**Rapports de stage – Technord**

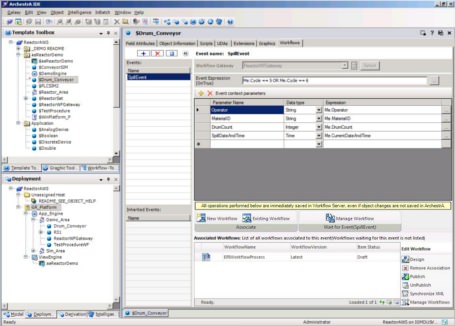
Semaine 01/02 – Installation, doc, découverte ArchestrA ;   
 – GRAccess, datacontext, récupérer galaxies, templates.

A mon arrivée, j’ai reçu un ordinateur portable pour travailler. Malheureusement, l’ordinateur n’était absolument pas prêt : pas de session à mon nom (je travaille toujours sur la session d’un ancien étudiant le 14/02), machines virtuelles à installer, programmes à installer (Visual Studio, ArchestrA IDE, GRAccess, accès à un NAS de données partagées, versioning SVN, Dropbox avec documentation).

Il a donc fallu consacrer les deux premiers jours du stage à installer tous les éléments nécessaires. Une machine virtuelle distante, avec Windows Server 2012, a été créée pour y stocker le programme ArchestrA, sa base de données SQL Server, ainsi que l’IDE d’ArchestrA. L’accès à la VM se fait via le bureau à distance (ou via VMWare WorkStation, pour les installations).

L’IDE Visual Studio 2013 ainsi que le toolkit GRAccess (permettant de communiquer en C# et C++ avec ArchestrA) ont été installés localement sur l’ordinateur portable. Le but était de rendre possible un accès à distance des programmes développés (localement) vers ArchestrA (sur machine virtuelle). Malheureusement, ce type d’accès nécessitait un compte administrateur sur le système. Etant dans l’impossibilité d’accéder ou de créer un compte administrateur sur l’ordinateur, il m’était donc impossible de travailler ainsi. J’ai donc également installé Visual Studio et GRAccess sur la machine virtuelle distante. Un second écran a été mis à ma disposition, afin de faciliter le travail simultané avec l’ordinateur local et la machine virtuelle.

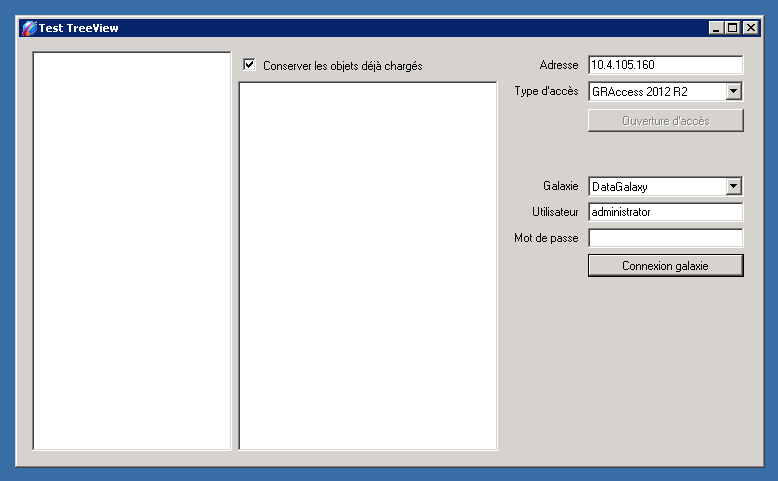
A ce stade, aucun cahier des charges concret ne m’avait été communiqué. Avant de commencer à développer, il a d’abord fallu commencer par découvrir ArchestrA et ses principes. Une banque de données (« galaxie ») m’a été fournie pour ArchestrA, afin de pouvoir regarder par moi-même comment utiliser ce programme lorsque des données sont présentes. La documentation officielle d’ArchestrA et GRAccess m’a également été fournie.



Après avoir épluché la documentation, j’ai mieux compris l’organisation des données dans ArchestrA. Les projets sont nommés « galaxies ». Chaque galaxie contient une série de « templates » de base (il s’agit de modèles de paramétrage pouvant être utilisés par des objets concrets). Une série de templates personnalisés peuvent être créés à partir des templates de base. On obtient ainsi une hiérarchie de templates, visible sous forme d’arbre dans ArchestrA IDE. Une fois les templates en place, on peut créer des « instances », qui sont des objets concrets reprenant les paramètres d’un template précis (et désignant, par exemple, une machine ou un moteur).

Ma première mission était de découvrir l’usage de GRAccess, permettant la communication avec ArchestrA à partir d’une application C# ou C++. Malheureusement, les personnes présentes autour de moi n’avaient guère de connaissances au sujet de GRAccess. La documentation officielle s’est également révélée très limitée et souvent ambigüe. E n supplément, j’ai constaté que GRAccess n’était pas un toolkit très connu, puisque même les recherches sur Internet ne retournaient que très peu de résultats (parfois aucun). La seule solution a donc été de réaliser moi-même des tests pour comprendre son fonctionnement. Heureusement, la fonctionnalité d’autocomplétion de Visual Studio m’a permis d’obtenir la liste des appels de fonctions disponibles pour les différents objets GRAccess. J’ai donc analysé le comportement et le paramétrage de ces fonctions, jusqu’à trouver un moyen d’obtenir à peu près les résultats souhaités.

Après de laborieuses recherches, je suis parvenu à réaliser la connexion de mon application vers une galaxie ArchestrA. Le résultat obtenu est un « datacontext » reposant sur des principes similaires à l’outil LinQ (pour les bases de données). Le principe est le suivant : des fonctions propres au toolkit sont utilisées pour récupérer des données ou pour en ajouter ou supprimer. Les données récupérées sous forme d’objets restent liées à leur source. Ainsi, si on modifie une propriété d’un objet récupéré, les modifications seront répercutées dans la base de données.

Il m’a été demandé d’utiliser la version 2012 R2 de GRAccess, car la plupart des projets utilisent ArchestrA 2012 R2. Cepen-dant, il se pourrait qu’une autre version de GRAccess soit ajoutée ultérieurement (2014, par exemple). Pour cette raison, j’ai choisi d’organiser l’accès selon le patron de conception « factory ». Ce dernier permet qu’une demande d’instanciation puisse potentiellement créer des objets différents, tout en offrant une abstraction vis-à-vis de l’interfaçage de l’objet.

Après avoir chargé GRAccess, il m’a été possible de récupérer la liste des galaxies présentes dans ArchestrA. Le login vers cette galaxie permet d’obtenir un datacontext.

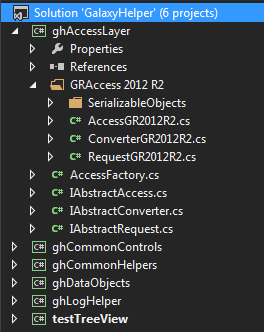
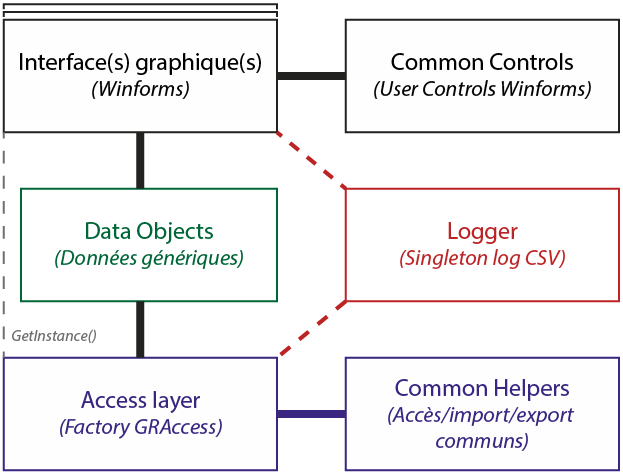
J’ai choisi de répartir la structure du programme en plusieurs projets, afin d’en faciliter le dévelop-pement, la maintenance et l’évolution. L’application est développée en C#.

Un projet est dédié à la communication avec ArchestrA, reprenant la « factory » pour le type d’accès (version). Cette « factory » contient elle-même des « singletons », permettant de s’assurer que chaque type d’accès ne puisse être instancié qu’une seule fois, et que ce même accès puisse être récupéré n’importe où. Les types d’objets récupérés sont propres à chaque type/version.

Pour que l’interface graphique puisse rester la même dans tous les cas, il a donc fallu ajouter une couche d’abstraction, reprenant des objets génériques équivalents (data objects). D’autres fonction-nalités y ont également été ajoutées, telles que des classes permettant la réalisation d’un arbre de données.

Un troisième projet a été créé pour l’interface graphique et la gestion d’événements. D’autres projets pourront être créés dans ce même but, selon les applications souhaitées. Ces projets utilisent des objets d’accès (couche de communication) et des objets de données (couche d’abstraction) des deux premiers projets. Comme certains contrôles seront communs à ces projets graphiques (par exemple, l’arbre de templates ou la liste d’instances), ces contrôles ont été regroupés sous forme de « user controls » dans un quatrième projet.

Enfin, un projet de « log » a été ajouté. Il renferme un « singleton », accessible depuis tous les autres projets. Son rôle est d’enregistrer un historique des événements et des erreurs.

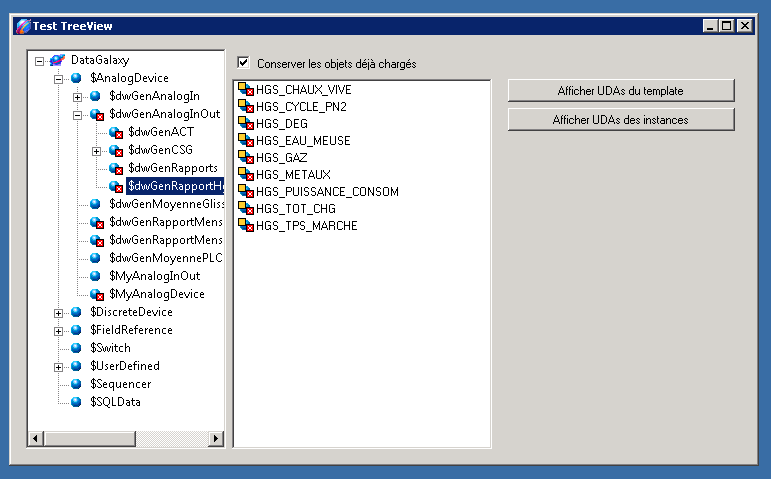
 

La première tâche à réaliser, en attendant de recevoir un cahier des charges complet, était la récupération de la hiérarchie des templates d’une galaxie (sous forme d’arbre), ainsi que la récupé-ration de la liste des instances, pour un template donné. Plusieurs classes ont été créées dans le projet « data objects » pour y parvenir (notamment une classe pour les nœuds d’un arbre, com-prenant notamment un algorithme pour créer un arbre hiérarchique à partir d’une liste. Un contrôle d’arbre a également été ajouté au projet « common controls ».

A partir de copies d’écran de l’IDE d’ArchestrA, il m’a été possible de rassembler les différentes icônes des éléments (galaxies, templates, instances). Après les avoir organisées l’une à la suite de l’autre, au sein d’un fichier image, j’ai mis au point un système de « sprites » permettant de récupérer chaque icône dans un objet de l’application. Elles seront par la suite utilisées pour les différents contrôles (arbre de templates, liste d’instances, …).

