# TP3. Interface Client et Web Service.

## Objectifs :

* Créer l’interface cliente en fonction du cahier des charges
* Concevoir une application cliente (WindowsForm)
* Valider le fonctionnement du client en fonction du diagramme d’état.
* Affecter les actions utilisateur aux méthodes

## Déroulement :

### Fonctionnement de l’application.

Le bouton ‘nouveau bagage’ permet la réinitialisation de l’interface graphique (cela correspond dans un fonctionnement réel à l’arrivée d’un nouveau bagage sur le tapis).

Au début du cycle d’indentification seul la zone de saisie CodeIata est accessible à l’utilisateur l’ensemble des autres champs sont visibles mais non modifiable.

L’appui sur le bouton rechercher va appeler la méthode GetBagage(string CodeIata) via l’intermédiaire de la Factory, si un bagage est trouvé l’ensemble des informations du dit bagage sont affichées et les champs restent en lecture seul, le champ CodeIata compris.

Si la recherche ne trouve aucun bagage avec le code Iata demandé alors les champs deviennent accessibles en écriture permettant ainsi lors de l’appuie sur le bouton ‘créer’ de demander la création du bagage à la factory, dans cette état le champs CodeIata est non modifiable. L’interface affiche une boite de dialogue indiquant si l’action a bien été prise en compte ou non. Dans le cas où le bagage à bien été enregistré l’interface est réinitialisé. Sinon les champs de saisis sont réinitialisé exception du codeIata bien entendu.

Enfin si la recherche nous retourne plusieurs résultats, un popUp est alors présenté à l’utilisateur afin de lui permettre de sélection le bon bagage (cette fonctionnalité est pour ‘instant laissée de côté nous n’afficherons qu’une popUp pour indiquer que plusieurs bagages ont été trouvé).

L’application peut donc se trouver dans différents états de progression dans l’identification d’un bagage. Ces états sont ici au nombre de 5 (Déconnecté, AttenteBagage, SelectionBagage, CréationBagage et AffichageBagage)

### Création de la machine a état.

Afin de simplifier la gestion de l’interface graphique et de l’enchaînement des différents états nous allons créer une énumération que nous nommerons PimState, la classe PIM va implémenter une instance privée de cette énumération et une propriété qui servira à déclencher un événement à chaque modification de la valeur PimState. De cette façon nous pouvons centraliser la mise à jour de l’interface uniquement par changement de la variable d’état.

private PimState state = PimState.Deconnecter;

private PimState State

{

get { return this.state;}

set { OnPimStateChanged(value); }

}

La méthode OnPimStateChanged est pour l’instant indiquée en défaut car elle n’est pas encore implémentée.

Ajouter un nouvel élément PimState de type class au projet ClientPim.

public enum PimState

{

Deconnecter,

AttenteBagage,

SelectionBagage,

CreationBagage,

AffichageBagage

}

Dans la classe Pim créer un évènement PimStateChanged de type PimStateEventHandler, ajouter donc la déclaration du délégué PimStateEventHandler ayant pour prototype void en retour et comme premier argument un objet de type object et comme second argument un objet de type PimState. Ajouter ensuite la méthode permettant le déclenchement de l’évènement PimStateChanged, méthode nommée OnPimStateChanged.

public event PimStateEventHandler PimStateChanged;

public delegate void PimStateEventHandler(object sender, PimState state);

private void OnPimStateChanged(PimState newState)

{

if (newState != this.state)

{

this.state = newState;

if (this.PimStateChanged != null)

{

PimStateChanged(this, this.state);

}

}

}

Et enfin connecter la méthode Pim\_PimStateChanged du formulaire Pim à l’événement PimStateChanged

public PIM()

{

InitializeComponent();

this.PimStateChanged += PIM\_PimStateChanged;

}

void PIM\_PimStateChanged(object sender, PimState state)

{

MessageBox.Show("StateChanged");

