Projet Impératif :
Gestionnaire de location de vélos à la « Vélo Toulouse »

Philippe LELEUX
Groupe C

3 janvier 2011

Résumé :
Dans le cadre d'un système de location de vélos, ce projet va consister en l'implémentation d'une interface utilisateur permettant :

- la location de vélos
- la création et la suppression de stations
- la gestion des utilisateurs abonnés

Table des matières

1 Présentation générale

1.1 Introduction

L'objectif de ce projet est de créer un logiciel gestionnaire de location de vélo du type Vêlo Toulouse permettant à des utilisateurs abonnés de se déplacer dans la ville. Il y a deux parties distinctes à implémenter :

D'une part une partie dite administrateur, permettant à un employé connaissant le mot de passe de se connecter à la station et de gérer la création/suppression de vélos et de station ainsi que la gestion des utilisateurs abonnés.

D'autre part une partie dite utilisateur, contenant toutes les actions que peuvent faire les utilisateurs telles que la création de compte, la location de vélo ou encore la gestion de leur propre compte.

Nous ne nous intéressons ici qu'à la couche logiciel du système, considérant que l'on se trouve à une station, il faut créer l'interface graphique qui permet d'utiliser les différentes fonctions du logiciel. On ne s'intéresse pas aux moyens de mettre en commun les informations entre les stations, seulement aux moyens de mettre en place des structures de données efficaces pour enregistrer les informations et interagir avec les différents vélos, stations et comptes.

Il faut créer trois structures de données différentes pour chacun de ces paramètres et implémenter les fonctions permettant de les manipuler avant de s'intéresser à l'interface graphique.

Cette interface doit être conviviale mais fiable. On va utiliser ici une programmation défensive pour toutes les fonctions susceptibles d'être utilisées par un utilisateur extérieur, considérant qu'il ne respectera pas forcément le contrat entre programmeur et utilisateur tandis que les fonctions destinées exclusivement aux administrateurs seront crées en considérant que celui-ci respectera les préconditions établies.

1.2 Packages

Pour une meilleure structuration du programme et une plus grande ré-utilisabilité, le logiciel va être construit en plusieurs modules : des packages pour définir les structures de données et les fonctions permettant de les manipuler dans le programme principal sans pour autant interagir avec l'élément même.

Pour changer le type de structure, il « suffira » alors de changer le package, si on se rend compte que la structure n'est pas entièrement adaptée par exemple. Pour ajouter une fonctionnalité à l'interface, on aura alors qu'à implanter la fonction dans le package et faire appel à elle aux endroits souhaités.

Il y a trois packages à définir : vélo
station
compte

2 Structures de données

Ne pouvant pas connaître à l'avance le nombre d'élément à enregistrer dans la base de données, il faut implémenter des structures de données dynamiques. Nous allons implanter ces structures grâce à des pointeurs en créant des liste simplement chaînées.

2.1 Les vélos

Un vélo est identifié par un numéro positif ou nul unique dans la totalité du système : ident_velo.
Il est pris dans une station et devra être restitué dans une autre (ou dans la même) station après utilisation, on doit donc pouvoir accéder à la station actuelle du vélo : ceci correspondra à un entier positif (l'identificateur de la station) ident_station.
De plus, un vélo ne peut être affecté qu'à un utilisateur au plus à un instant donné : l'identificateur 0 pour les stations sera affecté au vélo si il est déjà loué.
Lors du dépôt d'un vélo loué, on doit pouvoir vérifier que l'utilisateur ne se trompe pas de numéro de vélo rien qu'en accédant à la structure des vélos : un entier supérieur ou égal à 1 ident_compte.

On aura alors la structure définie en langage algorithmique:

```
type velo

type liste_velos est access velo

type velo = enregistrement
        ident_velo : entier
        ident_station : entier
        ident_compte : entier
        suivant : liste_velos
fin enregistrement
```

2.2 Les stations

Les stations de vélos sont également identifiées par un numéro : ident_station.
Elles peuvent contenir un nombre fixé de vélo qui peut être différent d'une station à l'autre et on doit pouvoir connaître le nombre de velos disponibles : deux entiers positifs nombre_velos_maximum et nombre_velos.
On souhaite également pouvoir accéder aux informations concernant les stations les plus proches, on a donc besoin de définir une notion de distance et pour cela de connaître les coordonnées de chaque station dans le plan : deux réels x et y.