Semaine 08/02 – Arbre de templates, liste d’instances, attributs UDA ;  
 – Optimisation (index, stockage), log CSV.

La première partie de la semaine a été consacrée à la récupération des templates et des instances, et à leur affichage sous forme d’arbre (templates) et de liste (instances du template sélectionné). Les icônes réalisées précédemment ont été associées aux contrôles d’arbre et de liste créés.



Récupérer les templates n’a pas été facile. Aucune fonction de GRAccess ne permet de récupérer la liste complète des templates (une condition de filtrage devant absolument être spécifiée, par exemple par nom, ou par ancêtre). Dans un premier temps, j’ai contourné ce problème en hard-codant les noms des templates de base, et en récupérant, pour chaque template de base, la liste de ses descendants. Toutefois, je me suis ensuite aperçu que les noms des templates de base pouvaient être modifiés dans l’IDE ArchestrA, ce qui rendait cette première méthode insuffisante. De plus, la récupération prenait plusieurs minutes.

La solution finale était donc d’utiliser une condition « prétexte » qui soit toujours vraie : « ne doit pas être dérivé de », utilisé avec une valeur qui n’existera jamais (car elle ne commence pas par le symbole « $ » requis pour les templates). Cette méthode permettait de récupérer l’intégralité des templates d’un seul coup, ce qui évitait non seulement le problème de renommage potentiel, mais accélérait aussi considérablement la durée du procédé (passant de plusieurs minutes à une vingtaine de secondes). Toutefois, un nouveau problème survenait : la liste comprenait à présent des « templa-tes » fonctionnels, tels que les « vues » InTouch et les serveurs ArchestrA. Le maître de stage m’a demandé de ne pas lister ces éléments, c’est pourquoi j’ai ajouté une vérification supplémentaire pour les éliminer avant de construire l’arbre hiérarchique.

Pour construire un arbre hiérarchique, chaque élément a été ajouté à un dictionnaire/hashtable (dont la clé d’identification est le nom de l’élément) sous forme d’objet générique (data object). Cette conversion des objets est non seulement nécessaire pour obtenir un résultat indépendant du type de version de GRAccess, mais également pour ne plus être dépendant du datacontext (et ainsi pouvoir manipuler les objets sans altérer la base de données). Pour chaque objet, le nom de l’objet parent a été récupéré. Il a ensuite suffit de construire un arbre en se servant du dictionnaire et du nom de parent de chaque élément.

La récupération des instances pour un template s’est révélée plus simple que la récupération des templates. En effet, il s’agissait d’une simple liste. De plus, la condition de récupération était évidente (le nom du template correspondant). Toutefois, cette phase du développement a mis en évidence un gros problème : la lenteur extrême du toolkit GRAccess. Si la récupération des templates ne nécessitait d’attendre qu’une seule fois (au démarrage), il en allait différemment pour la récupération des instances. A chaque sélection de templates, il fallait attendre pour une durée allant de quelques secondes à quelques minutes (selon le nombre d’instances du template choisi).

Le problème de lenteur lors d’une première récupération ne pouvait pas être contourné. En revan-che, il pouvait être évité pour récupérer à nouveau des instances déjà chargées précédemment. Pour ce faire, il suffisait de stocker en mémoire chaque élément chargé. Lors d’une demande de récupé-ration, il suffit alors de vérifier si l’objet est déjà chargé ou si une requête est nécessaire. Pour ne pas perdre de temps lors de la vérification, les objets sont stockés dans des dictionnaires/hashtables, ce qui permet des accès directs (en utilisant leur nom comme clé).

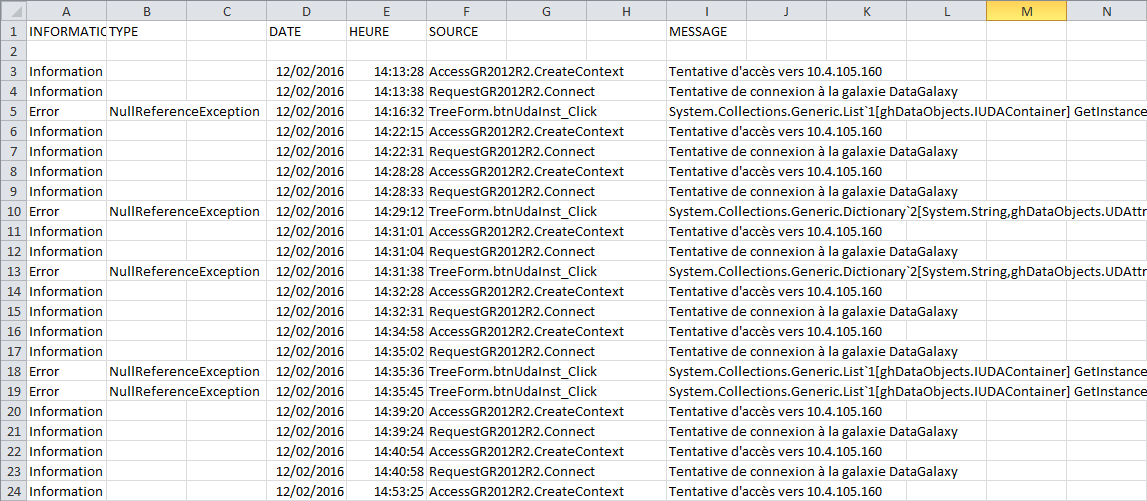
Ce système de stockage des éléments présente un gain de temps non négligeable. Toutefois, il pose un nouveau problème : les éléments stockés gardent l’état qu’ils avaient lorsqu’ils ont été chargés. Ainsi, si une autre personne modifiait un objet ArchestrA entre-temps, la version stockée ne serait plus à jour. Pour éviter ce problème, j’ai donc ajouté une option au programme permettant d’activer ou non le stockage des éléments (ce qui permet de choisir entre un gain de temps (si l’on sait que personne d’autre n’utilise la galaxie courante, ce qui sera généralement le cas) ou un gain de fiabilité (si plusieurs personnes manipulent la même galaxie au même moment).

Enfin, j’ai constaté qu’en cas d’erreur lors d’une requête GRAccess, celle-ci ne se terminait jamais, ce qui bloquait totalement l’application. Lorsque le projet final à réaliser sera connu, il conviendra d’y ajouter un thread dédié aux requête, afin de garder la main sur l’application (et de mettre au point un éventuel système de timeout).

Après avoir terminé la première fonctionnalité (arbre de templates et liste d’instances), le cahier des charges n’était toujours pas disponible. J’ai tout de même été informé de la prochaine fonctionnalité à développer. Parmi les attributs contenus dans un template ou une instance, on retrouve notamment des UDAs (User Defined Attributes), qui sont des types d’attributs propres à chaque élément (contrairement aux attributs standards, dont seule la valeur est propre aux éléments). Mon but serait donc de récupérer la liste des UDAs de chaque élément (divisée en deux parties : les UDAs de l’élément lui-même, et les UDAs hérités des parents).

Avant de découvrir les principes des UDAs, il m’a semblé judicieux de réaliser le contenu de la classe de log, afin de permettre un développement et un débogage plus simples. Pour ce faire, un « single-ton » a été mis en place, renfermant plusieurs propriétés (nom du fichier de destination et mode d’écriture). La classe de log est ainsi accessible partout, tout en garantissant qu’une seule instance soit présente (ce qui évite les problèmes d’accès concurrents au fichier de destination).

Cette classe de log permet d’enregistrer des événements informatifs (connexions, …), des avertisse-ments (élément demandé non trouvé, …) et des erreurs (exceptions et crashs). Les informations sont écrites sous forme de tableau, dans un fichier CSV.



Pour comprendre l’organisation des UDAs, j’ai utilisé l’auto-complétion de Visual Studio. Celle-ci m’a permis de réaliser que l’ensemble des attributs et UDAs d’un objet (template/instance) étaient regroupés au sein de sous-propriétés nommées Attributes et ConfigurableAttributes. Pour compren-dre la différence entre les deux, j’ai procédé à des affichages de leur contenu en mode console. Il s’est avéré que les ConfigurableAttributes ne sont en vérité qu’un sous-ensemble des Attributes. Ainsi, chaque ConfigurableAttribute est également présent dans Attributes (l’inverse n’est pas toujours vrai, en particulier pour les attributs de type « script »).

L’affichage de chacun des entrées présentes dans Attributes et de leurs valeurs m’a permis de repérer deux attributs nommés « *UDAs* » et « *\_InheritedUDAs* ». Ces attributs ne sont rien d’autre que des listes, dont la valeur prend la forme XML suivante :

<UDAInfo>

<Attribute **Name**="Object.Script.AffectInputSource"

**DataType**="MxBoolean"

**Category**="MxCategoryWriteable\_USC\_Lockable"

Security="MxSecurityOperate"

IsArray="false"

HasBuffer="false"

ArrayElementCount="0"

InheritedFromTagName=""/>

<Attribute ... />

...

</UDAInfo>

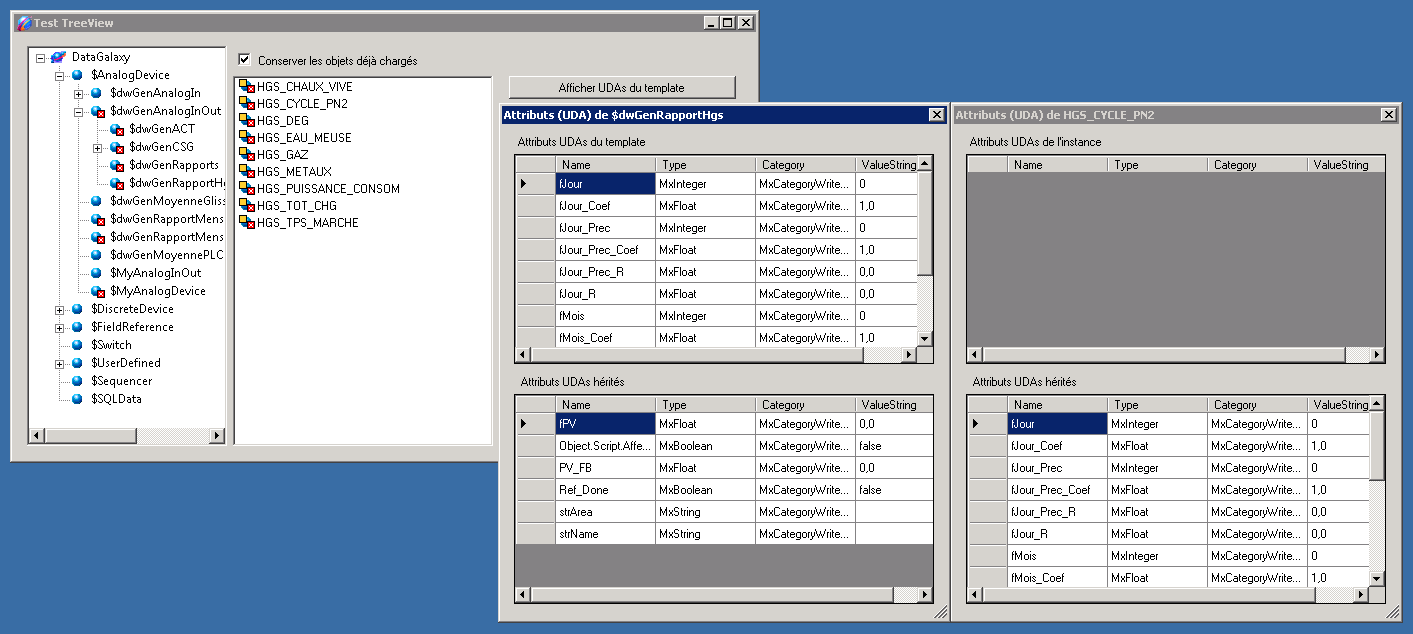
Ces deux listes permettent ainsi de connaître les noms, les types et les catégories des attributs UDAs propres et hérités. Elles ne fournissent pas la valeur de ces attributs. Ces listes peuvent être vues comme des sortes de références vers une série d’autres attributs. Pour chaque attribut mentionné au sein des listes UDAs et \_InheritedUDAs, on peut retrouver un attribut du même nom dans la liste générale Attributes (et aussi ConfigurableAttributes, pour certains d’entre-eux).

