FICHE DESCRIPTIVE DE LA PRATIQUE PROFESSIONNELLE

CORRESPONDANT A L'ACTIVITE TYPE N° 2 *(Voir le mode d'emploi)*

Intitulé de l'activité-type : CCP n°2 : Développer des composants d'interface, et la persistance des données.

1 - **Indiquez les résultats directs de votre action** : produits fabriqués, ouvrages, prestations de service ou autres productions que vous avez réalisés ou auxquels vous avez contribué :

Réalisation d'une application de gestion des vols pour une compagnie aérienne.

2 - **Décrivez les tâches et/ou opérations** que vous avez directement effectuées en vue des réalisations indiquées ci-dessus ainsi que leur fréquence de réalisation :

- Modélisation et création de la base de données ;
- Analyse des besoins et réalisation du cahier des charges correspondant ;
- Maquettage des IHM ;
- Codage d'une application fonctionnelle en Java 7 ;
- Interfaçage avec une base de données MySQL pour la persistance des données ;
- Réalisation d'un trigger ;
- Développement et mise en forme en conformité avec le cahier des charges défini ;
- Tests réalisés sur l'application pendant et après la phase de codage.

Fréquence de réalisation :

☐ Très fréquemment ☒ Fréquemment ☐ Rarement

3 - **Lieux où cette pratique professionnelle** a été exercée :

Nom de l'entreprise, organisme ou association	Lieu	Chantier, atelier, services ou autres (à préciser)
GRETA Paris Centre	Lycée Charles De Gaule 17 rue Ligner 75020 Paris	

4 - **Indiquez la période** de l'exercice de cette pratique professionnelle :

Du 10/09/2013 au 20/09/2013

5 - **Précisez les moyens** que vous avez utilisés pour accomplir les tâches décrites : matériels, outils, techniques, matériaux, produits, logiciels,… :

Environnements Linux (Debian et dérivées) et Windows (7)

Eclipse pour l'écriture du code

WindowBuilder, plugin d'Eclipse, pour le positionnement des éléments graphiques

Gestionnaire de versions Git

WinDesign pour le Modèle Conceptuel de données (MCD)

phpMyAdmin pour créer la base de données

Dia pour les diagrammes de cas d'utilisation dans le cahier des charges

6 - **Pour la réalisation de ces tâches ou opérations**, avez-vous travaillé seul ou en équipe, avec ou sans consignes, en relation avec d'autres personnes de votre entreprise ou extérieures à votre entreprise …) ? Si oui, précisez dans quelles circonstances :

J'ai réalisé cette application seule, dans le cadre de la formation de Développeur Logiciel au sein du GRETA. J'ai rédigé en autonomie un cahier des charges, que j'ai respecté. Je me suis imposée l'utilisation de la même base de données que pour le site web réalisé en PHP (cf activité type N°1), ainsi on peut par exemple créer un vol sur l'application en Java et le réserver sur le site.

7 - **Documents annexes**

7 A – Le cas échéant, documents prévus dans le règlement du titre visé :

7 B - Documents complémentaires en option **:** indiquez ici la liste des documents que vous souhaitez présenter au jury

Vous trouverez en annexe un rapport de projet qui contient notamment :

- le cahier des charges sur lequel l'application se base, qui lui-même contient :

  - un lexique des termes métiers utilisés,

  - une description du contexte, de l'objectif à atteindre et des contraintes,

  - des diagrammes de cas d'utilisation pour les fonctionnalités de gestion des aéroports et des vols,

  - les IHM de l'application,

  - les règles de gestion,

  - le Modèle Conceptuel de Données (MCD).

- une description des outils utilisés,

- l'arborescence des fichiers,

- une description du code source de l'application,

- des exemples de tests réalisés,

- une description du trigger utilisé.

Date : 22/09/2013

Vous pouvez ajouter une page complémentaire de commentaires.

RAPPORT DE PROJET

DEV-FLY (Java)

JULIE NUGUET

Table des matières

1) INTRODUCTION

Dans le cadre du Titre Professionnel Développeur Logiciel, j'ai réalisé en autonomie une application de gestion des vols pour une compagnie aérienne. Son but est de permettre à la compagnie DEV-FLY d'enregistrer, modifier ou supprimer les aéroports qu'elle utilise, d'enregistrer, modifier, ou supprimer des vols reliant ces aéroports, et d'affecter des employés aux différents vols prévus.

J'ai utilisé la même base de données que pour leur site web (dev-fly.fr), l'application en client léger (PHP) constituant le front office, et l'application en client lourd (Java) le back office. Ainsi, un vol créé par un employé est réservable en ligne par les clients !

Étant donné que le projet est un « exercice » (bien que réalisé dans un contexte professionnel, il ne concerne pas une réelle compagnie aérienne), j'ai décidé de m'imposer un contexte fictif. Ainsi, j'ai réalisé en amont un cahier des charges avec des exigences que je me suis efforcée de respecter. Le cahier des charges est présenté à la section suivante.

2) CAHIER DES CHARGES

Vous trouverez à la page suivante le cahier des charges de l'application, qui décrit les exigences et les besoins auxquels l'application répond.

CAHIER DES CHARGES
Application de gestion des vols pour la société DEV-FLY

Sommaire

Le présent cahier des charges a pour objet de définir les exigences et les besoins auxquels l'application de gestion des vols devra répondre.

1) Lexique

DEV-FLY est une compagnie aérienne, ce qui implique l'emploi de termes métiers propres à ce secteur. Voici un lexique du vocabulaire spécifique utilisé dans ce document :

Code AITA : code attribué par l'Association Internationale du Transport Aérien à un aéroport. Il est constitué de 3 lettres et désigne un aéroport unique (ex : BRN pour Berne ou SXB pour Strasbourg).

Copilote : second pilote affecté à un avion, travaillant conjointement avec le pilote. Sa présence est obligatoire sur tous les vols. Chaque copilote de DEV-FLY est identifié par un code employé unique commençant par la lettre « C ».

Équipe de vol : groupe d'employés travaillant sur un vol donné, composé d'un pilote, d'un copilote et de trois hôtesses ou stewards.

Hôtesse : une hôtesse de l'air, souvent abrégé en « hôtesse », a pour rôle d'accueillir les passagers, de les informer, et de s'assurer de leur confort et leur sécurité. On parle de « steward » lorsque ce poste est occupé par un homme. Chaque hôtesse ou steward de DEV-FLY est identifié(e) par un code employé unique commençant par la lettre « H ».

Pilote : désigne le commandant de bord, qui a la responsabilité de la navigation du vol. Il doit être titulaire d'une licence de pilote à jour. Chaque pilote de DEV-FLY est identifié par un code employé unique commençant par la lettre « P ».

Steward : voir hôtesse.

Valider un vol : faire passer un vol du statut de « en attente » à « programmé ».

Vol en attente : un vol est dit « en attente » tant qu'une équipe de vol complète ne lui est pas affectée. Les réservations ne sont pas possibles sur un vol « en attente ».

Vol programmé : un vol passe au statut « programmé » dès lors qu'une équipe de vol complète lui est affectée. Ce changement de statut ouvre la possibilité de réserver des billets sur ce vol.

2) Contexte

DEV-FLY est une filiale d'Air Greta France créée en décembre 2012. Elle est spécialisée dans le marché low-cost du transport de passagers. Implantée à Paris, elle emploie 978 salariés. Sa flotte est constituée de 126 avions Airfly E321 de 170 places qui desservent plus de 40 destinations.
Auparavant, Air Greta France ne gérait pas les vols low-costs. C'est pour se faire une place sur ce marché porteur qu'elle a décidé de créer DEV-FLY. En effet, les vols low-costs devraient s'accroître de 50% d'ici 2020, et Air Greta France souhaite prendre part à cette croissance.

Jusqu'à mi-2013, la filiale DEV-FLY avait recours aux logiciels de sa maison-mère. Aujourd'hui, elle souhaite développer son propre système d'information, qui sera notamment composé d'une application web en client léger côté front office, et d'une application en client lourd côté back office. C'est dans ce

contexte que son nouveau site web (www.dev-fly.fr) a été inauguré en milieu d'année, et que le service des vols demande à présent la réalisation d'une application de gestion des vols.

3) Objectif

L'application de gestion des vols devra couvrir :

- la consultation, l'ajout, la modification et la suppression des aéroports utilisés par la compagnie. Pour chaque aéroport seront renseignés le code aéroport dit « AITA », la ville, et le pays.

- la consultation, l'ajout, la modification et la suppression d'un vol dit « en attente » (ce statut sera maintenu jusqu'à ce qu'une équipe de vol lui soit affectée). Tous les vols devront être en partance et en provenance des aéroports précités. Les éléments suivants devront être visibles pour chaque vol enregistré : aéroports de départ et d'arrivée, dates et heures de départ et d'arrivée, durée du vol, tarif, employés éventuellement affectés au vol. Tous les horaires sont indiqués en heure française.

- la possibilité d'affecter des employés sur un vol « en attente » en vue de le faire passer au statut de « vol programmé » (changement effectif dès l'affectation d'une équipe complète : 1 pilote, 1 copilote, et 3 hôtesses et / ou stewards). Les réservations des clients ne sont ouvertes que sur les vols « programmés ».

- la possibilité de modifier le tarif d'un vol « programmé », ou de supprimer le vol (si aucune réservation n'a été faite dessus).

Le tout prendra la forme d'une application tournant en local et reliée à la base de données de la compagnie. Le programme sera installé sur le poste des employés habilités à l'utiliser.

Remarques :

- Le prix du billet sur un vol est modifiable jusqu'à la veille du départ.

- En raison du temps de préparation d'un vol, un vol enregistré peut partir au plus tôt le lendemain de la date d'enregistrement.

- Il est possible qu'un vol ait un tarif nul (= à zéro) pour des événements particuliers.

- Pour des raisons de coûts logistiques, la compagnie n'utilise qu'un seul aéroport pour une même ville.