}

### Description de l’interface client.

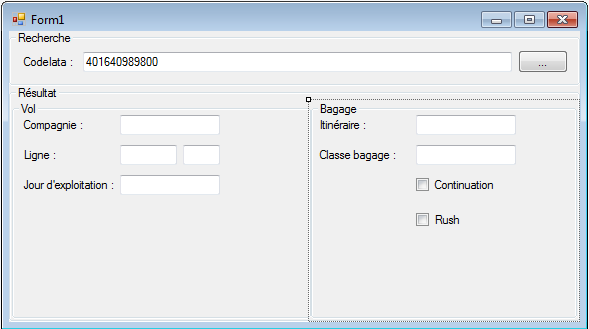
L’interface client de l’application doit nous permettre de rechercher un bagage via son code Iata et de visualiser le résultat, si une réponse a été trouvée.

Si aucune réponse n’est trouvée l’utilisateur est alors invité à saisir les informations nécessaires à la création du nouveau bagage.

Si plusieurs bagages sont trouvés l’utilisateur est invité à choisir le bagage auquel il souhaite associer sa saisie.

Il est donc nécessaire de créer un nouveau formulaire dans le projet Client.FormIHM, sélectionner un élélement du projet Client.FormIhm est faire CRT+SHIFT+A et choisir un élément de type WindowsForm et le nommer PIM.cs

, la capture ci-dessous présente une possibilité de présentation.



L’utilisation de groupBox permet de structurer l’interface en regroupant les éléments par thème, les groupBox peuvent ensuite être positionnés au bon vouloir dans l’interface via l’utilisation des propriétés ‘Dock’ et ‘Anchor’. L’utilisation de la propriété ‘TabIndex’ est d’une importance capitale dans l’ergonomie de l’application car elle permet de définir l’ordre de sélection des éléments graphiques via l’utilisation de la touche ‘TAB’.

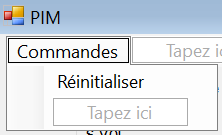
Dans l’exemple ci-dessus nous utilisons un menu (MenuStrip) qui nous permettra différentes commandes génériques comme la possibilité de réinitialiser l’interface utilisateur.

Et une barre de statut (status Strip) pour afficher différentes informations comme l’état de notre application.

Dans l’exemple nous utilisons également 3 groupBox pour organiser les informations bagages.

#### Le menu.

Nous allons pour l’instant ajouter une seule commande ‘Réinitialiser’ dans ce menu et elle sera rangée dans le sous menu ‘Commandes’.



La propriété ShortcutKeys vous permet de définir une combinaison de touche permettant l’exécution de la commande sans passer par le menu. Ici CTRL+R.

Un double click sur la commande permet de générer la méthode répondant à l’événement

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

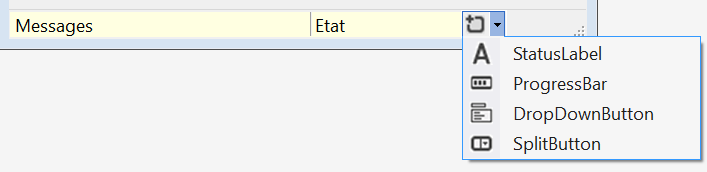
{

}

Nous y reviendrons plus tard.

#### La barre de statut.

La barre de statut est généralement positionnée dans le bas de la fenêtre on l’utilise pour indiquer l’état du system a l’utilisateur de manière non intrusive, on peut aussi l’utiliser pour indiquer ce que l’application est en train de faire pour des actions longues (connexion au réseau, traitement de données, requêtes en base, …). Ici nous l’utilisons pour indiquer les appels fait à la factory, ainsi que pour indiquer à quelle étape de traitement nous nous trouvons.

