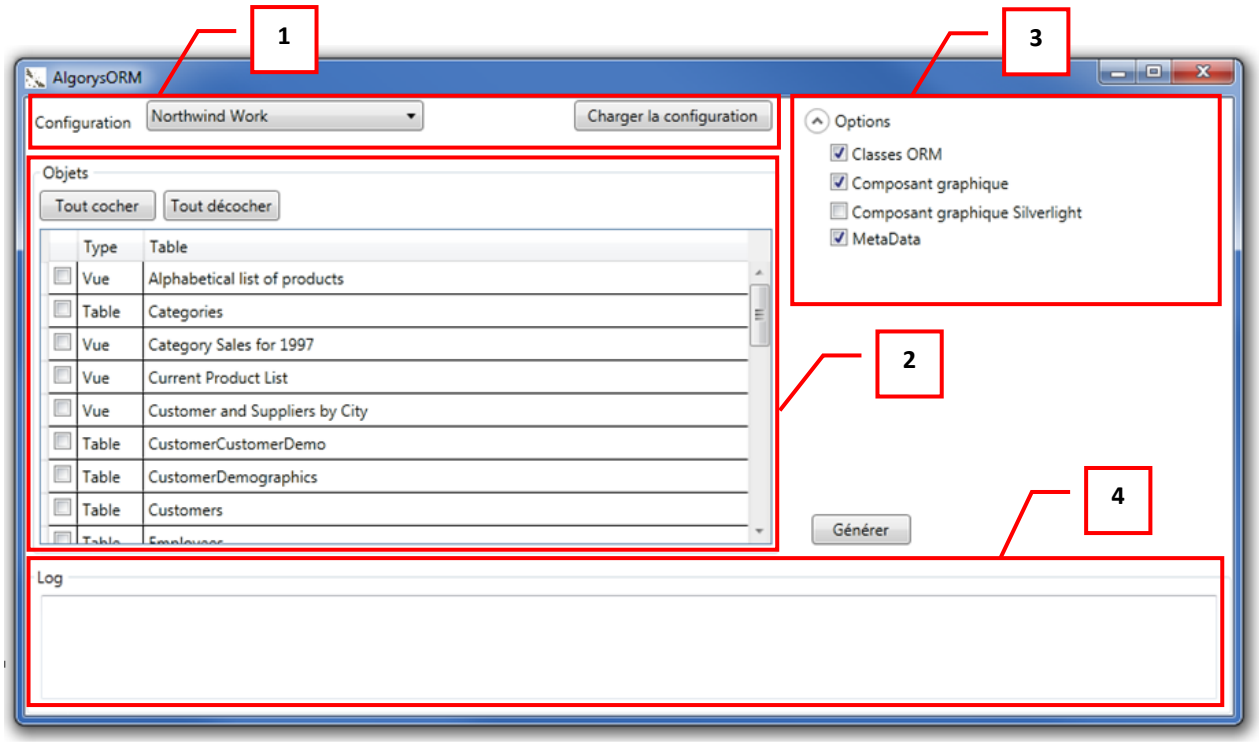

Figure 48 - Présentation de l'outil AlgorysORM

3.4. Configuration d'une application ASP.NET

Afin de pouvoir utiliser le code source généré dans une application, il faut paramétrer cette dernière. Après avoir inclus les nouvelles sources à la solution Microsoft Visual Studio, Il faut faire référence à la bibliothèque logicielle AlgorysORM.Reference.

Ajouter trois clés dans le nœud <appSettings> du fichier Web.config :

- SourcesNamespace : Espace de nom du code source ORM
- GUINamespace : Espace de nom du code source IHM
- AlgorysORMMetaDataPath : Répertoire contenant l'ensemble des métadonnées

Il suffit ensuite de créer une page contenant un des contrôles personnalisés généré et la référencer dans l'application.

3.5. Génération ORM

Pour la génération de la couche ORM, le principe est d'établir une correspondance entre le modèle relationnel de la base de données et le modèle objet utilisé dans l'application. Pour chaque table ou vue, deux classes sont générées : une contenant les attributs, une autre contenant les méthodes d'accès aux données. Les étapes de la génération sont les suivantes (voir Figure 49):

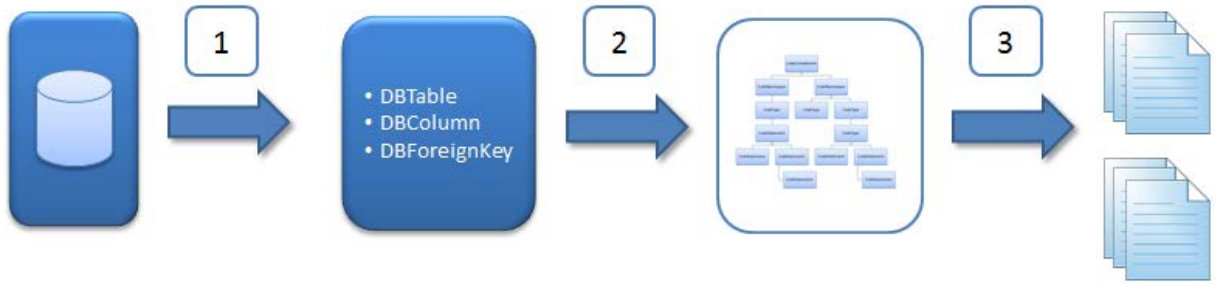

Figure 49 - Schéma de génération de la couche ORM

1. La première étape est la récupération des informations sur le schéma de la base de données. Pour cela le dictionnaire de la base de données est utilisé. Il contient toutes les informations sur la structure de la base. Les données sont ensuite stockées dans des objets représentant les métadonnées

2. Les objets créés précédemment sont convertis en arbre CodeDOM. Cet arbre est une représentation hiérarchique du code à générer

3. La dernière étape consiste à transformer l'arbre CodeDOM en classes sous forme de code source

Dans les paragraphes suivants, les étapes du processus de génération du code source de la couche ORM sont détaillées.

3.5.1. Interrogation du dictionnaire de données

La première étape est la récupération des informations sur le schéma de la base de données. Pour cela le dictionnaire de la base de données est utilisé. Il contient toutes les informations sur la structure de la base. Le dictionnaire de données est le référentiel d'organisation des données dans la base de données. Il contient les métadonnées qui décrivent l'organisation interne des bases de données au niveau structure et organisation.

Voici toutes les principales métadonnées qui sont mises à disposition par le dictionnaire de données :

- le nom des objets présents (tables, vues, procédures stockées …)
- pour chaque table et vue, toutes les informations concernant les colonnes (nom, type, position …)
- la liste des contraintes sur les objets (contraintes de clé primaire, de clé étrangère, …)

Dans le cahier des charges de l'outil, nous devions prendre en charge les bases de données Microsoft SQL SERVER à partir de la version 2000. Il a donc fallu s'assurer que toutes les méthodes et les objets utilisés pour récupérer les informations du dictionnaire de données de la base étaient compatibles avec toutes les versions.

Comme le préconise Microsoft, nous nous sommes appuyés sur le schéma INFORMATION_SCHEMA. Ce dictionnaire de données s'appuie sur la définition du catalogue de vues de la norme ISO. Il présente les informations dans un format indépendant de toute implémentation des tables du dictionnaire de données et des changements dans les tables sous-jacentes. Les applications qui utilisent ces vues sont portables entre systèmes de base de données compatibles ISO. Malgré cela, Microsoft met en garde sur l'utilisation de certains objets qui pourraient ne plus fonctionner selon la version. L'ensemble des objets utilisés a donc été testé sur toutes les versions supérieures de SQL SERVER (voir résultat Tableau 4).

Tableau 4 - Liste des vues du dictionnaire de données utilisées

Objet	Description	Version de SQL SERVER				
		2000	2005	2008	2008 R2	2012
TABLES	Liste des tables et des vues	✓	✓	✓	✓	✓
TABLE_CONSTRAINTS	Liste des contraintes des tables et des vues	✓	✓	✓	✓	✓
KEY_COLUMN_USAGE	Liste des colonnes qui sont contraintes par des clés.	✓	✓	✓	✓	✓
REFERENTIAL_CONSTRAINTS	Listes des clés étrangères	✓	✓	✓	✓	✓

La première étape de la génération ORM est donc l'interrogation du dictionnaire de données de la base de données. L'étape suivante consiste à manipuler ces données en vue de la génération. Pour cela il faut créer un modèle pour les données issues du dictionnaire.

3.5.2. Modélisation des métadonnées

Pour manipuler les données du dictionnaire de la base de données (appelé aussi métadonnées), 3 classes ont été créées (voir Figure 50):

DBTable
Représente une table ou une vue dans la base de données. Contient une liste de DBColumn.

DBColumn
Représente une colonne d'une table ou d'une vue avec toutes les informations relatives aux types de données (Type, valeur par défaut, numérotation automatique, valeurs nulles autorisées, et association à une contrainte de clé primaire/étrangère, etc.)

DbForeignKey
Représente une contrainte de clé étrangère entre deux tables.

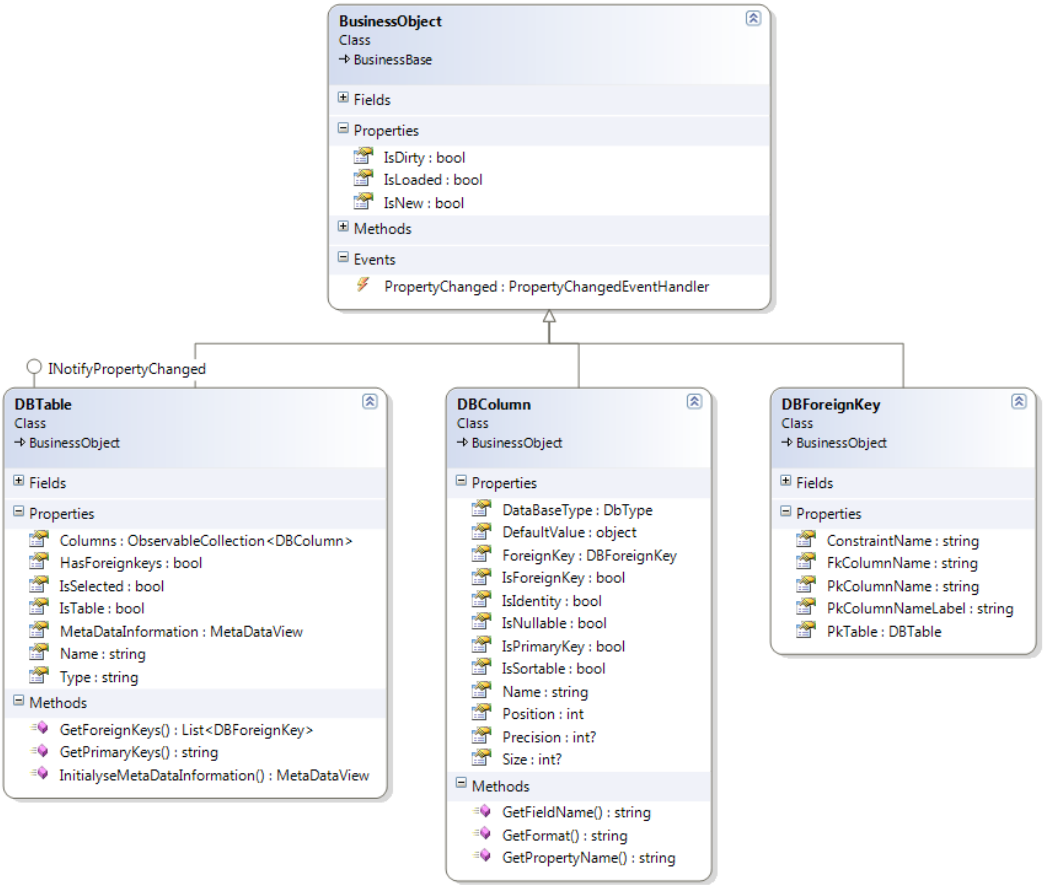

Figure 50 - Diagramme de classes représentant des métadonnées de la base de données

Le lien entre les colonnes et la table (ou la vue) s'effectue via une collection de type DBColumn incluse dans chaque objet de type DBTable. Les contraintes d'intégrités sont modélisées au niveau de la propriété représentant la colonne.

