|  |
| --- |
|  |
| 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](#_Toc433794687)

[Lexique 5](#_Toc433794688)

[Squelette d’application 6](#_Toc433794689)

[Application 8](#_Toc433794690)

[Contrôleur 8](#_Toc433794691)

[Model 8](#_Toc433794692)

[Vue 8](#_Toc433794693)

[Evénement 9](#_Toc433794694)

[Entité 10](#_Toc433794695)

[Etat 12](#_Toc433794696)

[Donnée 12](#_Toc433794697)

[Intégration 13](#_Toc433794698)

[Application 13](#_Toc433794699)

[Model 13](#_Toc433794700)

[Vue 13](#_Toc433794701)

[Contrôleur 13](#_Toc433794702)

[Entités 13](#_Toc433794703)

[Evénements 14](#_Toc433794704)

[Donnée 15](#_Toc433794705)

[Etat 16](#_Toc433794706)

[Patron MVC 17](#_Toc433794707)

[Séparation du code 17](#_Toc433794708)

[Variantes du parton MVC 17](#_Toc433794709)

[Préparation du projet 18](#_Toc433794710)

[Répertoire de travail 18](#_Toc433794711)

[Suivi de version 18](#_Toc433794712)

[Modélisation 19](#_Toc433794713)

[Diagramme de classes 19](#_Toc433794714)

[Implémentation 21](#_Toc433794715)

[C++ (wxWidget) 22](#_Toc433794716)

[C# (WPF) 27](#_Toc433794717)

[C# (WinForms) 32](#_Toc433794718)

[Visionner 3 33](#_Toc433794719)

[Génération avec PowerDesigner 34](#_Toc433794720)

[Type de projet 34](#_Toc433794721)

[Templates 34](#_Toc433794722)

[Stéréotypes 34](#_Toc433794723)

[Options de génération 35](#_Toc433794724)

[Persistance 35](#_Toc433794725)

# Introduction

Ce document propose une méthode de développement générique et évolutive applicable au plus grand nombre de projets informatiques mettant en relation les données et les interfaces utilisateurs dans une architecture logiciel.

Cette méthode n’a pas pour prétention de fournir une architecture logicielle finie mais plutôt de définir les principaux éléments d’une application (entité, message, événement, …) pour ensuite les appliquer à différentes architectures logiciel (l’implémentation).

Les architectures sont généralement dérivées du patron MVC pour les langages orienté-objet (C#, C++, Java) et de simple patron de fonction pour les langages procéduraux (C, Basic,). Dans tous les cas la méthode tire partie des avantages des langages et des technologies pour arriver au but commun qui est la simplicité de développement et la cohérence des systèmes.

L’intérêt pour le programmeur est de se concentrer sur l’essentiel :

* La modélisation des données
* La logique 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)

Et toutes autres concepts jugé utile au programmeur…

# Lexique

ORM : Mapping objet-relationnel. Technique de programmation permettant de faire correspondre des données entre les systèmes de types incompatibles. Par exemple le mapping d’une base de données relationnelle en objets du langage de programmation.

# Squelette d’application

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

Le squelette définir la structure de base de l’implémentation sur laquelle est basée cette méthode. En implémentant une partie ou l’ensemble de ces interfaces, une architecture suit un processus global. Certaines de ces interfaces sont indispensables au fonctionnement d’une architecture et d’autres sont totalement inutiles dans certains contexte.