On a alors la structure :

```
type station

type liste_station est access station
```

```
type station = enregistrement
        ident_station : entier
        nombre_velos_maximum : entier
        nombre_velos : entier
        x : flottant
        y : flottant
        suivant : liste_station
fin enregistrement
```

2.3 Les comptes

Les utilisateurs abonnés sont décrits par un entier positif unique : ident_compte.

L'administrateur possède un identifiant spécial, le numéro 0.

Dans le compte de l'utilisateur, le code postal et un mot de passe, permettant de sécuriser les comptes et donnant des droits spéciaux à l'administrateur, sont également enregistrés comme deux entiers. Code postal doit être compris entre 1000 et 99000 tandis que le mot de passe fait six chiffres minimum (100000) : code_postal et Mot_de_passe.

Les utilisateurs ont accès à un crédit qu'ils dépensent au cours de l'utilisation et une caution prélevée en cas de perte, vol ou détérioration de vélo : credit et caution.

Lors de la location d'un vélo, on doit pouvoir accéder à l'identifiant du vélo (entier positif) et savoir que l'utilisateur est en train de louer (un booléen) depuis la structure vélo mais également connaître l'heure de retrait du vélo (deux flottant, l'heure comprise entre 0 et 23 et les minutes entre 0 et 59) : en_location, ident_velo, hh et mm.

D'où la structure :

```
        type compte;

type liste_compte est access compte

type compte est enregistrement
        ident_compte : entier
        code_postal : entier
        Mot_de_passe : entier
        credit : entier
        caution : entier
        en_location : booléen
        ident_velo : entier
        hh : entier
        mm :  entier
        suivant : liste_compte
fin enregistrement
```

3 Fonctions des Packages

3.1 Fonctions génériques des liste

Une certaine quantité des fonctions implantées dans les trois pacquages sont des fonctions génériques de manipulation des listes et sont donc presque identiques d'un package à l'autre. Les seules différences se trouvent principalement au niveau des paramètres à modifier, il n'y en a pas le même nombre dans chacune des structures par exemple. D'autres différences notables sont présentes qui seront détaillées plus loin dans chacun des packages.

On considère le type générique suivant :

```
type nœud
type liste est access nœud
type nœud = enregistrement
        identificateur : entier
        element : type
        suivant : liste
fin enregistrement
```

Ces fonctions sont :

```
(*creer une liste vide*)
procedure créer_liste_vide
(*teste si une liste l est vide*)
fonction est_vide retourne booléen
(*insere en tete de la liste l (liste vide ou non vide)*)
procedure insérer_en_tête ((*D/R*) l : liste ; (*D*) éléments : type)
(*insere dans la liste en donnant le premier identificateur non utilise*)
procedure ajouter ((*D/R*) l : liste ; (*D*) identificateur : entier)
(*afficher les elements de la liste l*)
procedure afficher_liste ((*D/R*) l : liste ; (*D*) identificateur : entier);
(*recherche si ident_compte est present dans la liste l, retourne son adresse ou null si la liste est vide ou si
ident_compte n'appartient pas a la liste*)
fonction rechercher ((*D/R*) l : liste ; (*D*) identificateur : entier) retourne liste
(*Enlever un élément identificateur de la liste l (liste vide ou non)*)
procedure enlever ((*D/R*) l : liste ; (*D*) identificateur : entier)
(*Retourner un élément spécifique d'e l'enregistrement*)
fonction retourner_élément ((*D*) l : liste ; (*D*) identifiant : entier) retourne type
(*Modifier un élément spécifique de l'enregistrement*)
procedure modifier_élément ((*D/R*) l : liste ; (*D*) identifiant : entier)
(*Initialiser une liste*)
procedure init_liste ((*R*) l : liste)
```

