|  |
| --- |
|  |
| Application Type |
| Modélisation UML et Patron de programmation pour applications orientées objets |
|  |
| **Thomas Auguey** |
| **20/08/2015** |

|  |
| --- |
| Ce guide définit une méthode de développement applicable aux langages de programmation orienté-objet et basé sur le patron Modèle-Vue-Contrôleur. |

SOMMAIRE

[Introduction 4](#_Toc433295935)

[Squelette d’application 5](#_Toc433295936)

[Application 7](#_Toc433295937)

[Contrôleur 7](#_Toc433295938)

[Model 7](#_Toc433295939)

[Vue 7](#_Toc433295940)

[Evénement 8](#_Toc433295941)

[Entité 9](#_Toc433295942)

[Etat 11](#_Toc433295943)

[Donné 11](#_Toc433295944)

[Préparation du projet 12](#_Toc433295945)

[Répertoire de travail 12](#_Toc433295946)

[Suivi de version 12](#_Toc433295947)

[Modélisation 13](#_Toc433295948)

[Entité 13](#_Toc433295949)

[Association 13](#_Toc433295950)

[Intégration 15](#_Toc433295951)

[Types de données 15](#_Toc433295952)

[Sérialisation 15](#_Toc433295953)

[Base de données 15](#_Toc433295954)

[Evénements 16](#_Toc433295955)

[Implémentations 18](#_Toc433295956)

[C++ (wxWidget) 19](#_Toc433295957)

[C# (WPF) 24](#_Toc433295958)

[C# (WinForms) 29](#_Toc433295959)

[Visionner 3 30](#_Toc433295960)

[Génération avec PowerDesigner 31](#_Toc433295961)

[Type de projet 31](#_Toc433295962)

[Templates 31](#_Toc433295963)

[Stéréotypes 31](#_Toc433295964)

[Options de génération 32](#_Toc433295965)

[Persistance 32](#_Toc433295966)

[Projet type 35](#_Toc433295967)

[Présentation 35](#_Toc433295968)

[Cycle de vie 35](#_Toc433295969)

[Définition 35](#_Toc433295970)

[UML 36](#_Toc433295971)

[Références 39](#_Toc433295972)

# Introduction

Le but de ce document est de définir une méthode de développement générique applicable au plus grand nombre de projets informatiques mettant en relation les bases de données, les interfaces utilisateurs et l’architecture logiciel.

Cette méthode n’a pas pour prétention de fournir une architecture idéale mais plutôt les bases d’une application robuste et facile à développer.

Le programmeur se concentre sur :

* Le model de données (UML)
* La logique d’implémentation métier

L’implémentation définit:

* L’interface avec les bases de données SQL
* Les domaines de valeurs
* Les entités :
  + Clé primaire
  + Sérialisation des données
  + Validation des champs
  + Persistance des données
* Les événements :
  + Les mécanismes de traitement des événements
  + Historisation des états
  + Interaction Interface Utilisateur / Model de données
    - Create, Update, Delete, Copy, Paste, Change, …
* Les modèles de données :
  + Maintien d’une liste de références
  + Gestion de model multiples (MDI, modèles concurrents, asynchronisme)

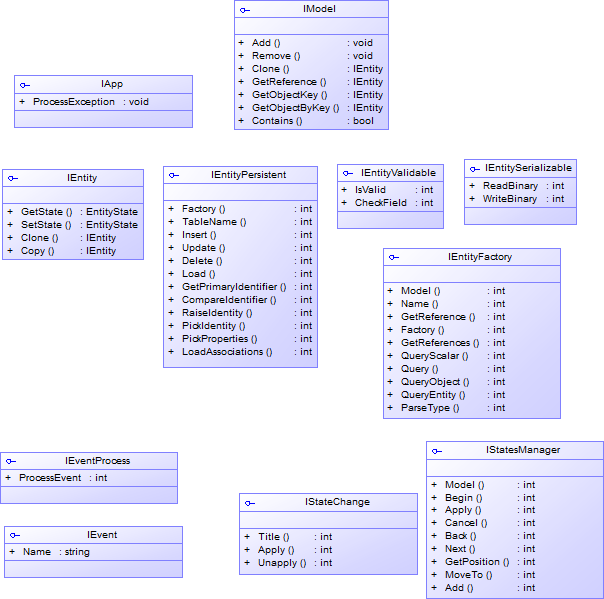
# Squelette d’application

L’implémentation repose sur un squelette de classes abstraites (interfaces). Les langages de programmation concernés sont donc obligatoirement de type Orienté-Objet.