- Un vol a un numéro unique (ex : le Paris - Tunis du 02/03 n'aura pas le même numéro que le Paris – Tunis du 26/03).

4) Contraintes

Les contraintes suivantes doivent être prises en compte :

- Le projet sera réalisé par un seul développeur. Celui-ci sera entièrement dédié au projet, du 10 au 20 septembre 2013.

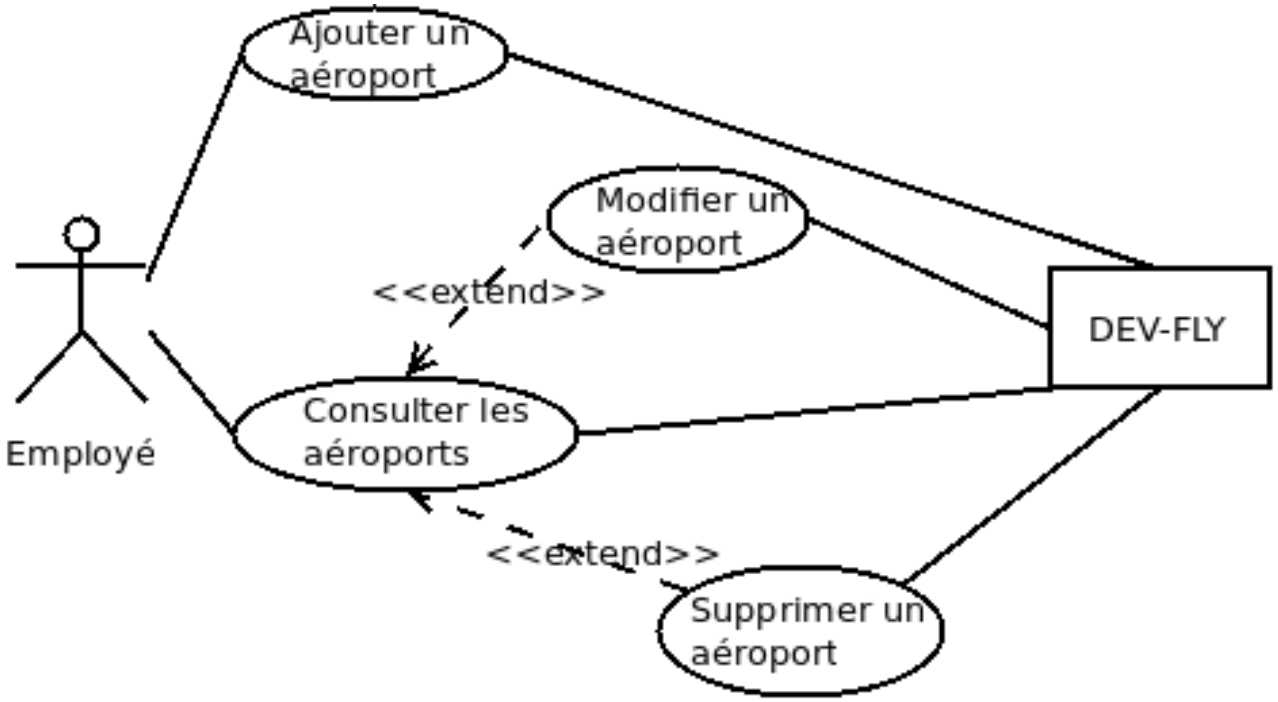
- Hormis pour la couleur de police, les couleurs de l'application devront se restreindre au bleu et / ou blanc et / ou rouge, afin de respecter la charte graphique de la société. Le logo de DEV-FLY devra être visible au lancement de l'application.

- Le projet devra être réalisé en Java version 7 (ou 1.7), qui est la version stable actuellement (septembre 2013).

5) Fonctionnalités

a) Gestion des aéroports

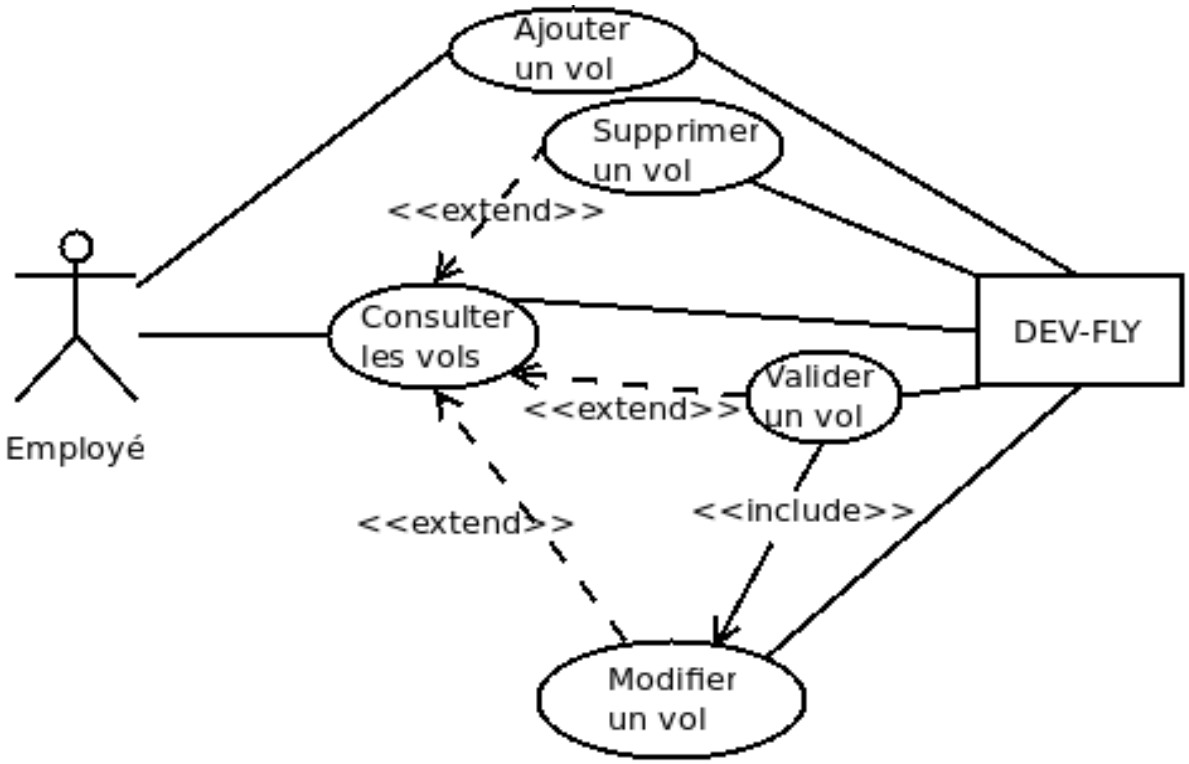
La fonctionnalité de gestion des aéroports est représentée au moyen du diagramme de cas d'utilisation suivant :



L'employé peut directement ajouter un aéroport. Il accède également directement à la liste des aéroports. Depuis cette liste, il peut ensuite modifier ou supprimer un aéroport.

b) Gestion des vols

La fonctionnalité de gestion des vols est représentée au moyen du diagramme de cas d'utilisation suivant :



L'employé peut directement ajouter un vol. Il accède également directement à la liste des vols. Depuis cette liste, il peut ensuite modifier ou supprimer un vol. Il peut aussi valider un vol (dans le cas d'un vol en attente), ce qui implique alors obligatoirement la modification du vol, en occurrence en lui affectant une équipe.

6) IHM & règles de gestion

Voici une présentation d'une utilisation standard de l'application. Pour chaque élément du scénario décrit sont présentées :

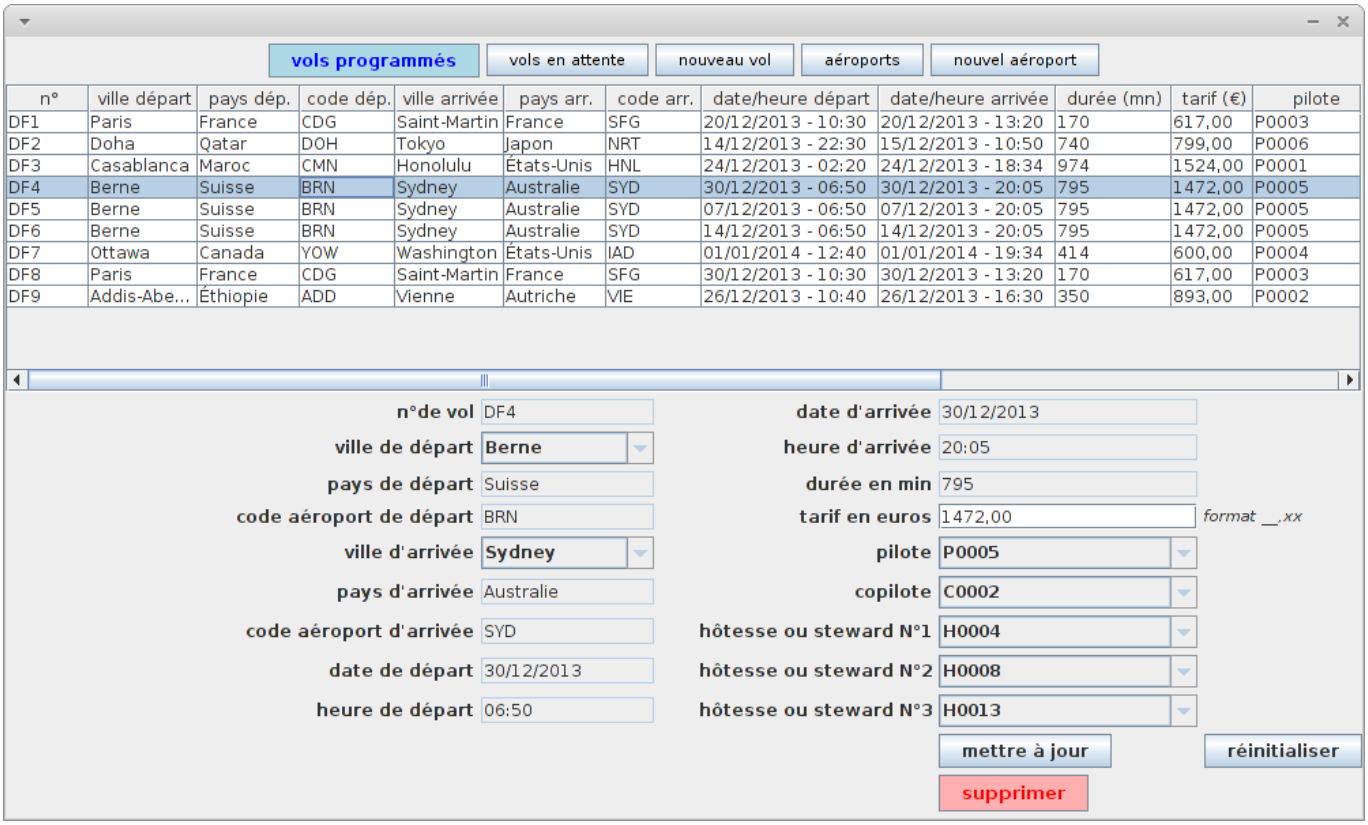
- la vue correspondante
- les règles de gestion.

