|  |
| --- |
|  |
| Application Type |
| Tests et validation |
|  |
| **Thomas Auguey** |
| **20/08/2015** |

|  |
| --- |
| Tests et validation du système. |

SOMMAIRE

[Introduction 3](#_Toc434323019)

[Lexique 4](#_Toc434323020)

[Modélisation 5](#_Toc434323021)

[Diagramme de classes 5](#_Toc434323022)

[Model physique de données 10](#_Toc434323023)

[Diagramme des cas d’utilisations 11](#_Toc434323024)

# Introduction

Ce document décrit la phase de modélisation. Dans cette phase nous exprimons le besoin sous forme de données et de relations.

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

* La modélisation des données
* La logique métier

La méthode s’appuie sur conception descendante (haut-bas) la haut modélisant les besoin pour arriver en bas l’implémentation du code. On part d’une implémentation simple on la valide puis on incorpore petit à petit les nouveautés en réitérant le cycle de développement, chaque cycle impose une incrémentation de la version mineur jusqu'à atteindre la version majeur validée par le client.

Ce cycle de développement est répété à chaque modification du projet.

Le développeur reprend les phases de conceptions :

1. Recueil du besoin
2. Expression du besoin (uses cases)
3. Modélisation du besoin (MCD/MPD/MOO)
4. Implémentation du besoin (Codage, génération de code)
5. Tests et validation
6. Incrémentation de la version

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.

# 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.

# Modélisation

Le but est de définir les règles de modélisations qui serons appliqué à l’implémentation.

Cette méthode couvre la modélisation des principaux éléments d’un système. Nous utilisons plusieurs schémas de la notation UML/Merise pour réaliser nos besoins :

* Le diagramme de classe (pour implémenter le model applicatif)
* Le model physique de données (pour implémenter la base de données)
* Le diagramme des cas d’utilisations (pour implémenter la logique métier)

Pour rappel la modélisation doit de nous aider à définir les limites du système et son implémentation, cependant l’UML reste une vision subjective du projet et ne doit détailler que les points intéressant du projet du point de vue du programmeur. Le but étant de gagner du temps et pérenniser les phases de conception et non le contraire.

## Diagramme conceptuel des données

## Diagramme de classes

Nous utiliserons le diagramme de classe pour modéliser la partie applicative du système. Dans l’application cela se traduit généralement par des classes et des propriétés.

L’intérêt de modéliser le model applicatif est multiple :

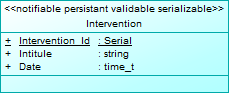
* Obtenir une vision relationnelle des entités aidant la conception
* Identifier le mappage entités/tables des données persistantes « ORM »
* Définir les relations entres entités

### Classe

Nous nommerons les classes des entités car elles seront utilisées en ce sens, c'est-à-dire : Un ensemble de propriétés groupés au sein de classes ayant des relations communes.

Une entité se présente sous la forme suivante :

Les stéréotypes



Les propriétés

Le nom

Recommandations

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

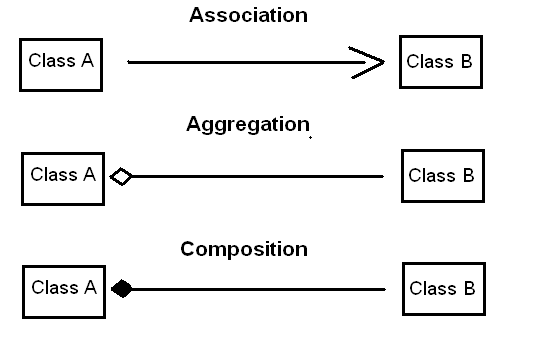
Stéréotypes

Les stéréotypes sont un moyen de définir l’usage final d’une classe dans le code. Les stéréotypes ajoutent généralement des méthodes durant la phase d’implémentation du code source.

|  |  |
| --- | --- |
| Mot clé | Description |
| persistant | Indique que les données de l’entité sont stockées en base de données.  La persistance répond au besoin de transporter les données de la mémoire avec les données d’un SGBD. |
| serializable | Indique que les données de l’entité peuvent être formatées dans un format binaire ou texte pour être intégrer à un flux de données.  La sérialisation répond au besoin d’échanger des données entre processus ou avec un support de stockages (presse-papier, fichiers de données, …) |
| notifiable | Indique que les données de l’entité sont soumises à notification en cas de changement.  La notification répond au besoin de synchronisation dans une application dotée d’une interface utilisateur. |
| validable | Indique que les données de l’entité sont soumises à validation par un format définit.  La validation répond au besoin de contrôler les entrées d’un utilisateur dans un modèle de données cohérant. |

