# TP 1 Conception de la solution.

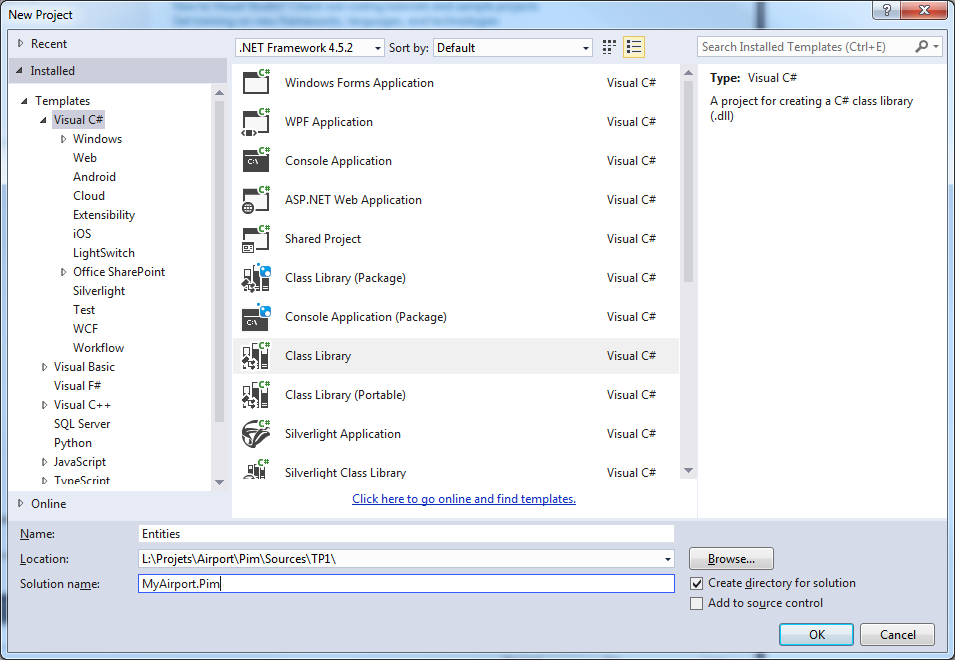
## Objectifs :

* Rafraichissement des connaissances C# et Visual Studio.
* Découverte de l’environnement de travail.
* Maitrise et compréhension du projet PIM.
* Conception des éléments de base de la solution PIM.

## Déroulement :

Dans Visual Studio, démarrer un nouveau projet Visual C# de type Bibliothèque de classe.

Nommer la solution MyAirport.Pim et le projet Entities.



Ce projet va contenir la déclaration de l’ensemble des objets extrait de la source de données est exposés au client.

Renommer la classe Class1 en BagageDefinition

Et ouvrer le menu contextuel du projet Entities et sélectionner propriétés

Dans la fenêtre propriété choisissez l’onglet Application et modifier l’espace de noms par défaut : à MyAirport.Pim.Entities

Dans la classe BagageDefinition modifier le nom du namespace en MyAirport.Pim.Entities

namespace MyAirport.Pim.Entities

{

/// <summary>

/// Définition d'un object bagage

/// </summary>

sealed public class BagageDefinition

{

public int IdBagage{get;set;} // Identifiant bagage en base de données permet l'identification unique d'un bagage

public string CodeIata { get; set; } // Numéro du bagage présent sur l'étiquette

public string Compagnie { get; set;} // Code Iata de la compagnie aerienne sur 2 lettres

public string Ligne { get; set; } // Numéro de vol 3 ou 4 digits et parfois une lettre a la fin

public DateTime DateVol { get; set; } // Jour et heure de depart du vol

public string Itineraire { get; set; } // Arrêt ou descend le passager pour ce vol

public bool Prioritaire { get; set; } // Est-ce que le passager est un passager prioritaire?

public bool EnContinuation { get; set; } // Si la destination est différente de l'itineraire, est-ce que le bagage doit être livré au passager au prochain arrêt?

public bool Rush { get; set; } // Bagage sans passager (pour les bagages ayant ratés un vol)

}

}

Ajouter un répertoire de solution à la solution MyAirport.Pim est le nommer ‘Models’.

Ajouter dans le répertoire un nouveau projet de type ‘bibliothèque de classes’ et lui donner le nom de ‘Models.AbstractDefinition’.

Nous allons maintenant créer une Factory qui va nous servir d’intermédiaire pour accéder à notre source de données. L’utilisation d’une Factory nous permet de faire abstraction du modèle de données physique. La Factory a en charge de créer le modèle physique (indiquer dans le fichier de configuration de l’application) et de l’exposer via une classe abstraite. Toutes les implémentations liées à un modèle physique de données devront donc dériver de cette classe abstraite. La Factory, la classe abstraite et chacune des implémentations spécifiques d’un modèle de données doivent être situées dans des projets distincts afin de limiter les dépendances.