a) Vue « bienvenue »

Au lancement de l'application, l'employé arrive sur une page de présentation, qui contient le logo de la compagnie et l'adresse de son site web. Il clique sur la page pour accéder à la vue suivante.

Règles de gestion : aucune ici.

b) Vue « vols programmés »



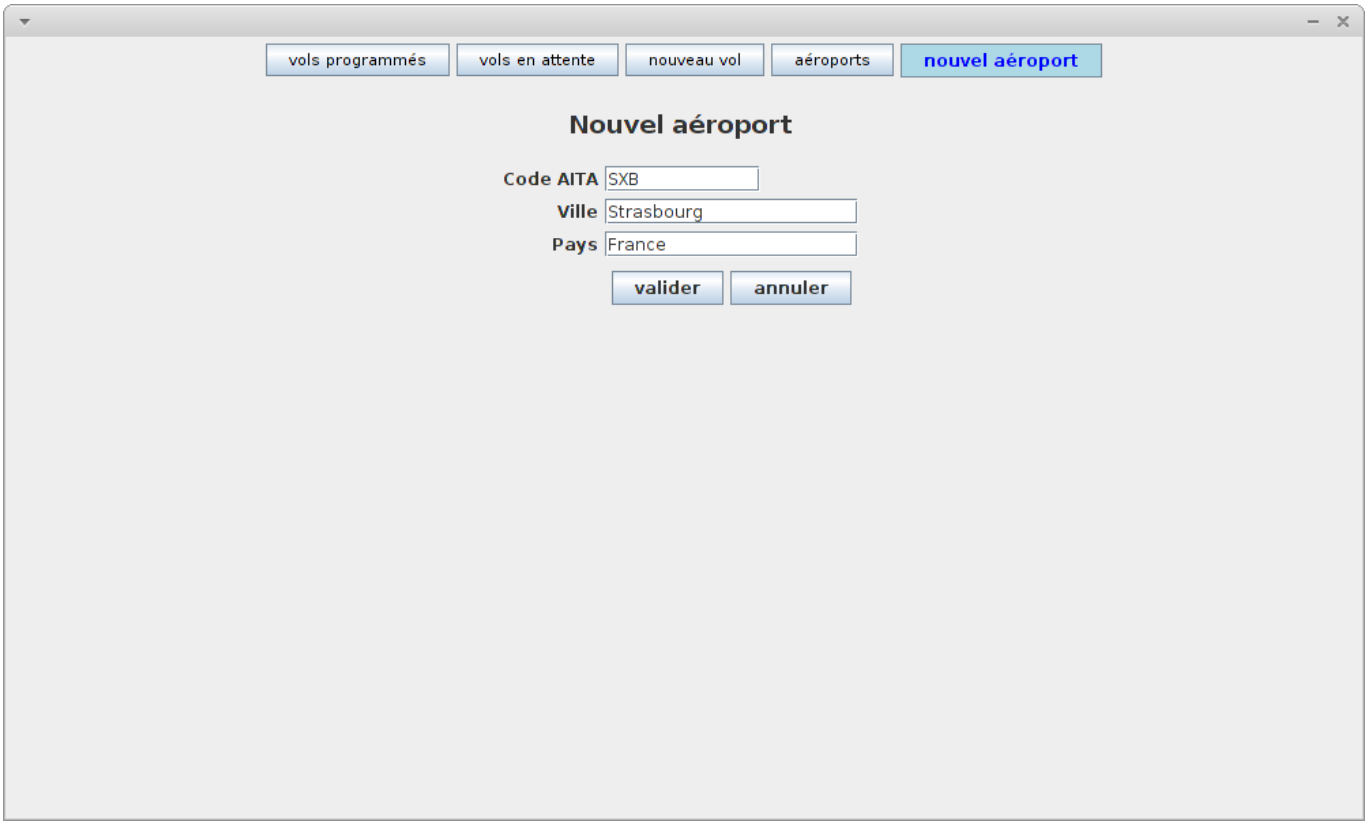
Après un clic sur la page « d'accueil », il arrive sur la vue ci-dessus. Elle est également disponible en cliquant sur « vols programmés » dans le menu en haut. Elle présente le récapitulatif des vols « programmés ». Au clic sur l'un des vols dans le tableau du haut, le détail de ce vol apparaît en bas. Ici, l'utilisateur choisit le vol « DF4 ». Il lui est alors possible de modifier son prix, ou de supprimer le vol. Pour rétablir le tarif original, l'utilisateur clique sur le bouton « réinitialiser ».

Règles de gestion :

- pour la modification du tarif : le nouveau prix indiqué doit être au format valide et supérieur ou égal à zéro euros (pas de tarif négatif !). Le vol pour lequel le prix est modifié doit partir au plus tôt le lendemain (pas de modification de tarif sur un vol passé...).

- la suppression d'un vol n'est possible que si aucune réservation n'a été effectuée sur ce vol.

c) Vue « nouvel aéroport »



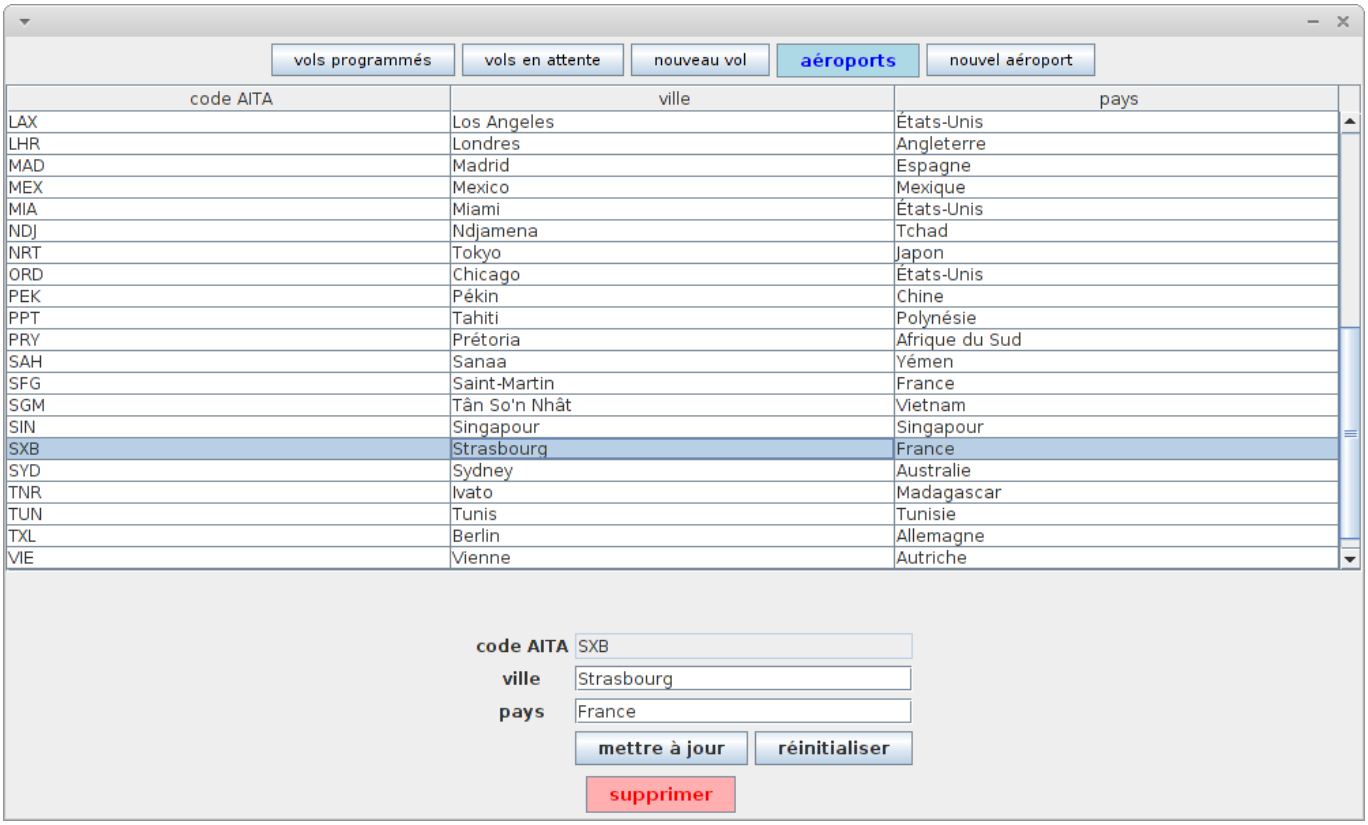
Cette vue s'affiche lorsque l'employé clique sur « nouvel aéroport » dans le menu en haut. Il peut y renseigner un nouvel aéroport en indiquant son code AITA, sa ville et son pays, et en validant. Il pourra effacer les champs en cliquant sur le bouton « annuler ». Ici, par exemple, il renseigne l'aéroport de Strasbourg en France, dont le code AITA est SXB.