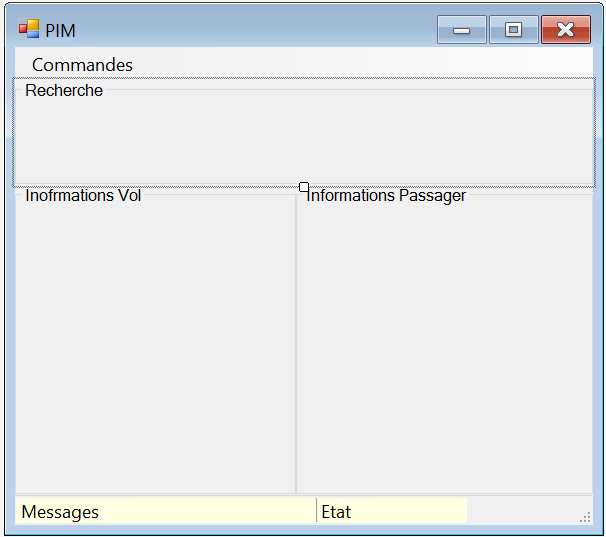


Les longueurs sont respectivement fixées à 300 et 150.

#### Les GroupBox.

3 groupes Box respectivement nommées respectivement Recherche, Informations Vol et Informations Bagages sont positionnée dans le formulaire. Recherche est positionné dans le haut du formulaire via la propriété Dock, Informations Vol est positionné à Gauche et Informations Bagage occupe le reste de l’espace.

Votre interface devrait ressembler à ceux-ci



Attention L’ordre de positionnement des éléments dans le formulaire à son importance car elle définit la priorité d’application de la propriété dock dans le formulaire. Vous pouvez donc avoir un positionnement qui semble chaotique. Dans ce cas il est nécessaire de modifier directement dans le fichier xxxx.Designer.cs l’ordre dans lequel les contrôles sont ajoutés au formulaire. Généralement à la fin de la méthode InitializeComponent(). Pour notre cas nous avons l’ordre suivant :

this.Controls.Add(this.groupBox3);

this.Controls.Add(this.groupBox2);

this.Controls.Add(this.groupBox1);

this.Controls.Add(this.statusStrip1);

this.Controls.Add(this.menuStrip1);

#### Autres composants.

Il ne reste plus qu’à ajouter différents label et autre textbox pour compléter l’affichage.

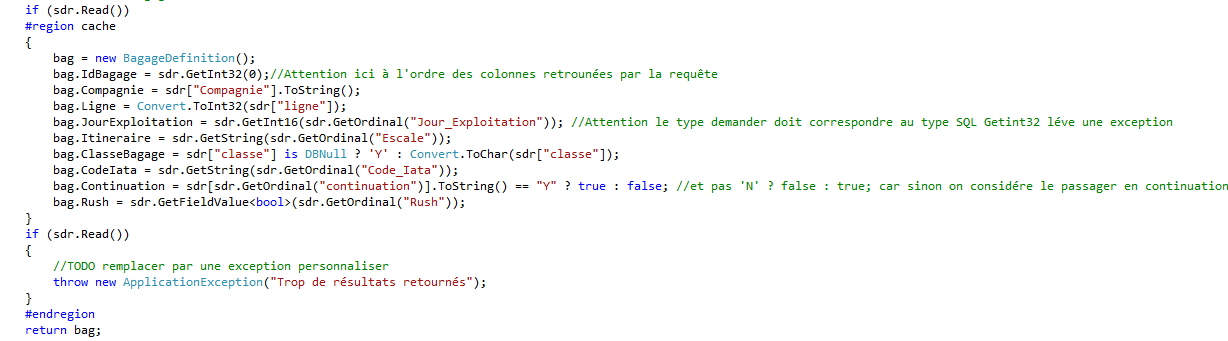
#### Gestion des états.

### Traitement des données (Rappel TP 2).

Il convient maintenant de convertir le résultat retourné par la requête SQL en un objet ‘BagageDefinition’ celui-ci étant le seul type de donnée compris par le client et commun à chacun des modèles. Puis dans un second temps il est nécessaire d’associer chacune des propriétés de l’objet à un élément de l’interface graphique. Il a été vu dans le précédent TP que le résultat de la requête SQL peut être récupéré via un objet de type ‘SqlDataReader’. Cet objet permet de lire ligne par ligne chacun des résultats via la méthode ‘read’ qui retourne un booléen indiquant s’il y a une ligne à lire. Il est ensuite possible de lire chacune des colonnes constituant cette ligne. Il existe une multitude de méthodes permettant la lecture de la valeur d’une colonne.