Ce squelette permet de définir la structure de base de l’implémentation sur laquelle est basée cette méthode, certaines interfaces sont indispensables et d’autres facultatives. Ce sont les éléments les plus communs à une application (événements, entités, application, contrôleur, model, ….)

Suivant l’implémentation finale du programmeur certaines de ces interfaces seront remplacées par des classes du Framework ou des éléments du langage utilisé. Ce squelette a uniquement pour but de définir l’ensemble des interfaces nécessaires à une architecture de base.

Cette section présente les interfaces et leurs méthodes dont le schéma ci-dessous en est une représentation UML :

****

## Application

Point d'entrée du programme et données globales dont les modèles, vues et contrôleurs

### Interface IApp

|  |  |
| --- | --- |
| **ProcessException** | Traite une exception utilisateur retournée par l’application |

## Contrôleur

Classe d'interaction entre la vue et le model

### Interface IControler

…

## Model

Manage les données et implémente la logique métier

### Interface IModel

Model de données.

|  |  |
| --- | --- |
| **CreateEntity** | Crée une entité.  *Cette méthode n’implémente pas la logique métier.* |
| **Remove** | Supprime une entité des références.  *Cette méthode n’implémente pas la logique métier.* |
| **Add** | Ajoute une entité aux références.  *Cette méthode n’implémente pas la logique métier.* |
| **Clone** | Clone le model et les entités qu'il contient.  *Cette méthode n’implémente pas la logique métier.* |
| **GetReference** | Obtient une référence de l'objet, si celui-ci n'existe pas il est automatiquement ajouté à la liste.  *Cette méthode n’implémente pas la logique métier.* |
| **GetObjectKey** | Obtient la clé d'un objet |
| **GetObjectByKey** | Obtient un objet par sa clé |
| **Contains** | Test si une référence de l’objet existe dans le model. |
| **Update** | Met à jour du model.  *Cette méthode implémente la logique métier (validation, persistance, …).* |
| **Create** | Crée une entité du model.  *Cette méthode implémente la logique métier (validation, persistance, …).* |
| **Delete** | Supprime une entité du model.  *Cette méthode implémente la logique métier (validation, persistance, …).* |

## Vue

Classe d'interaction avec l’utilisateur

### Interface IView

…

## Evénement

Les événements participes à la communication liée aux changements d’état interne et externe à l’application. L’un des plus courant est par exemple d’avertir une vue d’un changement d’état dans le model.

Les événements sont définit par le programmeur selon les besoins de sont application. Certains plus courants sont implémenté par défaut (ex : changement d’une entité)

### Interface IEvent

Evénement de base.

|  |  |
| --- | --- |
| **Name** | Nom de l’événement |

### Interface IEventProcess

Interface de traitement des événements.

|  |  |
| --- | --- |
| **ProcessEvent** | Traite l’événement |

### Interface IEventManager

Manage une collection d’objets notifiables.

|  |  |
| --- | --- |
| **NotifyRegister** | Abonne un objet aux notifications |
| **NotifyUnRegister** | Désabonne un objet notifié |
| **NotifyEvent** | Notifie un événement |

## Entité

Une entité est une composante du model de données, elle représente une structure composée de variables élémentaires.