Pour résumer, les propriétés Attributes et ConfigurableAttributes contiennent une série d’attributs (dont certains sont des attributs UDAs), ainsi que deux listes qui référencent tous ces UDAs (nom-mées UDAs et \_InheritedUDAs). La valeur de UDAs et \_InheritedUDAs est une liste en XML, alors que la valeur des autres attributs désigne leur contenu respectif.

Le code XML mentionné précédemment fait donc référence cet attribut de la liste générale :   
Object.Script.AffectInputSource (dont la valeur de type MxBoolean est false).

Pour pouvoir récupérer la liste des UDAs propres et hérités, il est nécessaire d’interpréter le code XML récupéré dans les attributs UDAs et \_InheritedUDAs. Pour ce faire, j’ai créé une classe *UDAInfo* et une classe *Attribute*, dont les variables membres et le contenu respectent la même structure et les mêmes noms le code XML. L’outil de désérialisation XML du C# a ensuite pu être utilisé, afin de récupérer un objet UDAInfo contenant une liste d’objets Attribute. Un simple parcours de liste a ensuite permis de récupérer les valeurs des attributs, en se servant de leur nom pour les trouver.

Des fonctions ont été ajoutées pour permettre la récupération (et le stockage, si demandé) des UDAs pour une série de templates ou d’instances. Il a ensuite suffi d’ajouter les contrôles appropriés pour les interfaces graphiques. La liste d’instances de l’interface graphique permet une sélection multiple, ce qui a nécessité (dans le cas d’une demande d’UDAs pour une sélection multiple) une fonction ne conservant que les UDAs communs (et n’affichant leur valeur que si elle est commune également).



Pour terminer, j’ai ajouté au projet un manifeste réclamant des droits d’administrateur. En effet, la connexion GRAccess ne peut fonctionner que si l’application dispose de droits d’administrateur. L’ajout du manifeste permet qu’une demande d’autorisation soit effectuée automatiquement au lancement de l’application (plutôt que de devoir, manuellement, lancer en tant qu’administrateur).

Semaine 15/02 – Recherche de templates/instances, export liste, sélection.

Avant de créer davantage de composants, j’ai commencé par optimiser ceux créés précédemment. Après avoir ajouté une multitude de traces et effectué de nouveaux tests, j’ai réalisé qu’environ 75% du temps de récupération des objets via GRAccess était consacré à la vérification de leur statut (erreurs ou non, avertissements, checkout). Il s’est avéré que les objets retournés par le datacontext ne contenaient en vérité qu’un lien symbolique vers ces informations. En d’autres termes, pour chaque élément dont le statut était vérifié, une nouvelle requête s’exécutait implicitement.

Après avoir ajouté temporairement une option pour rendre la lecture du statut optionnelle (ce qui permettait de réduire le temps de chargement par quatre), j’ai ajouté un bouton de rafraîchissement à l’application. Ce dernier permet de recharger l’intégralité des objets stockés en mémoire (la liste des templates, l’arbre, et les instances stockées). Cette fonctionnalité vient compléter l’option de stockage ajoutée précédemment (si chaque objet obtenu reste stocké en mémoire, il suffit de rafraîchir pour qu’il soit mis à jour en fonction de nouveaux changements survenus entre-temps).

J’ai par la suite réalisé, qu’outre les vérifications d’erreurs et avertissements, les objets possédaient une propriété (nommée « validation ») reflétant leur statut via une valeur d’une énumération. Ainsi, il était possible de connaître le statut des objets sans vérifier leurs erreurs et avertissements. Ceci permettait donc d’obtenir le statut des objets sans ralentir la récupération, rendant ainsi mon option (lecture ou non du statut) totalement inutile. J’ai donc supprimé l’option et procédé à une lecture systématique du statut à partir de la propriété appropriée.

La principale nouveauté de cette semaine consistait à créer une fonction de recherche. Celle-ci permettrait de rechercher des templates et/ou instances à partir de leur nom, ou d’une partie de leur nom. Une fois les résultats obtenus, ils seraient affichés dans une liste. Il serait alors possible d’exporter la liste des noms des résultats obtenus (sous forme de fichier CSV), ou encore d’activer le nœud correspondant à la sélection dans l’arbre de templates et dans la liste des instances.

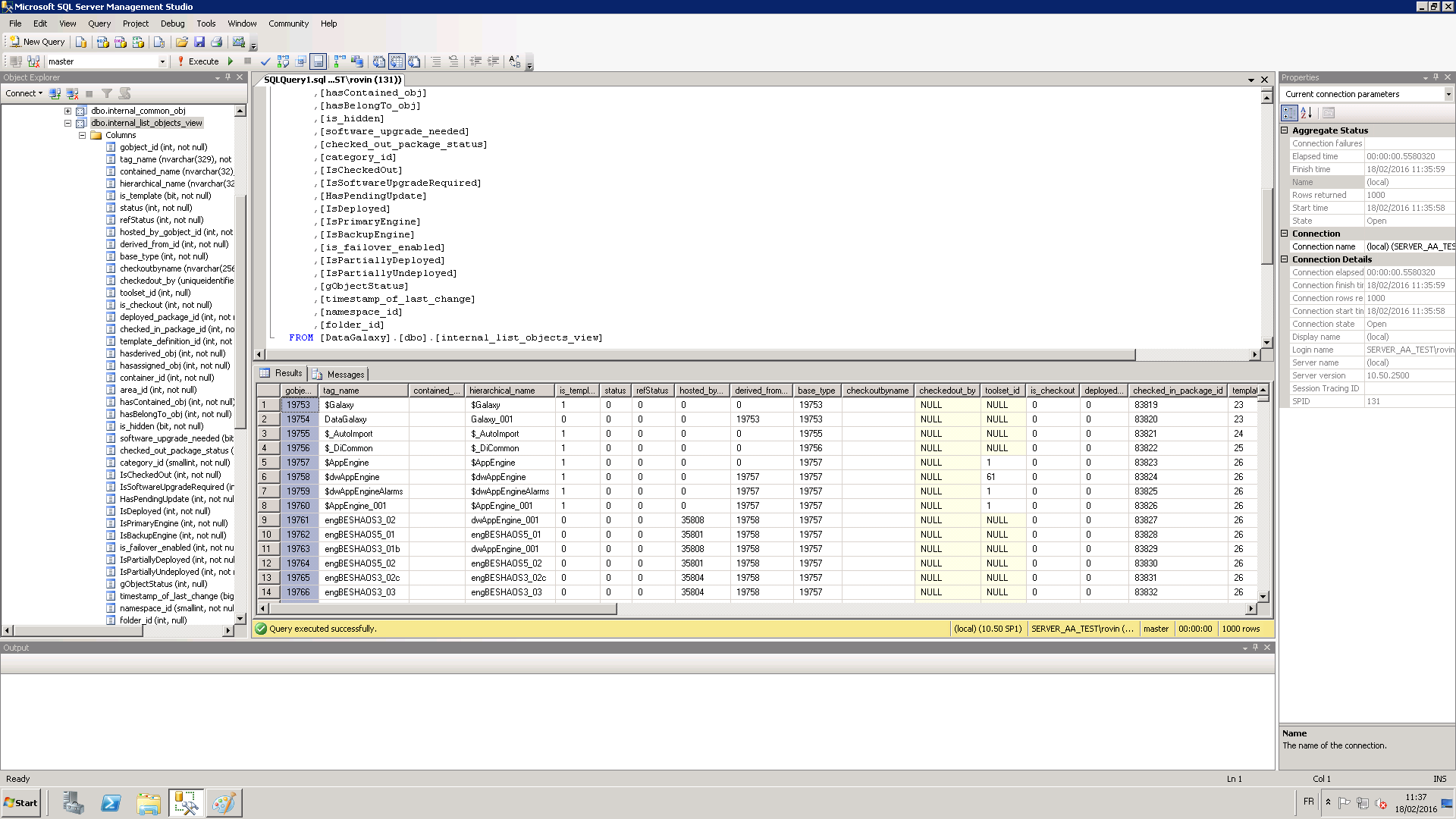
L’utilitaire GRAccess s’est rapidement révélé bien trop limité pour réaliser une telle fonctionnalité. Il ne permet en effet que la récupération d’un objet par son nom exact (et sensible à la casse). Une possibilité aurait pu être de récupérer l’intégralité des objets, et de les vérifier un à un via une expression régulière. Cependant, cette solution aurait été inutilement complexe, et la recherche aurait été d’une lenteur extrême (ce qui est gênant pour une fonctionnalité utilisée fréquemment).

Pour cette raison, j’ai décidé de réaliser la recherche en SQL, directement dans la base de données ArchestrA, sans passer par GRAccess. Cette solution s’est révélée nettement plus efficace et bien plus simple à mettre en œuvre. Elle a toutefois nécessité l’analyse détaillée du contenu de la base de données, ainsi qu’une recherche importante de documentation pour parvenir à s’y connecter via une application tierce.

La phase d’analyse de la base de données a été réalisée en explorant celle-ci via le logiciel Microsoft SQL Server Management Studio. Il semble qu’ArchestrA crée une base de données pour chaque « galaxie ». Ces bases de données contiennent un nombre de table gigantesque et peu claires. Cependant, elles contiennent également des « vues » SQL, beaucoup plus explicites et beaucoup moins nombreuses. C’est en utilisant l’une de ces vues (*dbo.internal\_list\_objects\_view*) que je suis parvenu à réaliser la fonctionnalité de recherche.

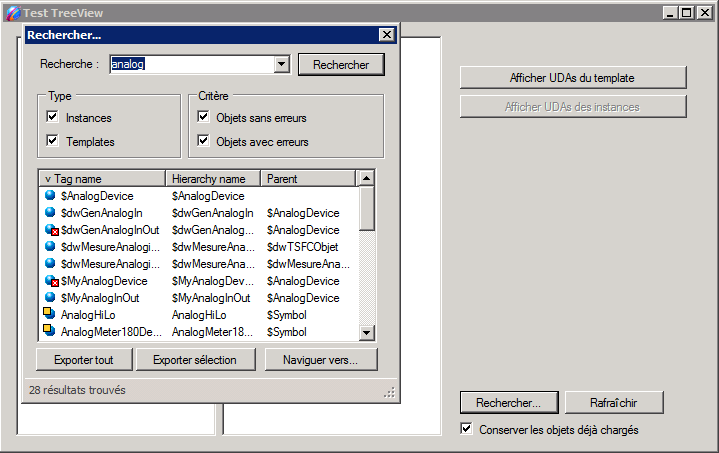
L’affichage des données de la vue permet de repérer facilement les différentes colonnes et ce qu’elles contiennent. Ceci m’a aidé à écrire la requête SQL de recherche, que j’ai pu directement tester au sein de Sql Server Management Studio.



