



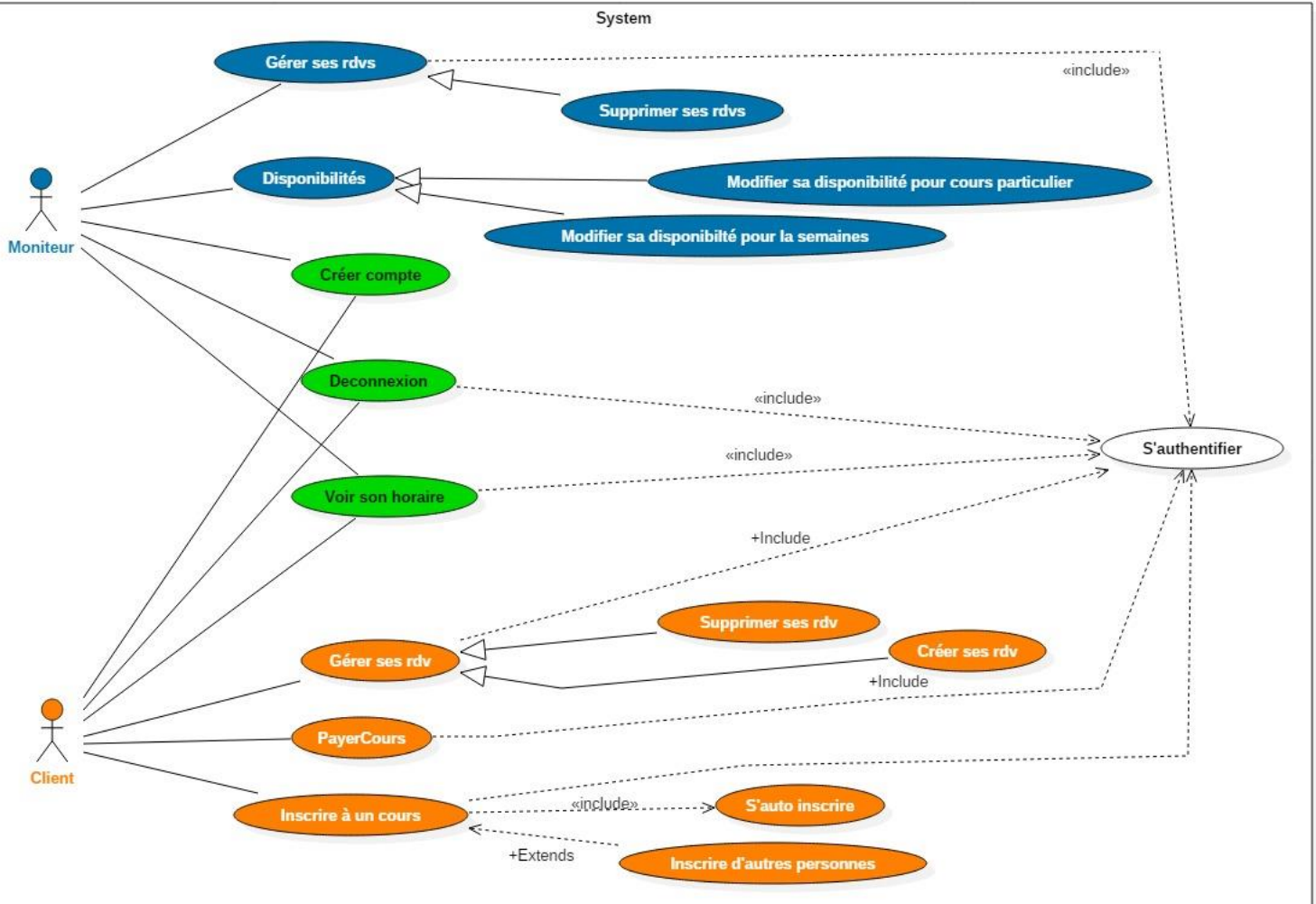
# PROJET JAVA STATION DE SKI

## Résumé

Application visant à gérer les réservations pour des cours collectifs ou particuliers dans une station de ski.

Adrien Mousty  
moustyadrien@hotmail.com

# 1 DIAGRAMME DE CLASSE



## 2 DECOUPE DU PROJET - PACKAGES

Afin de respecter le DAO ainsi que les principes de la programmation-objet, il a fallu séparer le projet en 3 parties. De plus, j'ai ajouté un 4ème package, « utilitaire » ainsi qu'un 5ème package contenant toute la partie visible par l'utilisateur du programme.