### Domaine EntityState

Constantes des états possibles d’une entité.

|  |  |
| --- | --- |
| **Unmodified** | Non-Modifiée  Indique une entité existante dont les données ne sont pas modifiées |
| **Modified** | Modifiée  Indique une entité modifiée mais pas encore sauvegardée |
| **Added** | Ajouté  Indique une entité nouvellement ajoutée mais pas encore sauvegardée |
| **Deleted** | Supprimée  Indique une entité supprimée qui n’est plus utilisée |

### Interface IEntity

Entité de base.

|  |  |
| --- | --- |
| **EntityState** | Etat actuel de l’entité, cette valeur est gérée par le programmeur pour identifier l’état actuel de l’entité et donc les opérations à lui appliquer |
| **Model** | Model parent de l’entité |
| **EntityName** | Nom de l’entité |
| **Clone** | Duplique les propriétés de l’entité dans une nouvelle instance |
| **Copy** | Copie les propriétés d’une entité dans une autre |

### Interface IEntityFactory

Classe d’interface entre les entités et une base de données SQL.

|  |  |
| --- | --- |
| **Model** | Obtient le model de données contenant les références des entités |
| **Name** | Obtient le nom du fournisseur d'accès |
| **GetReference** | Obtient une référence d'entité |
| **Factory** | Enumère les références en base de données |
| **GetReferences** | Obtient une liste des références du model de données |
| **QueryScalar** | Exécute une requête SQL et retourne le premier champ de la première colonne trouvée |
| **Query** | Exécute une requête SQL sans retourner de résultat |
| **QueryObject** | Initialise les propriétés d'un objet générique avec les données d'une requête SQL |
| **QueryEntity** | Retourne une entité de référence initialisée avec les données d'une requête SQL |
| **ParseType** | Converti un type natif en type SQL |

### Interface IEntityPersistent

Entité dont les membres sont hébergés dans une base de données SQL.

|  |  |
| --- | --- |
| **Factory** | Obtient l'instance d'interface avec la base de données |
| **TableName** | Nom de la table correspondante à l’entité |
| **Insert** | Insert l'entité en base de données |
| **Update** | Actualise les champs en base de données |
| **Delete** | Supprime l'entité en base de données |
| **Load** | Charge les relations d'entités |
| **GetPrimaryIdentifier** | Obtient l’identificateur primaire |
| **CompareIdentifier** | Compare l'identifiant primaire avec une autre entité |
| **RaiseIdentity** | Rase l'identifiant primaire |
| **PickIdentity** | Extrait l'identifiant primaire d'un résultat de requête |
| **PickProperties** | Extrait les propriétés d'un résultat de requête |
| **LoadAssociations** | Charge une relation d'entités |

### Interface IEntitySerializable

Entité dont les membres sont exportables dans un flux de données (par exemple un fichier).

|  |  |
| --- | --- |
| **ReadBinary** | Lit les propriétés de l'entité depuis un flux de données binaire |
| **WriteBinary** | Ecrit les propriétés de l'entité depuis un flux de données binaire |

### Interface IEntityValidable

Entité dont les membres sont soumis à une validation des données.

|  |  |
| --- | --- |
| **Validate** | Test la validité de tous les champs |
| **CheckField** | Test la validité d'un champ |

## Etat

Les états permettent d’identifier les changements apportés à une application (changement de données dans une entité, exécution d’une commande, …)

Un changement d’état est généralement initié par une action de l’utilisateur (clique, saisie, copier/coller, …) ou une action du système (transformation, importation,..).

Permettre de contrôler ces états donne la possibilité de :

* Naviguer dans l’historique des modifications
* Sauvegarder des états avant de les appliquer au model persistant (mémoire, BDD, Fichiers,…)

### Interface IStateChange

Etat ou action. Il peut s’agir d’une commande, de l’état d’une entité ou d’une donnée quelconque définit par le programmeur.

|  |  |
| --- | --- |
| **Title** | Titre |
| **Apply** | Applique la modification dans le model de données |
| **Unapply** | Annule la modification dans model de données |

### Interface IStatesManager

Gestionnaire des états. Consiste en une liste ordonnée d’états. Le gestionnaire permet d’appliquer ou annuler les modifications apportées à un modèle de données.