3.2 Package vélo

Fonctions modifiées :
La procedure afficher_velo_dispo est une fonction de type afficher_liste avec un test sur l'élément

ident_station pour n'afficher que les vélos présents dans la station.
La procedure retirer_velo est une fonction de type changer element, où on donne 0 comme emplacement et le numéro du compte de l'utilisateur, avec en plus un test sur l'emplacement du vélo (loué ou présent dans une autre station).
La procédure retirer_velo est une fonction de type changer element, où on donne l'identificateur de la station, avec en plus un test sur l'état du vélo (en location ou non) et sur l'identité de l'utilisateur (est-bien lui qui a loué ce vélo).

3.3 Package station

Nouvelles fonctions :
(*affiche les informations a propos de la station (coordonnees, velos disponibles, places libres)*)
procedure info_station ((*D*) l : liste_station ; (*D*) ident_station : entier)
(*affiche les informations a propos des stations alentours*)
procedure info_station_alentours ((*D*) l : liste_station ; (*D*) ident_station : entier)
(*renvoie les numeros des stations distantes de moins de X metres d'une station*)
procedure stations_moins_x_metres ((*D*) l : liste_station ; (*D*) ident_station : entier ; (*D*) x : réel)

3.4 Package compte

Fonctions modifiées :
Dans ajouter, on ajoute une boucle pour contrôler l'acquisition au clavier des paramètres code_postal et Mot_de_passe. On ajoute aussi une boucle de contrôle dans les fonctions changer_Mot_de_passe et changer_code_postal.
Dans crediter_compte, une fonction de type modifier_element, on effectue un contrôle sur le credit pour respecter le critère : la première demi-heure est gratuite.

Nouvelles fonctions :
(*verifie les identifiant et mot de passe entres par l'utilisateur renvoie 0 pour un admin, 1 pour un utilisateur, 2 pour une erreur*)
fonction check_identity ((*D*) l : liste_compte ; (*D*) ident_compte : entier ; (*D*) Mot_de_passe : entier)
(*calcule la somme que doit payer l'utilisateur lorrsqu'il rend un velo*)
fonction calcule_credit ((*D*) l : liste_compte; (*D*) ident_compte : integer; (*D*) hh : integer; (*D*) mm : integer)
(*permet d'obtenir des info sur le compte*)
procedure info_compte ((*D*) l : liste_compte; (*D*) ident_compte : integer);

4 Interface utilisateur

4.1 Programme principal

Le programme principal effectue plusieurs tâches élémentaires :

Il initialise d'abord les listes de vélos, stations et comptes avec des valeurs entrées directement dans les packages pour pouvoir effectuer les tests.

Ensuite, il permet d'entrer au clavier la station dans laquelle on se trouve parmis la liste de stations initialisées.

Enfin, dans une boucle permettant de rester toujours dans le programme jusqu'à ce que l'administrateur demande à l'éteindre, il fait appel au programme d'identification puis redirige en fonction de l'identité de l'utilisateur vers le menu approprié.

Intéressons nous au différents sous-programmes nécessaire et, justement, à ces menus spécifiques.

4.2 Sous-programmes

4.2.1 Les différents sous programmes

L'interface utilisateur comporte plusieurs sous programme :

L'identification permet à partir de l'identifiant et du mot de passe d'accéder au compte de l'utilisateur, d'identifier l'utilisateur comme un administrateur ou de s'orienter vers une création de compte.

Ceci nous amène aux autres sous programmes qui sont les différents menus auxquels on peut accéder : - le menu pour utilisateur sans compte qui permet d'afficher les infos sur le service ou de créer un compte

- le menu utilisateur permet d'afficher les infos sur le service, de louer/rendre un vélo, d'accéder au menu pour consulter/modifier son compte, d'obtenir des informations sur la station ou les stations environnantes, de déclarer le vol/dégradation/perte d'un vélo ou de changer de station. Cette dernière fonctionnalité est ajoutée pour pouvoir effectuer les tests, en effet sinon on ne peut que louer ou rendre un vélo dans une des stations, le logiciel n'étant compilé que sur un seul support physique. Dans la réalité, il y aurait plusieurs station fonctionnant en autonomie et partageant une même base de donnée.
- le menu consulter/modifier compte (auquel on accède via le menu utilisateur) permet d'afficher les infos du compte, de changer le code postal/mot de passe ou de créditer/clôturer le compte.
- le menu administrateur permet d'ajouter/supprimer une station/vélo, de supprimer un compte, d'éteindre la station (seul moyen d'arrêter le programme) et d'afficher la liste des stations.