Dans la gestion des contraintes de clés primaires, la classe DBColumn est composée d'une propriété de type booléenne. Si une table contient une clé primaire sur plusieurs colonnes l'information est répartie sur la propriété IsPrimaryKey de chaque colonne concernée. Pour les clés étrangères, une propriété de type simple n'est pas suffisante. La classe DBForeigneKey qui est référencée par la classe DBColumn contient donc toutes les informations nécessaires à la représentation d'une clé étrangère, c'est-à-dire :

- le nom de la contrainte (ConstraintName)
- le rappel du nom de la colonne ayant la contrainte de clé étrangère (FkColumnName)
- Le nom de la table référencée par la contrainte (PkTable)
- Le nom de la colonne de la clé primaire de la table référencé par la contrainte (PkColumnName)
- Le nom de la colonne qui a le rôle de libellé pour la clé primaire de la table référencée par la contrainte. Cette information est utilisée pour la gestion des listes déroulantes (PkColumnNameLabel)

Pour la gestion des clés étrangères, si une colonne constitue une référence vers une autre table (par exemple la colonne de l'identifiant d'un code produit dans la table des commandes), il est d'usage d'afficher dans l'interface utilisateur une propriété autre qui a un sens pour l'utilisateur (par exemple le nom du produit plutôt que son identifiant). Mais lors de la génération, il est impossible de savoir quelle propriété doit remplacer la propriété représentant la colonne référence (par exemple le nom du produit ou son prix). La

règle appliquée a été de positionner par défaut la propriété à afficher, soit PkColumnNameLabel, avec le nom de la colonne de la position +1 par rapport à la colonne référence.

Toutes les classes héritent de la classe BusinessObject, classe de base pour toutes les classes métiers. Elle contient notamment, un mécanisme de génération d'événements lors de la modification d'une propriété pour que les composants visuels puissent se mettre à jour. Elle contient également trois propriétés qui indiquent si l'objet est nouveau, modifié ou a été chargé à partir d'une source externe. A ce stade, les données du dictionnaire de données ont été chargées dans des objets DBTable, DBColumn, DBForeignKey. La génération du code source peut donc démarrer.

3.5.3. Modélisation du code source (CodeDOM)

Toute la génération du code source s'appuie sur les classes de l'espace de nom System.CodeDOM dont le fonctionnement a été détaillé au paragraphe (2.2.4. CodeDOM).

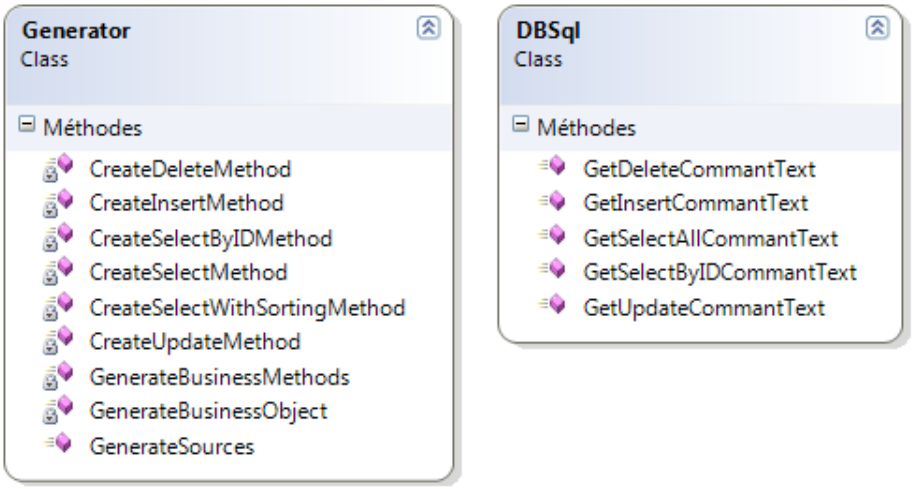

Figure 51 - Diagramme de classes - Génération du code source par la couche ORM

Pour la génération de code source, 2 classes sont utilisées : Generator et DBSql.

La classe **Generator** via la méthode GenerateSource génère, pour chaque objet de type DBTable, une classe « entité » et une classe « méthode » comme défini par le cahier des charges et préconisé par les bonnes pratiques de Microsoft. Les deux méthodes principales correspondantes sont GenerateBusinessObject pour la génération des classes « entité » et GenerateBusinessMethods pour des classes « Methods ».

La méthode **GenerateBusinessObject** prend en argument un objet de type DBTable. Elle génère une classe composée d'un attribut et d'une propriété par colonne. La correspondance passe par une énumération de type indépendante du fournisseur de données, cela en prévision de la gestion de bases de données autres que SQL SERVER (voir correspondance Tableau 5).

Tableau 5 - Correspondance des types Base de données / Fournisseur de données Microsoft / Objet

Type base de données SQL Serveur	Type fournisseur de données Microsoft	Type objet
tinyint	Byte	Byte
smallint	Int16	Int16
int	Int32	int
bigint	Int64	Int64
numeric	Decimal	decimal
decimal		
smallmoney		
money		
bit	Boolean	bool
float	Single	float
real		
datetime	DateTime	DateTime
date		
smalldatetime		
image	Binary	Byte[]
varbinary		
binary		
*	String	String

La méthode **GenerateBusinessMethod** prend également en paramètre un objet de type BDTable. Elle génère une classe sans attribut ni propriété mais avec les méthodes suivantes :

- SelectWithSorting (dédié pour le composant ObjectDataSource)
- Select (select générique)
- SelectByID (sélection d'une seule ligne via l'identifiant d'un objet)

Plus les méthodes suivantes si la propriété IsTable de l'objet DBTable vaut « vrai » :

- Update (mis à jour d'un enregistrement)
- Delete (suppression d'un enregistrement)
- Insert (insertion d'un nouvel enregistrement)

La génération de ces méthodes s'appuie sur la classe DBSql pour la génération des ordres SQL dynamiques.

3.5.4. Description des classes générées

Pour rappel, les classes « entités » générées héritent de la classe BusinessObject et implémentent également l'interface INotifyPropertyChanged pour permettre la notification, en cas de changement d'une des propriétés de l'objet. L'implémentation de cette interface est notamment obligatoire pour la liaison avec les composants graphiques Windows Presentation Foundation (application client lourd avec IHM à base de XAML) et les composants graphiques Silverlight (application client riche avec IHM à base de XAML) dont la prise en charge pourrait être une évolution de l'outil.

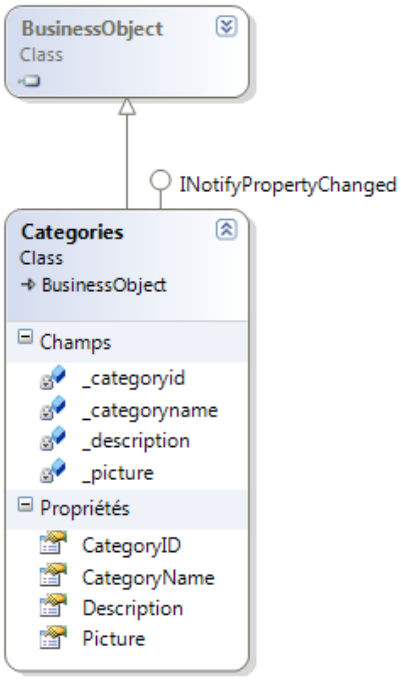

Figure 52 - Diagramme de classes - Exemple de classe "entité" générée

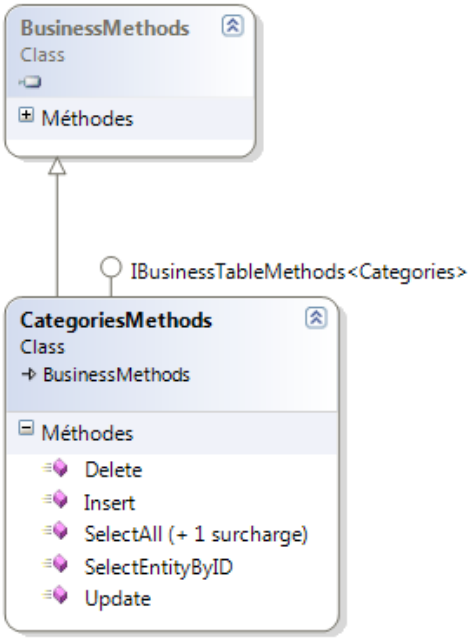

Figure 53 - Diagramme de classes - Exemple de classe "méthodes" générée

Les classes « méthodes » générées héritent toutes de la classe BusinessMethods qui contient un ensemble de méthodes pour la conversion des types issus de la base de données vers les types « objet » .Net (voir Figure 54).

Les classes générées implémentent l'interface IBusinessMethods dans le cas d'une classe correspondant à une vue et à l'interface IBusinessTableMethods.

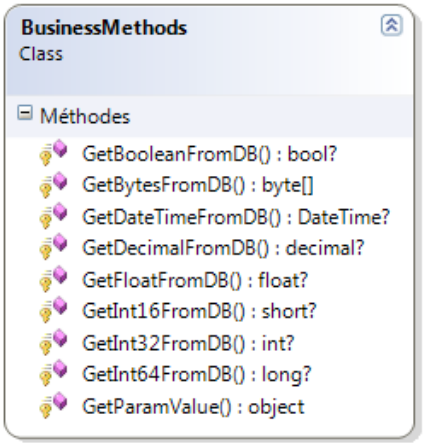

Figure 54 - Diagramme de classes BusinessMethods

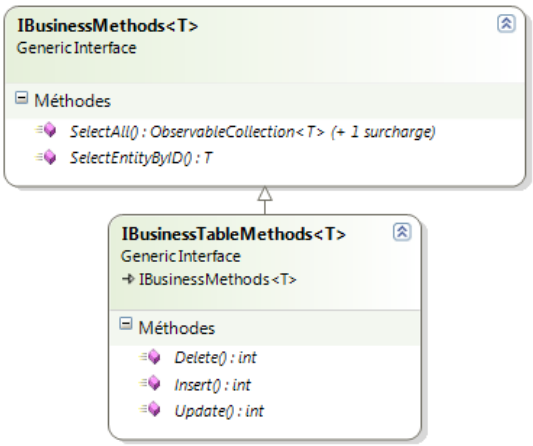

Figure 55 - Diagramme de classes - Interfaces pour les classes "méthodes"

Il est important de noter que les méthodes SelectAll retournent un type nommé « ObservableCollection<T> » qui est une préconisation d'usage pour les mêmes composants que ceux pour lesquels il faut implémenter l'interface INotifyPropertyChanged. Cet objet de type collection générique fournit des notifications lorsque des éléments sont ajoutés, supprimés ou lorsque la liste entière est actualisée.

3.6. Génération IHM

Pour rappel dans le paragraphe « Conception d'IHM », a été résolu le problème de la personnalisation des composants utilisés en ASP.Net via la génération uniquement en « Code-Behind » des composants personnalisés. Dans une approche plus complète de la génération de l'IHM et de son fonctionnement, ce paragraphe présente les composants mis en jeux, l'architecture, le principe de génération ainsi que la mise en œuvre.

3.6.1. Composants utilisés

Le but des composants à générer est de pouvoir fournir un formulaire ainsi que les fonctionnalités et services nécessaires aux opérations de base sur les données de référence. Ces opérations sont les suivantes :

- Affichage des données
- Création d'une nouvelle donnée
- Modification d'une donnée
- Suppression d'une donnée

La conception de composant s'appuie sur les patterns de Data grid et Form, détaillés dans le paragraphe 2.3.5. « Design Patterns : IHM ». De même que pour le paragraphe précédent, les explications portent sur un composant ayant pour fonction la mise à jour des données d'une table. Les composants correspondant aux vues ne sont que des versions ne présentant que leurs fonctionnalités d'affichage des données.

Au niveau présentation, le composant est constitué de plusieurs éléments visuels dont les principaux sont :

- Une grille de présentation des données
- Un formulaire de création de données

La **grille** fournit les fonctionnalités suivantes : affichage des données, modification d'une donnée et suppression d'une donnée, tandis que le **formulaire** constitue l'interface pour créer une nouvelle donnée (voir Figure 56 - Schéma de présentation d).