|  |  |
| --- | --- |
| **Model** | Obtient le model de données |
| **Begin** | Débute l’enregistrement des modifications sur un model |
| **Apply** | Applique toutes les modifications apportées depuis l’appel à Begin |
| **Cancel** | Annule toutes les modifications apportées depuis l’appel à Begin |
| **Back** | Annule la modification et recule le curseur de modification |
| **Next** | Applique la modification et avance le curseur de modification |
| **GetPosition** | Obtient la position du curseur de modification |
| **MoveTo** | Déplace le curseur de modification |
| **Add** | Ajoute un changement à la liste |

## Donné

Données

### Interface ITypedData

Représente une donnée typée

|  |  |
| --- | --- |
| **ToString** | Convertie le type en chaine de caractères |
| **FromString** | Convertie la chaine de caractères en type |
| **IsInitialised** | Indique si la valeur est initialisée |
| **GetData** | Obtient la donnée |
| **GetType** | Obtient le type de la donnée |

# Préparation du projet

Cette section décrit les étapes initiales de préparation du cycle de vie de l’application.

## Répertoire de travail

Le projet peux être créé depuis une nouvelle base ou copier depuis un exemple existant.

**Création d’un nouveau projet :**

1. Créer un dossier pour votre répertoire de travail
2. Créer le projet
3. …

**Reprise d’un projet minimaliste :**

1. Copier l’une des applications d’exemple disponibles dans le dossier « App » puis coller le dossier dans votre répertoire de travail
2. Ouvrir le projet
3. Renommer le projet
4. …

## Suivi de version

Le suivi de version permettra de borné les évolutions du programme et naviguer dans l’historique des modifications. Dans notre exemple le logiciel GIT sera utilisé.

**Création du projet :**

1. Ouvrir la console de commande
2. Se placer dans le répertoire de stockage
3. Créer un nouveau dépôt

# Modélisation

Cette section décrit les conventions utilisées pour modéliser les entités de votre programme dans un diagramme de classes UML.

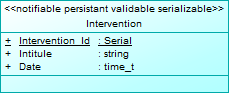
## Entité

On définit une entité pour créer une classe de données.

### Stéréotypes

|  |  |
| --- | --- |
| Mot clé | Description |
| persistant | Implémente les méthodes d’accès à la base de données.  (Interface **IEntityPersistent)** |
| serializable | Implémente les méthodes de conversions binaire et XML.  (Interface **IEntitySerializable)** |
| notifiable | Implémente les méthodes de notification des champs.  (Interface **IEntityNotifiable)** |
| validable | Implémente les méthodes de validation des champs.  (Interface **IEntityValidable)** |

Exemple de stéréotypes sur une entité :



## Association

La définition du type d’association influence le code en sortie.

### Composition

https://upload.wikimedia.org/wikipedia/commons/thumb/1/12/UML_Rel_composition.jpg/75px-UML_Rel_composition.jpg

On définit une relation de composition entre 2 entités lorsque qu’une dépendance est présente. Par exemple : **Employé** à une relation de composition avec **Adresse** car elle n’a pas lieu d’être si **Employé** n’existe plus.

Affecte:

* La sérialisation, les entités ayant une relation de composition sont également sérialisé avec l’objet parent.
* L’instance, les entités ayant une relation de composition sont également supprimés avec l’objet parent.

### Agrégation

https://upload.wikimedia.org/wikipedia/commons/thumb/5/50/UML_Rel_aggregation.jpg/75px-UML_Rel_aggregation.jpg

On définit une relation d’agrégation entre 2 entités lorsque qu’aucune dépendance n’est présente. Par exemple : **Employé** à une relation d’agrégation avec **Société** car celle-ci persiste même si Employé n’existe plus.

# Intégration

Cette section d’écrit les pratiques de programmation.

## Types de données

La plupart des types de données

## Sérialisation

La sérialisation définit la dépendance des données vis-à-vis de la classe parente qui l’implémente.