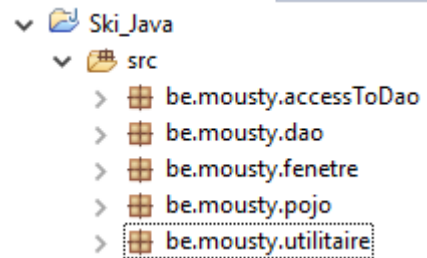
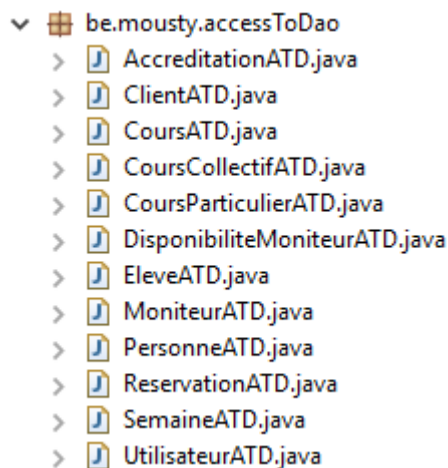


Figure 2 Découpe du projet

### 2.1 ACCESS TO DAO

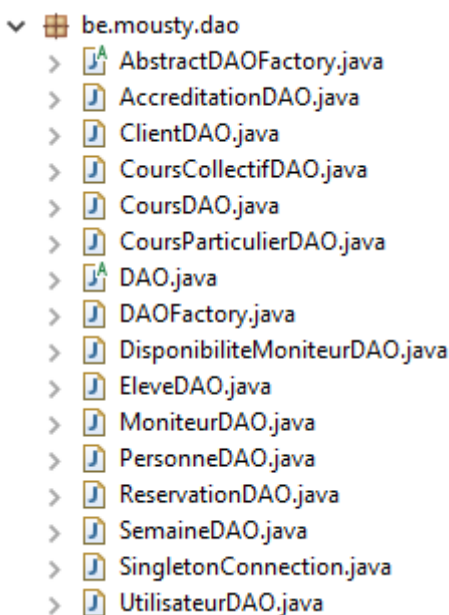


Les classes ATD sont en fait les classes métier.

Les classes ATD déclenchent un traitement du DAO via le polymorphisme. Cela permet d'ajouter une couche d'abstraction et de le rendre indépendant au type de base de données.

Figure 3 Classe métier

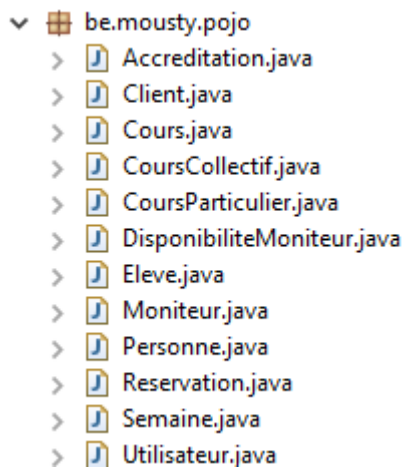
### 2.2 DAO



Le modèle DAO propose de regrouper les accès aux données persistantes dans des classes à part. Il n'y aura donc que dans les classes « DAO » que nous allons accéder aux données de la base de données.

Figure 4 DAO

## 2.3 POJO<sup>1</sup>



Les classes POJOs sont manipulées dans les classes DAO. Elles représentent l'état de la base de données. On peut donc y insérer les IDs, ce qui ne peut pas être fait dans les classes métier.

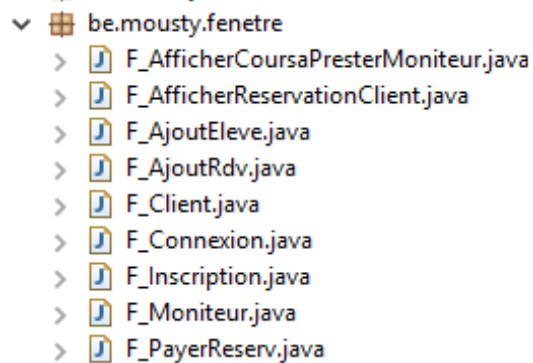
Ils ne possèdent **aucune méthode** ni d'autres constructeurs qu'un constructeur par défaut.