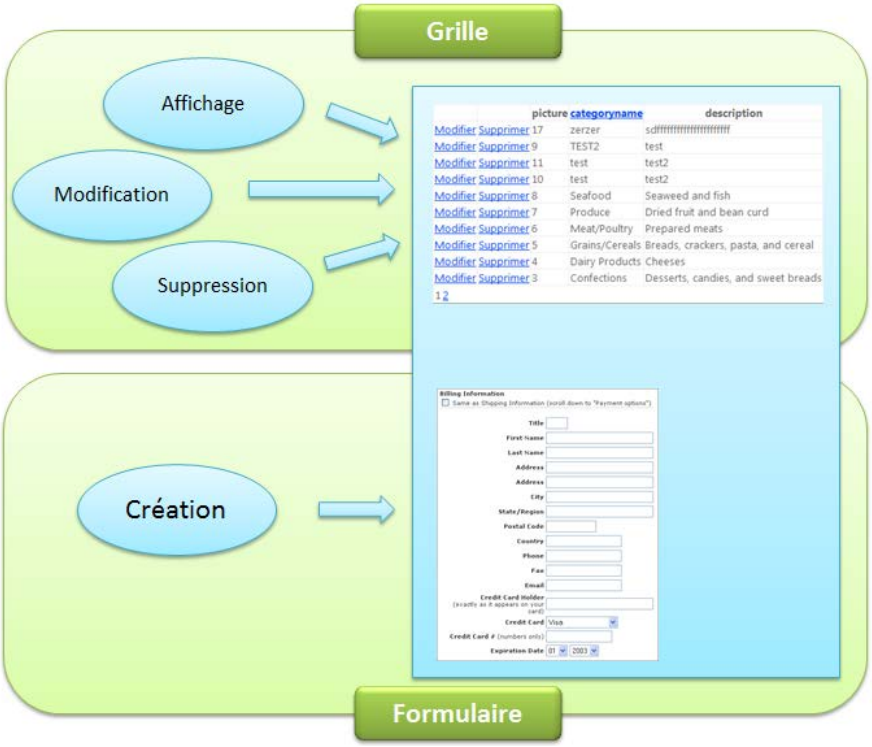

Figure 56 - Schéma de présentation de l'interface utilisateur du composant

3.6.2. Architecture

Du point de vue des données, l'architecture s'articule de la manière suivante. L'interface utilisateur du composant est, comme évoqué dans le paragraphe précédent, composée principalement d'une grille de données et d'un formulaire de création de données. Leurs sources de données sont contrôlées par des objets natifs du Framework appelé ObjectDataSource ayant le rôle de fournisseur de données. Ces derniers font le lien entre les données de la couche ORM et les contrôles visuels de la couche présentation. Les données de la couche ORM sont issues de la base de données. Les objets « méthodes » générés par l'outil interrogent la base de données afin de récupérer les informations qui sont converties en type de données compatibles avec le Framework .Net pour être stockées dans les objets « entités » également générés par l'outil (voir Figure 57).

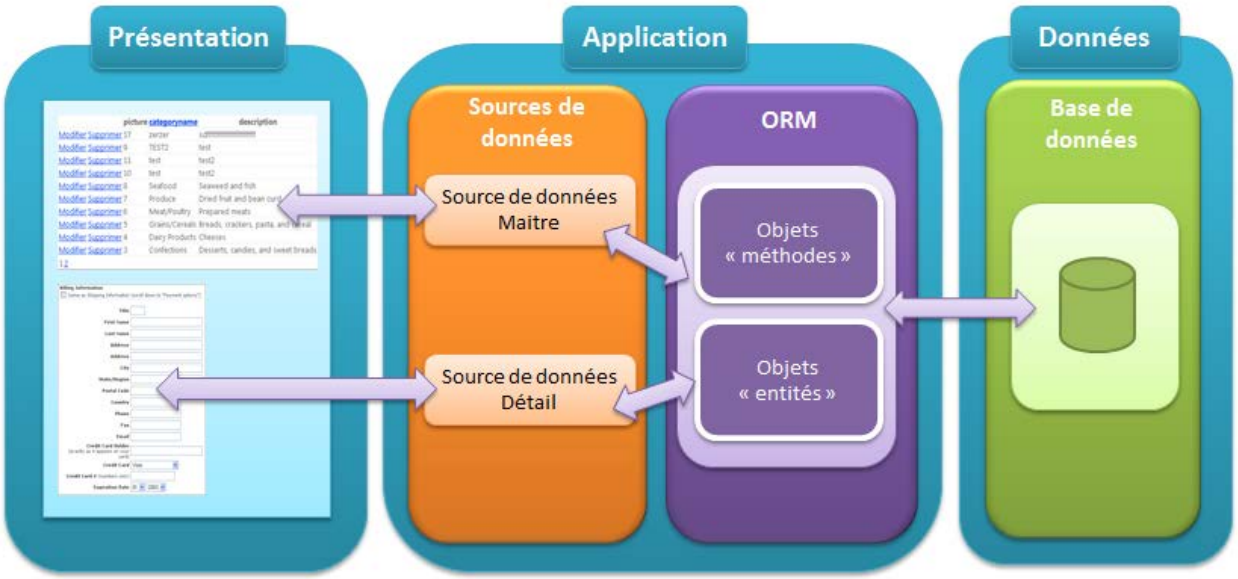

Figure 57 - Schéma de principe de l'architecture du composant

Au niveau du composant généré, la grille se compose en fait d'une collection de colonnes. De même pour le formulaire qui est constitué d'un ensemble de champs. Les colonnes et les champs sont, dans la plupart des cas, alimentés en données par le fournisseur de données de leurs éléments parents respectifs. Dans certains cas une colonne ou un champ a besoin de son propre fournisseur de données, par exemple dans le cas de l'affichage d'une liste déroulante ou d'un champ à choix multiples. Autant de sources de données sont alors nécessaires qu'il y a de champs ou de colonnes nécessitant une source de données spécifique.

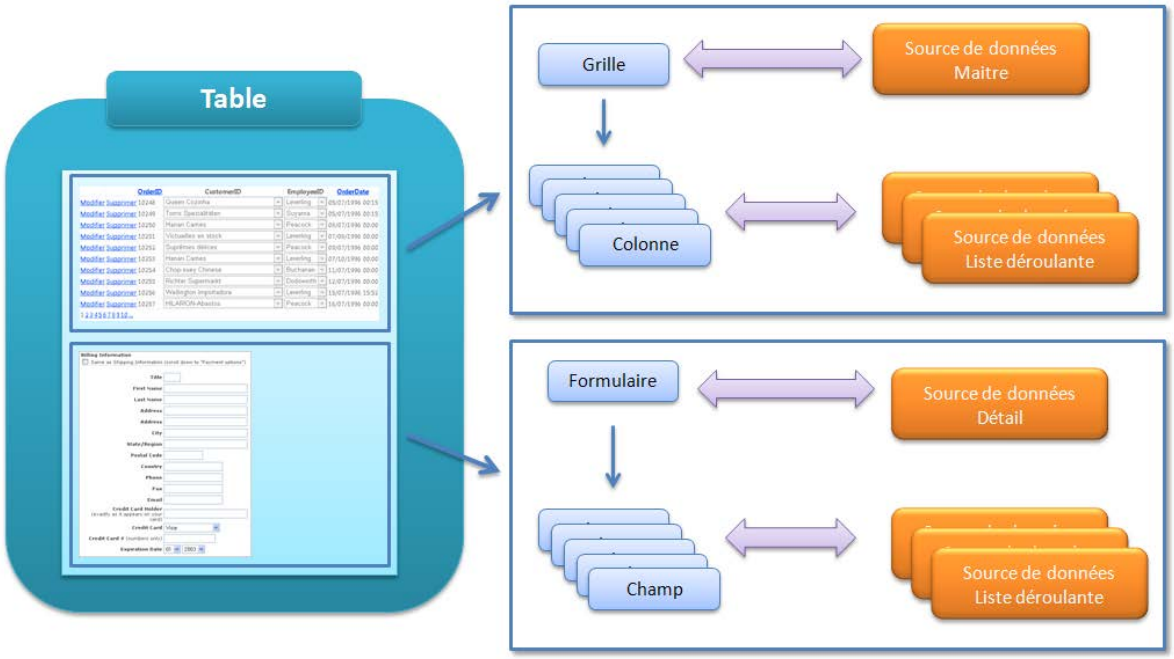

Figure 58 - Architecture du composant graphique

3.6.3. Principe de génération et de mise en œuvre

Le processus de génération des formulaires est plus complexe que celui de la génération de la couche l'ORM car les objets à générer sont plus complexes au niveau de leur arborescence. Il faut bien rappeler qu'il ne faut pas simplement les générer via l'espace de nom System.CodeDOM mais qu'il faut générer le code qui les crée à l'exécution des composants graphiques dans la page web. En effet, ils ne peuvent pas être déclarés dans le fichier de déclaration du composant en raison du dysfonctionnement de l'héritage si le composant n'est pas entièrement décrit dans le fichier « Code-Behind ». Ce dysfonctionnement est décrit dans le paragraphe 2.3.6. « Solution retenue ».

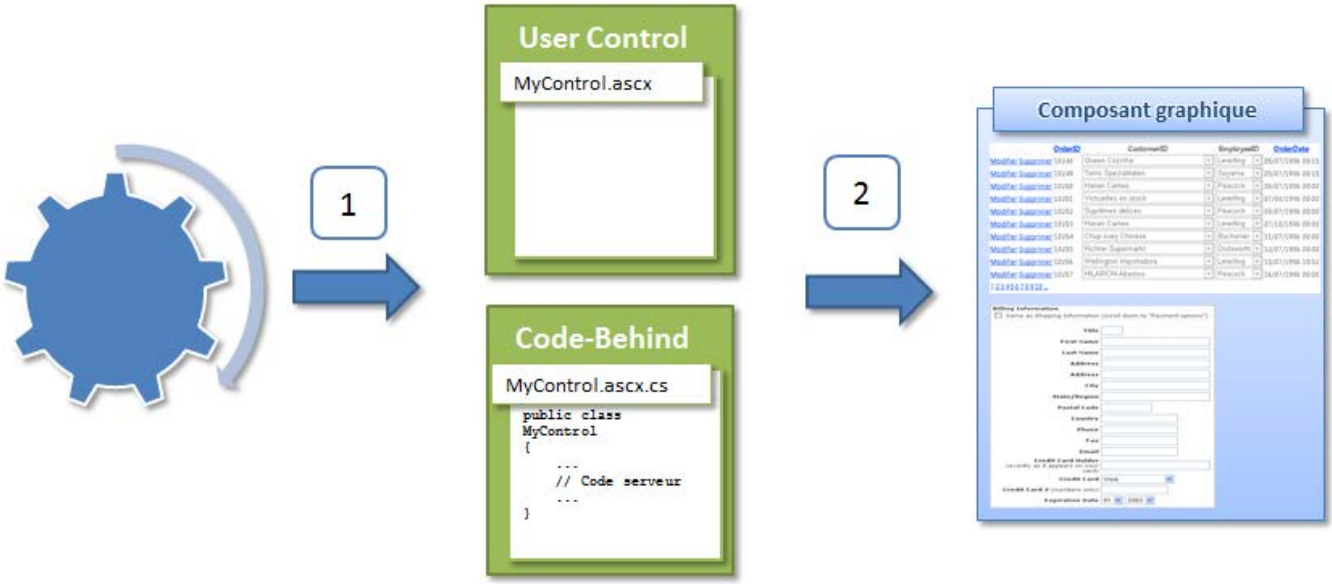

Figure 59 - Processus de génération et d'exécution du formulaire

Le schéma ci-dessus (voir Figure 59) illustre le principe de génération en étapes du composant personnalisé :

La première étape est celle de la génération. La classe UIGenerator construit en CodeDOM le code de création des composants dynamiques puis génère le Code-Behind. Le contrôle personnalisé (« user control ») étant un simple fichier texte, il est créé de façon classique sans passer par le système CodeDOM. Le résultat de cette opération est composé de deux entités et trois fichiers :

- Contrôle personnalisé
    - Un fichier de type déclaration de contrôle personnalisé visuel (UserControl) avec l'extension « ascx ». La lettre « c » signifiant « Control » à la différence des pages web à l'extension « aspx » dont le « p » signifie « Page ». Pour ce fichier, la déclaration du composant est, comme expliqué précédemment, vide dans ce cas.
    - Un fichier de type « Code-Behind » avec l'extension ascx.cs. Le « cs » de fin de l'extension signifie que le langage de programmation du fichier est le C# (CSharp). Il contient la déclaration des composants graphiques, leurs instanciations et leurs initialisations dans le composant.