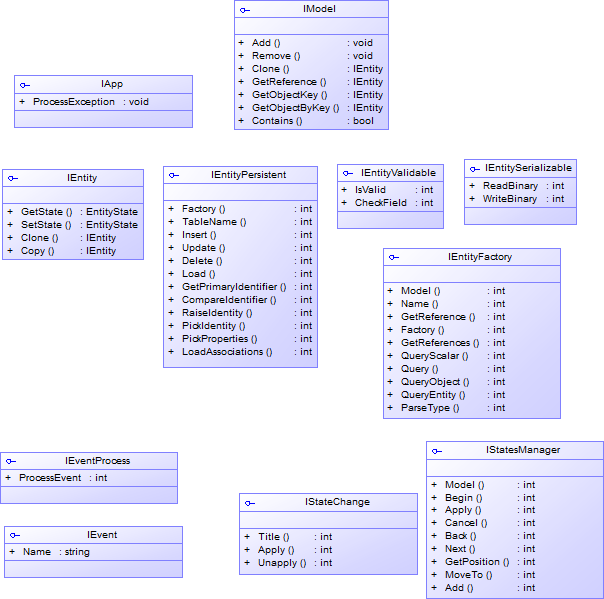
Suivant l’implémentation finale du programmeur certaines de ces interfaces seront remplacées par des classes d’un Framework externe ou des éléments du langage. Dans tous les cas ce squelette a uniquement pour but de poser une base à l’application finale, le programmeur restant maître de son architecture.

Les différents groupes d’interfaces

Les interfaces sont triées par groupe plus ou moins indépendant les uns des autres.

|  |  |
| --- | --- |
| GROUPE | DEFINITION |
| Application | Sommet de l’architecture gérant la globalité des ressources et ayant une |
| Contrôleur | Dans le patron MVC le contrôleur implémente la logique métier entre les données et l’utilisateur. |
| Model | Dans le patron MVC le model implémente l’interface avec les données de l’application. |
| Vue | Dans le patron MVC la vue réagit aux interactions de l’utilisateur et communique avec le contrôleur pour affecter les données. |
| Evénement | Les événements permettent de diffuser un message à travers les différents éléments de l’application (vue, model, contrôleur, entité, …). Son rôle est d’informer, son typage et les données qu’il transporte sont interpréter indépendamment par l’élément qui le reçoit. |
| Entité | Les entités sont les objets d’un model de données. Elles définissent des propriétés, des fonctions et des relations de dépendances et d’associations. |
| Etat | Un état définit une donnée à un moment T de sont existence. Les états sont principalement utilisé pour historiser les modifications d’un utilisateur ou d’un thread. |
| Donné | La donnée est l’élément le plus basique d’une application, elle peut être un type de base (entier, booléen, réel, ..) ou un type composé (chaine de caractères). L’intérêt et des pouvoir la transformer et la représenter de différente manières (dans une chaine SQL, dans un fichier XML, dans un flux binaire, …). Cette transformation est la base de l’échange de données entre applications. |

Représentation UML des interfaces

****

## Application

Point d'entrée du programme et des ressources logicielles.

### Interface IApp

|  |  |
| --- | --- |
| **ProcessException** | Traite une exception retournée par l’un des éléments de l’application. Implémente généralement un message d’erreur ou une voix alternative à un problème. |

## 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 |

## Donnée

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 |

# Intégration

Cette section d’écrit les éléments de l’architecture logicielle. L’interaction entre ces éléments permet de répondre à la plupart des besoins de l’application (rafraichissement des données, communication, contrôle des saisies, etc…)

## Application

Maintient l’ensemble des ressources de l’application, généralement les instances de vues, modèles, … C’est également le point d’initialisation des données et d’interaction avec le système (fichiers, …)

## Model

Un model est un ensemble d’entités ayant des relations communes. Une application peut gérer plusieurs modèles de données, par exemple les données persistantes d’un système (une base de données) et les données locales de l’application (configuration, fichier de projet, …). Cette séparation des entités en modèles et des modèles entre-eux permet au programmeur une grande flexibilité et un meilleur contrôle des interactions.

Pour les langages de haut niveau, le model fait également office de gestionnaire de mémoire en contenant la liste de toutes les entités allouées.

## Vue

La vue intercepte les actions de l’utilisateur puis les transmet au contrôleur pour traitement.

## Contrôleur

Le contrôleur interprète les commandes de la vue pour affecter le model. Le contrôleur à la responsabilité de maintenir un model cohérant en contrôlant les actions de l’utilisateur et en gérant les relations entre les entités.