Deux dernière fonction sont présentes, en effet elles sont appelées à différents endroits du programme et ainsi on peut les utiliser sans les réimplanter à chaque fois, ces fonctions sont info service qui affiche les infos sur le service et une fonction qui demande à l'utilisateur d'entrer l'heure en retirant/déposant un vélo.

En résumé, les sous-programmes sont :

(*Renvoie 0 pour un administrateur, 1 pour un utilisateur et 2 pour redémarrer l'identification*)
procedure identification ((*D/R*) liste_comp : liste_compte; (*R*) ident_compte : entier; (*R*)

option_identifiant : entier)
(*menu permettant a un utilisateur non abonne de s'abonner (renvoie 1) et/ou d'obtenir des info sur le service (renvoie 0) *)
procedure menu_utilisateur_sans_compte ((*D/R*) liste_comp : liste_compte; (*D/R*) option_identifiant : entier; (*D/R*) ident_compte : entier)
(*menu permettant d'obtenir des infos, de louer/rendre un velo, de consulter/modifier un compte, obtenir le nombre de place et de velos de la stations et des stations environnantes *)
procedure menu_utilisateur ((*D/R*) liste_vel : liste_velos; (*D/R*) liste_stat : liste_station; (*D/R*) liste_comp : liste_compte; (*D/R*) ident_station : entier; (*D*) ident_compte : entier)
(*Permet a l'administrateur d'ajouter/supprimer une station, des velos, de supprimer un compte ou d'eteindre la station*)
procedure menu_administrateur ((*R*) ContinuerAEffectuerDesActions : booléen; (*D/R*) liste_vel : liste_velos; (*D/R*) liste_stat : liste_station; (*D/R*) liste_comp : liste_compte)
(*affiche les info a propos des tarifs etc.. *)
procedure info_service
(*demande l'heure et modifie hh et mm *)
procedure heure ((*R*) hh : integer; (*R*) mm : integer)

4.2.2 Implantation

Tous les menus de cette interface ont une implantation similaire. Tout le programme est entouré par deux boucles : - la plus large permet de rester dans le menu même après avoir effectué une action, un utilisateur par exemple restera dans le menu utilisateur même après avoir loué un vélo.

- la seconde boucle, à l'intérieur de la première, permet d'effectuer un contrôle sur le choix d'action dans le menu : si l'utilisateur choisis une action qui n'existe pas, il peut réessayer jusqu'à demander une action possible.

Cette sorte d'implantation permet non seulement d'enchaîner plusieurs actions en restant identifié mais également d'utiliser une programmation plutôt défensive, on ne fait pas confiance à l'utilisateur.

Le menu est ensuite réalisé grâce à un case avec pour entrée un character.

5 Raffinage

5.1 Raffinage des fonctions génériques

Nous n'allons ici pas détailler le raffinage de toutes les fonctions, certaines étant des fonctions très simples et classiques ne nécessitant presque aucun raffinage. La seule fonction originale et nécessitant un raffinage est en fait ajouter.

Raffinage de la fonction ajouter :

R0=Spécification : Ajouter un nœud dans la liste en spécifiant au clavier les différents paramètres de l'enregistrement et lui donnant le premier identificateur non utilisé de la liste (par exemple pour une liste de vélos, si on a la liste existante d'identificateurs 0,1,2,5,6 alors on lui donnera l'identificateur 3).

