**Rapports de stage – Technord**

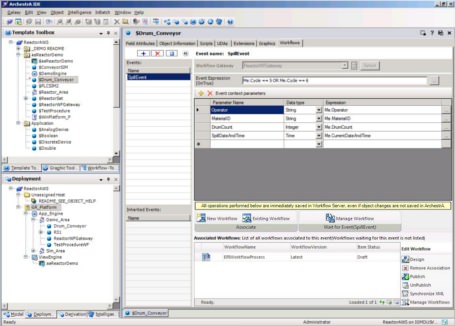
Semaine 01/02 – Installation, doc, découverte ArchestrA ;   
 – GRAccess, datacontext, récupérer galaxies, templates.

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.

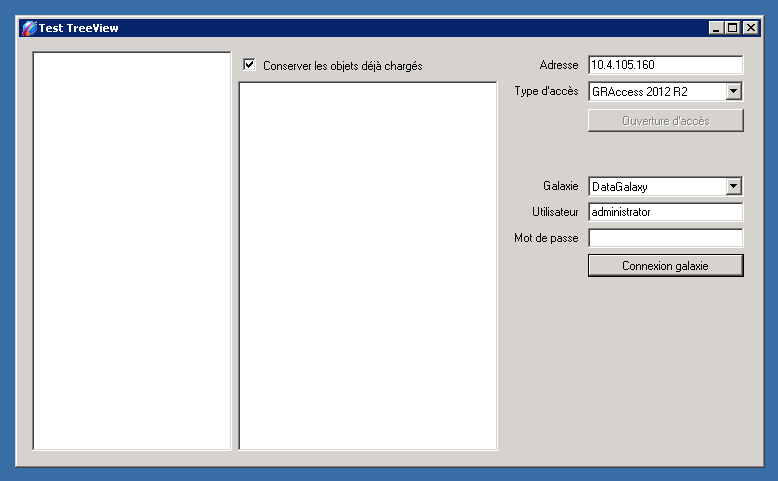
A ce stade, aucun cahier des charges concret ne m’avait été communiqué. Avant de commencer à développer, il a d’abord fallu commencer par découvrir ArchestrA et ses principes. Une banque de données (« galaxie ») m’a été fournie pour ArchestrA, afin de pouvoir regarder par moi-même comment utiliser ce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

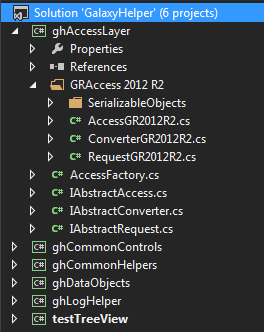
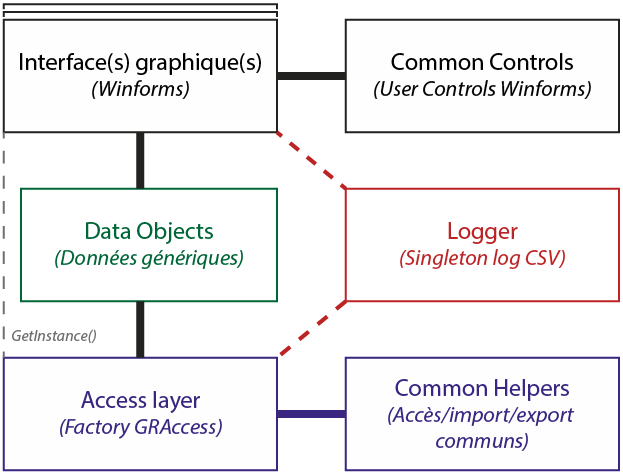
J’ai choisi de répartir la structure du programme en plusieurs projets, afin d’en faciliter le dévelop-pement, la maintenance et l’évolution. L’application est développée en C#.

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.

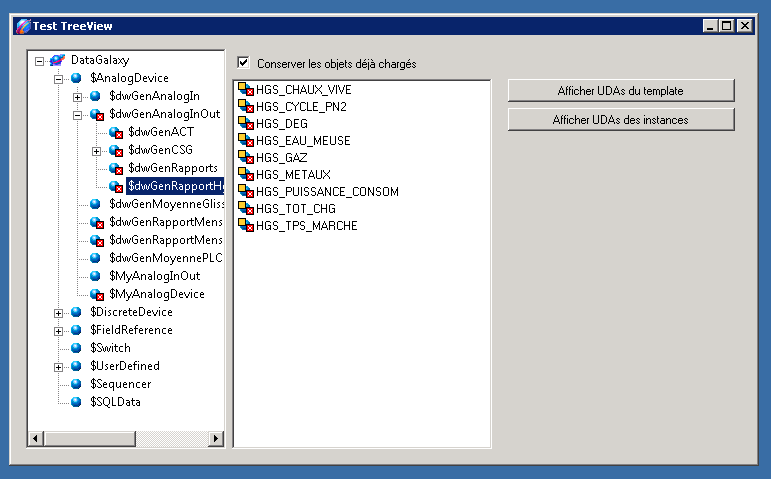
 

La première tâche à réaliser, en attendant de recevoir un cahier des charges complet, était la récupération de la hiérarchie des templates d’une galaxie (sous forme d’arbre), ainsi que la récupé-ration de la liste des instances, pour un template donné. Plusieurs classes ont été créées dans le projet « 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).