## Entités

Les entités regroupe une grande partie des fonctionnalités de l’application, elles sont les éléments de l’application dont d’écoule les vues, les actions et la logique métier d’une application.

Sérialisation

La sérialisation écrit dans un flux de données les propriétés de l’entité ainsi que ses associations de composition.

Typiquement le programmeur créera une hiérarchie d’entité en partant du haut (le projet) vers le bas (les données). Ainsi la sérialisation à elle seule permet de concevoir les formats de fichier d’une application ou encore l’échange de données entre applications ou même un réseau.

Persistance

La persistance associe aux entités à un système de gestion de base de données (SGBD). Chaque entité représente une table, chaque propriété une colonne et chaque pointeur une clé étrangère. L’implémentation des méthodes de l’interface IEntityPersistent permet l’insertion, la mise à jour et la suppression des données.

L’interface IEntityFactory ajoute une abstraction supplémentaire entre le SGBD propriétaire (SQL Server, Postgres, Oracle, …) et l’entité.

Validation

La validation concerne les propriétés d’une entité. Elle permet de contrôler les entrées d’un utilisateur avant d’applique la modification au model.

## Evénements

Les événements sont définit par le programmeur suivant les besoins de l’application. Cependant certains sont très largement utilisés dans la plupart des applications (Copier/Coller, Changement d’une données par un élément extérieur, action de l’utilisateur, …).

Voici quelques implémentations génériques des événements les plus courants. Bien entendu cette logique diffère selon l’implémentation réalisée :

Copier une entité

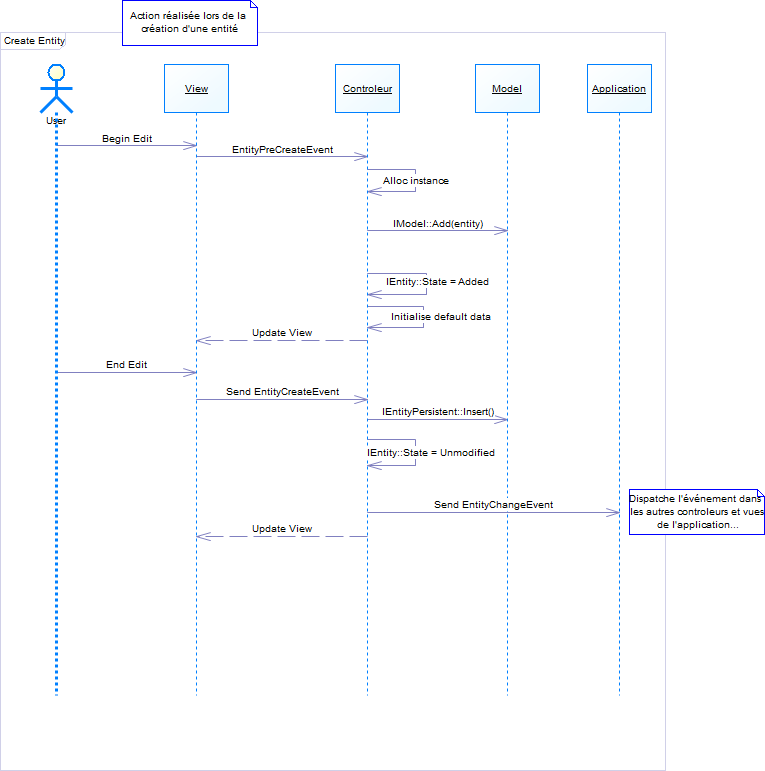
1. Sérialiser les données de l’entité dans un format XML
2. Copier le contenu XML dans le presse-papier

Coller une entité

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

* 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

…

## Donnée

La donnée est un élément fondamentale de l’architecture de l’application, toute donnée définit dans une entité doit pouvoir être transporté dans différent format. La donné est le lien entre les différentes implémentations logicielles (SGBD, Process, Fichier, …). Elle doit pouvoir être lue et écrite dans différents format pour être comprise de tout le monde.

En définissant une donnée on offre aussi au programmeur d’implémenter des types personnalisés manipulable par l’application.