Règles de gestion :

- les 3 champs doivent être remplis.
- la ville et le pays doivent être constitués de lettres (accentuées ou non) et éventuellement de traits d'union.
- la ville ne doit pas déjà exister en base.
- le code AITA doit être composé de 3 lettres, et ne doit pas déjà exister en base.

Remarque : le code AITA est automatiquement passé en majuscules, tout comme la première lettre des villes et pays, pour une meilleure cohérence entre les données.

d) Vue « aéroports »

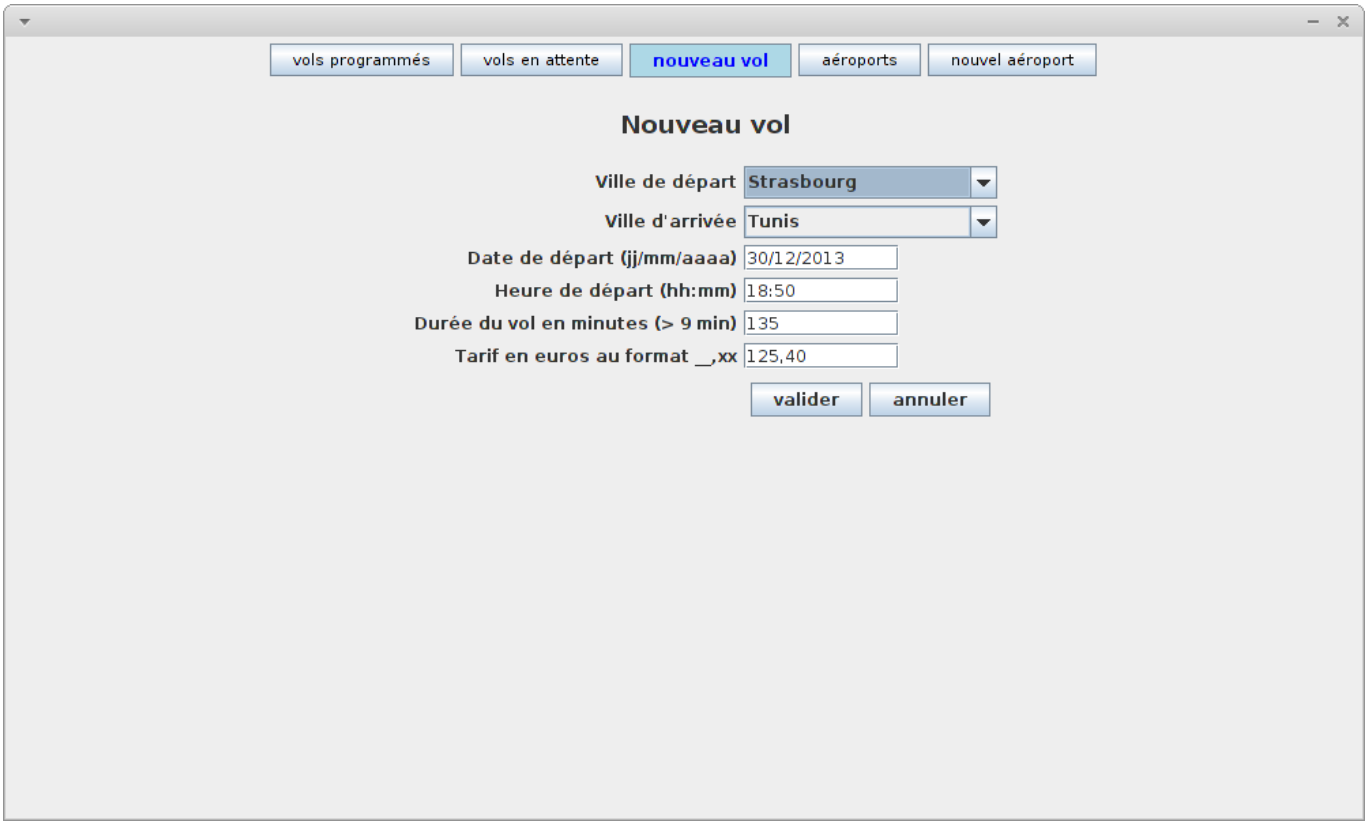


Cette vue s'affiche lorsque l'employé clique sur « aéroports » dans le menu en haut. Il y trouve la liste des aéroports enregistrés. L'aéroport de Strasbourg (SXB) enregistré à la vue précédente est directement visible ici. Au clic sur l'aéroport dans le tableau du haut, son détail apparaît en bas. Il peut alors être modifié ou supprimé. L'employé clique sur « réinitialiser » pour rétablir les informations d'origine.

Règles de gestion :

- règles identiques à l'ajout d'un nouvel aéroport.
- un aéroport déjà utilisé pour un vol ne peut plus être modifié ou supprimé.

e) Vue « nouveau vol »

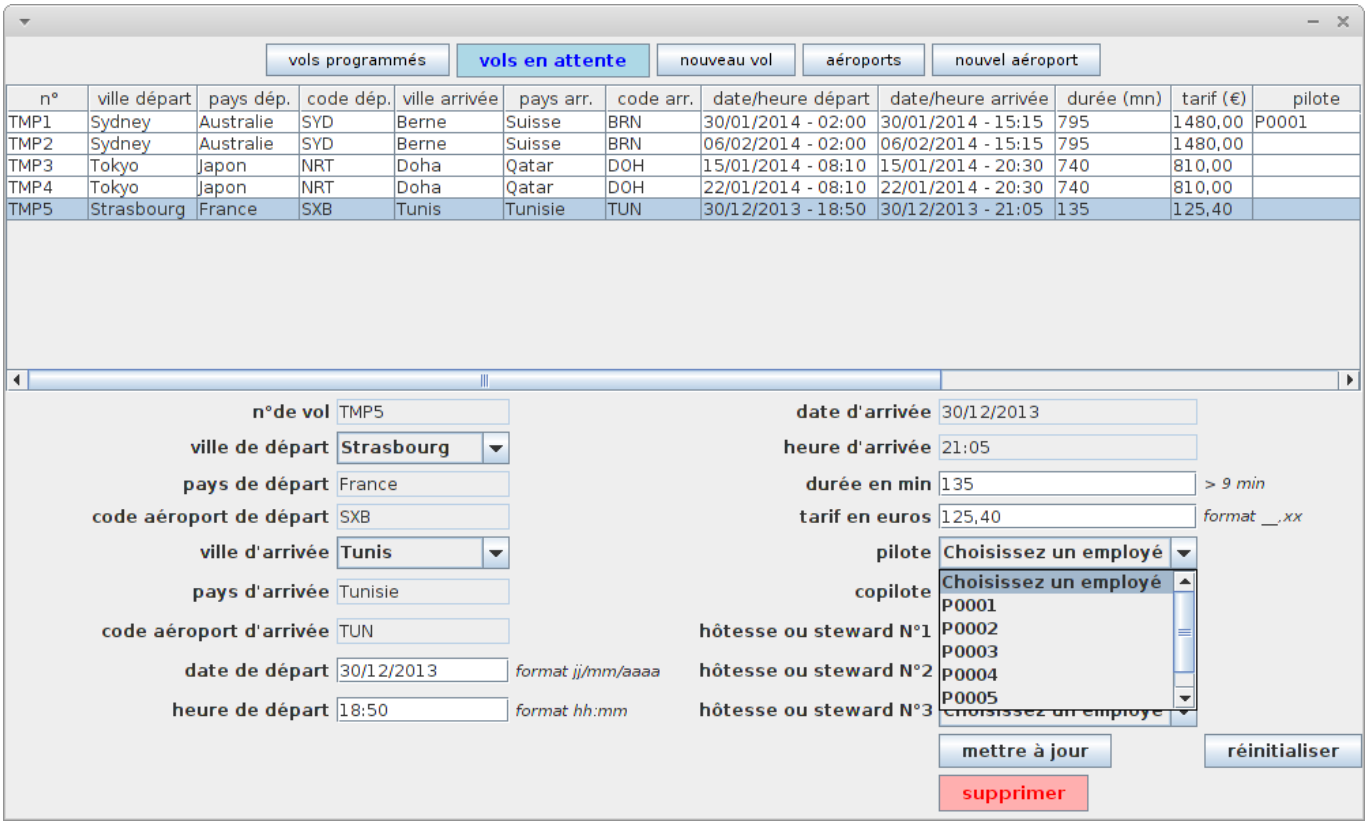


Cette vue s'affiche lorsque l'employé clique sur « nouveau vol » dans le menu en haut. Elle lui permet de créer un nouveau vol « en attente ». Les villes de départ et d'arrivée sont proposées via des listes déroulantes : elles correspondent aux aéroports enregistrés par la compagnie. Si un aéroport vient d'être enregistré ou modifié, la ville correspondante apparaît, c'est le cas ici pour la ville de « Strasbourg ». Un clic sur le bouton « annuler » efface les champs. L'employé crée ici un vol qui va de Strasbourg à Tunis.

Règles de gestion :

- les villes de départ et d'arrivée, prévues par la compagnie, doivent être différentes.
- la date et l'heure de départ doivent être au format valide.
- la date de départ ne peut pas être antérieure au lendemain.
- la durée du vol (en minutes) doit être indiquée en chiffres, et ne peut être inférieure à 10 min.
- le tarif a un format décimal (la virgule et le point sont acceptés comme séparateurs), et ne peut pas être négatif.

f) Vue « vols en attente »



Cette vue s'affiche lorsque l'employé clique sur « vols en attente » dans le menu en haut. Elle lui permet de visualiser la liste des vols en attente enregistrés. Il y retrouve le vol Strasbourg-Tunis qu'il vient d'enregistrer. Il clique sur le vol dans le tableau du haut afin de faire apparaitre son détail en bas. Il peut alors le modifier, le supprimer, ou le valider en affectant une équipe de vol complète dessus (le vol disparaîtra alors de la liste et sera immédiatement visible dans la liste des vols programmés). S'il clique sur le bouton « réinitialiser », il rétablit la valeur d'origine des champs pour le vol en cours de modification.