R1 :

```
(*procedure ajouter_velo                                              *)
(*semantique: ajoute un velo u la liste de velo en lui donnant le premier *)
(*identificateur non utilise                                          *)
(*parametres: l donnee/resultat type liste_velos                      *)
(*pre-condition: la liste est triee                                          *)
(*post-condition: la liste comporte un velo de pluss                  *)

procedure ajouter_velo ((*D/R*) l : liste) est
element : type
pins : liste_station (*pointeur de parcours *)
pecr : liste_station (*pointeur d'ecriture *)
interieur : boolean (*vrai si il y a un "trou" dans la liste*)
debut
        Acquisition d'element
        Inserer le nœud en lui donnant le premier identificateur non utilisé
        Afficher le numéro du nouveau nœud
```

R2 :

```
(*Acquisition d'element*)
ecrire(« Entrez element »)
lire(element)
(*inserer le nœud en lui donnant le premier identificateur non utilisé*)
Si Liste vide ou liste a un element alors traitement pour liste vide ou liste à un nœud
                                    sinon rechercher un « trou » dans la liste et inserer dans ce trou
Fin si
(*Afficher le numero du nouveau noeud
```

R3 :

```
Si (*Liste vide ou liste a un element alors traitement pour liste vide ou liste à un nœud*)
   liste=NULL alors inserer en tete
                sinon si liste.suivant=NULL, tester identificateur dernier noeud
Sinon   pins ← nouveau nœud
```

```
pecr ← nouveau noeud
pins ← l
pecr ← l
(*rechercher un « trou » dans la liste et inserer dans ce trou*)
Tant que pins/=NULL faire
        Si pins.suivant/=NULL alors tester si il y a un trou
                                    sinon tester si trou après dernier nœud
        Fin si
avancer le pointeur
Si il y a un trou alors inserer dans le trou
                   sinon inserer en tete
Fin si
```

R4 :

```
Si (*Liste vide ou liste a un element alors traitement pour liste vide ou liste à un nœud*)
   liste=NULL alors (*inserer en tete*)
                    inserer_en_tete(liste, element)
              sinon si liste.suivant=NULL, (*tester identificateur dernier noeud*)
                                    si liste.identificateur/=0 alors inserer_en_tete(l.suivant, element)
                                                               sinon inserer_en_tete(l, element)
                                    fin si
Sinon  pins ← nouveau nœud
       pecr ← nouveau noeud
       pins ← l
       pecr ← l
       (*rechercher un « trou » dans la liste et inserer dans ce trou*)
       Tant que pins/=NULL faire
              Si pins.suivant/=NULL alors (*tester si il y a un trou*)
                                    si pins.identificateur-pins.suivant.identificateur/=1
                                                alors interieur ← vrai
                                                     pecr ← pins
                                    sinon (*tester si trou après dernier nœud*)
                                          si pins.identificateur/=0 alors interieur ← vrai
                                                                          pecr ← pins
                                                                    sinon NULL
                                          fin si
              (*avancer le pointeur*)
              pins ← pins.suivant
              Fin si
       fin tant que
       Si (*il y a un trou*)
          interieur alors (*inserer dans le trou*)
                          inserer_en_tete(pecr.suivant, element)
                    sinon (*inserer en tete*)
                           inserer_en_tete(l, element)
       Fin si
```

5.2 Raffinage des Fonctions spécifiques

5.2.1 Package station

Procedure info_station :

R0 = spécification : affiche les informations a propos de la station (coordonnees, velos disponibles, places libres)

R1 : **(*** procedure info_station *)
(*sémantique: affiche les infos a propos de la station (coordonnées, vélo*)
(* disponibles, places libres) *)
(*paramètres: l donnée type liste_station *)
(* ident_station donnée type entier *)
(*précondition: aucune *)
(*postcondition: aucune *)

procedure info_station ((*D*) l : liste_station; (*D*) ident_station : entier) est
laux : liste_station (*pointeur vers la station courante*)
begin
Se placer au niveau de la bonne station
tester existance station
afficher identificateur, coordonnées, nombre de vélos, nombre de places libres

R2 :