Ils possèdent des propriétés.

Néanmoins j'ai ajouté pour chaque classe un constructeur prenant en paramètre une classe métier afin de faciliter la conversion de type.

Figure 5 POJO

## 2.4 FENETRE



Les fenêtres sont la partie visible du programme. Elles sont générées grâce à WindowsBuilder puis modelées, un peu comme nous avons fait avec Blend lors de projets précédents en C#.

Toutes les fenêtres utilisent JFrame. C'est l'équivalent de la classe Frame de l'AWT. Elles utilisent un panneau de contenu (ContentPane) pour insérer des composants.

<sup>1</sup> Plain Old Java Object

## 3 BOUTS DE CODE ISSU DU PROJET

### 3.1 LAMBDA EXPRESSIONS

Depuis Java 8, l'utilisation de lambda a fortement évolué, il était donc normal d'en implémenter dans mon projet.

#### 3.1.1 AnyMatch

```
/**
 * Objectif : Savoir si un moniteur a encore des cours particuliers à prêter.
 * @param numMoniteur
 * @return
 */
public boolean ceMoniteurDoitPresterCoursParticulier(int numMoniteur){
    return this.getMyList(numMoniteur).stream().anyMatch(t -> t.getCours().getPrix() < 90);
}
```

Figure 7 Lambda, anyMatch

- ➔ Si un cours ayant un prix inférieur à 90€ se trouve dans la liste, alors la fonction retournera true. Le cas échéant, il retournera false.

#### 3.1.2 Stream, filter & collect

```
/**
 * Objectif : Il faut d'abord savoir s'il faut update l'assurance / faire payer SANS update car on peut annuler la réservation.
 * @param numEleve
 * @param numSemaine
 * @param periode
 * @return un booléen pour savoir s'il faut update ou non.
 */
public boolean besoinDupdateOuNonAssurance (int numEleve, int numSemaine, String periode) {
    // On ne s'occupe que des cours collectifs, donc pas besoin des autres périodes.
    String oppose = periode.equals("09-12") ? "14-17" : periode.equals("14-17") ? "09-12" : "";
    // Lambda expression avec filtre permettant de retourner 0 ou une occurrence dans la liste.
    ArrayList<Reservation> listeFull = getListRes();
    ArrayList<Reservation> listTriee = listeFull.stream()
        .filter(p -> p.getEleve().getNumEleve() == numEleve)
        .filter(p -> p.getSemaine().getNumSemaine() == numSemaine)
        .filter(p -> p.getCours().getPeriodeCours().equals(oppose))
        .filter(p -> p.getAuneAssurance() == true)
        .collect(Collectors.toCollection(ArrayList::new));
    // s'il trouve un élément, il update l'assurance.
    return !listTriee.isEmpty();
}
//return ReservationDAO.besoinDupdateOuNonAssurance(numEleve, numSemaine, periode);
}
```

Figure 8 Stream, filter & collect

La variable « **oppose** » aura la valeur opposée à la **période** entrée en paramètre. Uniquement si la période correspond aux horaires des cours collectifs. On ne gère pas celle des cours particuliers, car pour ce type de cours l'assurance sera payée quoiqu'il arrive.

Ensuite, nous insérons dans la liste « **listeTriee** » les éléments correspondants aux différents filtrés entrés.

Nous retournons « **listeFull.stream()** » en type ArrayList via **.collect()**.

S'il y a des éléments dans cette liste, nous devons update l'assurance. Car elle a déjà été payée la première fois.

## 3.2 JTABLE

L'utilisation de JTable facilite beaucoup l'affichage d'une grande quantité d'information.