Après avoir mieux saisi les rouages de la base de données et avoir écrit la requête SQL appropriée, il fallait intégrer celle-ci au sein de mon application C#. Après être parvenu à une connexion vers la base de données, il a été relativement aisé d’implémenter la requête de recherche.

Prévoyant le cas où la base de données serait verrouillée et inaccessible, j’ai également ajouté une fonction alternative permettant la recherche via GRAccess. Cette alternative ne permet toutefois la recherche qu’en fonction du nom exact de l’objet.

La prochaine étape consistait à récupérer les résultats, les regrouper au sein d’une liste, et réaliser l’interface graphique de recherche. Cette interface devait permettre d’entrer un nom et de choisir de chercher des templates et/ou des instances. J’y ai également ajouté un historique des recherches (permettant de resélectionner un nom recherché précédemment, en utilisant une liste de sélection en guise de champ de recherche), ainsi qu’une option de récupération d’objets avec ou sans erreurs.



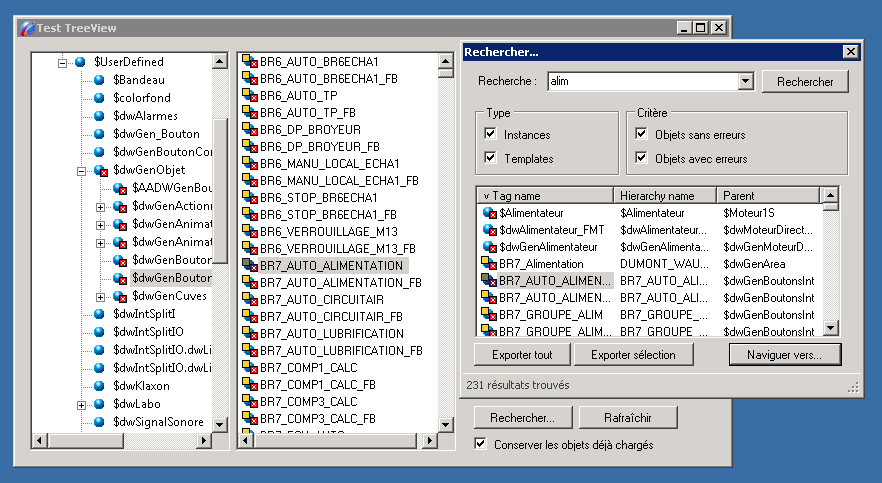
L’affichage des résultats s’est avéré relativement complexe. En effet, les « ListView » du C#, si elles permettent d’afficher plusieurs colonnes, ne permettent pas le tri en fonction d’une d’entre-elles.

Il s’est donc révélé nécessaire de créer une classe dérivée, afin de pouvoir la personnaliser. J’y ai ainsi ajouté la gestion des clics sur les en-têtes des colonnes, déclenchant le tri en fonction du contenu associé à cette colonne. En cas de clic répété sur une même colonne, le sens du tri est inversé. J’ai également inclus l’affichage d’icônes à la liste. Quant aux résultats, leur nombre est affiché en bas de la fenêtre, dans une barre de statut.

Il manquait toutefois un indicateur, afin de marquer la colonne de tri sélectionnée et le sens du tri. J’ai opté pour la simplicité en concaténant un symbole de flèche au nom de la colonne (et en le retirant, pour la colonne désélectionnée).

L’export des résultats de recherche en liste CSV s’est révélé très simple et ne présente guère d’intérêt. Cette fonctionnalité est temporaire, puisque l’application finale permettra différents types d’exportations. J’ai toutefois jugé utile de proposer à la fois un bouton pour exporter toute la liste, ainsi qu’un second bouton pour n’exporter que les éléments sélectionnés.

Il m’a été demandé de pouvoir naviguer dans l’application vers un résultat de recherche. Ceci permet un gain de temps à l’utilisation. Par exemple, si on ne se souvient pas de l’arborescence complète pour accéder à une instance, il suffit de la recherche, puis de presser le bouton « naviguer vers ». L’arbre des templates sélectionne alors l’élément (dans le cas d’un template recherché) ou son template parent (dans le cas d’une instance). Si l’élément demandé est une instance, celle-ci est également sélectionnée dans la liste des instances.



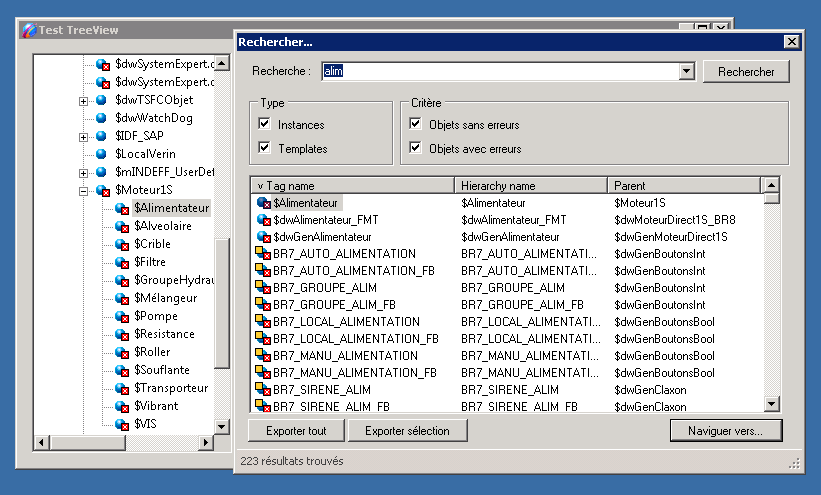
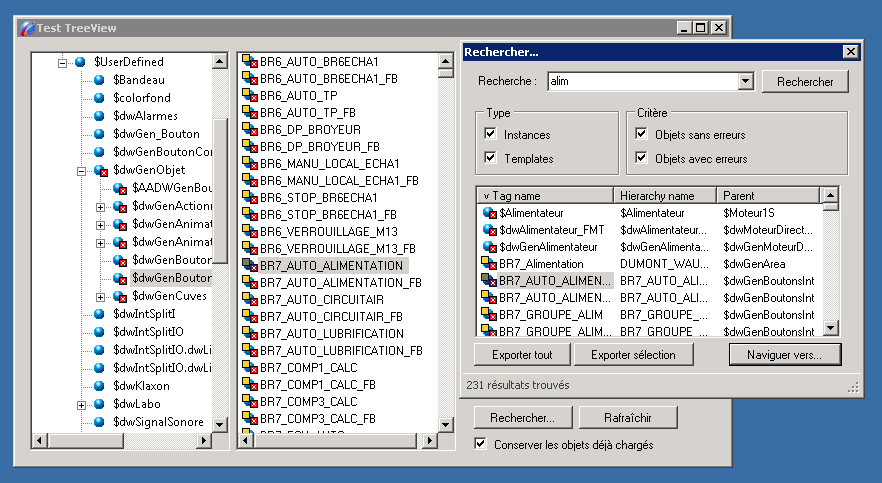
Pour réaliser cette fonctionnalité, un parcours de hiérarchie était nécessaire. Heureusement, chaque élément de l’arbre conteneur des templates contenait le nom de son parent parmi ses propriétés, et un index (dictionnaire) reprend la liste de tous les templates (c’est la méthode que j’utilisais pour obtenir des informations sur un nœud sélectionné dans l’arbre, par exemple pour récupérer ses instances).

Il suffisait donc d’accéder directement au template via l’index, puis de parcourir sa hiérarchie via la propriété « parent ». La liste des entrées à parcourir était ensuite envoyée à l’interface graphique, afin d’ouvrir chaque nœud intermédiaire de l’arbre graphique, pour enfin sélectionner la cible.

Si l’élément à sélectionner était une instance, c’était son template parent qui était sélectionné dans l’arbre graphique. Une fois la liste des instances obtenue, une recherche dichotomique a permis de localiser l’instance ciblée, afin de pouvoir la sélectionner dans la liste.

Pour terminer, j’ai pris en compte l’exclusion des objets descendants de certain parents. Comme il m’avait été demandé, pour la réalisation de l’arbre, d’ignorer certains éléments (moteurs, vues, …), il m’a semblé judicieux que ces éléments n’apparaissent pas non plus dans la recherche. Une simple modification de la requête SQL a permis ce filtrage.

Par ailleurs, j’en ai profité pour améliorer l’utilisabilité de la recherche. L’utilisation de la touche ENTER lorsque le champ de recherche est actif permet à présent de lancer la recherche sans devoir presser le bouton de recherche. J’ai également fait en sorte de rendre la fenêtre « responsive » (élastique) : le contenu s’adapte en fonction de la taille de la fenêtre.

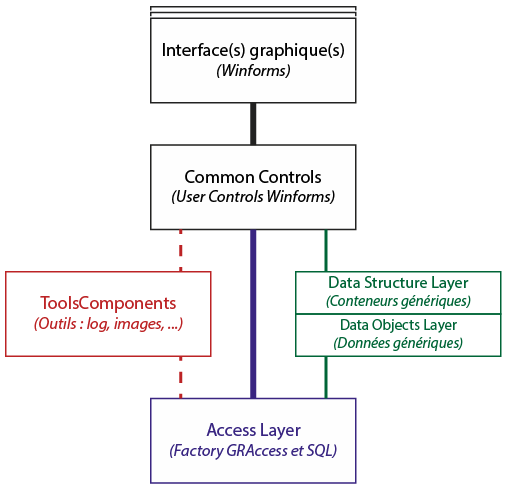


Pour terminer la semaine, j’ai fait en sorte que la sélection courante de l’arbre des templates soit mémorisée, de sorte qu’en cas de rafraîchissement, la sélection soit conservée et reste visible.

Semaine 22/02 – Cahier des charges, analyse, réorganisation ;   
 – Redémarrage du projet, connexion, récupération dossiers.

Un cahier des charges plus étoffé m’a été communiqué à la fin de la semaine dernière. Il n’est pas encore complet, mais contient suffisamment d’informations pour avoir un bon aperçu de la direction prise par le projet. Malheureusement, certaines directives sont en contradiction avec celles fournies précédemment. D’autres auraient pu permettre de mieux orienter le projet, si elles avaient été connues avant. Entre-temps, le collègue de mon maître de stage (qui est la personne qui utilisera mon application pour le traitement et l’import/export des objets et des UDAs) est revenu de vacances. Il m’a expliqué de manière plus détaillée ce qu’il souhaite, mais cela ne correspond pas aux directives inscrites dans le cahier des charges. Celles-ci seront donc modifiées.