Remarques :

- le choix d'une nouvelle ville de départ ou d'arrivée dans la liste déroulante modifie automatiquement le pays et le code aéroport en fonction.

- si la durée du vol en minutes est modifiée, la date et l'heure d'arrivée sont effacées et seront recalculées à la validation.

- les employés à affecter sur le vol (pilotes / copilotes / hôtesses / stewards) sont proposés depuis une liste déroulante dans laquelle les codes des employés de la compagnie sont listés.

- il est tout à fait possible de ne pas affecter une équipe de vol en une fois. Les employés déjà affectés sont enregistrés et le vol garde le statut « en attente ». Par ailleurs, leur affectation peut être modifiée et n'est effective qu'une fois le vol validé.

- Si une équipe de vol complète est affectée au vol, il passe au statut « programmé ».

Règles de gestion :

- règles identiques à l'ajout d'un nouveau vol.

- les 3 hôtesses / stewards affectés au vol doivent être différents.

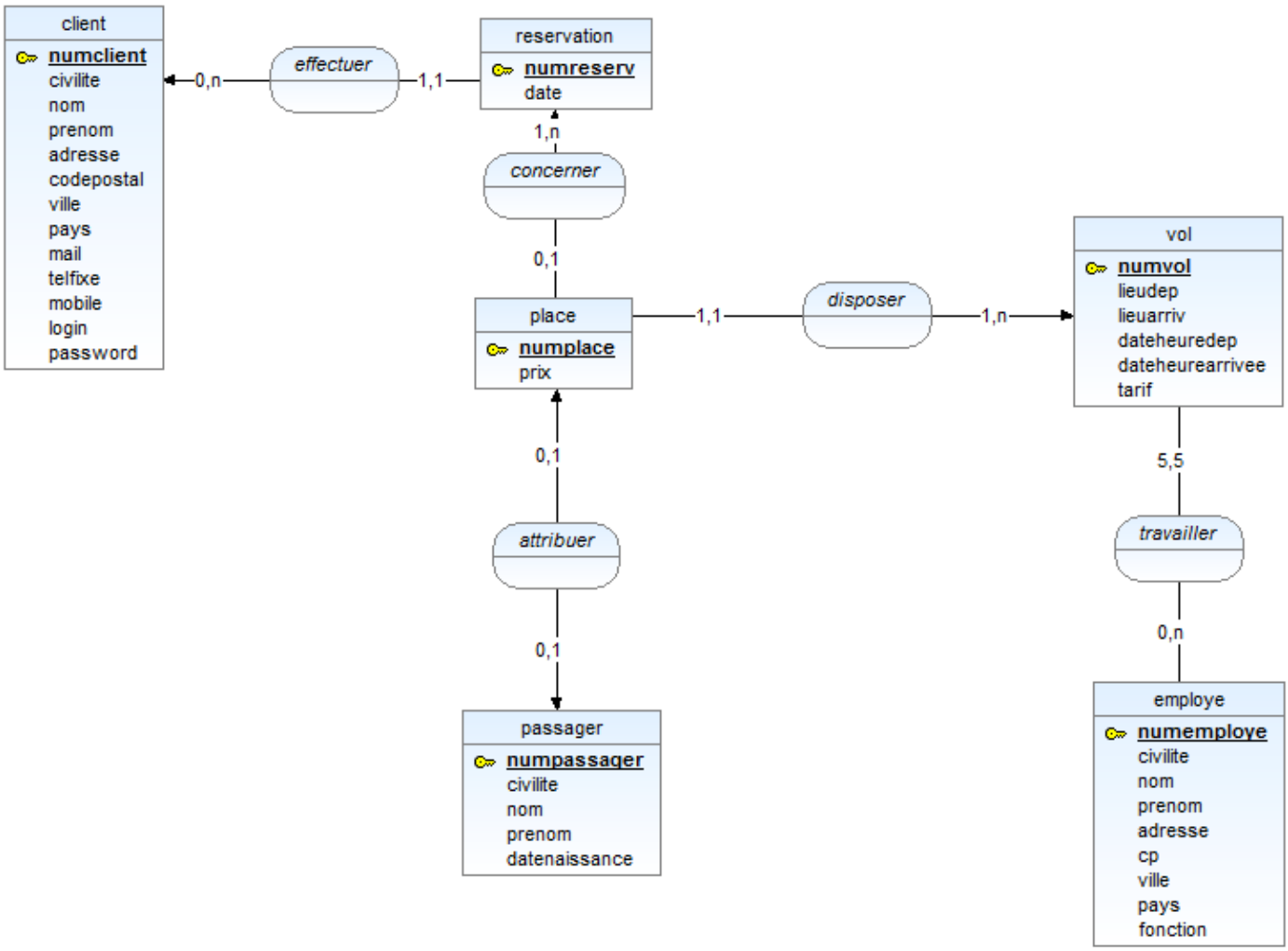
g) Remarques générales

- tout au long de la procédure, des messages sont affichés pour l'utilisateur, notamment pour l'informer que l'opération a bien été effectuée, ou pour lui demander de modifier une information saisie pour se conformer aux règles de gestions établies.

- les opérations peuvent être effectuées de manière indépendante. Ainsi, il n'est évidemment pas nécessaire par exemple de créer un nouvel aéroport avant de créer un vol, ou de créer un vol avant de pouvoir en modifier un autre !

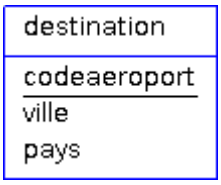
7) Modèle Conceptuel de Données

L'application s'est reliée au modèle de données existant de la compagnie, représenté page suivante selon la méthode Merise. Ainsi, les données sont liées : sur le site web dev-fly.fr, on pourra réserver un vol créé depuis l'application, ou voir quelle équipe de vol lui est affectée...



Pour les besoins de l'application, 2 tables supplémentaires ont été ajoutées :

- la table « destination » où sont stockés les différents aéroports enregistrés :



- la table « vol-tmp » où sont stockées les informations sur les vols en attente :

3) OUTILS UTILISÉS

J'ai réalisé ce site en partie dans un environnement **Linux** (Debian et dérivées), qui offre un panel d'outils adaptés à la programmation. J'ai également travaillé sous **Windows** 7, qui est le système d'exploitation utilisé sur mon lieu de formation. Le langage de programmation utilisé est **Java** 7 (ou 1.7).

Afin de mener à bien ce projet, j'ai utilisé :

- **Eclipse**, environnement de développement utilisé pour l'écriture du code, qui présente l'avantage de fonctionner de manière similaire sur les systèmes d'exploitation Unix et Windows,

- **WindowBuilder**, éditeur graphique qui est un plugin d'Eclipse, utilisé pour le positionnement des éléments graphiques,

- le gestionnaire de versions **Git**, afin de garder un historique du travail produit, et de gérer un backup des données,

- **WinDesign** pour réaliser le Modèle Conceptuel de Données,

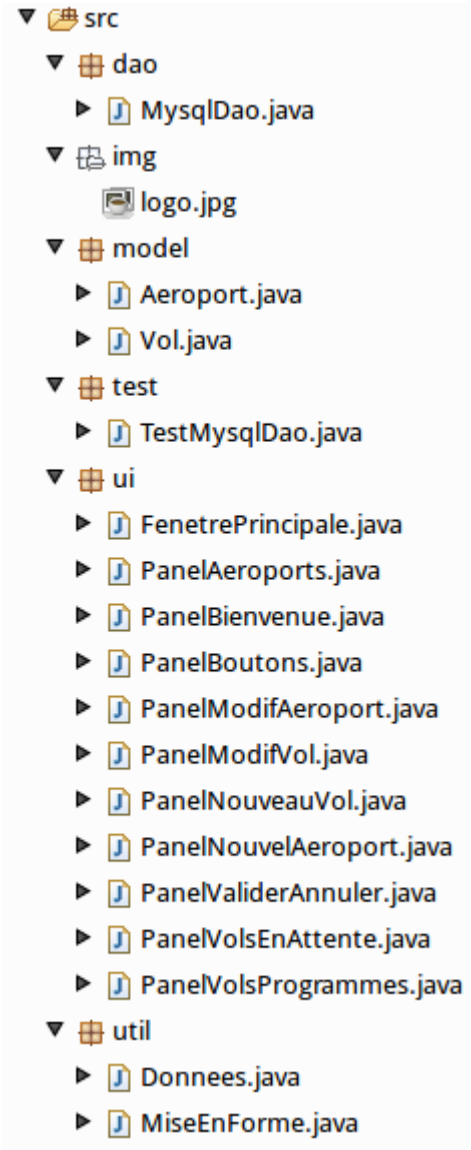
- **phpMyAdmin** pour créer et gérer la base de données (MySQL),

- **Dia**, pour la création des diagrammes de cas d'utilisation dans le cahier des charges.

4) ARBORESCENCE

Le code source de l'application a été organisé de manière logique, et séparé dans différents packages, comme on peut le voir sur l'arborescence.

(page suivante)



- « **dao** », pour Data Access Object (= objet d'accès aux données), contient le fichier MysqlDao.java dans lequel sont regroupées les méthodes qui « attaquent » la base de données pour en extraire de l'information utilisable.

- « **img** » contient l'image utilisée dans l'application, en occurrence le logo de DEV-FLY.

- « **model** » est utilisé pour les objets métiers, on y retrouve les classes « Vol » et « Aeroport ». Elles vont permettre de créer au sein de l'application un ensemble d'objets partageant les mêmes méthodes et attributs. Par exemple, chaque vol possède un lieu de départ et un lieu d'arrivée.