Mousty, Rue Hebert Hoover

Cours payés

N°	Jour début	jour fin	heure deb...	heure fin	Libellé	Elèves min	Elèves max	Elevés actuel...	Moniteur	Eleve	type	Prix	Action
31	2016-12-05	2016-12-11	09h	12h	Ski	6	10	3	MONITEUR 1	STEFANO N...	Collectif	130.0€	Annuler
71	2016-12-05	2016-12-11	09h	12h	Ski	6	10	3	MONITEUR 1	STEFANO St...	Collectif	130.0€	Annuler
74	2016-12-05	2016-12-11	09h	12h	Ski	6	10	3	MONITEUR 1	COLIN Jean	Collectif	130.0€	Annuler
81	2016-12-05	2016-12-11	09h	12h	Snowboard	5	8	2	MONITEUR 2	COLIN Franç...	Collectif	130.0€	Annuler
82	2016-12-05	2016-12-11	09h	12h	Snowboard	5	8	2	MONITEUR 2	MOUSTY Emry	Collectif	130.0€	Annuler
77	2016-12-05	2016-12-05	13h	14h	Ski	1	4	1	MONITEUR 1	MOUSTY Adr...	Particulier	50.0€	Annuler
78	2016-12-05	2016-12-05	13h	14h	Snowboard	1	4	1	MONITEUR 2	STEFANO N...	Particulier	50.0€	Annuler
70	2016-12-05	2016-12-11	14h	17h	Ski	6	10	2	MONITEUR 1	MOUSTY Adr...	Collectif	130.0€	Annuler
72	2016-12-05	2016-12-11	14h	17h	Ski	6	10	2	MONITEUR 1	STEFANO N...	Collectif	130.0€	Annuler
79	2016-12-05	2016-12-11	14h	17h	Snowboard	5	8	2	MONITEUR 2	STEFANO St...	Collectif	130.0€	Annuler
83	2016-12-05	2016-12-11	14h	17h	Snowboard	5	8	2	MONITEUR 2	CAREY Mariah	Collectif	130.0€	Annuler

Cours en attente de paiement

N°	Jour début	jour fin	heure debut	heure fin	Libellé	Elèves min	Elèves max	Elevés actuel...	Moniteur	Eleve	type	Prix	Action
80	2016-12-05	2016-12-11	09h	12h	Snowboard	5	8	2	MONITEUR 2	MOUSTY Adr...	Collectif	130.0€	Payer

Récapitulatif

Type	Prix cours	Assurances	Reduction	Prix total
Payé	1270	105	78.0	1297.0
Non payé	130	0	39.0	91.0
Total	1400	105	117.0	1388.0

Payer mon panier

Retour

Figure 9 JTable

### 3.2.1 Code couleur :

**Jaune** : les réservations **en suspens**, car le quota n'est pas encore rempli,

**Vert** : les réservations sont **validées**, le quota est rempli,

**Rouge** : les réservations sont **non validées**, car en attente de paiement.

Pour ce faire, voici le petit bout de code qui met en place tout ceci :

```
//changer la couleur
table.setDefaultRenderer(Object.class, new DefaultTableCellRenderer() {

    private static final long serialVersionUID = 1L;

    @Override
    public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected,
        boolean hasFocus, int row, int col) {
        super.getTableCellRendererComponent(table, value, isSelected, hasFocus, row, col);

        boolean estValide = false;
        // On va calculer la place des cours, ensuite on va voir si la palce correspond au maximum possible.
        // Si oui, on colorie en vert, si non on colorie en rouge.
        RATD = listReserv.get(row);
        //CoursATD CATD = RATD.getCours();

        if (RATD.calculerNombrePlaceRestanteMinPourValiderUnCours() == 0)
            estValide = true;
        if (estValide) { setBackground(new Color(102, 255, 51)); }
        else { setBackground(new Color(255,255,153)); }
        return this;
    }
});
```

Pour générer des couleurs en hexadécimal, j'ai utilisé ce site :

[http://www.toutimages.com/generateur\\_nr\\_2.htm](http://www.toutimages.com/generateur_nr_2.htm)

### 3.2.2 Insertion de boutons dans un JTable

Cela permet de gérer chaque réservation de manière individuelle, qu'on veuille supprimer ou payer une réservation.

Pour éviter de dupliquer du code, j'ai créé la classe `ButtonColumn`. Une partie du code provient d'internet.