- Métadonnées du contrôle
    - Ce fichier contient l'ensemble des informations concernant la personnalisation des contrôles et notamment les attributs de chaque propriétés (le nom à afficher dans le formulaire, la position du champ, etc.). Les valeurs de ses métadonnées sont initialisées lors de leurs générations. Elles sont destinées à être modifiées par la suite pour personnaliser l'affichage du composant sans utiliser la programmation.

L'étape finale est la création du composant au sens propre du terme dans le serveur web où s'exécute les instructions du « Code-Behind » afin de générer le composant graphique lors de l'exécution de la page sur laquelle ce dernier a été utilisé.

Ces deux étapes vont maintenant être détaillées dans les paragraphes suivants.

3.6.4. Fonctionnement du contrôle personnalisé

La première étape a donc la tâche complexe de générer le Code-Behind qui devra, une fois le composant exécuté, générer lui-même le formulaire. Pour comprendre le processus de génération, il faut déjà comprendre quelle est la logique d'exécution du Code-Behind pour la génération de la page.

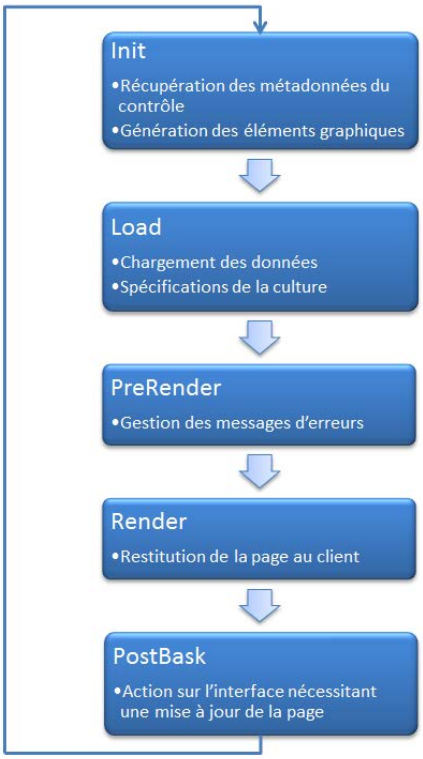

Figure 60 - Processus de vie d'un contrôle personnalisé

Le processus de vie du composant se déroule en 5 étapes majeures :

Init : Initialisation du composant. Dans cette étape, le contrôle récupère les métadonnées correspondant à la clé définie dans son attribut « MetaDataId ». Cela autorise l'exécution du même composant avec plusieurs métadonnées. Une fois les métadonnées récupérées, une série de méthodes est appelée pour instancier et paramétrer les différents éléments qui composent le contrôle.

- GenerateGridView : pour la grille
- GenerateDetailView : pour le formulaire
- GenerateObjectDataSource : pour la source de données de la grille
- GenerateObjectDataSourceDetail : pour la source de données du formulaire
- GenerateObjectDataSourceForeignKeys : pour les sources de données spécifiques.
- GenereateLiterralControlError : pour l'affichage des messages d'erreurs.

Toutes ces méthodes font appel aux méthodes de la classe (AlgorysORM.Reference.Web.ItemGenerator). Cette classe appartient à une bibliothèque logicielle à laquelle les solutions ASP.Net utilisant les mécanismes de génération ORM et IHM de l'outil doivent faire référence (voir Figure 47). Chaque méthode prend, au moins en paramètre, une référence sur l'objet représentant l'élément visuel associé et l'objet contenant les métadonnées. Parmi les paramétrages effectués par ces méthodes, il y a notamment la spécification des liaisons entre les sources de données et les éléments conteneurs de données (grille, formulaire, liste déroulante, etc.)

Load : Chargement des données. A cette étape, les contrôles ont été instanciés et configurés. Les sources de données chargent donc les données via les méthodes qui leurs sont définies. La culture qui représente les informations telles que le système d'écriture, le calendrier utilisé, la mise en forme des dates et le tri des chaînes, est placée à une valeur fixe de manière à ce que l'application ait un fonctionnement standard quel que soit la langue de l'installation du système d'exploitation.

PreRender : Pré-rendu. Gérés de manière native par le Framework, les éléments visuels qui disposent de leurs sources de données vont maintenant se générer. S'il y a eu des erreurs de chargement de données à l'étape précédente, elles sont mises à disposition dans une propriété de type liste de messages. Si la liste n'est pas vide il faut afficher les messages d'erreurs.

Render : Rendu. Toute la page est maintenant rendue. Elle est alors envoyée au client qui l'affiche sur le navigateur et peut la manipuler de son côté.

PostBack : Renvoi au serveur. Certaines fonctionnalités nécessitent un renvoi de la page au serveur comme la création, la modification ou la suppression d'une donnée. La page est alors renvoyée sur le serveur qui traite les informations dans la base de données et le cycle recommence jusqu'à la fermeture de la page.

3.6.5. Génération des fichiers

Lors de la génération des fichiers du contrôle personnalisé et du fichier de métadonnées du contrôle, deux classes interviennent :

- La classe UIGenerator qui génère à la fois la déclaration du contrôle personnalisé (UserControl) et la logique de programmation (« Code-Behind »).
- La classe MetaDataGenerator qui a pour but de générer le fichier XML de métadonnées.

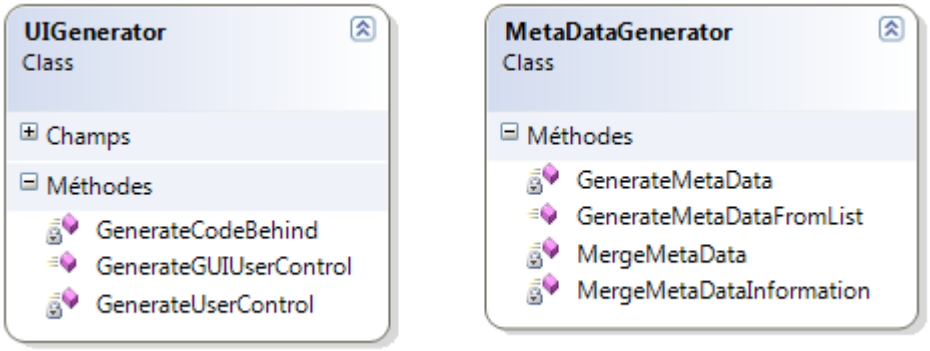

Figure 61 - Diagramme de classes – Génération du code source pour le formulaire

3.6.6. UIGenerator

La classe **UIGenerator** a donc pour fonction de générer le composant personnalisé. La logique de la classe est étudiée dans le cas d'une génération pour une correspondance avec une table. La correspondance avec une vue est similaire mais avec les fonctions de modification en moins. Comme pour la classe « Generator », la classe « UIGenerator » possède une méthode « GenerateUserControl » qui prend en paramètre un objet de type DBTable dont le but est de générer un fichier .ascx « déclaration » et un fichier .ascx.cs « Code behind » (voir Figure 44). Cette méthode fait appel aux deux autres méthodes de la classe : « GenerateGUIUserControl » et « GenerateCodeBehind ».

La méthode **GenerateGUIUserControl** crée donc le fichier de déclarations du contrôle personnalisé qui n'est qu'un conteneur vide afin de préserver le mécanisme d'héritage.

```
<%@ Control
    Language="C#"
    AutoEventWireup="true"
    CodeBehind="UCCategories.ascx.cs"
    Inherits="AlgorysORMWebTesting.UCCategories" %>
```

Figure 62 - Exemple de code de déclaration d'un contrôle personnalisé

Les informations définies sont (exemple voir Figure 62) :

- Language : Spécifie le langage utilisé dans des balises d'exécution en ligne (<% %> et <%= %>).

- AutoEventWireup : Définit un certain nombre de liaisons entre des méthodes et des évènements de la page, comme, par exemple, le fait qu'au chargement de la page, la méthode Page_Load soit automatiquement appelée sans qu'une surcharge soit nécessaire. C'est un mécanisme de confort coûteux en termes de performances mais sa désactivation peut être perturbante. Le choix de laisser l'option a donc été fait.

- CodeBehind : Spécifie le nom du fichier qui contient la classe associée au contrôle.

- Inherits : Définit la classe Code-Behind pour l'héritage du contrôle. C'est ce mécanisme qui permet de générer le composant en dynamique, entièrement depuis le Code-Behind.

La méthode **GenerateCodeBehind** génère toute la structure CodeDOM des éléments mise en œuvre lors de l'exécution du « Code-Behind » (voir Figure 60). Les étapes les plus importantes de la construction du CodeDOM sont les suivantes :

- Déclaration de la classe héritant de System.Web.UI.UserControl (Classe de base pour les contrôles personnalisés)
- Création des attributs
    - _metadataId : clé correspondant aux métadonnées du contrôle
    - _classeName : Nom de la classe (pour éviter de l'introspection)
    - _nsName : Nom de l'espace de nom (pour éviter de l'introspection)
    - _view : Référence les métadonnées (type : MetaDataView)
    - _listErrorMessage : Liste des messages d'erreurs
    - ods{nom_correspondance} : Source de données principale (type : ObjectDataSource)
    - dv{nom_correspondance} : Formulaire (type : DetailView)
    - gv{nom_correspondance} : Grille de données (type : GridView)
    - LitError : Zone de texte pour l'affichage des erreurs (LiteralControl)
- Création des méthodes
    - OnInit : Instanciation, configuration et personnalisation via les métadonnées des éléments graphiques. Liaison entre les éléments graphiques et les sources de données. Gestion des liaisons entre les évènements des contrôles et leurs délégués
    - OnLoad : Spécification de la culture
    - Page_PreRenderComplete : Gestion de l'affichage des messages d'erreurs (fait référence à la méthode GetMessageFromList de la classe (AlgorysORM.Reference.Tools)
    - GenerateGridView/ DetailView / DataSource / DataSourceDetail / DataSourceForeygnsKeys / LiterralControlError : Méthodes appelées dans la fonction OnInit pour la création et la configuration des éléments correspondants.
- Création des délégués
    - ods{nom_correspondance}_Deleted / _Updated / _Inserted / _Selected : Récupération du message d'erreur si besoin.
    - gv{nom_correspondance}_RowUpdating : Gestion du formatage des dates lors de la mise à jour d'une donnée
    - gv{nom_correspondance}_Sorting : Gestion des tris lors d'un clic sur une colonne