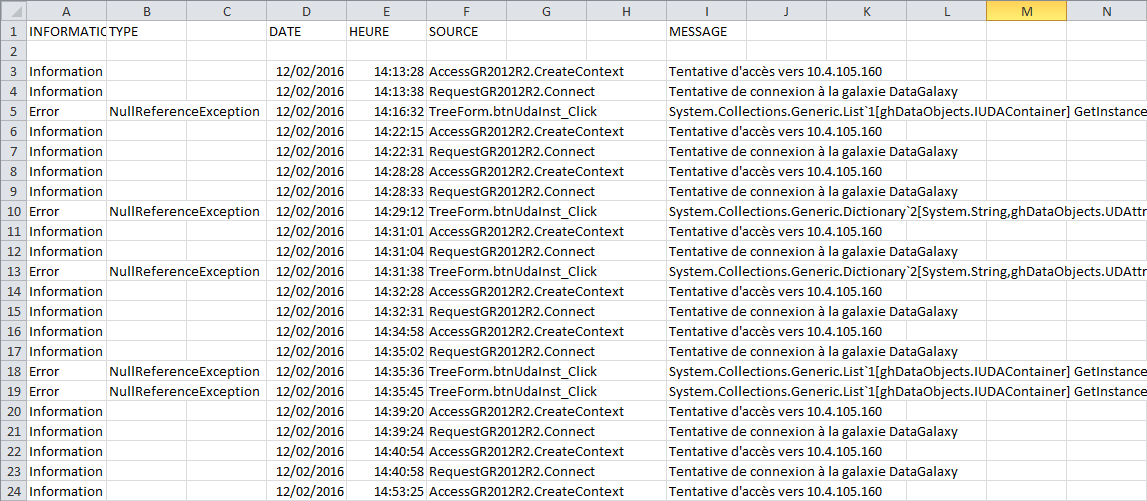
Ce système de stockage des éléments présente un gain de temps non négligeable. Toutefois, il pose un nouveau problème : les éléments stockés gardent l’état qu’ils avaient lorsqu’ils ont été chargés. Ainsi, si une autre personne modifiait un objet ArchestrA entre-temps, la version stockée ne serait plus 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>

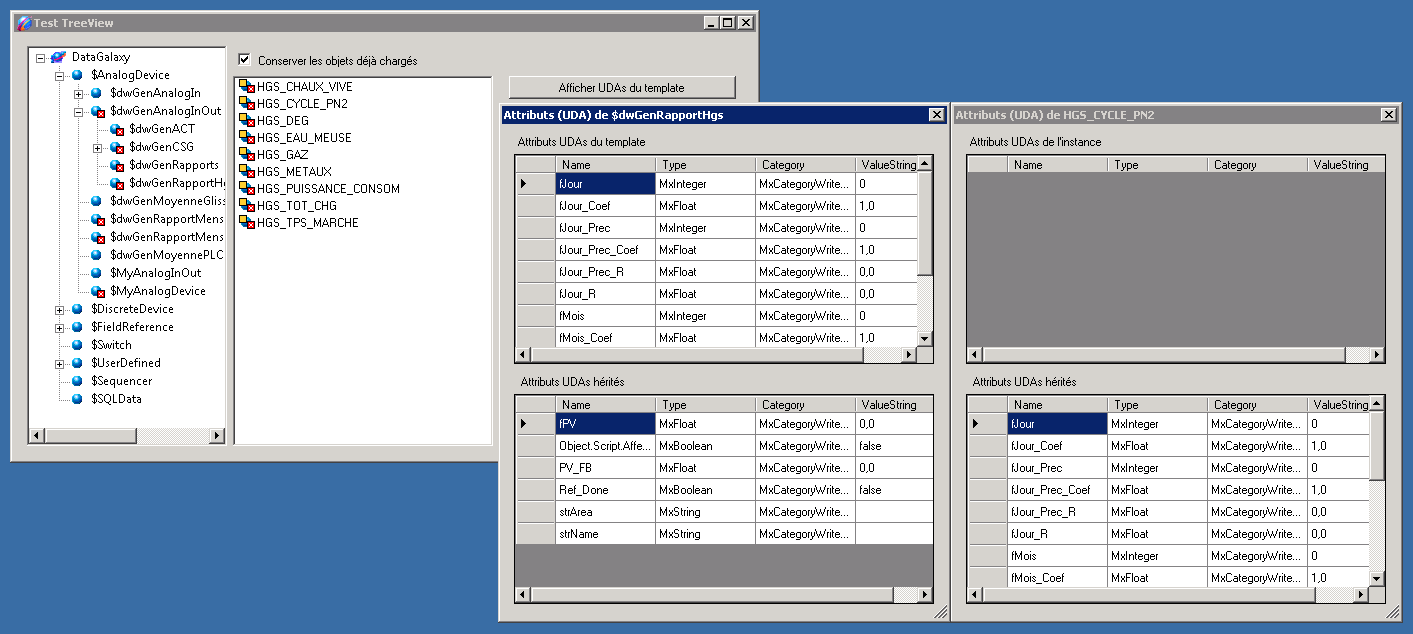
Ces deux listes permettent ainsi de connaître les noms, les types et les catégories des attributs UDAs propres et hérités. Elles ne fournissent pas la valeur de ces attributs. Ces listes peuvent être vues comme des sortes de références vers une série d’autres attributs. Pour chaque attribut mentionné 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 – Cahier des charges, analyse, planning ;   
 – Recherche de templates/instances, export liste, sélection.

…