Ensuite, dans le code `JFrame`, je gère le bouton comme suit :

```
new ButtonColumn(table, changerValeur, headerInfoCours.length-1);
```

Les paramètres sont :

1. La table sur lequel appliquer les boutons.
2. La fonction qui sera exécutée lors du clic.

```
// Action de modification
final Action changerValeur = new AbstractAction()
{
    private static final long serialVersionUID = 1L;

    @Override
    public void actionPerformed(ActionEvent e)
    {
        JTable mytableClicked = (JTable)e.getSource();
        Object numRes = mytableClicked.getModel().getValueAt(mytableClicked.getSelectedRow(), 0);
        //if(ReservationDAO.delete(ReservationDAO.find(Integer.parseInt(numRes.toString()))))
        if(RATD.delete(RATD.find(Integer.parseInt(numRes.toString()))))
            JOptionPane.showMessageDialog(contentPane, "Cours supprimé.");
        else
            JOptionPane.showMessageDialog(contentPane, "Une erreur est intervenue, le cours n'est pas supprimé.");
        setVisible(false);
        F_AfficherReservationClient frameA = new F_AfficherReservationClient(idPersonne);
        frameA.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);
        frameA.setVisible(true);
    }
};
```

- ➔ Via le numéro de la réservation relative à la ligne cliquée, il va tenter de supprimer la réservation.
  - ➔ `myTableClicked` représente la table sur laquelle agir.
  - ➔ `numRes` récupère le n° de réservation qui est le 1<sup>er</sup> élément de la lignée cliquée
  - ➔ Il faut « parser » le n° de la réservation avant de l'insérer en paramètre.
  - ➔ Après la tentative de suppression je ferme puis réaffiche la fenêtre afin d'actualiser les valeurs.
3. La colonne sur laquelle appliquer le bouton.

## 4 OUTILS

### 4.1 SQLITE STUDIO

Au détriment de Access, j'ai décidé d'utiliser une base de données de type SQLite, car je le trouve plus simple à utiliser et moins lourd.

Afin d'utiliser ce type de base de données, j'ai utilisé le logiciel **SQLite Studio**.

C'est un gestionnaire de base de données SQLite avec des fonctionnalités très intéressantes tel que :

- Portable - pas besoin d'installer ou de désinstaller.
- Interface intuitive
- Puissant, mais léger et rapide,
- Open source et gratuit (licence GPLv3).
- Possibilité de tester des requêtes SQL.

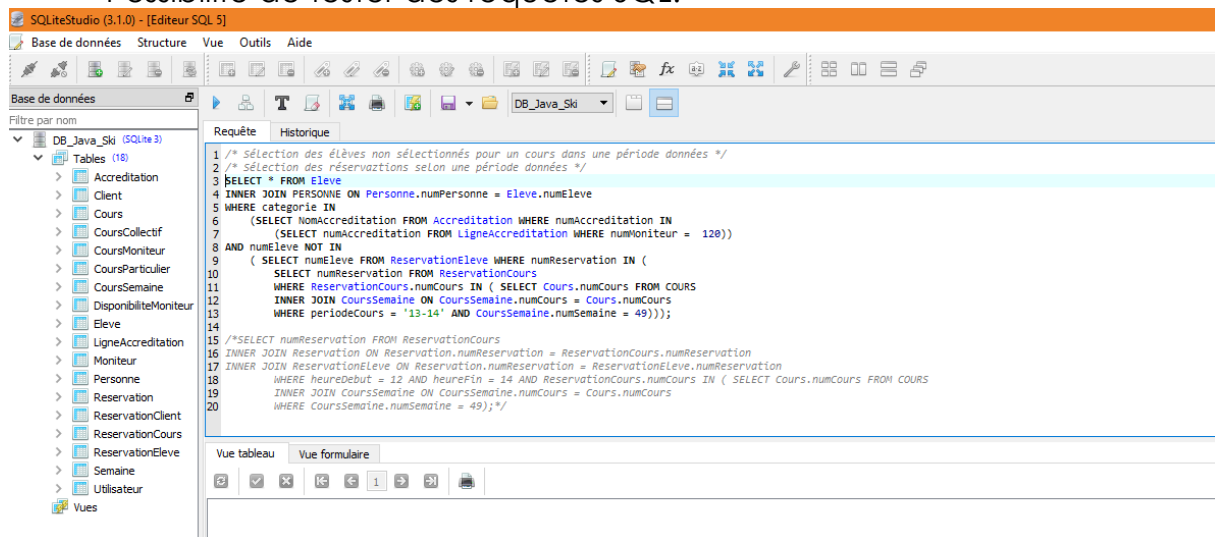



Figure 10 SQLite Studio

Afin d'utiliser ma base de données en JAVA, j'ai dû installer une librairie externe :

 [sqlite-jdbc-3.14.2.1.jar](#)

17/10/2016 14:55

Executable Jar File

4 221 Ko



## 4.2 GIT

En cours nous avons appris à utiliser Git. Bien que celui-ci ne soit pas vraiment utile dans un projet personnel, son importance est indéniable lorsque nous sommes plus d'une personne à développer le même projet.