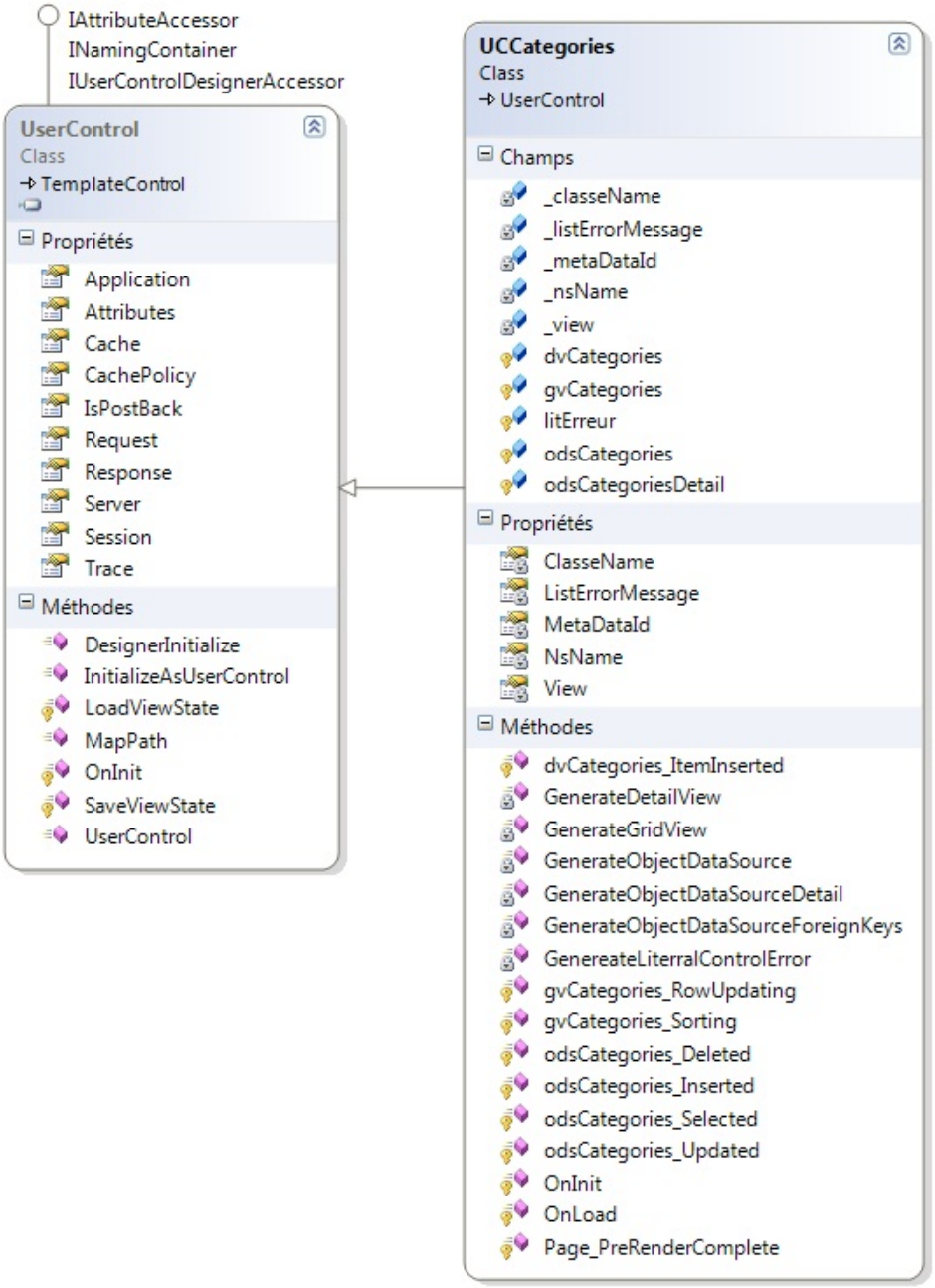

Figure 63 - Diagramme de classes - Exemple d'une classe correspondant à une table Categories

La Figure 63 présente le diagramme de classes correspondant au composant personnalisé correspondant à la table Categories.

3.6.7. MetaDataGenerator

La méthode principale de la classe est GenerateMetaData qui génère un fichier XML basé sur la sérialisation de classe MetaDataView (voir Annexe C). L'objet MetaDataView est constitué notamment d'une collection de MetaDataProperty dont chaque propriété est équivalente à une possibilité de personnalisation du rendu lors de l'affichage.

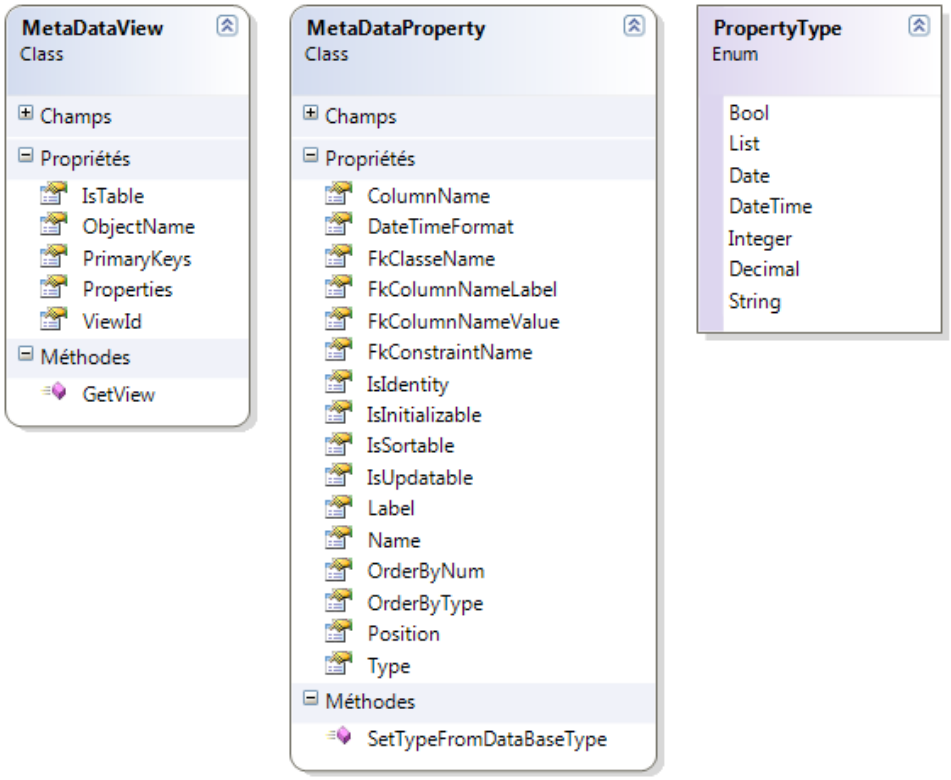

Figure 64 - Diagramme de classes - Représentation des métadonnées d'un contrôle personnalisé

Une des fonctionnalités implémentée est la fusion des métadonnées lorsque celles-ci sont déjà existantes à l'endroit spécifié pour la génération du fichier. Dans ce cas le fichier existant est désérialisé dans un objet de type MetaDataView puis une fusion est effectuée dans les deux listes des MetaDataProperty pour y ajouter, si besoin, une nouvelle propriété ou en supprimer une qui n'est plus active.

3.7. Tests

Il existe plusieurs approches en termes de test logiciel, notamment l'approche « Boîte blanche » et l'approche « Boîte noire ». Dans la première, une vue du système est créée pour déterminer les jeux de tests (données d'entrée pour les tests). Dans la seconde le logiciel à tester est vu comme une « boîte » hermétique dont on ne connaît pas l'intérieur. Il existe également une troisième catégorie de tests appelé « Boîte grise » qui est une combinaison des deux approches précédentes.

Le centre de recherche de Microsoft travaille, depuis 2008, sur un outil de génération de tests et de données de test, appelé Pex (pour « **P**rocess **Ex**ploration »). Son but est de fournir une suite de tests unitaires qui offrent le maximum de couverture de chemin en proposant le minimum de valeurs d'entrée. Pex utilise l'exécution « concolique » (combinaison d'exécutions symboliques et concrètes) et le solveur de contrainte Z3 de Microsoft pour générer des jeux de tests.

Le fonctionnement de Pex appartient à la famille des tests structurels car il se base principalement sur la structure du programme. Une représentation du code source appelée **Graphe de Flux de Contrôle** (GFC) est utilisée. Ce dernier permet de visualiser les différents chemins que pourrait parcourir le pointeur d'exécution du programme. Le programme, représenté dans la Figure 65, a pour but de d'effectuer le calcul « x à la puissance y ». Dans la partie gauche figure le code du programme et à droite son GFC.

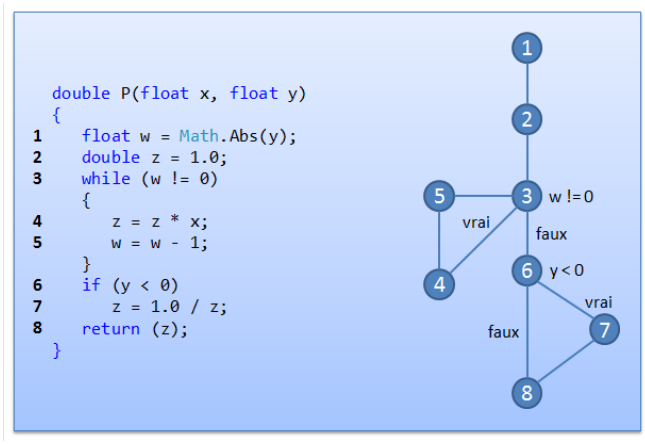

Figure 65 - Diagramme de flux de contrôle

En ce qui concerne l'exécution concolique, Pex utilise la méthode dite d'exécution symbolique dynamique. Il commence l'exécution de la méthode avec des valeurs simples tout en démarrant une exécution symbolique en parallèle permettant de récupérer les contraintes symboliques sur les chemins. Puis à l'aide de solveurs de contrainte Z3, il déduit des nouvelles valeurs de variables qui permettent d'explorer d'autres chemins. Dans le but d'offrir le meilleur niveau de couverture, Pex va privilégier une exécution rapide et s'arrêter au bout d'un temps défini par l'utilisateur plutôt que de privilégier une couverture de 100% du code sur un critère. Le temps défini par l'utilisateur ne constitue pas le seul critère d'arrêt. D'autres sont disponibles tels que le nombre d'exécution maximum, la profondeur maximum du graphe de flux de contrôle.

Dans certains cas Pex ne peut pas analyser correctement le code. Il part du principe que le système est déterministe. S'il détecte qu'il ne l'est pas, il propose à l'utilisateur de modifier le code pour le rendre déterministe. Le code natif appelé par le code .Net est considéré comme non déterministe. Les programmes multithreads ne sont pas gérés. Le solveur Z3 interdit également la résolution de contraintes comprenant des nombres en virgule flottante Pex pourrait tester tous les codes compilés avec la plateforme .Net mais à l'heure actuelle le complément de Visual Studio ne gère que le code C#.

L'utilisation de Pex se fait directement dans Visual Studio. En se plaçant dans la méthode à tester, l'exploration du code se lance directement via une option. Cela va générer automatiquement plusieurs entrées qui seront évaluées une à une. Les résultats des évaluations seront affichés dans une fenêtre spécifique. Pour chaque entrée ou pour l'ensemble il est possible de sauvegarder le script de test afin de mettre en place soit une politique de test unitaire, de test d'intégration ou de test de non régression.

La Figure 66 montre un exemple d'une méthode testée par Pex. L'objet de cette méthode est de fournir la requête SQL pour supprimer une ligne dans la base de données. Elle prend comme paramètre d'entrée un objet de type DBTable et retourne la requête sous forme d'une chaîne de caractères.

```
/// <summary>
/// Get the "Delete" commandtext to SQL
/// </summary>
/// <param name="table"></param>
/// <returns></returns>
public string GetDeleteCommantText(DBTable table)
{
    StringBuilder query = new StringBuilder();

    if (table != null && table.Name != null && table.Columns != null)
    {
        query.Append("DELETE ");
        query.Append(" FROM [" + table.Name + "]");
        query.Append(" WHERE 1=1 ");
        foreach (DBColumn col in table.Columns)
        {
            if (col!= null && col.IsPrimaryKey)
            {
                query.Append(" AND " + col.Name + " = @" + col.Name);
            }
        }
    }

    return query.ToString();
}
```

Figure 66 - Exemple de méthode testé par Pex

Lorsque Pex est exécuté sur la méthode, il va proposer un ensemble de jeux de tests et les exécuter afin de s'assurer qu'aucun d'entre eux ne génère une exception :

Pex Exploration Results - stopped

DBSql.GetDeleteCommantText(DBTable) | Run | Views | Follow Pe

7 0 33/33 Block, 0/0 asserts, 171 runs

	table	result	Summary/Exception
1	null	""	
2	new DBTable{Name=null,IsTable...	""	
3	new DBTable{Name=null,IsTable...	""	
4	new DBTable{Name="",IsTable=f...	""	
5	new DBTable{Name="",IsTable=f...	"DELETE FROM [] WHERE 1=1 "	
6	new DBTable{Name="",IsTable=f...	"DELETE FROM [] WHERE 1=1 "	
7	new DBTable{Name="",IsTable=f...	"DELETE FROM [] WHERE 1=1 AND = @"	

Figure 67 - Résultat de l'exécution de Pex

Le résultat de l'analyse par Pex a permis de montrer qu'il existe 33 chemins différents à notre méthode. Pex a effectué 171 exécutions de la méthode, ce qui lui a permis de déterminer qu'avec 7 jeux de tests, il était possible de couvrir l'ensemble des 33 chemins. Pour chaque jeu de tests, la valeur de la variable d'entrée est affichée ainsi que la valeur de retour de l'exécution de la méthode.

Il est alors possible de générer une bibliothèque de tests unitaires en fonction des tests générés par Pex. Cette procédure a été effectuée sur l'ensemble du projet afin de maintenir un haut niveau de couverture du code et de permettre la création d'un ensemble de test unitaire pour chaque module (ORM et IHM) pour faciliter leur intégration.