Le fait d’avoir reçu auparavant les instructions au compte-goutte a abouti à un projet incohérent et mal organisé. La contradiction des nouvelles instructions avec certains aspects ne ferait qu’empirer les choses. Par ailleurs, je m’aperçois que certaines choses peuvent être simplifiées : par exemple, plutôt que de séparer un arbre conteneur de données (pour obtenir les informations) et un arbre graphique (ne contenant que des labels), il serait plus intéressant de créer une classe dérivée des nœuds d’arbre graphique, afin de regrouper les deux aspects (et ainsi gagner en temps de traitement et en simplicité de code). L’ensemble de ces aspects m’a convaincu qu’il était plus sage de recom-mencer le projet à zéro. Le projet précédent passe donc au stade de « prototype ». J’en ai profité pour modifier l’organisation des projets Visual Studio comme suit :



La nature de certaines fonctionnalités demandées rendra l’accès SQL obligatoire. Plusieurs organisations sont possibles avec ArchestrA : une organisation hiérarchique, telle que je l’ai réalisée ; et une organisation par dossiers, ne tenant pas compte de la hiérarchie. Le fait de devoir reproduire ces deux organisations nécessite l’usage du SQL, car il n’existe par exemple aucun moyen de récupérer les dossiers d’organisation des templates via GRAccess. Ce n’est qu’un exemple parmi d’autres. L’accès à la base de données Sql Server n’est donc plus optionnel mais obligatoire.

Après avoir pris le temps d’analyser le projet à réaliser et de l’organiser, j’ai commencé par mettre en place toutes les structures principales. J’ai également récupéré ce qui pouvait l’être dans le précédent projet. J’ai ensuite commencé par recréer la couche d’accès GRAccess et SQL, car elle est nécessaire pour établir une connexion et réaliser des tests.

La couche d’accès a été réorganisée plus simplement. Les fonctions nécessitant GRAccess sont rassemblées dans une classe et se limitent au strict minimum. Idem pour les fonctions nécessitant l’accès SQL, qui sont regroupées dans une deuxième classe minimaliste. Ces deux classes respectent des interfaces, afin de pouvoir être sélectionnées par un pattern « factory » (ou que d’autres puissent être utilisées pour une autre version d’ArchestrA que la 2012 R2).

Les deux classes propres à une seule version d’ArchestrA sont minimalistes. Le reste du travail est rassemblé dans une classe principale de la couche d’accès. Cette classe rassemble l’ensemble des requêtes de logique métier sans se préoccuper du type de version utilisé.

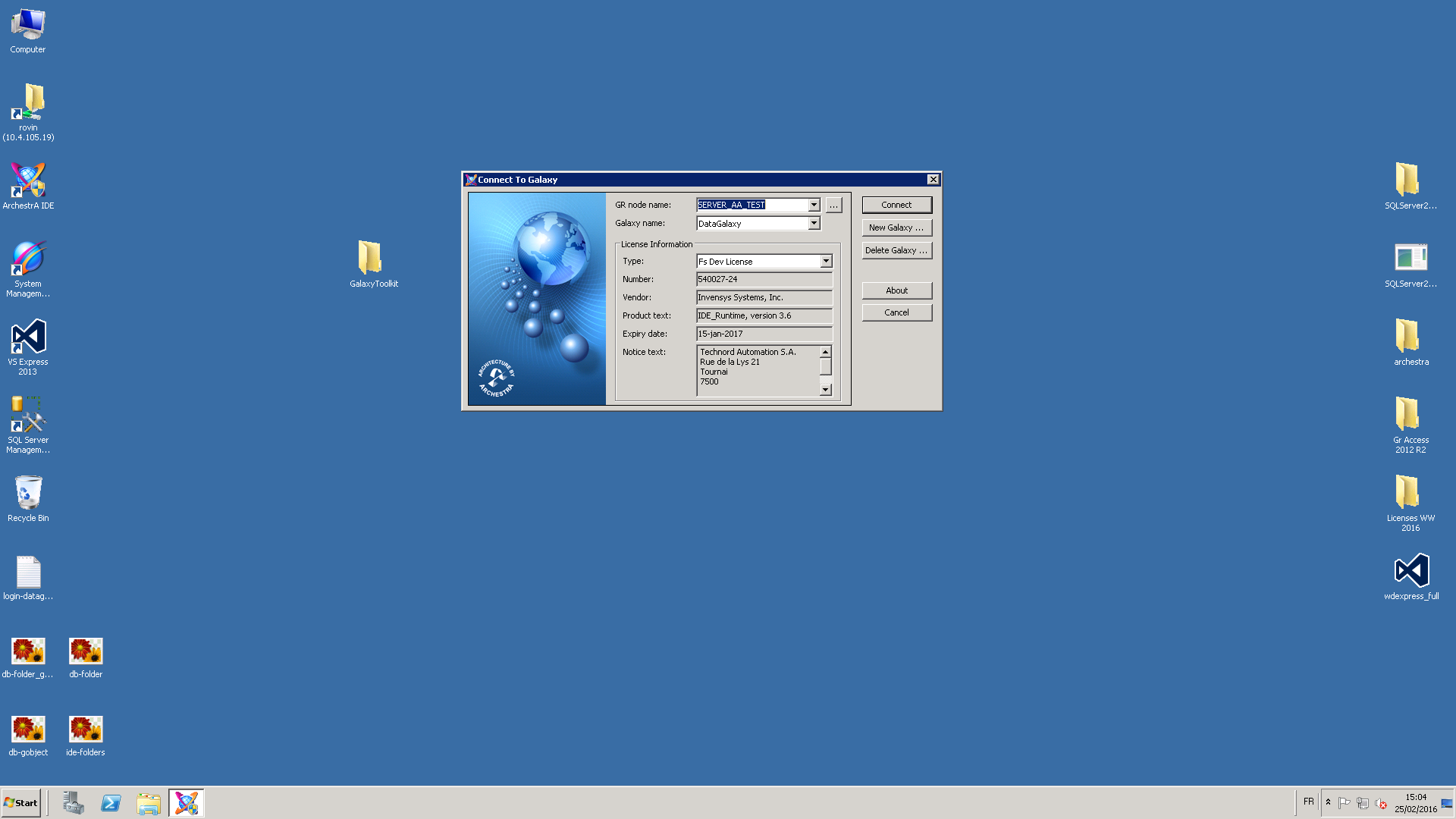
Par la suite, la classe de log a été recréée, afin de faciliter le débogage ultérieur. Les objets de données générique ont été séparés en deux classes : l’une contenant des objets structurels (tels que les arbres, nœuds d’arbres, …) et l’autre regroupant les objets de données (templates, instances, attributs, …). J’ai alors créé plusieurs types d’arbres : arbres simples et arbres indexés, pouvant tous deux gérer n’importe quel type de nœuds. Ainsi, l’index des templates pourra être directement construit dans l’arbre correspondant. Les méthodes de construction d’arbre du projet précédent ont été directement incluses dans ces nouvelles classes d’arbres. La généricité des deux types d’arbres implémentés me permettra de les réutiliser pour d’autres projets.

Plusieurs types de nœuds ont également été créés : nœuds simples (conteneurs basiques) et nœuds complexes (héritant de la classe de nœuds de l’interface graphique, de sorte qu’ils puissent être directement utilisés pour l’affichage, tout en contenant des données évoluées).

Côté données, les objets de templates et instances héritent à présent d’une classe d’objet commune. On évite ainsi la duplication de code. Une classe d’attributs a également été mise en place pour les UDA (et éventuels autres attributs ?). J’ai pris le temps d’observer les types d’ajouts de données possibles, afin de connaître la totalité des propriétés qu’il est possible de spécifier. Ceci m’a permis de compléter mes classes de templates, instances et attributs avec l’entièreté des possibilités. Des énumérations ont également été ajoutées pour les propriétés particulières, telles que les « data types » et les catégories.

Après avoir réécrit les classes de structures et de données génériques, j’ai pu recréer la classe d’accès SQL. J’ai repris une partie du code écrit précédemment, que j’ai adapté à la nouvelle organisation et aux nouveaux types d’objets génériques. Les requêtes SQL récupèrent à présent l’ensemble des propriétés des objets, et trient ces objets par nom. La classe d’accès GRAccess, elle, a été réécrite presque totalement, tout en étant grandement simplifiée. La sous-classe de conversion d’objets d’API en objets génériques a également été réécrite et simplifiée.

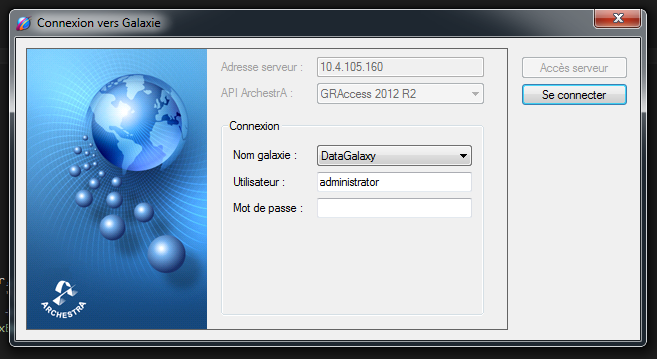
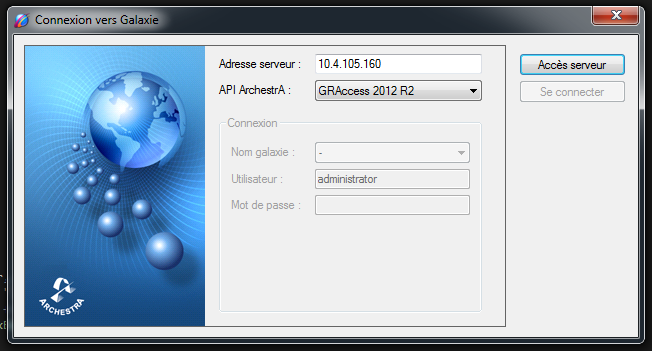
Des tests rapides ont été réalisés en mode console pour déboguer l’application. Il était temps de réaliser l’interface graphique de connexion avant d’aller plus loin. Le cahier des charges demandait une fenêtre de connexion du même style que celle de l’IDE Archestra (voir ci-dessous).



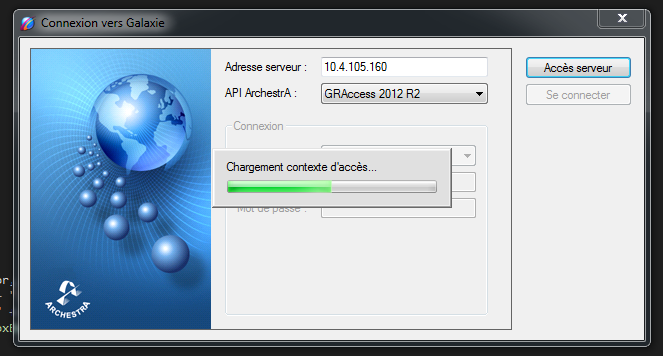
J’ai dès lors créé un formulaire dans la couche de présentation de mon application. J’ai également repris l’image de l’écran de connexion original (dont j’ai retiré une mention inappropriée).