(* Se placer au niveau de la bonne station*)
laux:=rechercher(l, ident_station);
(*tester existance station*)
Si laux=NULL alors écrire(« Cette station n'existe pas »)
sinon (*afficher identificateur*)
Ecrire(« La station «)
Ecrire(laux.ident_station)
(*coordonnées*)
Ecrire(" a pour coordonnees ")
Ecrire(laux.x)
Ecrire(" ")
Ecrire(laux.y)
Nouvelle_ligne
(*nombre de vélos*)
Ecrire("Elle a ")
Ecrire(laux.nombre_velos)
Ecrire(" velos disponibles et ")
(*nombre de places libres*)
Ecrire(laux.nombre_velos_maximum-laux.nombre_velos)
Ecrire(" places libres.")
Nouvelle_ligne

procedure info_station ((*D*) l : liste_station ; (*D*) ident_station : entier)
(*affiche les informations a propos des stations alentours*)
procedure info_station_alentours ((*D*) l : liste_station ; (*D*) ident_station : entier)
(*renvoie les numeros des stations distantes de moins de X metres d'une station*)
procedure stations_moins_x_metres ((*D*) l : liste_station ; (*D*) ident_station : entier ; (*D*) x : réel)

stations_moins_x_metres :

R0 = spécification : affiche les informations a propos des stations alentours situées à moins de X metres.

R1 :

(* procedure info_station_alentours *)
(*semantique: affiche les informations a propos des stations alentours *)
(*parametres: l donnee type liste_station *)
(*ident_station donnee type entier *)

```
(*precondition: aucune                                                    *)
(*postcondition: aucune                                                   *)
procedure stations_moins_x_metres ((*D*) l : liste_station ; (*D*) ident_station : entier ; (*D*) x : réel) est
paux : liste_station (*pointeur permettant de parcourir la liste *)
pcourant : liste_station (*pointeur vers la station courante*)
debut
        Se placer au niveau de la bonne station
        tester existance station
        parcourir la liste en testant la proximité entre les stations et la station courante
```

R2 :

```
(*Se placer au niveau de la bonne station*)
pcourant ← new station;
pcourant ← rechercher(l, ident_station)
(*tester existance station*)
si pcourant=NULL alors Ecrire(« Cette station n'existe pas »)
                sinon (*parcourir la liste en testant la proximité entre les stations et la station courante*)
                    tant que paux/=NULL faire
                            si sqrt((( paux.x)-(pcourant.x))**2+((paux.y)-(pcourant.y))**2)<=X et
paux/=pcourant
                                                    alors info_station(l, paux.ident_station);
                            sinon NULL
                            fin si
                            paux ← paux.suivant
                        fin tant que
fin si
```

info_stations_alentours :

Cette procedure est identique à la précédente avec X=1000 metres.

5.2.1 Package compte

check_identity :

R0 = spécification : verifie les identifiant et mot de passe entres par l'utilisateur renvoie 0 pour un admin, 1 pour un utilisateur, 2 pour une erreur.

R1 :

```
(*fonction check_identity                                                *)
(*semantique: verifie les identifiant et mot de passe entres par l'utilisateur    *)
(*          renvoie 0 pour un admin, 1 pour un utilisateur, 2 pour une erreur     *)
(*parametres: liste_comp donnee type liste_compte                         *)
(*          ident_compte donnee type integer                               *)
(*          Mot_de_passe donnee type integer                               *)
(*pre-condition: aucune                                                    *)
(*post-condition: aucune                                                   *)
procedure info_compte ((*D*) l : liste_compte; (*D*) ident_compte : integer) est
laux : liste_compte (*pointeur de parcours *)
option_identifiant : entier(*0 : administrateur, 1 : utilisateur, 2 : aucun des deux*)
debut
        considérer par défaut que l'utilisateur n'est pas abonné
        vérifier si l'utilisateur est un administrateur
        parcourir la liste en vérifiant si l'utilisateur est abonné
        retourner l'option d'identification
fin check_identity
```

R2 :

```
(*considérer par défaut que l'utilisateur n'est pas abonné*)
option_identifiant ← 2
(*vérifier si l'utilisateur est un administrateur*)
si ident_compte=0 et Mot_de_passe=789789 alors option_identifiant=0
                    sinon (*parcourir la liste en vérifiant si l'utilisateur est abonné*)
                        laux ← nouveau compte
                        laux ← l
                        tant que laux/=NULL et option_identifiant/=1 loop
                            si laux.ident_compte et laux.Mot_de_passe=Mot_de_passe alors
option_identifiant ← 1
                            sinon NULL
                            fin si
                        fin tant que
fin si
(*retourner l'option d'identification*(
retourne option_identifiant
```

info_compte :

Cette procedure est du même type que info_station avec les paramètres du compte en paramètre.