Dans la classe Sql implémentez la méthode GetBagage(string codeIata) afin de convertir la première ligne retournée par l’objet SqlDataReader en un objet BagageDefinition comme ci-dessous :

Notre requête peut retourner plusieurs bagages, mais notre interface n’accepte qu’un seul affichage, c’est pourquoi il est impératif de contrôler que la requête ne retourne qu’un seul bagage, en cas de présence d’autre résultat, une exception sera retournée à l’utilisateur.



Coté client il sera donc nécessaire de prévoir le traitement de cette exception.

### Affichage du résultat.

Le client ayant appelé la méthode GetBagage(string codeIata) va recevoir :

1. Un objet BagageDefinition contenant la description du bagage.
2. Un objet BagageDefinition valant null si aucun bagage ne correspond au code Iata demandé.
3. (Implémenté dans les prochains TP) Un objet ApplicationException dans le cas ou plusieurs bagages ont été trouvés en base.
4. (Implémenté dans les prochains TP) Un objet dérivé de la classe Exception pour tous les autres cas de disfonctionnement (base de données non disponible par exemple).

Il convient donc de traiter l’ensemble de ces cas coté client.

Pour le cas 1, il va être nécessaire d’associer chacun des objets graphiques à l’une des propriétés de l’objet BagageDefinition et de passer ces objets en lecture seul afin que l’utilisateur ne puisse pas modifier son contenu.

Pour les cas 2 et 3 l’utilisateur sera invité à saisir les informations de description du bagage. L’application appellera dans ce cas la méthode CreateBagage sur le serveur afin de créer un bagage dans le model de données.

Enfin dans le dernier cas l’utilisateur sera invité à reformuler sa demande ultérieurement.

Exemple possible de traitement.

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

{

try

{

var bagage2 = MyAirport.Pim.Model.Factory.Model.GetBagage(this.textBox1.Text);

this.tbAlpha.Text = bagage2.LigneAlpha.ToString();

this.tbAlpha.Enabled = false;

this.tbClasseBag.Text = bagage2.ClasseBagage.ToString();

this.tbClasseBag.Enabled = false;

this.tbCompagnie.Text = bagage2.Compagnie;

this.tbCompagnie.Enabled = false;

this.tbItineraire.Text = bagage2.Itineraire;

this.tbItineraire.Enabled = false;

this.tbJourExploitation.Text = bagage2.JourExploitation.ToString();

this.tbJourExploitation.Enabled = false;

this.tbLigne.Text = bagage2.Ligne.ToString();

this.tbLigne.Enabled = false;

this.cbContinuation.Checked = bagage2.Continuation;

this.cbContinuation.Enabled = false;

this.cbRush.Checked = bagage2.Rush;

this.cbRush.Enabled = false;

}

catch(ApplicationException appEx)

{

this.tbAlpha.Text = this.tbClasseBag.Text = this.tbCompagnie.Text = this.tbItineraire.Text = this.tbJourExploitation.Text = this.tbLigne.Text = "";

this.cbContinuation.Checked = this.cbRush.Checked = false;

this.tbAlpha.Enabled = this.tbClasseBag.Enabled = this.tbCompagnie.Enabled = this.tbItineraire.Enabled = this.tbJourExploitation.Enabled =

this.tbLigne.Enabled = this.cbContinuation.Enabled = this.cbRush.Enabled = true;

}

catch

{

MessageBox.Show("Une erreur s’est produite dans le traitement de votre demande.\nMerci de bien vouloir re tester ultérieurement ou de contacter votre administrateur.", "Erreur dans le traitement de votre demande", MessageBoxButtons.OK, MessageBoxIcon.Error);

}