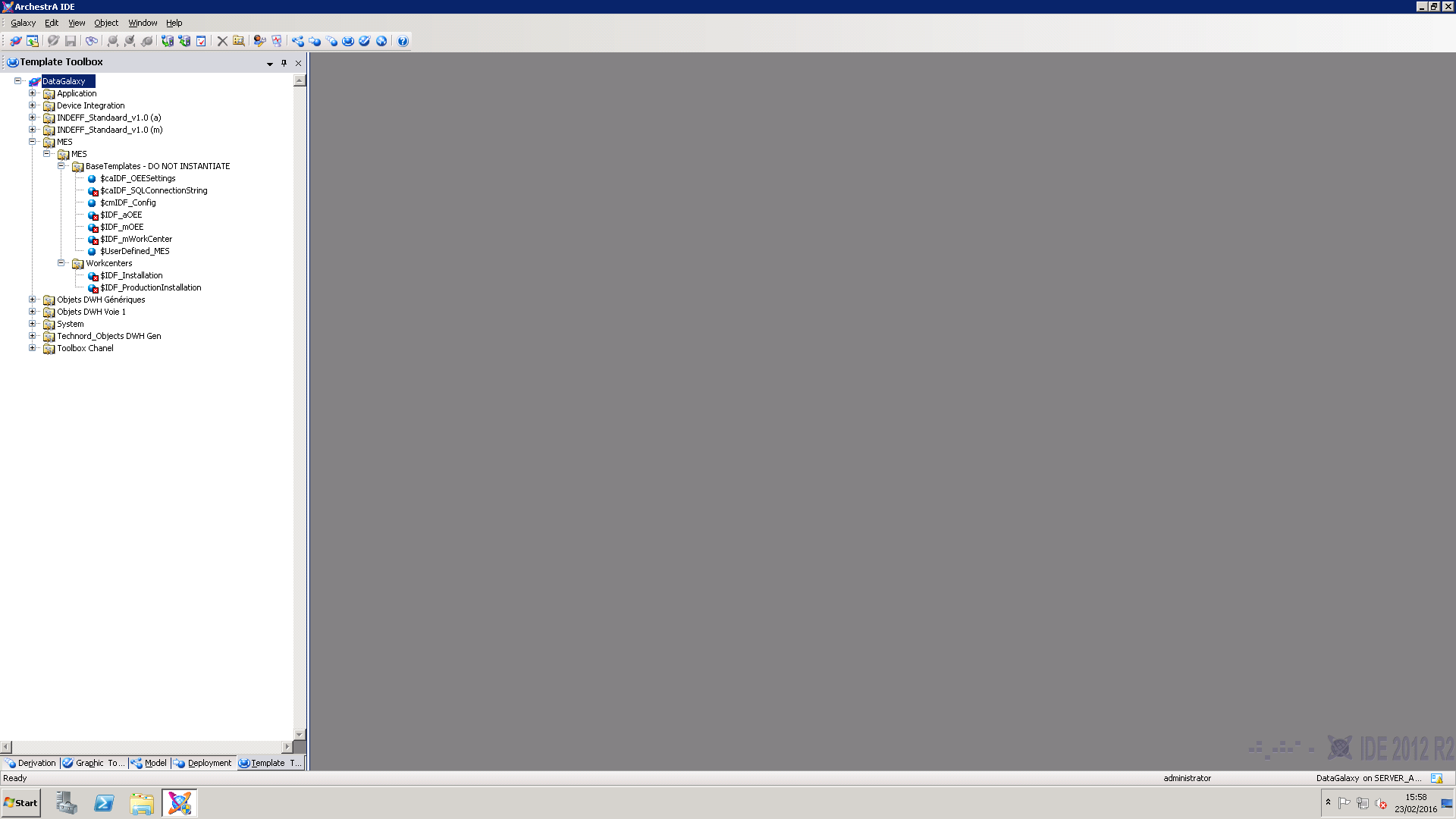
Ma version de l’écran de connexion est assez similaire à la version d’ArchestrA. Ses ingrédients restent les mêmes que dans l’application « prototype ». Selon l’étape de connexion, les éléments inappropriés sont grisés. L’utilisation est ainsi facilitée.



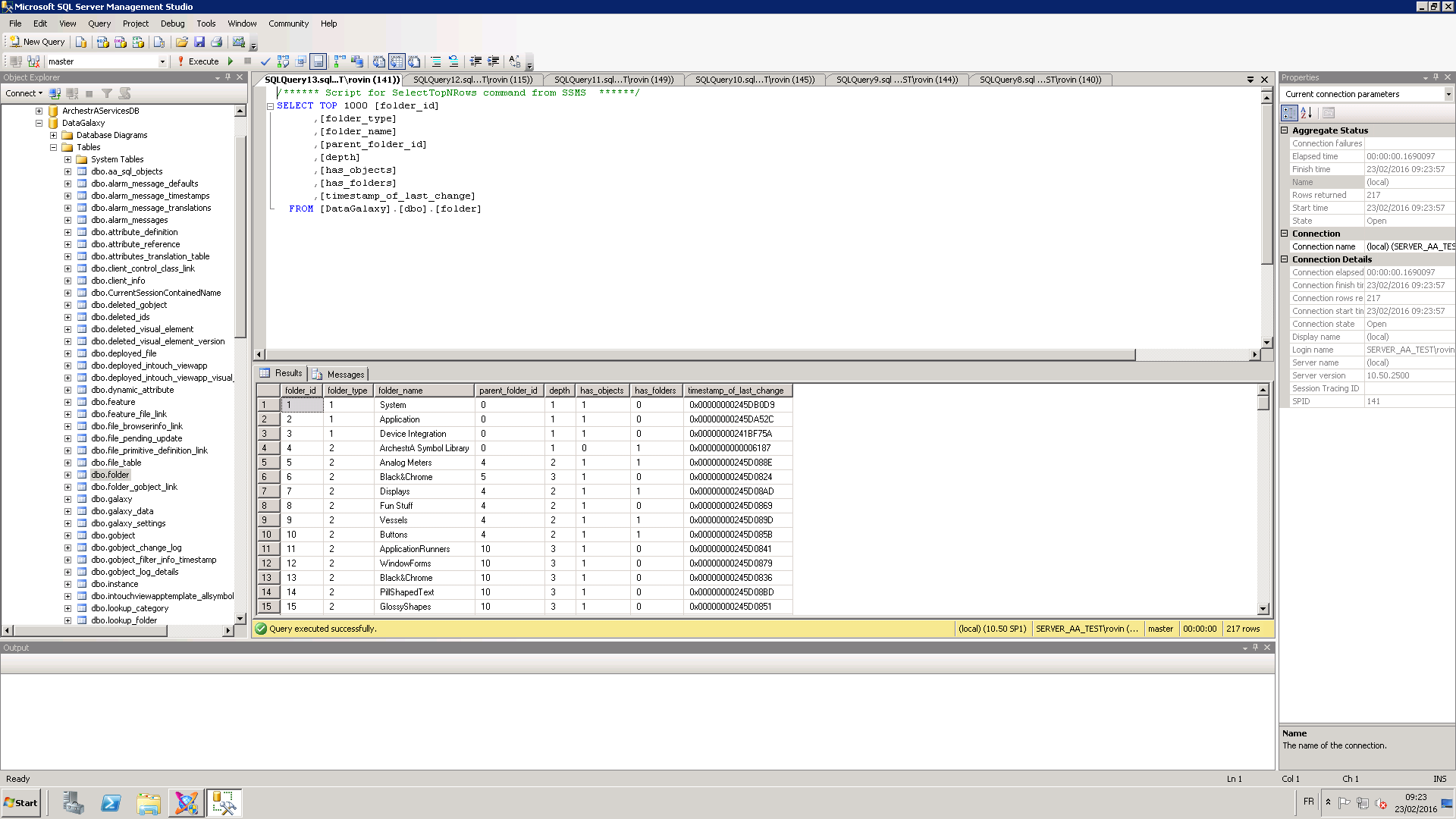
Le cahier des charges demandait de renseigner l’utilisateur au sujet de la progression. Un nouveau formulaire a donc été créé pour mettre en place une boîte de progression. J’ai alors créé un singleton permettant de gérer une liste de plusieurs boîtes de progression, et d’y accéder depuis n’importe quel endroit du code. Elles sont instanciées au lancement de l’application. Ainsi, l’interface graphique se chargerait d’afficher et masquer les boîtes de progression, alors que la couche d’accès se chargerait de leur envoyer des événements. Ce dernier point a nécessité la création d’une classe d’événements personnalisés, afin de transmettre un message et un pourcentage de progression.