### Composition

* Les instances de données enfants sont libérées avec la suppression la classe parent

### Association

* Les instances de données enfants sont conservées après la suppression de la classe parent

## Base de données

### Connexions

Le programmeur à généralement besoin d’accéder de façon concurrente à une base de données, dans cette situation nous pouvons dissocier 2 cas :

1. Travailler sur une base de données avec plusieurs modèles de données en mémoire. Dans ce cas nous utiliserons 2 instances de base **IEntityFactory** implémentant chacune une liste de référence des entités. Ainsi une entité modifiée dans un model n’affectera pas l’autre empêchant toute collisions dans les listes de référence.
2. Travailler sur une base de données avec un modèle de données commun en mémoire. Dans ce cas nous utiliserons 1 instance de base **IEntityFactory** avec plusieurs connexions**.** Ainsi une entité modifié dans un model affectera l’autre et empêchant toute duplication dans les listes de référence.

## Evénements

### Copier une entité

* Sérialiser les données de l’entité dans un format XML
* Copier le contenu XML dans le presse-papier

### Coller une entité

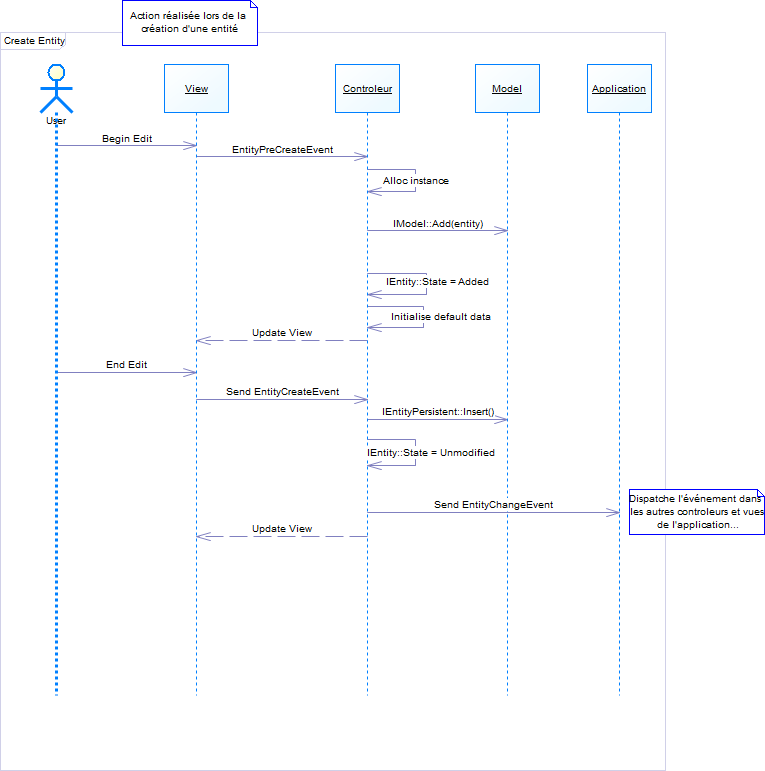
Le mécanisme permettant de coller une ou plusieurs entités est composé de plusieurs opérations :

1. Récupéré le texte XML depuis le presse-papier
2. Enumérer les nœuds des éléments XML
3. Pour chaque nœud
   1. Alloue une instance de l’entité (**IModel::CreateEntity**)
   2. Dé-sérialiser les données dans l’instance allouée (**IEntity::ReadXML**)
   3. Ajouter l’entité au model de données (**IModel::Add**)

### Créer une entité

Le mécanisme permettant de créer une nouvelle entité est composé de plusieurs opérations :

* Alloue une instance de l’entité (**IModel::CreateEntity**)
* Initialiser les données de base
* Ajouter l’entité au model de données (**IModel::Add**)



### Modifier une entité

…

### Supprimer une entité

…

## C# (WPF)

### Architecture

L’implémentation est basée sur le model MVVM.