### Relation

Pour représenter les relations entre les entités nous utilisons les 3 associations de l’UML :



Recommandations

* Il est préférable de nommer les relations pour plus de clarté dans l’implémentation

Association



Une association est non restrictive dans le sens où l’entité A n’a pas besoin de l’entité B pour subsister. Par exemple : **Ordinateur** à une relation d’association avec **Clé\_USB** car ils peuvent s’échanger des données mais l’ordinateur n’a pas besoin de la clé pour fonctionner.

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

Une composition crée une dépendance forte dans les sens où l’entité B appartient à l’entité A (ou vice versa). Par exemple : Un **Employé** à une relation de composition forte avec son **Adresse** car elle lui est dédié et ne sera pas partagé avec un autre employé (en effet si cet employé déménage il n’y a que son adresse qui change et non l’adresse d’un autre employé).

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’entité parent.

Agrégation

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

L’agrégation est similaire à une composition mais dans le cas ou les entités forme un tout mais est moins forte car l’entité contenue n’est pas indispensable. Par exemple : Un **Client** à une relation de composition avec son **Adresse** (de livraison) car elle est indispensable pour acheminer le produit. Au contraire, le **Client** à une relation de composition avec son **Adresse** (de facturation) car elle est facultative (la facture peut être envoyé à l’adresse de livraison). Dans tous les cas, ces 2 relations appartiennent uniquement au **Client** ce qui n’en fait pas une relation d’association.

Dans le code:

* Les entités ayant une relation d’agrégation se comportent comme une composition mais peuvent être absentes « null ».

### Cardinalité

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

Cardinalités



Les cardinalités sont formulée ainsi :

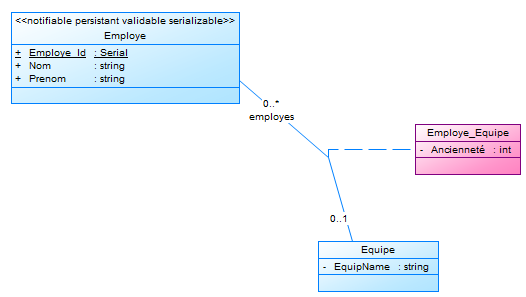
* Employé appartient obligatoirement (1..) une (..1) Société
* Société peut (0..) posséder plusieurs (..\*) employés

Dans le code:

* La cardinalité décide si l’attribut peut être NULL ou obligatoire
* La cardinalité décide si l’attribut est une liste de référence ou une référence unique

### Classe d’association

La classe d’association représente une données lié à une relation (qui n’appartient ni à l’une, ni à l’autre des entités). La classe se représente comme une entité attachée par des pointillés à une relation d’association.



Règle

* On ne définit qu’une classe d’association par association

### Opération

Les opérations représentes les actions réalisable par une classe, nous les utiliserons pas dans cette méthode car elles peuvent être nombreuses et gêner la lecture du schéma.

### Héritage

L’héritage définit une entité enfant héritant des propriétés d’une entité parente. La notion est similaire à l’héritage dans les langages de programmation objet.

Règle

* Une entité est limité à un héritage à la fois (l’héritage multiple est généralement impossible dans les langages orientés objet)

## Model physique de données

Nous utiliserons le model physique de données (PDM issue de la méthode Merise) pour modéliser une base de données relationnelle.

### Table

Les tables sont représentées sous la forme suivante :



Les propriétés

Le nom

Recommandations

* Eviter de nommer un attribut avec le nom de la table

### Association

Les relations représentent les clés étrangères entre les entités:



La flèche pointe vers la table de référence dont la clé primaire deviendra une clé étrangère dans la table pointant.

### Associations Multiples

Les associations N\*N sont représentées par des tables tampons.

Dans l’exemple ci-dessous les tables **Itineraire** et **Atterrissage** représentent 2 relations N\*N entre **Avion** et **Aeroport**.



### Données associés

Les tables « d’associations » peuvent disposer de données supplémentaires.

## Diagramme des cas d’utilisations

…