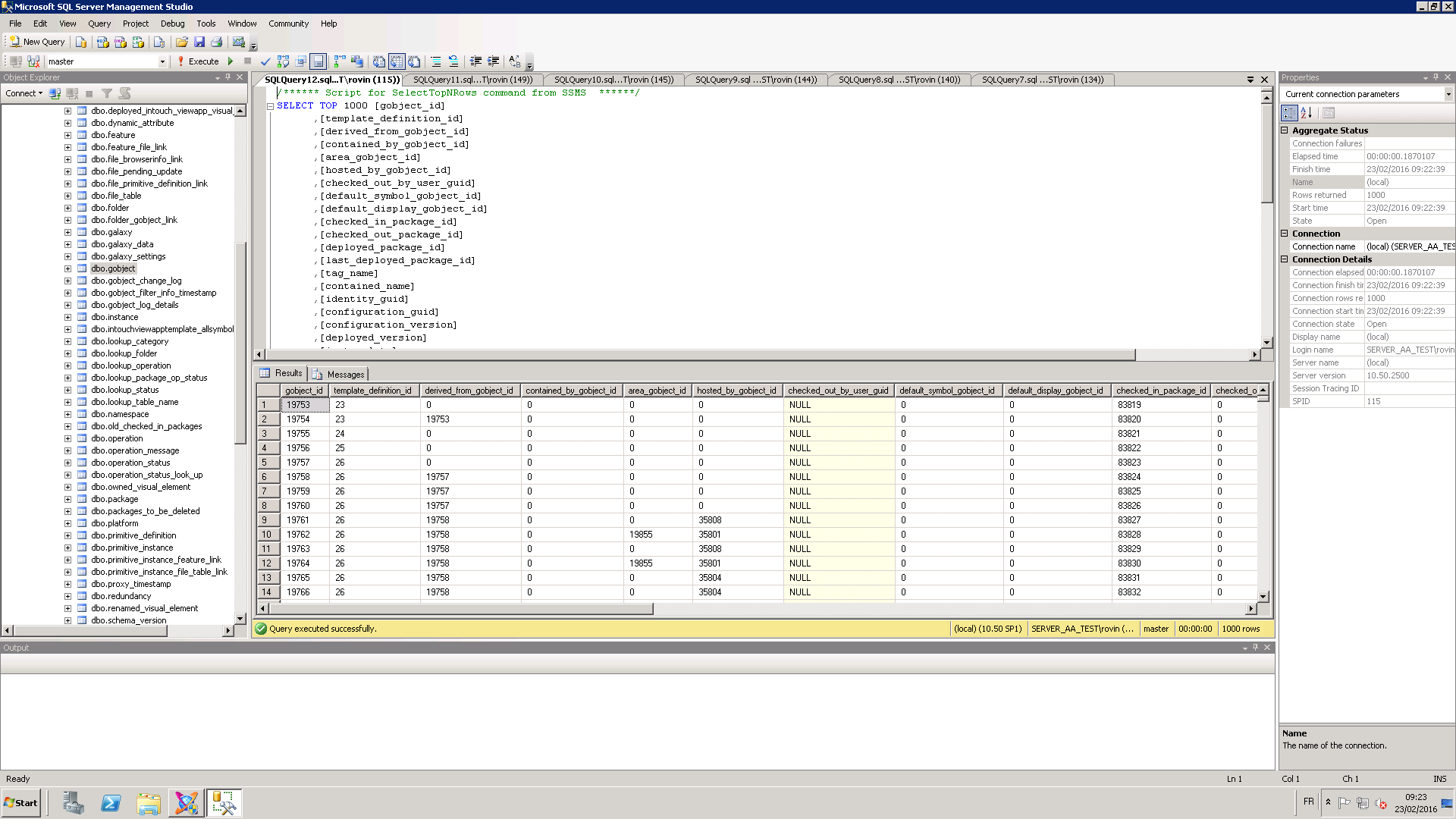
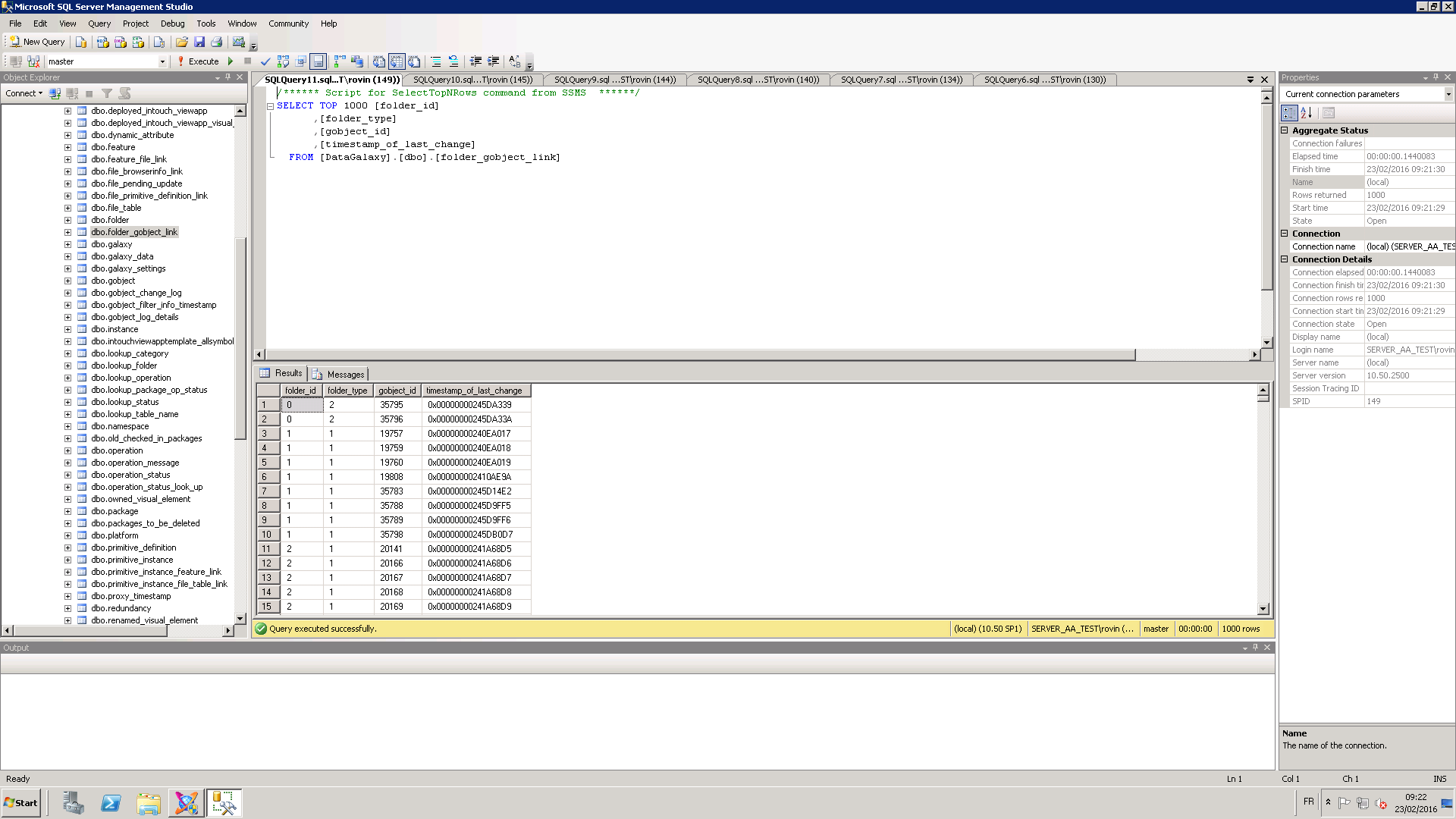


J’ai réorganisé le démarrage de l’application, afin d’utiliser un contexte d’application gérant les formulaires (plutôt que de placer le code de gestion de formulaires à l’intérieur d’un formulaire, ce qui était le cas du « prototype »). La prochaine étape, après la connexion, était la création des arbres de templates. Il ne s’agit plus cette fois de créer un seul arbre hiérarchique, mais bien deux arbres : le premier est hiérarchique, et le second se base sur un classement par dossiers, comme le montre l’image ci-dessous.



Une nouvelle fois, j’ai parcouru le contenu de la base de données ArchestrA. Malheureusement, aucune « vue » ne permettait de récupérer directement les dossiers. Il m’a donc été nécessaire d’analyser les différentes tables : celle des dossiers, celle des objets, ainsi qu’une table d’association. Il a été assez aisé d’écrire une requête utilisant des jointures pour récupérer les noms des dossiers et les noms des objets associés à chaque dossier.





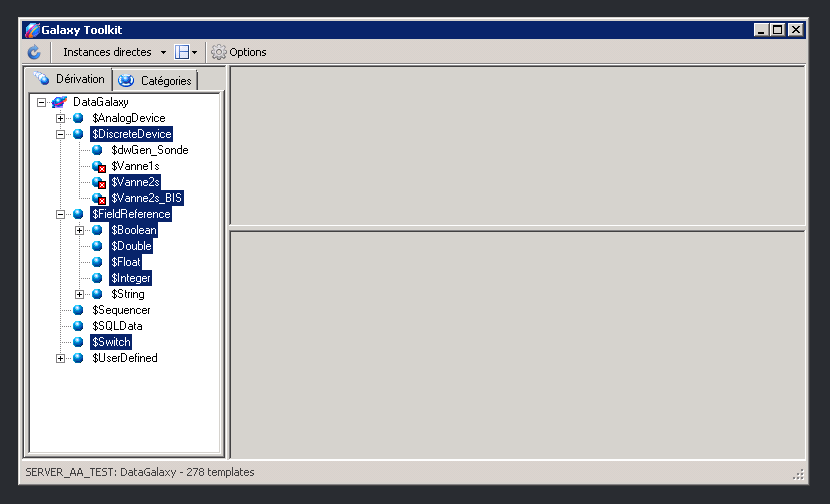
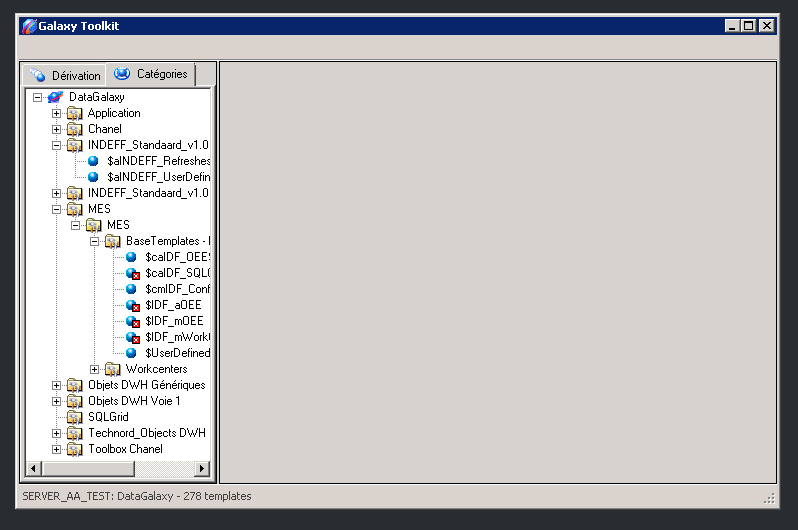
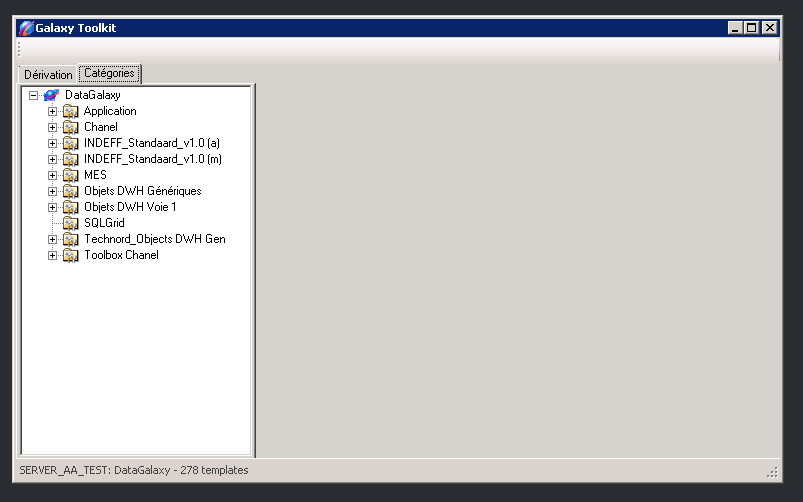
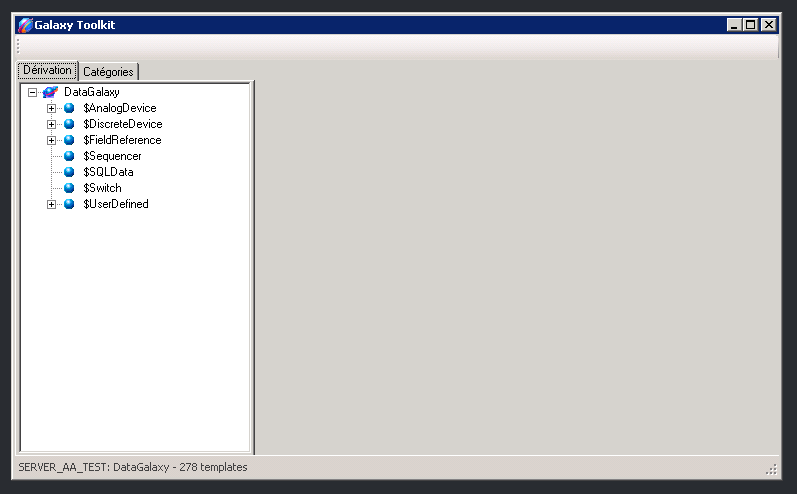
Certains dossiers « systèmes » devaient être ignorés (avec les moteurs et les vues). Il a donc fallu ajouter des exclusions dans la requête. Cela n’excluait toutefois que les dossiers non souhaités et leurs enfants directs. Une fonction récursive a donc été écrite pour éliminer les dossiers dont le parent a déjà été éliminé.