### Avantages

* Temps de compilation rapide
* Code simplifié

### Inconvénients

* Support OS limité (principalement Windows)

### Dépendances

Le code utilise :

* La librairie .Net Framework

### Arborescence des dossiers

Ci-dessous l’arborescence des répertoires du code source :

|  |  |
| --- | --- |
| + RootDir  + [%namespace%]  + Domain  + Format  + Entity  + Interface  + View  + Event  + Lib  + ModelView  + View  + Control | Dossier racine  Dossier correspondant à l’espace de nom du model cible  Classes des domaines de valeurs  Classes de validation des formats de données  Classes des entités de données  Interfaces additionnelles du model  Vues du model  Classes d’événements  Interfaces et librairies  Contrôleurs  Vues  Contrôles de vue |

### Implémentation

#### Relations

|  |  |
| --- | --- |
| Association UML | Implémentation C++ |
| A\*<-- >B\*  (RoleA)<--> (RoleB) | Class A{  public  readonly Collection<B> bList ;  public  Add *[RoleB]*(B entity) ;  public  Remove*[RoleB]*(B entity) ;  }  Class B{  public  readonly Collection<A> aList ;  public  Add *[RoleA]*(A entity) ;  public  Remove*[RoleA](*A entity) ;  } |
| A1 <-> B\*  (RoleA)<--> (RoleB) | Class A{  public  readonly IEnumerable<B> bList;  public  Add *[RoleB]*(B entity) ;  public  Remove*[RoleB]*(B entity) ;  }  Class B{  Public readonly A a;  public  Set*[RoleA]*(A entity) ;  } |
| A1 <-> B1  (RoleA)<--> (RoleB) | Class A{  public  readonly B b;  public  Set*[RoleB]*(B entity) ;  }  Class B{  public  readonly A a;  public  Set*[RoleA]*(A entity) ;  } |

#### Evénements

Les événements sont définit par l’utilisateur selon les besoins, les plus communs sont déclarés ainsi :

|  |  |  |
| --- | --- | --- |
| Classe | Base | Usage |
| Event.EntityUpdateEvent | Lib.IEvent | Lorsqu’une entité a était modifiée |
| Event.EntityDeleteEvent | Lib.IEvent | Lorsqu’une entité a était supprimée |
| Event.EntityCreateEvent | Lib.IEvent | Lorsqu’une entité a était créée |
| Event.EntityPreCreateEvent | Lib.IEvent | Lorsqu’une entité doit être créée |
| Event.EntityChangeEvent | Lib.IEvent | Lorsqu’une entité a était modifiée |
| Event.EntityCopyPasteEvent | Lib.IEvent | Lorsqu’une entité est copiée/collée dans/depuis le presse-papier |

#### Librairies

|  |  |  |
| --- | --- | --- |
| Classe | Base | Description |
| Lib.SqlODBCFactory | Lib.IEntityFactory | Interface avec une base de données ODBC |
| Lib.SqlPostgresFactory | Lib.IEntityFactory | Interface avec une base de données PostgreSQL |
| Lib.SqlServerFactory | Lib.IEntityFactory | Interface avec une base de données SQL Server |
| Lib.EditableDatagrid | System.Windows.Controls .DataGrid | Fournit les commandes de bases pour éditer un tableau d’entités : **Copier, Coller, Créer, Modifier, Supprimer**  **Note :** Définir la propriété de Binding **UpdateSourceTrigger** à **PropertieChanged** ou **LostFocus** pour permettre la modification des données dans l’entité avant le traitement par le model. |
| Lib.EditWindow | System.Windows.Window | Fenêtre générique encapsulant la vue EditView |
| View.EditView | System.Windows.Controls.Grid | Vue générique utilisée pour générer rapidement une vue de contrôles utilisateur. |
| Lib.EntitiesModel | Lib.IModel | Fournit une implémentation générique d’un model de données |
| Lib.EventProcess | **-** | Fournit des implémentations type des événements pour les contrôleurs de données (copier/coller, création, suppression, modification, …) |
| Lib.StatesManager | Lib.ViewModelBase, Lib.IStateManager | Fournit une implémentation générique d’un gestionnaire d’état (historique des modifications) |

### PowerDesigner

Le Template de langage **Tpl\_CSharp.xol** propose la génération du diagramme de classes en code source exploitable par le compilateur C#.