Git est un outil permettant de gérer l'évolution du contenu d'une arborescence.

J'ai préféré utiliser le GUI de Git (Gitk) afin de mieux visualiser mes précédents « commits ».

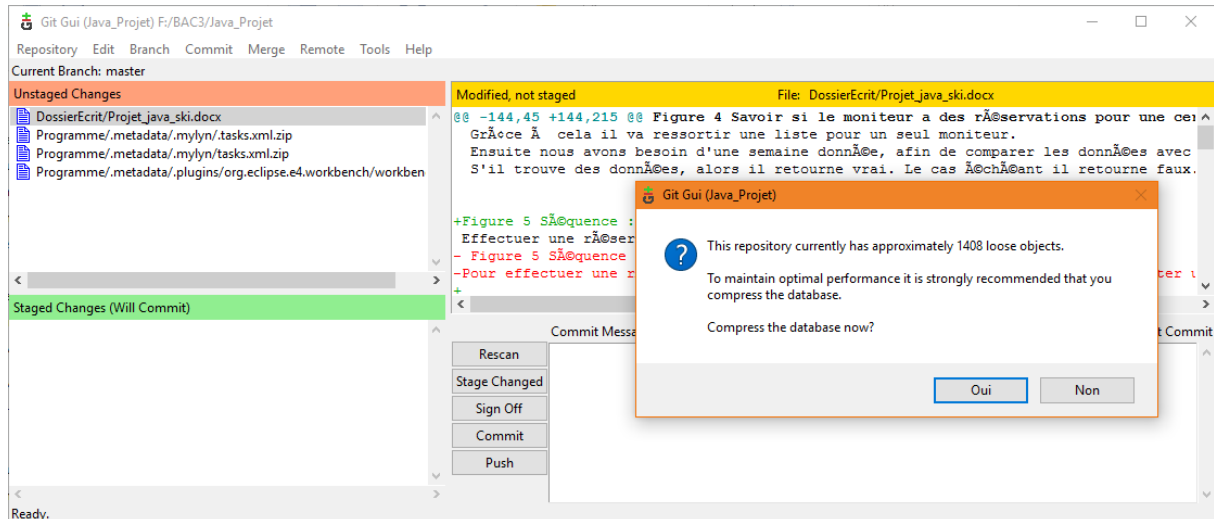


Figure 11 GitK

## 4.3 GIT HUB

Git Hub est le prolongement de Git. Il permet de mettre nos « commits » sur un serveur.

Dans un travail à plusieurs, cela pourrait être quelque chose d'intéressant.