Conclusion

Le projet s'est terminé pour sa première phase. Les fonctionnalités de génération d'une couche ORM ainsi que d'une IHM destinée à la modification des données de référence pour les applications Web ASP.Net sont réalisées, testées et validées. La prochaine évolution du projet AlgorysORM est la génération de composants visuels destinés aux applications Silverlight. Pour cela il sera nécessaire de créer une nouvelle bibliothèque logicielle dans la solution qui sera spécifique pour cette plateforme client riche de Microsoft, car les bibliothèques standards ne sont pas compatibles avec cette dernière. Il sera également nécessaire de modifier la structure de la bibliothèque AlgorysORM.Reference de manière à éviter toute duplication du code.

Ce projet reflète une tendance actuelle qui est l'augmentation des solutions permettant de faire abstraction de la récupération des données et de la gestion de la persistance de celles-ci. Les outils d'ORM disponibles de nos jours ont atteint une certaine maturité et proposent des solutions fiables et stables pour les grands projets.

La réalisation de ce projet m'a permis d'explorer la possibilité de génération dynamique de codes via la bibliothèque logicielle System.CodeDOM fourni par Microsoft, expérience techniquement intéressante et enrichissante. Par la gestion du projet, j'ai également pu mettre en valeur mes compétences issues de mes expériences ainsi que de la formation dispensée par le Conservatoire National des Arts et Métier au niveau de la conception, du pilotage et de la réalisation du projet en tant qu'ingénieur informatique.

Bibliographies

Ouvrages

[CAR03] Carrol, John M. HCI models, theories, and frameworks. Morgan Kaufmann. 2003

[DAV10] Bertrand DAVID. IHM. CNAM, 2010

[FOW02] Fowler Martin. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002.

[GOF94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1994

[JUL10] Lerman, Julia. Programming Entity Framework: Building Data Centric Apps with the ADO.NET Entity Framework. O'REILLY, 2010.

[ROE11] Roebuck, Kevin. Object-relational mapping (ORM). Tebbo, 2011

[ROW02] Rowland, Roger. Generating Design Patterns Using CodeDOM. ASP TODAY, 2002

[ZEH90] Zehnder, Carl-August. Développement de projet en informatique. PPUR presses polytechniques, 1990, p. 174

Sites internet

http://en.wikipedia.org/wiki/Object-relational_mapping : Article Web, WikiMedia. Object-relational mapping. 2011. Consulté le 21/02/2011.

http://en.wikipedia.org/wiki/List_of_object-relational_mapping_software : Article Web, WikiMedia. List of object-relational mapping software. 2011. Consulté le 21/02/2011.

http://en.wikipedia.org/wiki/Object-relational_impedance_mismatch : Article Web, WikiMedia. Object-relational impedance mismatch. 2011. Consulté le 21/02/2011.

http://www.dotnetguru.org/articles/Persistance/livreblanc/ormapping.htm : Article Web, DotNetGuru, Sébastien Ros, Directeur technique de la société Evaluant. Mapping objet-relationnel, Couches d'accès aux données et Frameworks de persistance. 2011. Consulté le 21/02/2011.

http://www.agiledata.org/essays/mappingObjects.html : Article Web, AgileData, Scott W. Ambler, IBM. Mapping Objects to Relational Databases: O/R Mapping In Detail. 2010 (Consulté le 21/02/2011)

http://www.objectarchitects.de/ObjectArchitects/orpatterns/ : Article Web, ObjectArchitects, Wolfgang Keller, ObjectArchitects. Patterns for Object / Relational Mapping and Access Layers. 2004. Consulté le 21/02/2011.

http://www.objectarchitects.de/ObjectArchitects/papers/Published/ZippedPapers/mappings04.pdf : Article Web, ObjectArchitects , Wolfgang Keller, Mapping Objects to Tables : A Pattern Language, 1997, Consulté le 21/02/2011.

http://www.dotnetguru.org/articles/articlets/choixmapping/mapping.htm : Article Web, DotNetGuru, Fabrice Marguerie, Architecte .Net chez Alti/Masterline. Mapping Objects to Relational Databases: O/R Mapping In Detail. 2011. Consulté le 21/02/2011.

http://www.alachisoft.com/resources/articles/orm.html : Article Web, Alachisoft, Iqbal M. Khan, Alachisoft. 2011. Consulté le 21/03/2011.

http://www.artima.com/intv/abstract3.html : Interview, Anders Hejlsberg, Bill Venners et Bruce Eckel Inappropriate Abstractions : Object-Relational Mappings. 2003. Consulté le 22/03/2011.

http://morpheus.ftp-developpez.com/architecture/architecture.pdf : PDF, LEBRUN Thomas, Microsoft Most Valuable Professional. Introduction au développement en couches. Consulté le 18/04/2011.

http://dotnetslackers.com/articles/ado_net/A-Feature-driven-Comparison-of-Entity-Framework-and-NHibernate-2nd-Level-Caching.aspx : Article Web, DotNetSlackers, Dino ESPOSITO, trainer and software consultant. A Feature-driven Comparison of Entity Framework and NHibernate-2nd Level Caching. 2010. Consulté le 18/04/2011.

http://www.service-architecture.com/object-relational-mapping/articles/index.html : Article Web, Service Architecture, Douglas BARRY, Cutter Consortium Consultant. Object-relational mapping articles. 1998. Consulté le 18/04/2011.

http://community.jboss.org/wiki/NHibernateforNET : Article Web, NHibernate. Nhibernate. Consulté le 19/04/2011.

http://wiki.fluentnhibernate.org : Article Web, Fluent Hibernate. Wiki Article Web. 2010. Consulté le 19/04/2011.

http://subsonicproject.com : Article Web, SubSonic. Subsonic site web. 2009. Consulté le 19/04/2011.

http://www.infoq.com/presentations/Data-Grid-Design-Patterns-Brian-Oliver : Conférence, Brian Oliver, Global Solutions Architect at Oracle. Data grid pattern. 2009. Consulté le 20/04/2011.

http://deptinfo.unice.fr/~grin/mescours/minfo/modpersobj/supports/sgbdoo6.pdf : SGBDOO, Richard Grin, Université de Nice Sophia-Antipolis. 2011. Consulté le 24/05/2011.

http://www1.acm.org/sigs/sigchi/chi97/proceedings/paper/jcc.htm : Article Web, Gaëlle Calvary, Joëlle Coutaz, Laurence Nigay, From Single-User Architectural Design to PAC. Consulté le 04/06/2011.

http://www.irsn.fr/FR/base_de_connaissances/Installations_nucleaires/La_surete_Nucleaire/Les-accidents-nucleaires/three-mile-island-1979/ Article Web, G. Cenerino, F. Pichereau, E. Raimond, M. Dubreuil, L. Esteller, C. Pignolet, F. Bigot, P. Quentin IRSN/DSR, R. Gonzalez, B. Clement IRSN/DPAM, K. Herviou IRSN/DEI, IRSN. Consulté le 04/06/2011.

http://people.rennes.inria.fr/Arnaud.Gotlieb/enseignement/MASTER_COT_05.pdf : Arnaud Gotlieb, IRISA / LANDE. Test structurel de programmes impératifs. Consulté le 17/04/2011.

Table des annexes

Annexe 1 Modèle physique de la base de données de tests 76
Annexe 2 Schémas des principaux objets de la bibliothèque logicielle CodeDOM 77
Annexe 3 Modèle physique de données du schéma INFORMATION_SCHEMA 82