#### Options de génération

|  |  |  |
| --- | --- | --- |
| Nom | Option | Description |
| namespace | Espace de nom réservé pour le model | Indique le « **namespace** » de base pour toutes les classes générées |
| serial\_datatype | Type de valeur associée au type Serial en base de données | Type de donnée C# a faire correspondre avec le type Serial du model de données conceptuel. Il s’agit des index auto incrémenté en base de données (généralement du type **System.Int32** ou **System.Int64** suivant l’architecture de la base de données). |
| useDotNetValidation | Implémenter l’interface de validation .Net | Implémente également l’interface standard **IDataErrorInfo** pour les entités validable (stéréotype). |
| useWPF | Exporter le code pour Windows presentation fundation | Exporte les vues, convertisseurs et autres code pour les applications Windows basé sur WPF.  **Note : La génération des vues éditables requière une référence à la librairie « Xceed.WPF.Toolkit »** |

# Génération avec PowerDesigner

Cette section est un support à l’outil de modélisation PowerDesigner. Les Templates développés dans les implémentations existantes ont pour but de générer le code correspondant aux modèles de données et à l’architecture cliente.

L’utilisation d’un outil de modélisation tel que PowerDesigner n’est pas indispensable mais permet un gain de temps et de pérennité pour le projet.

Notez que le code généré reste totalement indépendant du logiciel dans le cas ou PowerDesigner ne serait plus utilisé. Dans ce cas un logiciel de substitution ou le programmeur reprendra l’édition du code générique.

## Type de projet

Suivant le but recherché plusieurs méthodes existent.

La base de données et l’application sont crées sur la base d’un model de données commun:

* Créer un model générique de données (c’est celui-ci qui sera mis à jour en cas de modification du model)
* Générer un model physique de données
* Générer un model orienté objet

L’application est développée autour d’une base de données existante:

* Créer un model orienté objet et faire corresponde les noms des champs et tables avec la base de données existante (voir persistance)

La base de données est développée autour d’une application existante:

* Créer un model physique de données et faire corresponde les noms des champs et tables dans le model de l’application existante

## Templates

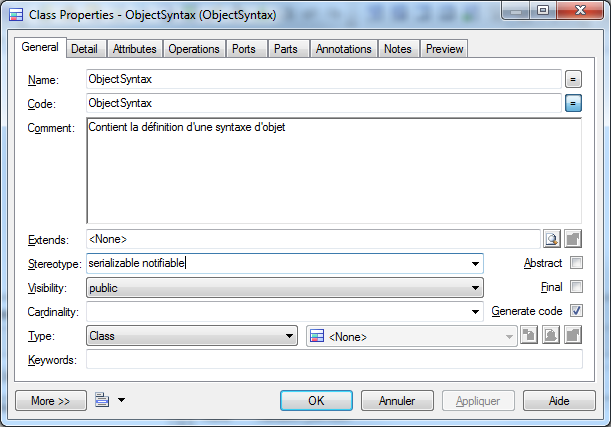
Ce tableau résume les modèles cibles d’applications :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Langage | Version | Framework | Fichier Template | Modèle |
| C# | 5.0 | - | Tpl\_CSharpApp.xol | OOM |
| C# | 5.0 | .Net | Tpl\_CSharpApp.xol | OOM |
| SQL Sever | Std 2012 | - | Tpl\_SQLServer2012.xdb | PDM |

## Stéréotypes

Les stéréotypes sont utilisés dans les objets pour définir l’usage final dans le code. Par exemple, si les données d’une classes sont persistantes, sérialisables, validables, etc…