- « **test** » contient des tests qui ont été réalisés au fur et à mesure du développement, lors de la phase de programmation. Des exemples sont donnés à la section « tests » du présent rapport.

- « **ui** », pour User Interface (= interface utilisateur), contient les différents éléments graphiques avec lesquels l'utilisateur interagit. Il regroupe la frame principale (nommée FenetrePrincipale) qui contient le « main », c'est-à-dire le point d'entrée dans l'application, et les différents panels qui viennent s'y greffer.

- « **util** » regroupe des méthodes dites « utilitaires ». Ce sont des méthodes statiques utilisées à plusieurs reprises dans le code, par différentes classes (par exemple une méthode pour savoir si une date est dans le futur ou non).

5) CODE SOURCE DE L'APPLICATION

Voici quelques **extraits** du code source de l'application.

J'ai fait en sorte de commenter tout le code que j'ai produit, afin de faciliter sa compréhension par toute personne ayant à le consulter.

A) DAO

Quelques méthodes issues du « dao » (Data Access Object) sont présentées ici.

On voit ici la méthode du dao qui permet de mettre à jour un vol « en attente » :

```
// met à jour le vol "en attente" en paramètre. Renvoie "vrai" si la mise à jour s'est bien passée
public boolean updateVolEnAttente(Vol v) throws SQLException {
    // on se connecte à la BDD
    Connection connection = DriverManager.getConnection(datasource,user,password);
    // requête SQL pour mettre à jour le vol
    String sql = "UPDATE vol_tmp SET lieudep=?, lieuarriv=?, dateheuredep=?, dateheurearrivee=?,"
            + "tarif=?, pilote=?, copilote=?, hotesse_steward1=?, hotesse_steward2=?,"
            + "hotesse_steward3=? WHERE numvol=?";
    PreparedStatement stmt = connection.prepareStatement(sql);
    // On valorise les paramètres
    stmt.setString(1, v.getAeroportDepart().getVille());
    stmt.setString(2, v.getAeroportArrivee().getVille());
    // on transforme la date util en timestamp SQL (on utilise getTime() pour récupérer
    // le timestamp de la date util).
    // (rq : avec une java.sql.Date, on ne récupèrerait pas les heures et minutes)
    stmt.setTimestamp(3, new java.sql.Timestamp(v.getDateHeureDepart().getTime()));
    stmt.setTimestamp(4, new java.sql.Timestamp(v.getDateHeureArrivee().getTime()));
    stmt.setFloat(5, v.getTarif());
    stmt.setString(6, v.getCodePilote());
    stmt.setString(7, v.getCodeCopilote());
    stmt.setString(8, v.getCodeHotesseSt1());
    stmt.setString(9, v.getCodeHotesseSt2());
    stmt.setString(10, v.getCodeHotesseSt3());
    stmt.setString(11, v.getId());

    int result = stmt.executeUpdate(); // retourne le nb d'enregistrements impactés
    connection.close();
    if(result == 1){ // la mise à jour s'est bien passée
        return true;
    }
    return false;
}
```

J'ai fait en sorte, comme pour de nombreuses méthodes du dao, que la valeur de retour soit un booléen. Cela permet d'agir différemment selon que l'action se soit bien passée ou non (afficher un message ou poursuivre).

On commence par se connecter à la base de données (à la toute fin, on fermera la connexion avec connection.close()).

La requête SQL utilisée est une requête « UPDATE » pour mettre à jour la table vol_tmp avec les nouvelles données. On a recours à une requête préparée pour se protéger des injections SQL, d'où le « PreparedStatement », puis la valorisation des paramètres.

Pour ces derniers, on utilise les données de l'objet de type Vol passé en paramètre de la méthode (on utilise les getters définis dans la classe Vol pour les obtenir, par exemple V.getTarif() donne le tarif du vol V en paramètre).
Pour récupérer les villes de départ et d'arrivée, on doit d'abord récupérer les aéroports (ex : v.getAeroportDepart()), puis les villes depuis les objets « Aeroport » obtenus.
Pour les dates, on utilise un Timestamp afin de récupérer les données complètes de la date, et pas uniquement le jour, le mois, et l'année.

Une fois les paramètres valorisés, on utilise executeUpdate() pour exécuter la requête de mise à jour. Cette méthode renvoie le nombre d'enregistrements impactés. Logiquement, la modification d'un vol impacte une ligne. Si c'est bien le cas, la méthode peut donc renvoyer « vrai ».

Ci-dessous, cette méthode du dao supprime un vol programmé :

```
// Supprime le vol "programmé" dont le code est passé en paramètre si il n'y a pas
// de réservation dessus. Renvoie vrai si la suppression s'est bien passée
public boolean deleteVolProgramme(String numVol) throws SQLException {
    // on se connecte à la BDD
    Connection connection = DriverManager.getConnection(datasource,user,password);
    // on vérifie au préalable qu'aucune réservation n'a été faite sur le vol
    // si c'est le cas, on renvoie "false" et on ne supprime pas le vol
    if(volReserve(numVol)){
        return false;
    }
    // requête SQL pour supprimer le lien entre les employés et le vol à supprimer :
    // (rappel : il y a forcément des employés affectés sur un vol "programmé")
    String sql1 = "DELETE FROM travailler WHERE vol = ?";
    // requête SQL pour supprimer le vol
    String sql2 = "DELETE FROM vol WHERE numvol = ?";
    // on prépare les 2 requêtes
    PreparedStatement stmt1 = connection.prepareStatement(sql1);
    PreparedStatement stmt2 = connection.prepareStatement(sql2);
    // on valorise le paramètre pour les 2 requêtes
    stmt1.setString(1, numVol);
    stmt2.setString(1, numVol);
    if(stmt1.executeUpdate() == 0){ // renvoie le nb de lignes impactées.
        connection.close();
        return false; // Si aucune, il y a eu un pb, on renvoie false.
    }
    if(stmt2.executeUpdate() == 0){ // renvoie le nb de lignes impactées.
        connection.close();
        return false; // Si aucune, il y a eu un pb, on renvoie false.
    }
    connection.close();
    return true; // la suppression s'est bien passée
}
```

Les vols programmés ne sont supprimables qu'en l'absence de réservation dessus. On commence donc par vérifier cela à l'aide de la méthode volReserve() décrite plus loin. On arrête ici et on renvoie « faux » si une réservation est trouvée.

On va ensuite supprimer l'association entre les employés et le vol à supprimer, en éliminant l'entrée correspondante dans la table « travailler ». On se sert une nouvelle fois du fait qu'executeUpdate() renvoie le nombre de lignes impactées pour vérifier si tout s'est bien passé. Un vol programmé a, par définition,

forcément une équipe de vol affectée. Si executeUpdate() renvoie zéro, on sait qu'il y a eu un problème., dans ce cas on ne poursuit pas.

Sinon, on peut continuer en supprimant cette fois le vol (procéder dans l'autre sens n'aurait pas été logique, et n'aurait d'ailleurs pas fonctionné du fait que la clé primaire du vol est utilisée en tant que clé étrangère dans la table « travailler »). Là encore, on renverra « false » en cas de problème.

Si tout s'est bien déroulé, on renvoie « true ».

Voici ici la méthode volReserve() évoquée précédemment. Elle permet de savoir si (au moins) une réservation a été effectuée sur le vol dont le numéro est en paramètre :

```
// renvoie vrai s'il y a au moins une réservation sur le vol dont l'id est en paramètre
public boolean volReserve(String numVol) throws SQLException{
    // on se connecte à la BDD
    Connection connection = DriverManager.getConnection(datasource,user,password);
    // on cherche une réservation sur le vol
    String sql = "SELECT numreservation FROM place WHERE numvol = ?";
    PreparedStatement stmt = connection.prepareStatement(sql);
    // on valorise le paramètre
    stmt.setString(1, numVol);
    // On exécute la requête :
    ResultSet result = stmt.executeQuery();
    // si au moins une réservation a été trouvée, on renvoie vrai.
    if (result.next()) {
        connection.close();
        return true;
    }
    connection.close();
    return false;
}
```

On utilise tout simplement une requête « SELECT » pour chercher les réservations correspondant au vol. A partir du moment où un résultat est trouvé, la condition « if(result.next()) » est remplie, on peut renvoyer « true ». Sinon, on renvoie « false ».

Tout comme « volReserve() », d'autres sous-méthodes sont utilisées par différentes méthodes de l'application. C'est le cas par exemple de « getNextIdVolTmp() » qui donne la valeur de l'ID du prochain vol « en attente » à insérer en base (page suivante).

```java
// renvoie le prochain ID à insérer dans la table vol_tmp
public String getNextIdVolTmp() throws SQLException{
    // on se connecte à la BDD
    Connection connection = DriverManager.getConnection(datasource, user, password);
    // On récupère les numvol de la table vol_tmp.
    String sql = "SELECT numvol FROM vol_tmp";
    PreparedStatement stmt = connection.prepareStatement(sql);
    ResultSet result = stmt.executeQuery();
    // On va chercher l'id max de la table. On initialise idMax à zéro.
    int idMax = 0;
    // On parcourt les résultats de la requête.
    while (result.next()) {
        // On ne prend que la fin de la chaîne. Ex : pour le vol "TMP12", on veut récupérer "12".
        // On récupère donc la chaine à partir du 4ème caractère (on enlève "TMP")
        String numvol = result.getString("numvol").substring(3);

        // On transforme la chaîne récupérée en int
        int nb = Integer.parseInt(numvol);

        // On récupère la plus grande valeur de la liste
        if(nb > idMax){
            idMax = nb;
        }
    };
    // le prochain ID à insérer correspondra à l'idMax + 1
    int prochainId = idMax + 1;
    String prochainIdString = "TMP" + prochainId; // on ajoute le préfixe "TMP"
    connection.close();
    return prochainIdString;
}
```

Les vols en attente ont des identifiants constitués de TMP + un nombre (ex : TMP42).
On va d'abord récupérer les identifiants dans leur ensemble, puis on va en extraire uniquement la partie « numérique » grâce à « substring() ». La valeur récupérée sera transformée en entier avec « Integer.parseInt() ». On pourra alors comparer les valeurs récupérées entre elles et récupérer la plus élevée.
On pourra enfin lui ajouter 1, et lui redonner le préfixe « TMP ».