Annexe 1
Modèle physique de la base de données de tests

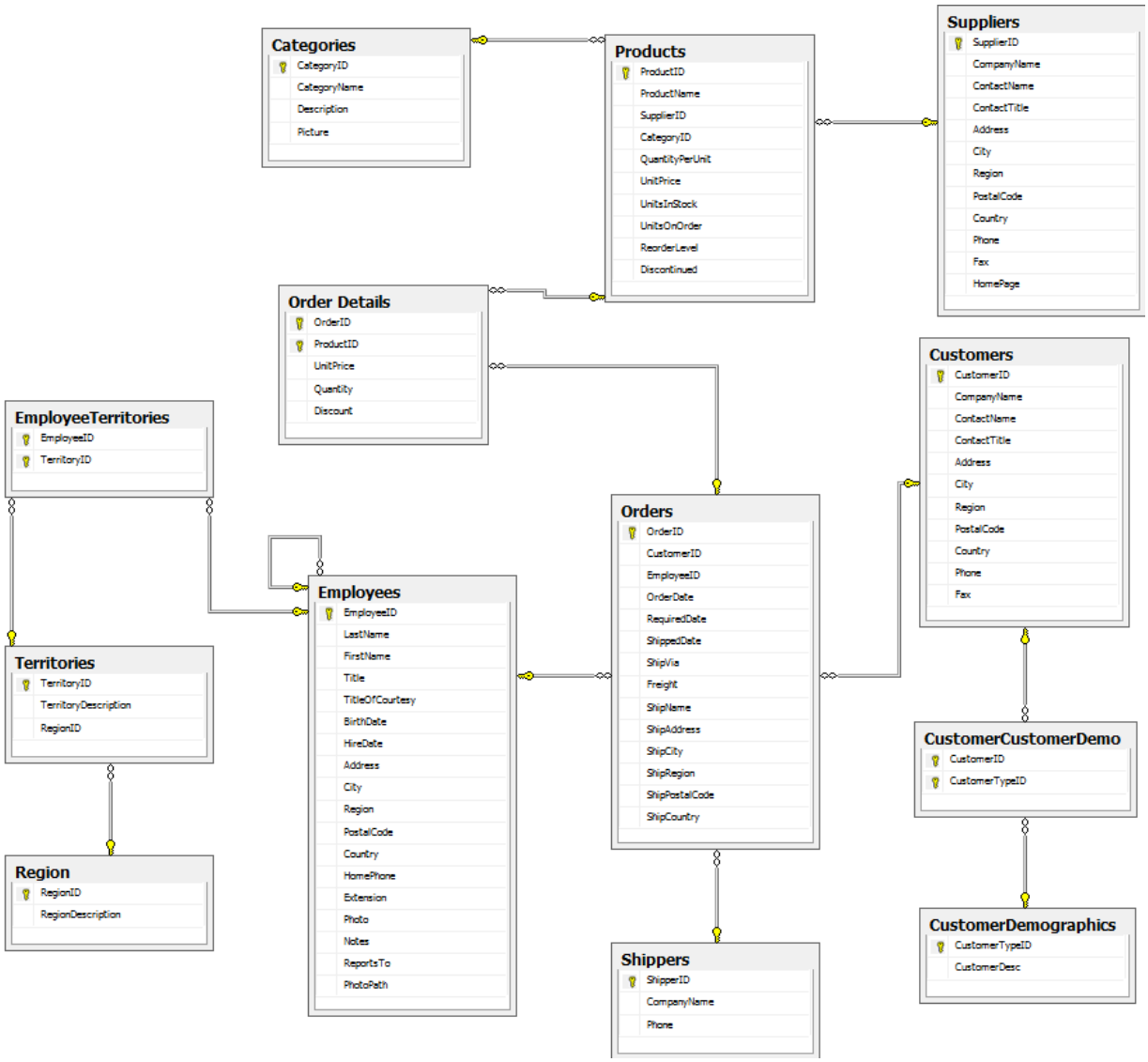

Figure 68 - Base de données de test : Northwind

Annexe 2
Schémas des principaux objets de la bibliothèque logicielle CodeDOM

Figure 69 - Classes principales de la bibliothèque logicielle CodeDOM

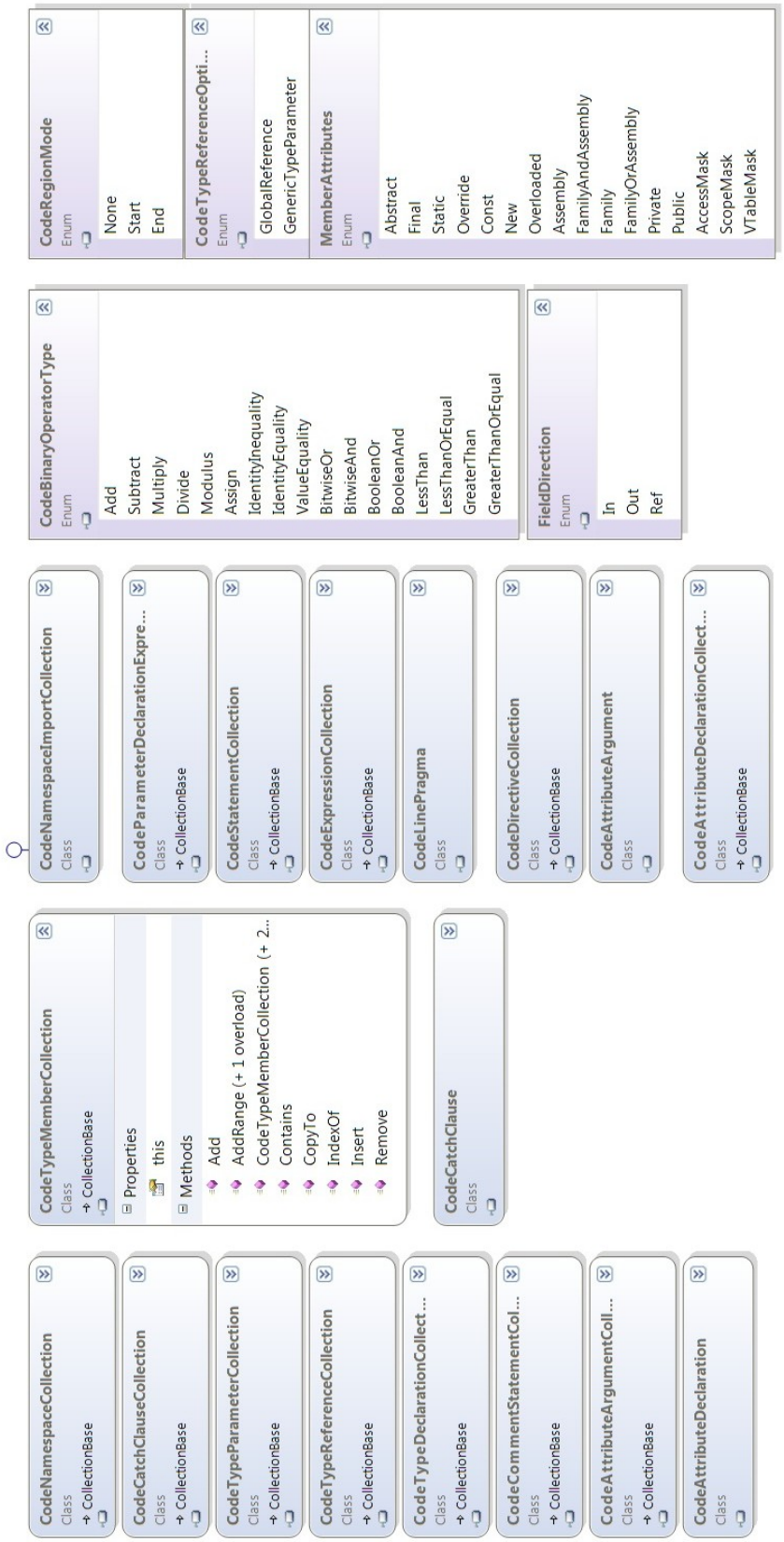

Figure 70 - Collection et énumérations principales de la bibliothèque logicielle CodeDOM