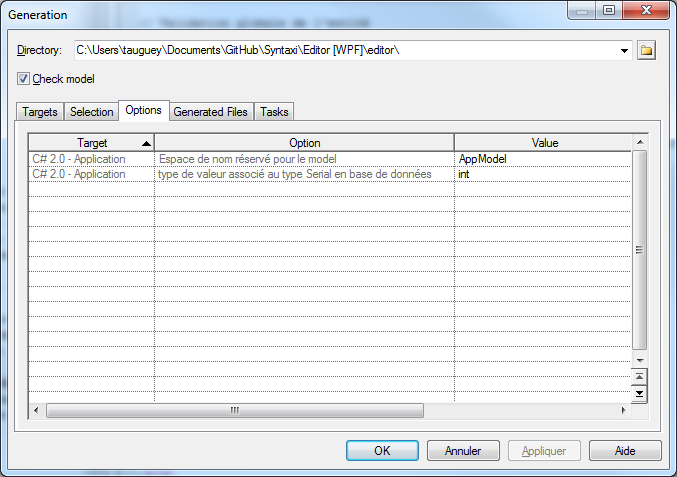
Les stéréotypes sont définit dans les propriétés de classe :



***Propriété d’une classe contenant les stéréotypes « serializable » et « validable »***

## Options de génération

Les options de générations sont spécifiques au langage cible, elles permettent d’ajuster la génération du code. Chaque option est définit plus loin dans son model.



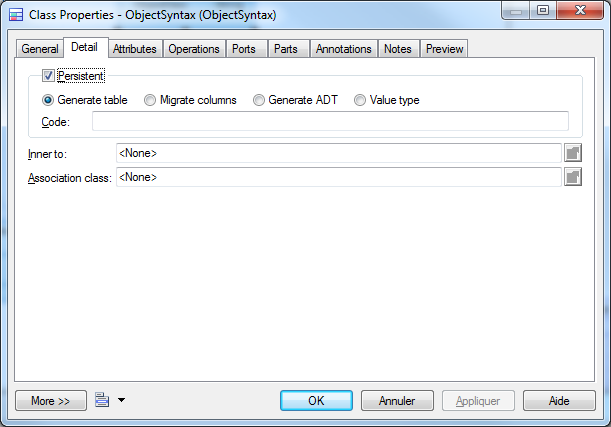
***Options de génération dans le dialogue « Générer »***

## Persistance

Il est possible de spécifier explicitement le nom des tables et des champs en correspondance avec la base de données. Par défaut le nom des tables et des champs est hérité des classes.

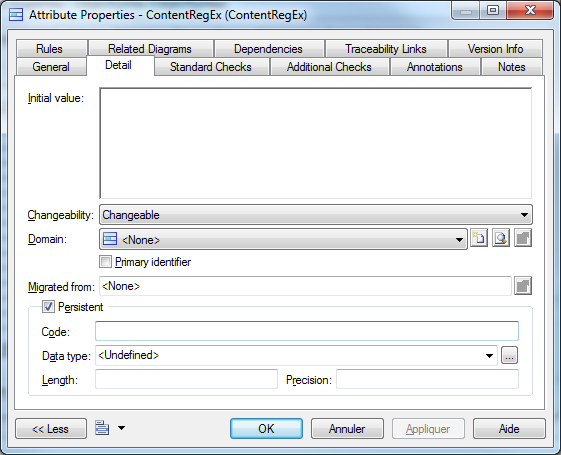
Spécifier le nom d’une table :

Rendez-vous dans le dialogue de propriétés de classe sous l’onglet « **Détail** » et éditer le champ « **Code** » sous le groupe « **Persistent** ».



Spécifier le nom d’un champ :

Rendez-vous dans le dialogue de propriétés de l’attribut sous l’onglet « **Détail** » et éditer le champ « **Code** » sous le groupe « **Persistant** ».



Classe d’association

Dans les types d’associations multiples (N-N) il est nécessaire de représenter la table de correspondance en base de données par une classe d’association.