6 Tests

Pour les tests, j'ai choisi de tester toutes les fonctions une par une afin d'être sûr de leur exécution en commençant par les fonctions qui n'appellent aucune autre fonction lors de leur exécution. Mon programme de test se trouve avec les fichiers sources : test.adb. Chacune des fonctions doit être testée sur les différents cas possibles que l'on va expliciter.

6.1 Fonctions génériques

Les fonctions génériques sont toutes identiques au nom de la liste près, les tester pour chaque package reviendrait donc en fait à les tester trois fois ce qui n'a pas grand intérêt, on va donc les implanter dans le module de test avec les types liste et nœud (element étant un entier).

créer_liste_vide : testée sur une liste pas encore initialisée.
est_vide : testée sur une liste vide et une liste non vide.
insérer_en_tête : idem.
Ajouter : testée sur une liste vide, une liste à un nœud dont l'identificateur est ou n'est pas 0, testée sur une liste à plus d'un nœud avec trou (à la fin ou non) et sans trou.
afficher_liste : testée sur une liste vide ou non.
Rechercher : testée sur une liste vide, sur une liste non vide mais ne contenant pas le nœud et sur une liste contenant le nœud
Enlever : testée sur une liste où le nœud n'est pas présent et une où il est présent.
retourner_élément : testé sur une liste avec le nœud.
modifier_élément : testé sur une liste où le nœud est présent ou non.
init_liste : simplement testée

6.2 Package station

info_station : testée sur une liste où la station est présente ou non.
info_station_alentours : testée sur une liste où la station n'est pas présente, sur une liste où la station est présente et n'a pas de station à proximité et sur une liste avec des stations à proximité.
stations_moins_x_metres : idem.

6.3 Package compte

check_identity : testée dans le cas où l'identifiant est 0 mais le mot de passe est faux, dans le cas où les deux sont bon, dans le cas où on a un utilisateur enregistré ou non.
calcule_credit : testée sur une liste où le compte n'est pas présent et sur une liste où il est présent avec comme durée moins d'une demi-heure puis plus d'une demi-heure
info_compte : même fonctionnement que info_station.

7 Conclusion

Ce projet a été une bonne expérience pratique de développement de logiciel dans une situation réelle. Il m'a montré a quel point un sujet qui pourrait paraître simple au premier abord peut s'avérer compliqué en rentrant dans les détails de sa programmation. Mélangeant de nombreuses facettes de la programmation impérative, et notamment la manipulation de pointeurs dans l'optique de créer et manipuler des structures de données dynamiques, ce projet m'a permis d'approfondir ma connaissance du langage ADA dans son ensemble.

Le choix des listes simplement chaînées n'est pas forcément le meilleur choix possible. Si le nombre de données à manipuler est petit, comme c'est le cas pour nos tests puisque l'on rentre un par un chaque élément, c'est acceptable, mais si le nombre d'éléments devenait très grand peut-être serait-il plus efficace d'implanter des structures plus complexes telles-que des arbres n-aires.

Beaucoup d'aspects pourraient être améliorés dans ce projet et d'autres points pourraient constituer un prolongement intéressant, on pourrait par exemple s'intéresser à la manière dont les données vont être mises en commun. Pourquoi pas créer un fichier texte extérieur qui contiendrait les informations sur les différentes stations, vélos et comptes. J'ai voulu implanter cette fonctionnalité au travers d'un 4ème package : fichier, qui au début du programme principal récupèrerait les données et serait modifier au fur et à mesure de l'exécution, cependant certaines fonctions devenaient longues à implanter et le temps pour le faire est réduit.