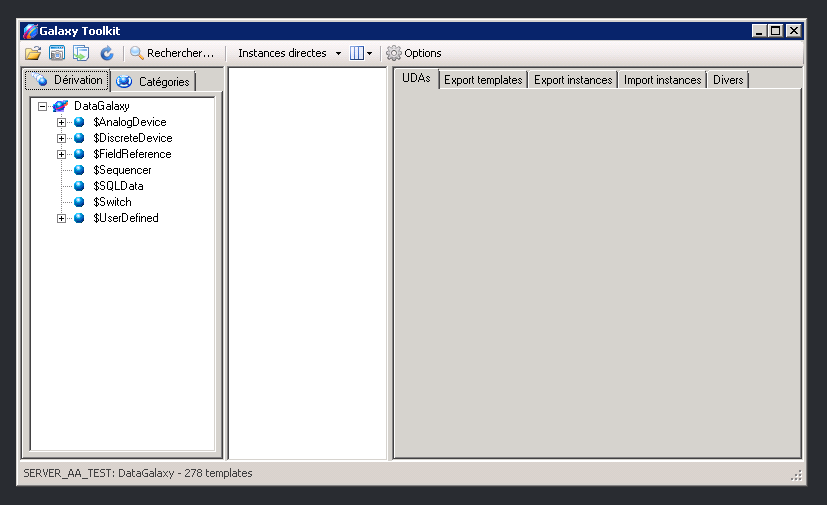
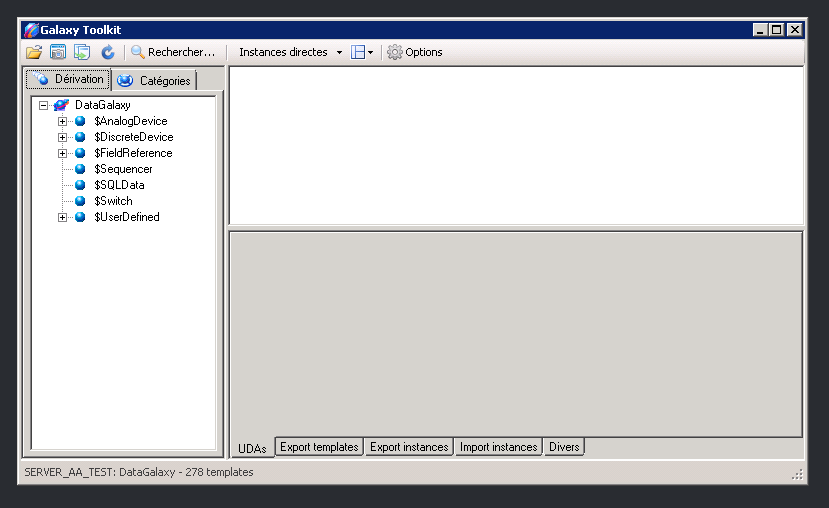
J’ai également ajouté des requêtes SQL de récupération de statistiques : nombre de templates et d’instances, … qui seront affichées dans une barre de statut, comme demandé.

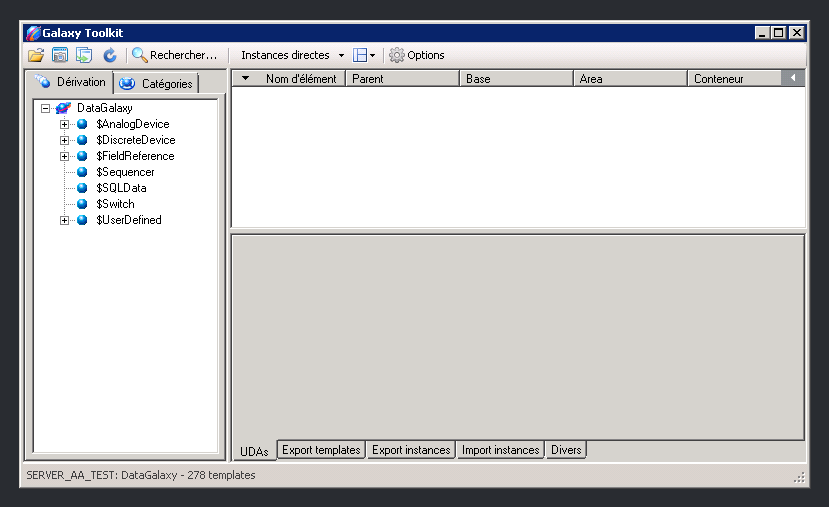
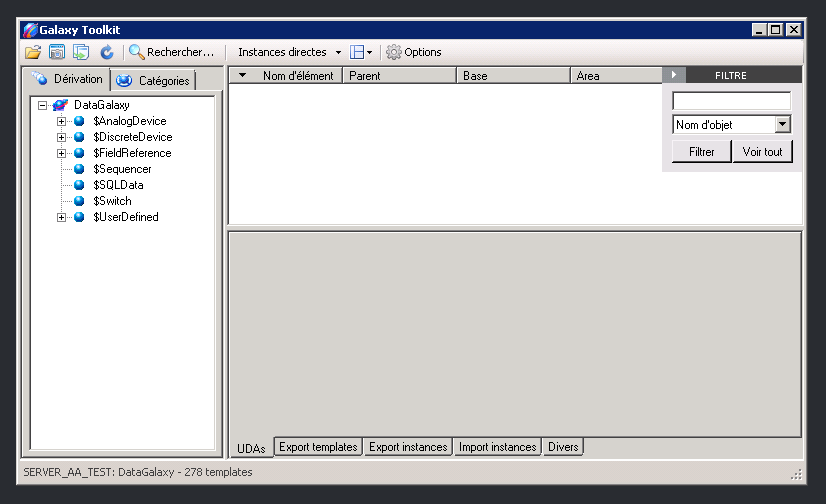
Après avoir récupéré les noms des dossiers et des éléments contenus dans chaque dossier, je me suis basé sur ces derniers pour créer un second arbre, en récupérant les éléments (sur base de leur nom) dans l’index du premier arbre (l’arbre hiérarchique). J’ai ainsi réalisé la création d’un arbre hiérarchique et d’un arbre par dossiers. Ces deux arbres seront accessibles dans l’interface graphique via deux onglets.

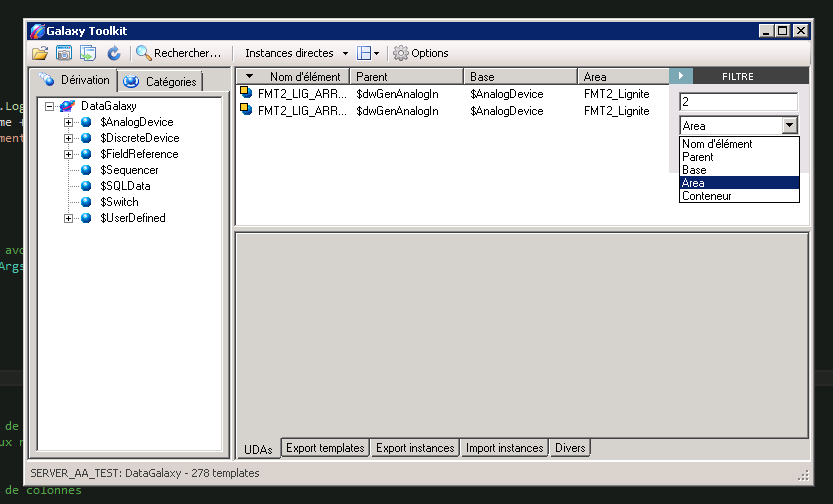
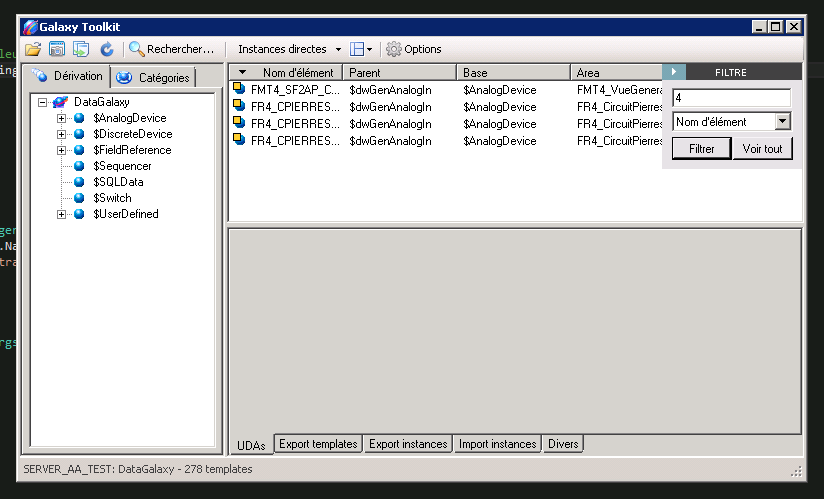
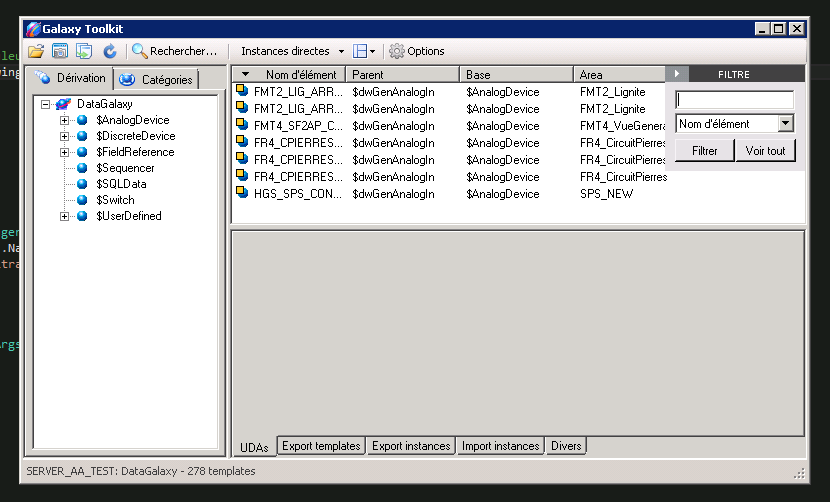
J’ai également ajouté des fonctions d’obtention d’informations générales (nom de la machine, nom de la galaxie actuelle) et de statistiques (utilisant les requêtes de la classe gérant le SQL). J’ai enfin adapté la fonction du « prototype », qui permettait d’obtenir les instances d’un template. Elle permet à présent d’obtenir les instances d’une liste de templates. J’ai également ajouté une fonction permettant de lister l’ensemble des templates descendants d’un template. En combinant les deux fonctions, il devient ainsi possible d’obtenir la liste des instances d’un ou plusieurs templates et leurs descendants.

Pour terminer, j’ai souhaité mettre en place un système de threads, permettant d’effectuer les requêtes de manière asynchrone, sans paralyser l’interface graphique. Après de nombreuses recherches, il s’est avéré que la solution la plus avantageuse est d’utiliser des objets C# appelés « tasks » et gérant automatiquement un pool de threads. Ces objets, s’ils permettent de faire abstraction du mécanisme des threads, sont toutefois extrêmement compliqués à manier (du moins lorsqu’on souhaite en tirer profit, par exemple en récupérant des objets d’un thread à l’autre). Je n’ai donc pas trouvé le temps d’implémenter leur utilisation. Cette problématique devrait m’occuper une partie de la semaine prochaine

Semaine 29/02 – Mécanisme de requêtes asynchrones ;  
 – Construction arbres (dérivation et catégories) + onglets  
 – Arbre multi-sélection + liste multi-colonnes triable  
 – Menus et icônes + splitters + layout + rafraîchissement  
 – Filtrage de liste  








Semaine 07/03 – Réorganisation projet ;

– Requêtes asynchrones interruptibles et remplaçables ;

– Boite de progression avec titre, étapes, annulation ;

– Récupération asynchrone d’instances selon sélection et type de descendants à charger + optimisation vitesse ;

– Barre de statut selon templates, instances et sélection ;

– Boite de recherche asynchrone + triable + filtres ;

– Historique de recherche configurable ;

– Navigation vers sélection de recherche (arbre/liste) ;

– Synchronisation de sélection des arbres ;

– UI de liste d’export et d’affichage d’UDAs