## Etat

Un état doit pouvoir être appliqué ou annulé à l’ensemble de l’application. Il affecte tous les éléments, notamment les données et l’interface utilisateur. Par exemple, dans une application de dessin le changement de couleur d’une forme est un état dont l’application remplie de rouge un triangle et l’annulation restaure la couleur précédente.

Pour être cohérant les états sont obligatoirement ordonnés dans le temps et applicable uniquement l’un après l’autre (le changement de couleur d’une forme ne peu pas être appliqué avant la création de celle-ci).

# Patron MVC

L’application d’un patron de développement crée une séparation des rôles dans l’architecture de l’application. Cette séparation contribue à maintenir un code efficace et permet le partage des tâches.

## Séparation du code

La séparation du code par couche est l’élément capitale du patron MVC elle permet au programmeur de rendre la code plus lisible et plus facile a entretenir lorsque la taille et la complexité de l’application augmente.

Les 3 couches du patron MVC sont :

* Les données (model)
* L’interface utilisateur (vue)
* La logique métier (contrôleur)

Ces trois couches communiquent directement (appel de méthodes) ou indirectement (par événements) suivant les besoins de l’application.

## Variantes du parton MVC

Suivant les technologies utilisées des variantes plus ou moins complexes du patron MVC existe (MVP, MVVM, …). L’application du patron n’a pas besoin d’être appliqué rigoureusement le but étant d’obtenir une application robuste et efficace dans le contexte de développement c’est pourquoi chacune des implémentations proposés dans cette méthode adapte le patron MVC aux technologies qu’elles emploient.

# 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

La première étape d’un projet consiste à modéliser son système (généralement une application, une base de données et les interactions).

Dans cette méthode nous utilisons plusieurs schémas de la notation UML pour modéliser nos besoins :

* + Le diagramme de classe (pour générer le model de données « ORM »)
  + Le model physique de données (pour générer la base de données)
  + Le diagramme des cas d’utilisations (pour visualiser la logique métier)

UML est le point d’orgue de toute l’implémentation logicielle, c’est sur elle que nous nous basons pour écrire nos objets et la logique métier.

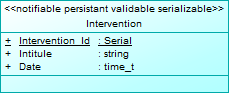
## Diagramme de classes

Le diagramme de classe modélise les entités du modèle ainsi que leurs relations. Dans l’application cela se traduit en classes de données ayant des propriétés distinctes et des pointeurs de classes.

### Entité

Toute classe UML est une entité du model, elle se présente sous la forme suivante :

Les stéréotypes



Les propriétés

Le nom

Recommandations

* Eviter de nommer un attribut avec le même nom que l’entité (impossible en C#)

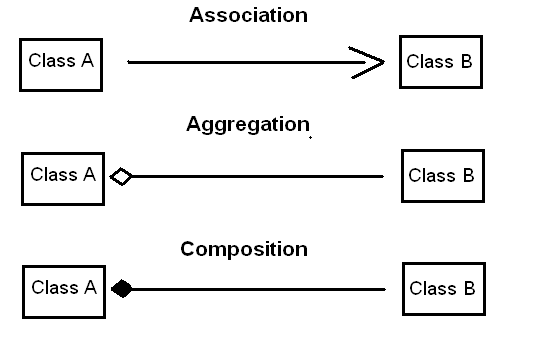
Stéréotypes

Les stéréotypes sont utilisés pour définir l’usage final d’une classe dans le code. Toutes les classes sont interprétées comme une entité, les stéréotypes ci-dessous ajoutent des implémentations à l’entité de base.

|  |  |
| --- | --- |
| 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)** |

### Association

Une association est un lien de dépendance entre 2 entités, il en existe plusieurs types de la forme suivante :



Recommandations

* Il est préférable de nommer les relations pour plus de clarté dans le code généré

Association



Une association simple met en relation 2 entités indépendantes l’un de l’autre. Leur relation est fondé sur un échange exemple : **Employé** à une relation d’association avec **Employé** car ils peuvent s’échanger des informations utiles.