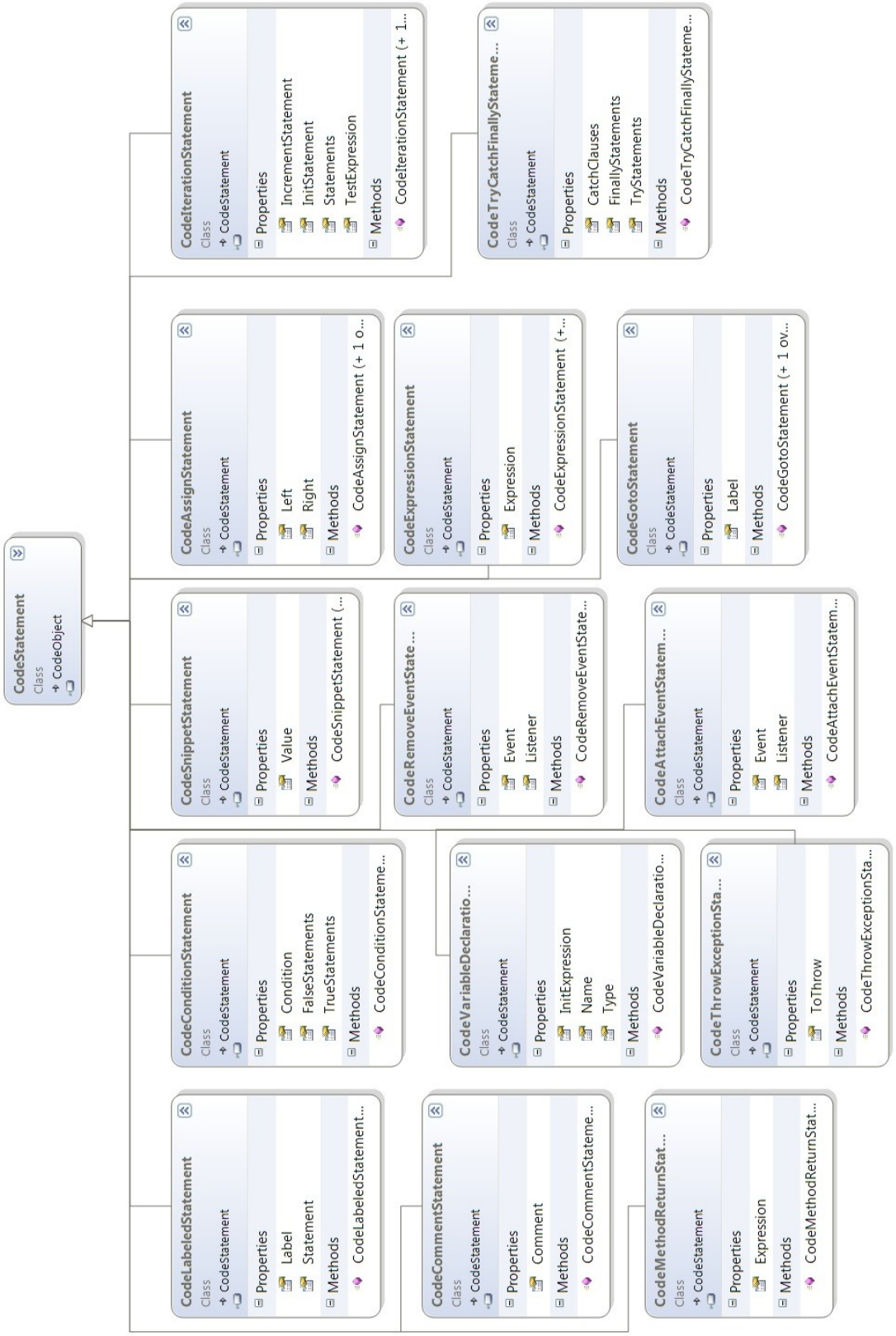

Figure 71 - Détails de la classe CodeStatement

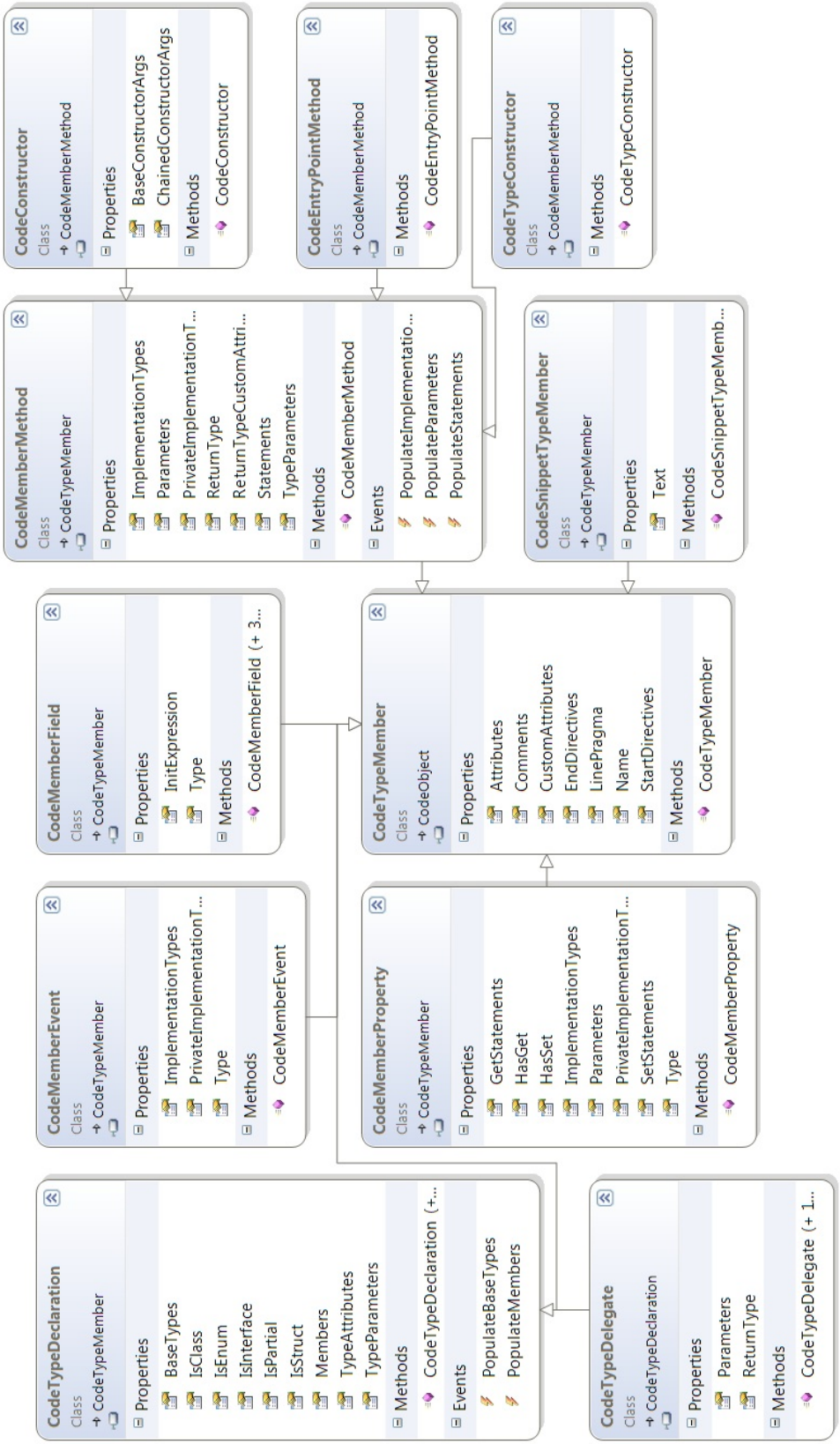

Figure 72 - Détails de la classe CodeType

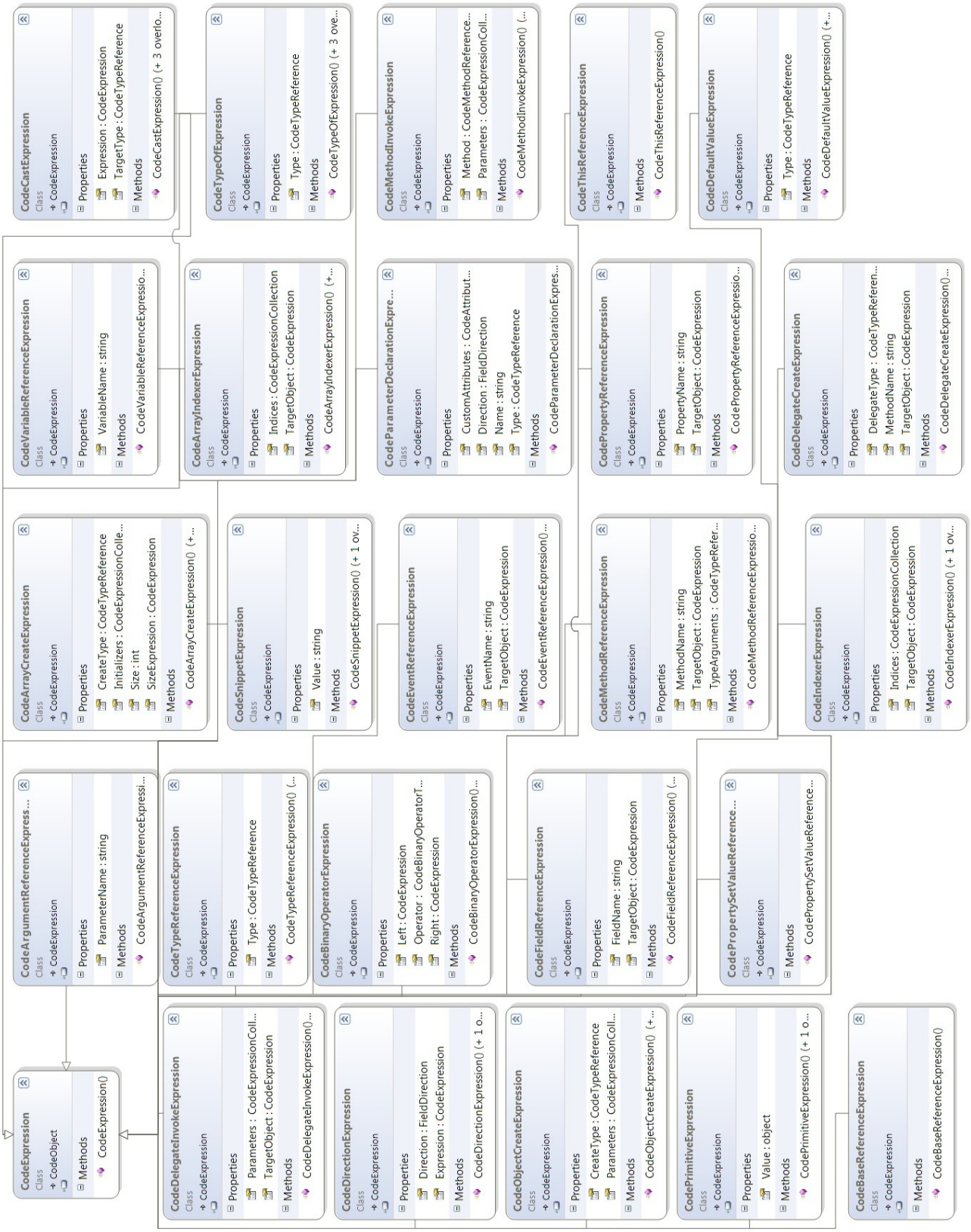

Figure 73 - Détails de la classe CodeExpression

Annexe 3
Modèle physique de données du schéma INFORMATION_SCHEMA

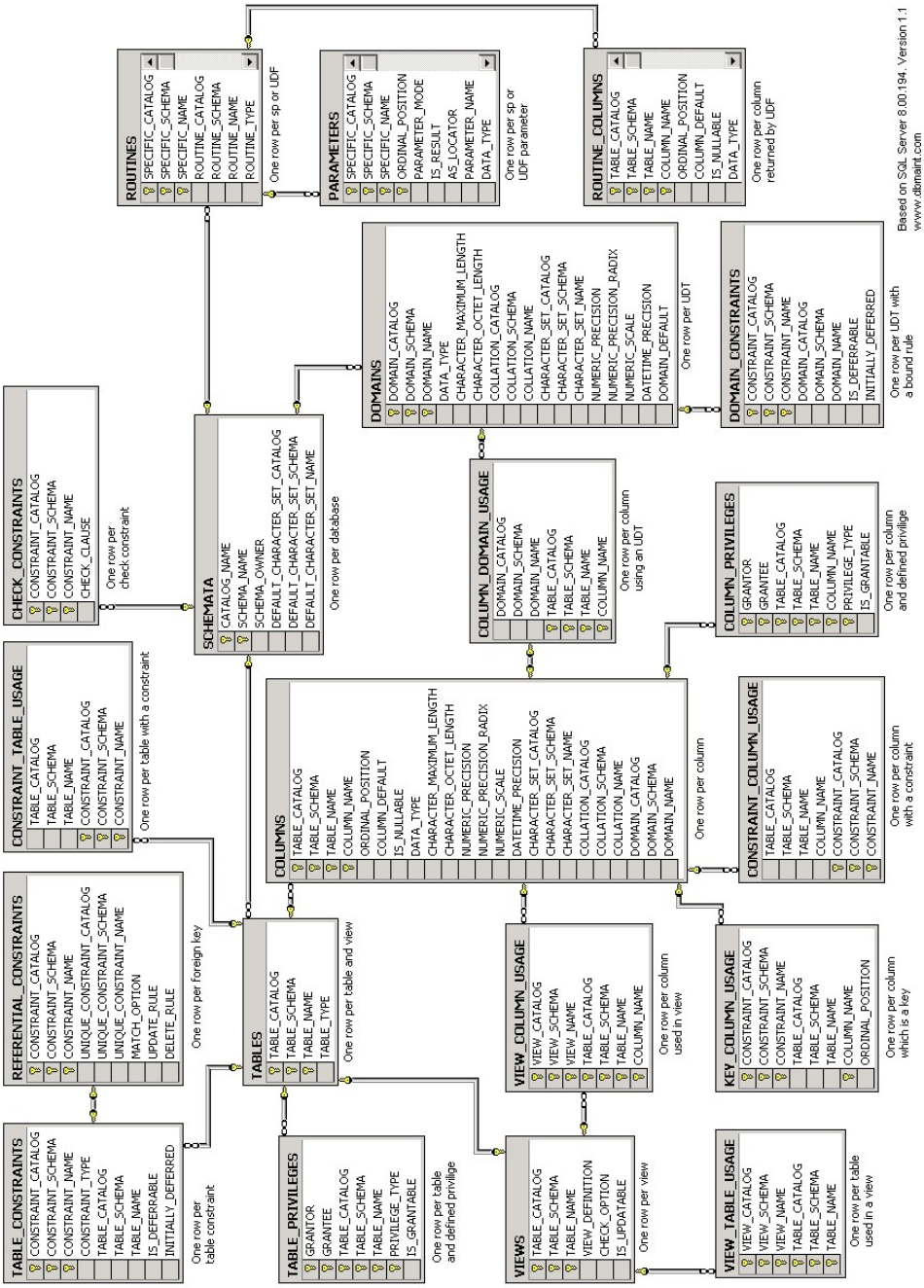

Figure 74 - Modèle physique de données du schéma INFORMATION_SCHEMA

Table des figures

Figure 1 - Evolution des effectifs d'Algorys 6
Figure 2 - Evolution du chiffre d'affaire d'Algorys 6
Figure 3 - Organigramme général Algorys 2011 7
Figure 4 - Diagramme des cas d'utilisation des applications de gestion 8
Figure 5 - Exemple de formulaire d'utilisation et de paramétrage 8
Figure 6 - Architecture 3-tiers 9
Figure 7 - Définition des modules à générer 12
Figure 8 - Description des versions 13
Figure 9 - Ordonnancement des tâches 14
Figure 10 - Planning du projet 15
Figure 11 - Schéma de l'approche n-uplet 19
Figure 12 - Architecture de référence 20
Figure 13 - Concepteur de modèle LINQ to SQL 21
Figure 14 - Exemple de requête LINQ to SQL : Chargement 22
Figure 15 - Exemple de requête LINQ to SQL : Mise à jour 22
Figure 16 - Schéma d'architecture d'Entity Framework 23
Figure 17 - Architecture Entity Data Model 23
Figure 18 - Exemple de schéma conceptuel sous Visual Studio 24
Figure 19 - Schémas d'interrogation des données dans Entity Framework 25
Figure 20 - Schéma de haut niveau de l'architecture de NHibernate 26
Figure 21 - Exemple de correspondance XML 27
Figure 22 - Exemple de correspondance "Fluent" 27
Figure 23 - Exemple de requête HQL 27
Figure 24 - Exemple de requête Critera Queries 28
Figure 25 - Processus de génération avec un modèle T4 "Run Time" 30
Figure 26 - Classe générée par le modèle T4 "Run Time" 31
Figure 27 - Processus de génération d'un modèle T4 "Design Time" 32
Figure 28 - Schéma de flux d'une transformation XSLT 33
Figure 29 - Exemple d'une transformation XSLT 33
Figure 30 - Schéma de principe CodeDOM 34
Figure 31 - Exemple de code CodeDOM pour la génération d'une classe 36
Figure 32 - Exemple d'un arbre dans CodeDOM 37
Figure 33 - Résultat de la génération de source de l'exemple de l'arbre CodeDOM 38
Figure 34 - Organisation hiérarchique du modèle PAC 40
Figure 35 - Composition modèle PAC 41
Figure 36 - Technique de conception « Monolithique » 41
Figure 37 - Modèle de programmation avec modèle 42
Figure 38 - Technique de génération de formulaire à l'aide de métadonnées 42
Figure 39 - Exemple de génération d'IHM 43
Figure 40 - Exemple de mise en œuvre du Data grid pattern 44
Figure 41 - Exemple d'une mise en œuvre du « Form pattern » 44
Figure 42 - Détail du schéma XML des métadonnées – MetaDataColumn 45
Figure 43 - Architecture globale ASP.NET 47
Figure 44 - Web Forms : Structure des fichiers 48
Figure 45 - Architecture logique des modules AlgorysORM 50
Figure 46 - Dépendances des bibliothèques AlgorysORM 51

Figure 47 - Schéma de référencement entre l'application et la bibliothèque logicielle AlgorysORM.Reference.. 52
Figure 48 - Présentation de l'outil AlgorysORM 53
Figure 49 - Schéma de génération de la couche ORM 54
Figure 50 - Diagramme de classes représentant des métadonnées de la base de données 56
Figure 51 - Diagramme de classes - Génération du code source par la couche ORM 57
Figure 52 - Diagramme de classes - Exemple de classe "entité" générée 59
Figure 53 - Diagramme de classes - Exemple de classe "méthodes" générée 59
Figure 54 - Diagramme de classes BusinessMethods 59
Figure 55 - Diagramme de classes - Interfaces pour les classes "méthodes" 59
Figure 56 - Schéma de présentation de l'interface utilisateur du composant 61
Figure 57 - Schéma de principe de l'architecture du composant 61
Figure 58 - Architecture du composant graphique 62
Figure 59 - Processus de génération et d'exécution du formulaire 62
Figure 60 - Processus de vie d'un contrôle personnalisé 64
Figure 61 - Diagramme de classes – Génération du code source pour le formulaire 65
Figure 62 - Exemple de code de déclaration d'un contrôle personnalisé 66
Figure 63 - Diagramme de classes - Exemple d'une classe correspondant à une table Categories 67
Figure 64 - Diagramme de classes - Représentation des métadonnées d'un contrôle personnalisé 68
Figure 65 - Diagramme de flux de contrôle 69
Figure 66 - Exemple de méthode testé par Pex 70
Figure 67 - Résultat de l'exécution de Pex 70
Figure 68 - Base de données de test : Northwind 76
Figure 69 - Classes principales de la bibliothèque logicielle CodeDOM 77
Figure 70 - Collection et énumérations principales de la bibliothèque logicielle CodeDOM 78
Figure 71 - Détails de la classe CodeStatement 79
Figure 72 - Détails de la classe CodeType 80
Figure 73 - Détails de la classe CodeExpression 81
Figure 74 - Modèle physique de données du schéma INFORMATION_SCHEMA 82

Liste des tableaux

Tableau 1 - Comparaison des outils d'ORM 29
Tableau 2 - Résultat du comparatif des performances 29
Tableau 3 - Comparaison des méthodes de génération de code source 39
Tableau 4 - Liste des vues du dictionnaire de données utilisées..... 55
Tableau 5 - Correspondance des types Base de données / Fournisseur de données Microsoft / Objet 58

Résumé

Etude d'une solution de génération d'IHM pour la mise à jour des données de référence utilisant l'approche ORM.

Mémoire d'Ingénieur C.N.A.M., Lyon 2012

RESUME

L'étude d'une solution de génération d'IHM a permis d'analyser les différentes approches de l'ORM ainsi que les différentes techniques de conception d'IHM.

La réalisation du projet a confirmé le gain de productivité induit par, d'un côté la génération de la couche d'accès aux données en se basant sur le modèle structurel de la base de données et de l'autre, la génération de composants de type formulaire pour la mise à jour des données de référence.

Mots clés : ORM, correspondance objet relationnel, Interaction Homme Machine, Présentation Abstraction Contrôle, System.CodeDOM.

SUMMARY

The study of a generation of HMI solution was used to analyse the different approaches to the ORM and the various techniques for designing GUI.

The project confirmed the productivity gain induced by the one hand the generation of the data access layer based on the database structural model and the other the generation of the form components to modify the master data.

Key words : ORM, object relational mapping, Human Machine Interaction, Presentation Abstraction Control, System.CodeDOM.