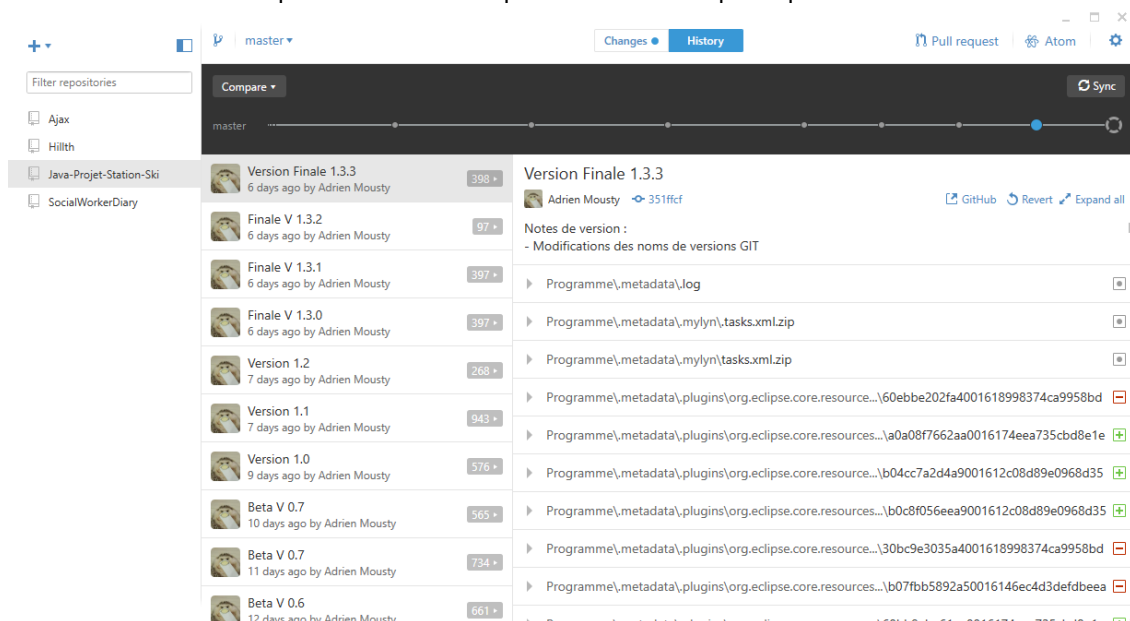


Figure 12 Git Hub

## 4.4 WINDOWBUILDER

WindowBuilder est composé de SWT Designer et Swing Designer. Il facilite la création d'applications Java GUI en diminuant le temps passé à écrire du code.

Il permet aussi bien de créer des fenêtres complexes que des fenêtres plus simples. Le code Java est généré automatiquement.

Vis-à-vis de son implémentation dans Eclipse, il se fait de la même façon que « Glassfish ».

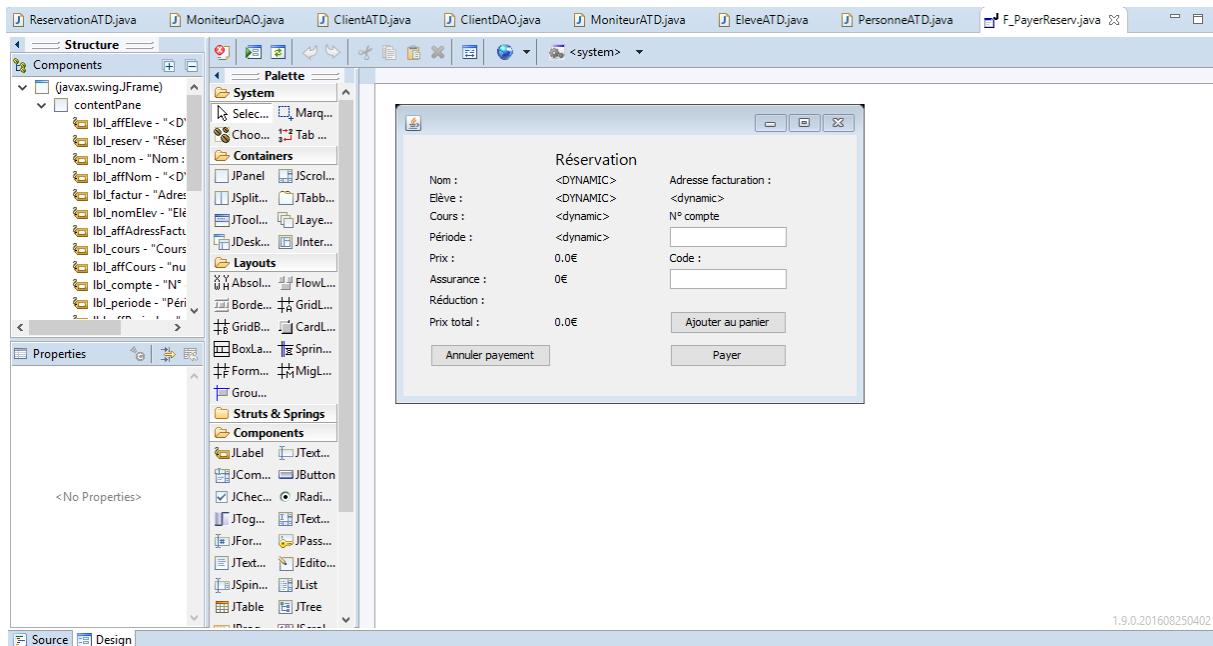


Figure 13 WindowBuilder

## 5 RESUME

---

- Projet effectué seul
- Mise en place d'un DAO
- Langages utilisés : Java, SQL
- Implémentation d'un DatePicker, utilisation d'un GUI
- Utilisation de divers diagrammes + schéma conceptuel, de Git ainsi que SQL Studio
- Note finale : 14/20