Dans le code:

* Les entités ayant une relation d’agrégation sont sérialisé indépendamment.
* Les entités ayant une relation d’agrégation persistent après la suppression de l’entité associée.

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 forte existe. Par exemple : Un **Employé** à une relation de composition forte avec son **Adresse** car elle n’a plus lieu d’exister dans le model si l’**Employé** n’existe plus.

Dans le code:

* Les entités ayant une relation de composition sont sérialisé avec l’entité parent.
* Les entités ayant une relation de composition sont 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 : Un **Employé** à une relation d’agrégation avec sa **Société** car celle-ci persiste même si Employé n’existe plus.

Dans le code:

* Les entités ayant une relation d’agrégation sont sérialisé indépendamment.
* Les entités ayant une relation d’agrégation persistent après la suppression de l’entité associée.

### Cardinalité

Les cardinalités représentent la quantité d’entités associées à une relation, elle se présente sous la forme suivante :

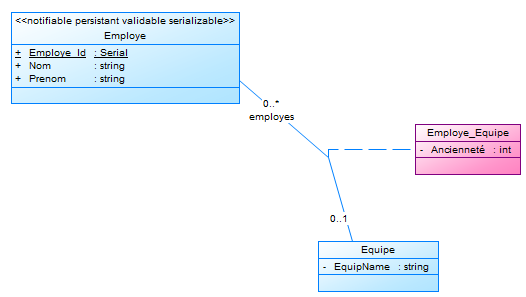
Cardinalités



Dans le code:

|  |  |
| --- | --- |
| Association UML | Association dans le Code |
| A\*<-- >B\*  (RoleA)<--> (RoleB) | A{  List<B> RoleB ;  }  B{  List<A> RoleA ;  } |
| A1 <-> B\*  (RoleA)<--> (RoleB) | A{  List<B> RoleB ;  }  B{  A RoleA;  } |
| A1 <-> B1  (RoleA)<--> (RoleB) | A{  B RoleB;  }  B{  A RoleA;  } |

### Classe d’association



Il ne peut avoir qu’une classe par association

### Héritage

…

## Model physique de données

…

## Diagramme des cas d’utilisations

…

# Implémentation

Cette section décrit les différentes implémentations réalisées.

Suivant le choix du langage et du Framework l’implémentation du patron vari selon les besoins (MVC, MVP, MVVM, …). Dans tous les cas le but est d’implémenter les notions de bases avec les outils à disposition.

|  |  |  |  |
| --- | --- | --- | --- |
| Langage | Framework | UI Framework | Implémentation |
| C# | .Net | - | MVC |
| C# | .Net | WPF | MVVM |
| C++ (VC) | stdlib | - | MVC |
| C++ (VC) | stdlib | wxWidgets | MVC |

Chaque exemple propose une implémentation du squelette adapté au Framework utilisé. Ces implémentations peuvent êtres utilisées comme base aux projets de développement.

## C++ (wxWidget)

Implémentation standard du C++ avec une interface utilisateur multiplateforme basé sur la librairie wxWidgets.

### Architecture

L’implémentation est basée sur le model MVC.



### Avantages

* Multi-OS (principalement Windows, Linux, Macintosh)

### Inconvénients

* Temps de compilation long
* Code plus complexe

### Dépendances

Le code utilise :

* La librairie standard C++
* La librairie RapidXml pour la sérialisation XML

### Arborescence des dossiers

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

|  |  |
| --- | --- |
| + RootDir  + [%namespace%]  + Domain  + Format  + Entity  + Interface  + 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  Classes d’événements  Interfaces et librairies  Contrôleurs  Vues  Contrôles de vue |