B) VÉRIFICATIONS

Dans les panels, l'utilisateur est amené à renseigner des données (les informations sur l'aéroport ou le vol qu'il souhaite créer ou modifier…). Avant d'appeler les méthodes du dao correspondantes, des vérifications sont faites pour s'assurer que les saisies dans les champs sont conformes à ce qui est attendu.

On voit ici par exemple des extraits des vérifications effectuées lors de la modification d'un vol en attente. Les commentaires au sein du code expliquent la démarche.

```java
// On récupère au préalable le code du vol en cours de modification
// pour s'assurer qu'un vol est bien sélectionné
String id = panelModifVolEnAttente.getTextFieldNdeVol().getText();
if(id.isEmpty()){
    // Si aucun vol n'est sélectionné, on affiche un message et on ne continue pas
    panelModifVolEnAttente.getLblMessage().setText("Vous devez sélectionner un vol ci-dessus !");
```

Ici, on prévoit le fait qu'un utilisateur puisse cliquer sur « mettre à jour » sans avoir sélectionné un vol au préalable.

```
// On initialise un booléen à vrai. Dès lors qu'un critère n'est pas rempli,
// on le passe à faux. C'est lui qui déterminera si la mise à jour peut se faire.
boolean miseAJour = true;
```

```
// On définit les formats voulus pour la date, l'heure, la durée du vol :
String regexDate = "^(0[1-9]|1[0-9]|2[0-9]|30|31)/(0[1-9]|1[0-2])/[0-9]{4}";
String regexHeure = "^([0-1][0-9]|2[0-3]):[0-5][0-9]$";
String regexDuree = "^[1-9][0-9]+$"; // la durée du vol ne peut pas être inférieure à 10 min
// Le tarif est un nombre décimal (rq : on laisse la possibilité à la compagnie d'indiquer
// un tarif à zéro pour les événements particuliers).
String regexTarif = "^[0-9]+\\.[0-9]{2}$";

// On vérifie que les villes de départ et d'arrivée sont différentes
if(villeDepart.equals(villeArrivee)){
    panelModifVolEnAttente.getLblMessage().setText("Le trajet indiqué n'est pas correct !");
    miseAJour = false;
}
// On vérifie que la date est au bon format ET dans le futur
if(!dateDepart.matches(regexDate) || !util.Donnees.futureDate(dateDepart)){
    // (on ne vérifie que la date est dans le futur que si elle a un format valide)
    panelModifVolEnAttente.getLblMessage().setText("<html><p>Vérifiez le format de la date svp.
            + "Attention, la date ne peut pas être antérieure à demain !</p></html>");
    miseAJour = false;
}
// On vérifie que l'heure est au bon format
if(!heureDepart.matches(regexHeure)){
    panelModifVolEnAttente.getLblMessage().setText("Vérifiez le format de l'heure svp !");
    miseAJour = false;
}
// On vérifie que la durée est OK
if(!duree.matches(regexDuree)){
    panelModifVolEnAttente.getLblMessage().setText("Vérifiez la durée du vol svp !");
    miseAJour = false;
}
```

Ci-dessus, on utilise des expressions régulières pour définir précisément le format attendu pour chaque donnée. Par exemple, pour la durée, on veut un nombre au moins égal à 10. On a donc [1-9] pour le premier chiffre de 1 à 9, suivi d'un autre chiffre de 0 à 9 [0-9] au moins une fois (« + »). Ainsi, la saisie « 9 » ou « 09 », par exemple, ne sera pas acceptée.

Les expressions régulières sont comparées aux données récupérées grâce à la méthode « matches ».
Pour la date, on vérifie également qu'elle n'est pas antérieure au lendemain. On utilise pour cela la méthode futureDate() codée dans le package « util ».

```
// On vérifie que les 3 hôtesses / stewards sélectionnés sont différents.
// La vérification se fait uniquement si l'employé n'a pas la valeur "Choisissez un employé".
String choixEmploye = "Choisissez un employé";
if((!codeHotesseSt1.equals(choixEmploye) && codeHotesseSt1.equals(codeHotesseSt2)) ||
        (!codeHotesseSt1.equals(choixEmploye) && codeHotesseSt1.equals(codeHotesseSt3)) ||
        (!codeHotesseSt2.equals(choixEmploye) && codeHotesseSt2.equals(codeHotesseSt3))){
    panelModifVolEnAttente.getLblMessage().setText("Vous devez choisir des hôtesses ou"
            + "stewards différents.");
    miseAJour = false;
}
```

On peut affecter des hôtesses et stewards sur un vol, mais ils doivent être différents. La subtilité ici est que leur valeur peut en fait être identique dans un cas précis : s'ils valent « Choisissez un employé », ce qui correspond en fait à une absence de choix. C'est pourquoi on ne vérifie que les chaînes de caractères sont équivalentes (grâce à la méthode equals()) uniquement si un employé a été sélectionné.

```
// On concatène la date et l'heure de départ
String dateHeureDepart = dateDepart + " " + heureDepart;

// On tranforme le résultat de String en Date
Date dateDeDepart = null;
try {
    dateDeDepart = new SimpleDateFormat("dd/MM/yyyy HH:mm").parse(dateHeureDepart);
} catch (ParseException e1) {
    panelModifVolEnAttente.getLblMessage().setText(e1.getMessage());
}

// On transforme la durée récupérée en int
int dureeInt = Integer.parseInt(duree); // en minutes

// pour calculer la date d'arrivée, on convertit la date de départ en timestamp
// et la durée en millisecondes, et on les additionne
long departMillisecondes = dateDeDepart.getTime();
long dureeMillisecondes = dureeInt * 60_000;

long arriveeMillisecondes = departMillisecondes + dureeMillisecondes;
// On transforme le long obtenu en Timestamp
Timestamp dateDArrivee = new Timestamp(arriveeMillisecondes);

// On transforme le tarif récupéré en float
float tarifFloat = Float.parseFloat(tarif);

// on crée un objet Vol avec toutes les données récupérées
Vol vol = new Vol(id, aeroportDepart, aeroportArrivee, dateDeDepart, dateDArrivee, dureeInt,
        tarifFloat, pilote, copilote, hotesseSt1, hotesseSt2, hotesseSt3);
```

On a récupéré les données qui nous intéressent. Dans l'extrait de code ci-dessus, on va faire en sorte de passer les données récupérées au bon format.

La date et l'heure sont renseignées séparément dans le formulaire, on commence donc par les concaténer pour avoir une donnée unique, qu'on va ensuite passer au format voulu grâce à SimpleDateFormat().

Les données récupérées du formulaire sont des chaînes de caractères, ce qui n'est pas le format attendu pour toutes les données. C'est pourquoi on va utiliser Integer.parseInt() sur la durée, et Float.parseFloat() sur le tarif pour les transformer respectivement en nombre entier et en nombre flottant.

Concernant la date d'arrivée, elle est calculée en récupérant le timestamp de la date de départ (en millisecondes) et en lui ajoutant la durée du vol également en millisecondes. Le « long » obtenu est ensuite passé en paramètre de la méthode Timestamp() pour obtenir le timestamp de la date d'arrivée.

Une fois toutes les données récupérées, on crée un objet Vol avec.

```
// on modifie le vol "en attente"
try {
    if(dao.updateVolEnAttente(vol)){ // renvoie vrai si la mise à jour s'est bien passée
        panelModifVolEnAttente.getLblMessage().setText("Le vol " + id + " a bien été mis à jour !");

        // Si tous les employés sont renseignés, on passe le vol de "vol en attente" à
        // "vol programmé". Les données du vol "en attente" fraichement modifié sont
        // utilisées par un TRIGGER qui va l'identifier comme étant le vol "temporaire" à
        // supprimer lors de l'ajout en base du vol "programmé" correspondant.
        if(!pilote.isEmpty() && !copilote.isEmpty() && !hotesseSt1.isEmpty() &&
                !hotesseSt2.isEmpty() && !hotesseSt3.isEmpty()){
            try {
                if(dao.confirmVol(vol)){ // renvoie vrai si ça s'est bien passé
                    panelModifVolEnAttente.getLblMessage().setText("Le vol a bien été validé !");
```

L'objet Vol précédemment créé est passé en paramètre de la méthode du dao « updateVolEnAttente() » qui met à jour le vol en attente.
On vérifie ensuite si une équipe de vol complète lui a été affectée, si c'est le cas, on appelle la méthode « confirmVol() » qui va créer le vol programmé correspondant. Un trigger, dont le détail est visible à la section « trigger » du présent document, se charge de supprimer le vol en attente correspondant.

C) AUTRES

Quelques autres parties du code sont décrites ici.