La Factory expose la classe abstraite via l’utilisation d’un pattern singleton permettant au client un accès simple aux données sans avoir à gérer le cycle de vie de l’instance Abstraite.

Renommer la classe générée en AbstractDefinition,

Définir la classe comme étant une classe abstraite.

Cette classe devra contenir l’ensemble des méthodes accessibles par chacune des implémentations physiques de notre modèle de données.

Modifier le nameSpace en MyAirport.Pim.Models

Ajouter la définition d’une méthode abstraite GetBagage qui prend en argument un idBagage de type int et qui retourne un objet de type BagageDefinition.

Ajouter une seconde méthode abstraite GetBagage qui prend en argument de type string nommé codeIataBagage et qui retourne une liste de BagageDefinition

namespace MyAirport.Pim.Models

{

public abstract class AbstractDefinition

{

public abstract BagageDefinition GetBagage(int idBagage);

public abstract List<BagageDefinition> GetBagage(string codeIataBagage);

}

}

L’objet ‘BagageDefinition’ est souligné en rouge par VisualStudio, cela signifie que le projet Model.AbstractDefinition ne connait pas la déclaration de l’objet BagageDefiniton.

Corriger cette erreur.

Ajouter au répertoire de solution Model, un projet ‘Bibliothèque de classe’ nommé ‘Models.Natif’ et une classe ‘Natif’. Modifier le namespace en MyAirport.Pim.Models

Cette classe sert uniquement en phase de test afin de vérifier le bon fonctionnement de notre architecture, les données y sont inscrites en dure directement dans le code, ce qui en fait un modèle plus que limité.

Ajouter les références à Entities et Models.AbstractDefinition

Faites dériver la classe Natif de la classe abstraite : MyAirport.Pim.Models.AbstractDefinition et implémenter les méthodes BagageDefinition.

namespace MyAirport.Pim.Models

{

public class Natif: AbstractDefinition

{

public override BagageDefinition GetBagage(int idBagage)

{

return new BagageDefinition() { CodeIata = "2452637", Compagnie = "LH" };

}

public override List<BagageDefinition> GetBagage(string codeIataBagage)

{

List<BagageDefinition> bags = new List<BagageDefinition>();

bags.Add(new BagageDefinition() { CodeIata = "2452637", Compagnie = "LH" });

return bags;

}

}

}

Ajouter un autre projet ‘Bibliothèque de classe’ nommé ‘Models.Factory’ et renommer la classe ‘Classe1’ en Factory.

Cette classe ne contient qu’une seule propriété publique et statique ‘Model’. Cette propriété est l’unique point d’accès aux données, accessible par le client. Si aucune source de données n’a déjà été créée, une instance implémentant le model AbstractDefinition sera automatiquement créée et retournée au client lors du premier à Model, c’est l’implémentation du pattern Singleton.

L’utilisation de la classe ConfigurationManager nécessite l’ajout d’une référence à la librairie : System.Configuration. La méthode AppSettings[« Factory »] permet la lecture de la balise Factory dans le fichier de configuration de l’application.

namespace MyAirport.Pim.Models

{

public static class Factory

{

private static AbstractDefinition singleton = null;

public static AbstractDefinition Model

{

get

{

if(singleton == null)

{

switch(ConfigurationManager.AppSettings["Factory"])

{

case "Sql":

// singleton = new MyAirport.Pim.Models.Sql();

break;

case "Natif":

singleton = new MyAirport.Pim.Models.Natif();

break;

default:

singleton = new MyAirport.Pim.Models.Natif();

break;

}

}

return singleton;

}

}

}

}

Ajouter à la solution un nouveau projet de type ‘Windows Forms Applcation’ Nommé ‘Client.FormIhm’.

Ce projet va nous servir à tester notre environnement, il sera remplacé par une application WPF, lors du prochain TP.

Dans un premier temps ajouter simplement un bouton à l’interface graphique et ajouter le code suivant en réaction à l’appui sur le bouton.

private void button1\_Click(object sender, EventArgs e)

{

var bagage = MyAirport.Pim.Models.Factory.Model.GetBagage("023232546100");

}

Ajouter dans le fichier app.config du projet Client.FormIhm les instructions suivantes :

<appSettings>

<add key="Factory" value="Natif"/>

</appSettings>

Cela permet la prise en compte du paramètre Factory utilisé dans la classe Factory.

Pensez à ajouter les références aux projets Models.AbstractDefinition, Models.Factory et Entities.

Placer un point d’arrêt sur la méthode BagagesDefiniton du projet MyAirport .Pim.Model.Natif et tester l’application.

Votre solution devrait ressembler à cela