### Implémentation

#### Relations

|  |  |
| --- | --- |
| Association UML | Implémentation C++ |
| A\*<-- >B\*  (RoleA)<--> (RoleB) | Class A{  Private :  List<B\*> roleB ;  Public :  Remove*[RoleB]*(B\* entity)  Add *[RoleB]*(B\* entity)  }  Class B{  Private :  List<A\*> roleA ;  Public :  Remove*[RoleA](*A\* entity)  Add *[RoleA]*(A\* entity)  } |
| A1 <-> B\*  (RoleA)<--> (RoleB) | Class A{  Public :  Remove*[RoleB]*(B\* entity)  Add *[RoleB]*(B\* entity)  }  Class B{  Public :  Set*[RoleA]*(A\* entity)  } |
| A1 <-> B1  (RoleA)<--> (RoleB) | Class A{  Public :  Set*[RoleB]*(B\* entity)  }  Class B{  Public :  Set*[RoleA]*(A\* entity)  } |

#### Evénements

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

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

#### Librairies

|  |  |  |
| --- | --- | --- |
| Classe | Base | Description |
| Lib::EntitiesModel | Lib::IModel | Fournit une gestion d’un model de données générique |
| Lib::EventManager | Lib::IEventManager | Fournit une gestion d’une liste d’objets notifiables |
| Lib::TypedData | Lib::ITypedData | Fournit une gestion des données typées |

### PowerDesigner

Le Template de langage **Tpl\_CPlusPlus.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 **int** ou **long** suivant l’architecture de la base de données). |
| xml\_parser | Nom de la librairie utilisée pour intégrer la serialisation XML. | Prend l'une des valeurs suivantes ():  - RapidXml  Si aucun laisser les champs vide, dans ce cas soit la sérialisation XML n’est pas implémenté soit le programmeur définit sa propre implémentation dans Class::Template::XmlParser. |

## C# (WPF)

Implémentation .Net du C# avec une interface utilisateur Windows Presentation Fundation.

### 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

#### Attributs

|  |  |
| --- | --- |
| UML | C# |
| A  Integer Name  … | Class A{  protected  int name ;  public  int Name{ get ; set ; } ;  } |

#### Relations

|  |  |
| --- | --- |
| UML | 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.EntityPreUpdateEvent | Lib.IEvent | Lorsqu’une entité doit être modifiée |
| 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 »** |

## C# (WinForms)

Implémentation .Net du C# avec une interface utilisateur Windows Forms.

### Architecture

L’implémentation est basée sur le model MVVM.

## Visionner 3

Implémentation du model de données pour le langage Basic dans Visionner 3.

### Architecture

L’implémentation Visionner est basée sur l’accès aux model de données.

### Avantages

* Génération des fonctions d’accès à la base de données et mappage des objets en base de données

### Structure des dossiers

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

|  |  |
| --- | --- |
| + RootDir  + Model.txt  + Model.bas | Dossier racine  Model des structures de données  Fonctions « Basic » d’accès aux données |

### Implémentation

|  |  |  |
| --- | --- | --- |
| UML |  | Visionner |
| Table | Type dans la base de données |
| Entité | Fonctions dans le module Basic |

#### Tables

Chaque table définit un type de données structuré dans la base de données de Visionner.

#### Base de données

Chaque table définit une fonction de sélection pour chaque clé primaire et alternative.

### PowerDesigner

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

Le Template de langage **Tpl\_Visionner.xdb** propose la génération du diagramme de données en structure de données Visionner.