On voit ici le processus pour faire en sorte qu'au clic sur une ligne d'un tableau, les données correspondantes soient visibles dans le formulaire en bas de page :

```
tableAeroports = new JTable();
tableAeroports.addMouseListener(new MouseAdapter() {
    @Override
    public void mouseClicked(MouseEvent arg0) {
        // on récupère l'endroit où a eu lieu l'événement (= le clic)
        Point p = arg0.getPoint();
        int row = tableAeroports.rowAtPoint(p); // renvoie la ligne sous le point
        // On convertit les row du tableau en row du modèle pour maintenir la cohérence
        // entre les cellules de la présentation et les cellules du model (source de données)
        int modelRow = tableAeroports.convertRowIndexToModel(row);
        TableModel model = tableAeroports.getModel();
        // String qui représentent les valeurs récupérées :
        String code = (String) model.getValueAt(modelRow, 0);
        String ville = (String) model.getValueAt(modelRow, 1);
        String pays = (String) model.getValueAt(modelRow, 2);

        // On place les valeurs récupérées dans les champs du formulaire
        panelModifAeroport.getTextFieldCode().setText(code);
        panelModifAeroport.getTextFieldVille().setText(ville);
        panelModifAeroport.getTextFieldPays().setText(pays);

        // on supprime le message éventuellement saisi
        panelModifAeroport.getLblMessage().setText("");
    }
});
```

Avec « addMouseListener() », on ajoute un évènement sur le tableau « tableAeroports » lors d'un clic de souris. On utilise ensuite une classe interne anonyme, plus concrètement cela signifie que le MouseAdapter passé en paramètre est directement créé et utilisé ici, au lieu d'être créé ailleurs et appelé depuis ce panel. Le code est exécuté directement mais il ne peut pas être utilisé ailleurs dans le code.

On récupère d'abord la position du pointeur de la souris, puis la ligne sous le pointeur. On s'assure ensuite de la cohérence entre les cellules de présentation (du tableau) et les cellules du « model » (la source de données), en convertissant les lignes du tableau en ligne du model associé.

On récupère le tout sous forme de chaînes de caractères qui est le format attendu par les textfields (champs textes) dans lesquels on va insérer le texte récupéré grâce à la méthode setText().
C'est également cette méthode qui est utilisée pour effacer l'éventuel message d'annonce ou d'erreur affiché précédemment, afin d'avoir une page « propre ».

Voici pour finir un exemple de méthode « statique » qu'on trouve dans le package « util » (pour « utilitaires »). Le principe est qu'on peut l'appeler de n'importe où sans avoir à instancier la classe correspondante.

```
// prend en paramètres un tableau de String (villes, codes employés...) et une JComboBox
// insère les villes / codes employés(...) dans la comboBox
public static void comboBoxCreation(String[]donnees, JComboBox<String> maComboBox){
    // on donne le tableau de données au model :
    DefaultComboBoxModel<String>model = new DefaultComboBoxModel<>(donnees);
    // on ajoute le model à la combobox :
    maComboBox.setModel(model);
    // on pourra faire défiler les données avec la molette de la souris :
    maComboBox.setMaximumRowCount(6); // 6 données visibles à chaque fois
}
```

On voit comment créer une comboBox, c'est-à-dire une liste déroulante, avec un tableau de chaînes de caractères passé en paramètre. L'intérêt d'isoler ces quelques lignes dans une méthode est d'éviter d'avoir à les répéter dans le code. En effet, de nombreuses comboBoxes sont crées avec, pour lister les villes de départ, d'arrivée, les codes employés… le tout sur différents panels.

6) TESTS

Cette section décrit brièvement le fichier TestMysqlDao.java dans le package « test ».
Y sont rassemblés les tests réalisés tout au long du développement. L'idée était de vérifier, dès codage d'une méthode dans le dao, que celle-ci renvoyait bien le résultat voulu, avant de l'utiliser dans l'application.

On utilise JUnit pour les tests, ce qui explique la présence de cette ligne…

```
@RunWith(JUnit4.class)
```

… et des imports suivants :

```
import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.JUnit4;
```

Ci-dessous, on vérifie que la méthode getAllAeroports() renvoie un résultat cohérent et conforme aux attentes, c'est-à-dire la liste des aéroports en base.

```
@Test
public void getAllAeroports() throws SQLException{ // doit retourner une liste des aéroports
    MysqlDao dao = new MysqlDao();
    List<Aeroport>aeroports = dao.getAllAeroports();
    // on peut tester que le nombre d'aéroports retournés correspond bien
    // au nombre d'aéroports en base (à réajuster au fur et à mesure) :
    Assert.assertEquals(40, aeroports.size());
    // on teste que le premier élément de la liste est bien une instance d'Aeroport
    Assert.assertTrue(aeroports.get(0) instanceof Aeroport);
}
```

Chaque méthode du fichier commence par @Test, ce qui signifie que le test sur cette méthode sera lancé. Si la ligne « @Test » est commentée, le test est ignoré (il suffit de décommenter la ligne et de relancer les tests pour que le test soit réalisé).
On appelle la méthode du dao getAllAeroports() et on stocke le tout dans une variable « aeroports ».
Les tests se basent sur des assertions, c'est-à-dire qu'on va vérifier si des expressions sont vraies. On compte par exemple le nombre d'aéroports en base - 40 à l'instant T- , et on va vérifier que la taille de la liste « aeroports » obtenue est bien égale aussi à 40. Si ce n'est pas le cas, on sait qu'il y a un problème. Bien évidemment, ce nombre est à ajuster avant de relancer un test, si le nombre d'aéroports en base a évolué depuis.
De la même façon, on sait qu'on doit récupérer une liste d'objets « Aeroport », on vérifie donc que le premier élément de la liste (choix arbitraire) est bien une instance d'Aeroport.

Ci-dessous, on vérifie que la méthode de mise à jour des vols en attente fonctionne.

```
@Test
public void updateVolEnAttente() throws Exception{
    MysqlDao dao = new MysqlDao();
    // On peut le modifier avant de relancer un test :
    Date dateDepart = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").parse("2014-04-25 04:00:00");
    Date dateArrivee = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").parse("2014-04-25 17:15:00");

    Aeroport aeroportDepart = dao.getAeroportByVille("Casablanca");
    Aeroport aeroportArrivee = dao.getAeroportByVille("Honolulu");
    // à réajuster à chaque test :
    Vol volTest1 = new Vol("TMP1", aeroportDepart, aeroportArrivee, dateDepart, dateArrivee, 795, 1000, "P0001",
            "C0006", "H0002", "", "");
    Vol volTest2 = new Vol("TMP42", aeroportDepart, aeroportArrivee, dateDepart, dateArrivee, 795, 1000, "P0001",
            "C0006", "H0002", "", "");
    boolean result1 = dao.updateVolEnAttente(volTest1);
    boolean result2 = dao.updateVolEnAttente(volTest2); // n'existe pas, ne peut pas être modifié
    Vol v1 = dao.getVolEnAtteenteById("TMP1"); // on récupère le vol modifié
    Assert.assertTrue(result1); // true
    Assert.assertFalse(result2); // false
    Assert.assertEquals(volTest1.getCodePilote(), v1.getCodePilote()); // les codes pilotes coïncident
}
```

On crée 2 vols, le premier est modifiable (on lui a passé des données cohérentes et son identifiant existe en base), le second n'est pas modifiable (son identifiant n'existe pas en base).
On vérifie grâce à Assert.assertTrue() et Assert.assertFalse() que la méthode renvoie bien « vrai » avec le premier vol et « faux » avec le second.
Pour le premier vol, on va également s'assurer que les données ont bel et bien été modifiées. On récupère dans une variable « v1 » l'objet Vol qui correspond au vol fraîchement modifié en base.
Avec assertEquals, on vérifie ensuite l'égalité entre le code pilote (choix arbitraire) du premier vol et celui du vol v1.

7) TRIGGER

Un trigger est utilisé lors de la validation d'un vol, c'est-à-dire son passage du statut « vol en attente » (stocké dans la table vol_tmp) au statut « vol programmé » (stocké dans la table vol).

Concrètement, lorsqu'une insertion est faite dans la table vol (= un nouveau vol programmé), on va supprimer le vol « en attente » correspondant, c'est-à-dire l'entrée correspondante dans la table vol_tmp.

```
DROP TRIGGER IF EXISTS suppr_vol_tmp;
-- on passe le delimiter à $$ le temps de la requête
DELIMITER $$
CREATE TRIGGER suppr_vol_tmp
    BEFORE INSERT ON vol FOR EACH ROW
    BEGIN
        -- on supprime de la table vol_tmp un vol qui a exactement les mêmes critères
        -- que le nouveau vol inséré dans la table vol
        DELETE FROM vol_tmp WHERE lieudep = NEW.lieudep AND lieuarriv = NEW.lieuarriv
        AND dateheuredep = NEW.dateheuredep AND dateheurearrivee = NEW.dateheurearrivee
        AND tarif = NEW.tarif;
    END$$
-- on rétablit le point-virgule comme delimiter
DELIMITER ;
```

On constate qu'on a passé le délimiteur à $$ le temps de la requête. En effet, le trigger contient une commande à effectuer (en occurrence DELETE), qui se termine par un point-virgule. Si le point-virgule était maintenu en tant que délimiteur, l'instruction « CREATE TRIGGER » aurait pris fin au mot « tarif ; » et serait donc incomplète.

Les données NEW.xxx correspondent aux données du nouveau vol programmé, qui va être inséré en base.

8) CONCLUSION

Cette application a été un projet intéressant à réaliser, et m'a permis d'avoir une approche concrète d'un développement en Java.

J'ai constaté avec plaisir que j'ai pu réaliser une application fonctionnelle en peu de temps. Les exigences du cahier des charges ont été respectées, et le rendu obtenu est conforme à ce qui était attendu.

Bien sûr, il y aurait encore des points à améliorer. Pour l'affectation des employés sur un vol, on pourrait par exemple ne proposer que les employés disponibles à la date du vol. On pourrait également faire en sorte que la durée d'un vol soit calculée automatiquement en fonction du trajet choisi, plutôt que d'être renseignée à la main.

Davantage de fonctionnalités auraient pu être implémentées si le temps consacré au projet (10 jours) avait été plus long, mais cette application constitue malgré tout une très bonne première expérience du langage Java !

9) WEBOGRAPHIE

J'ai consulté différents sites web pour m'aider dans ce projet, parmi lesquels :

La documentation officielle du langage Java, en anglais
http://docs.oracle.com/javase/7/docs/api/

Developpez.com, qui offre des réponses sur des questions de programmation
http://www.developpez.com

Wikipédia, pour des informations diverses, sur l'UML par exemple
http://fr.wikipedia.org