# 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

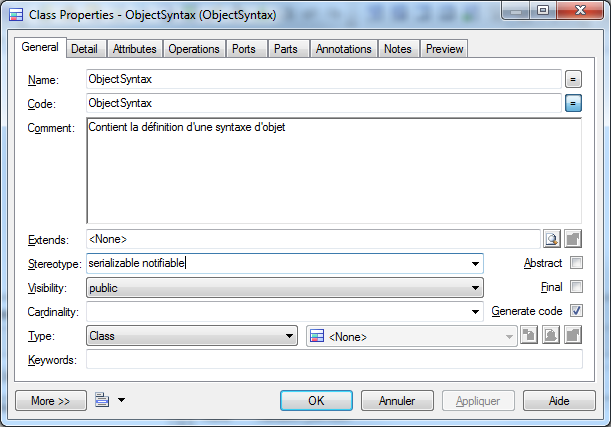
Ce tableau résume les modèles cibles d’applications :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Langage | Version | Framework | Fichier Template | Modèle |
| Visionner – Basic | 3.x | - | Tpl\_Visionner.xdb  Tpl\_Visionner.xol | OOM  PDM |
| C# | 5.0 | - | Tpl\_CSharpApp.xol | OOM |
| C# | 5.0 | .Net | Tpl\_CSharpApp.xol | OOM |
| C++ | VC2013 | - | Tpl\_CPlusPlus.xol | OOM |
| C++ | VC2013 | wxWidget | Tpl\_CPlusPlus.xol | OOM |
| SQL Sever | Std 2012 | - | Tpl\_SQLServer2012.xdb | PDM |
| Postgres SQL | 9.x | - | Tpl\_pgsql9.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…

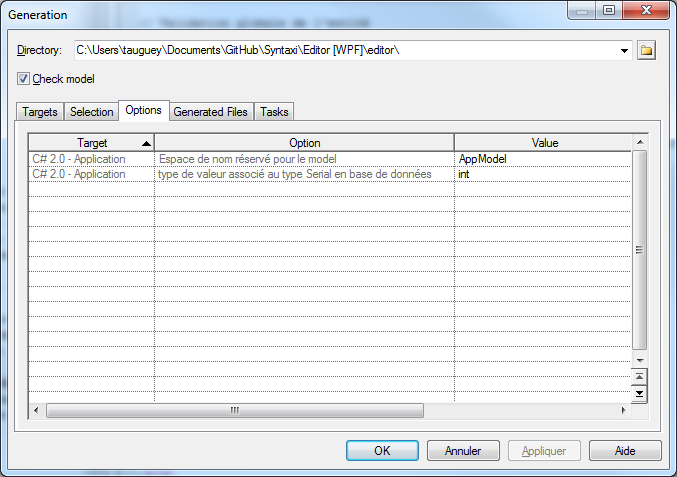
Les stéréotypes sont définit dans les propriétés de classe :



***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.



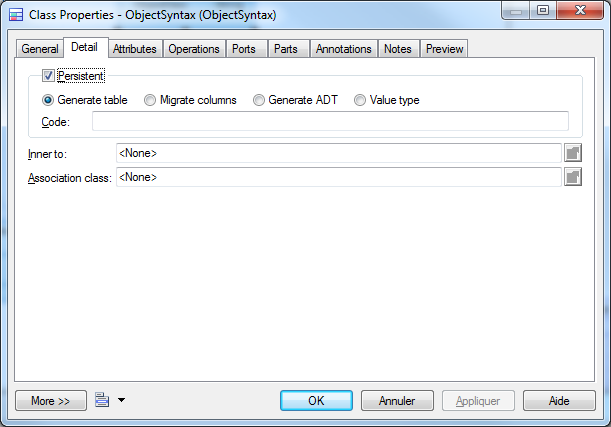
***Options de génération dans le dialogue « Générer »***

## 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.

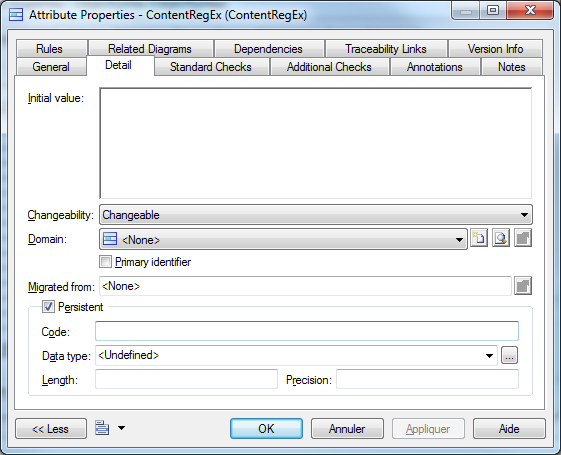
Spécifier le nom d’une table :

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.

