

Filière : Développement des Systèmes d'Information - DSI
Épreuve : Développement des Applications informatiques DAI

Durée : 4 Heures
Coefficient : 45

CONSIGNES

- Le sujet comporte 4 dossiers ; chaque dossier doit être traité dans une feuille séparée.

BARÈME DE NOTATION

Dossier 1 : Étude de l'application desktop JAVA	13 points
Dossier 2 : Étude de l'application desktop VB.Net	11 points
Dossier 3 : Étude de l'interface client WEB	7 points
Dossier 4 : Étude de l'application distribuée	9 points
Total	40 points

- Il sera pris en considération la qualité de la rédaction lors de la correction.
- Aucun document n'est autorisé.

PROBLÉMATIQUE

La gestion d'un parc automobile peut être découpée en diverses tâches, parmi lesquelles :

- la gestion de la documentation (*Certificat de visite, Attestation d'importation temporaire*) ;
- Le suivi des réservations (*valider la disponibilité du véhicule demandé et garantir la livraison en bonne condition*)...

La réalisation de cette gestion reste difficile compte tenu de la diversité des tâches à accomplir. En plus, l'absence d'une base de données et le non archivage des documents papiers utilisés pour les différentes tâches rendent quasiment impossible l'établissement de statistiques fiables. Pour mieux gérer son parc automobile, une société souhaite mettre en place des solutions informatiques. Pour ce faire, notre travail consiste à mettre en place un système dont les fonctionnalités sont :

- Une mise à jour locale par une application desktop développée sous VB.Net est installée dans le serveur se trouvant au siège de la société.
- Un accès distant par une application JAVA distribuée installée dans les différents bureaux de la société.
- Un accès client via un site web dynamique.

Ces différentes applications doivent assurer la rapidité, la fiabilité, la facilité des traitements et une circulation des informations en temps réel.

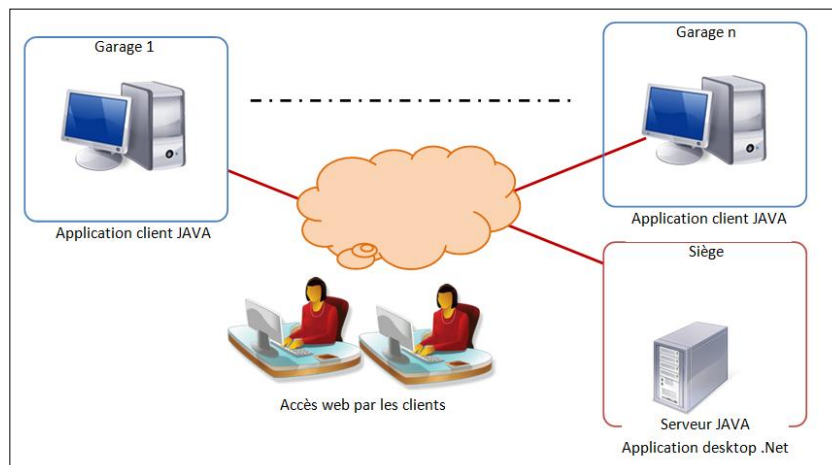


Figure 1 : Architecture du réseau informatique de la société

Une modélisation UML de ce sujet à aboutit au diagramme de classes suivant :

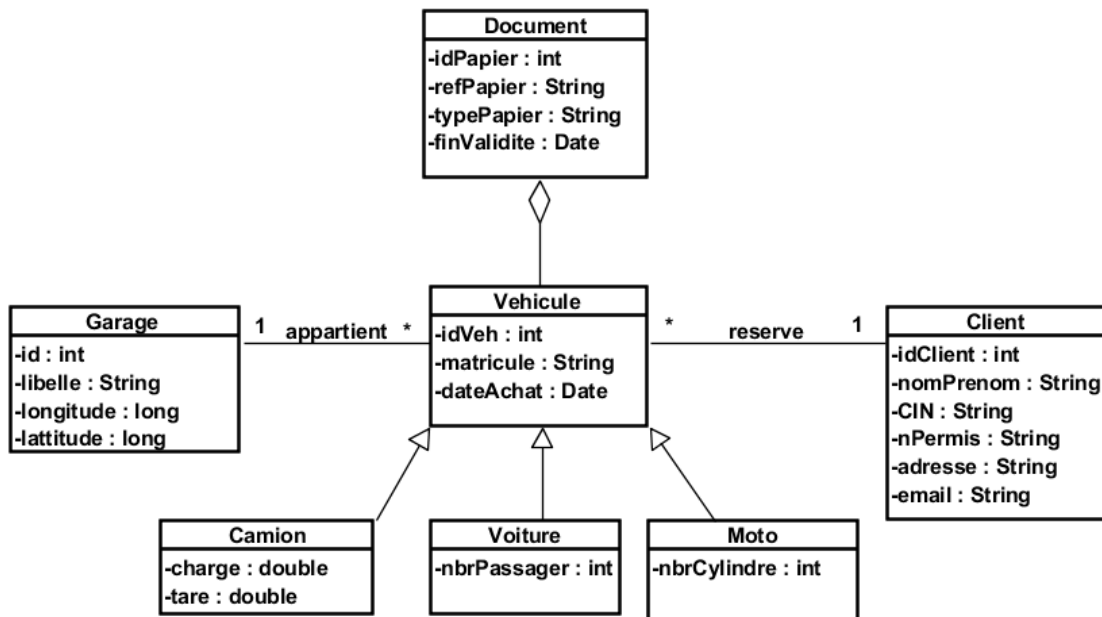


Figure 2 : Diagramme de classes

DOSSIER I : ÉTUDE DE L'APPLICATION DESKTOP JAVA

(13 pts)

Dans cette partie, on se contente de créer les 3 classes :

- « Vehicule »
- « Garage »
- « Camion »

1- Classe « Vehicule » :

Cette classe a comme canevas :

```

public class Vehicule {
    private int idVeh ;
    private String matricule ;
    private Date dateAchat;
    public Vehicule(int idVeh, String matricule, Date dateAchat) {
        .....
    }
    public Vehicule(int idVeh, String matricule) {
        .....
    }
    @Override
    public String toString() {
        .....
    }
    @Override
    public boolean equals(Object ob) {
        .....
    }
}
    
```

Compléter cette classe par l'implémentation du code des diverses méthodes :

- 1.1. Constructeur avec 3 arguments. (0,5 pt)
- 1.2. Constructeur avec 2 arguments (*la date d'achat prend la date système*). (0,5 pt)
- 1.3. « `toString` » qui retourne une chaîne de caractères ayant la forme suivante : (1 pt)
ID de véhicule : `xxxxxx`, Matricule : `xxxxxx`, Acheté le : `xxxxxx`
- 1.4. « `equals` » qui compare deux véhicules selon leurs identificateurs. (1 pt)

2- Classe « **Camion** » :

Cette classe a comme canevas :

```
public class Camion extends Vehicule {
    private double charge ;
    private double tare ;
    public Camion (int idVeh, String matricule, Date dateAchat, double charge, double tare)
    {
        .....
    }
    public Camion (int idVeh, String matricule, double charge, double tare)
    {
        .....
    }
    @Override
    public String toString()
    {
        .....
    }
}
```

Compléter cette classe par l'implémentation du code des diverses méthodes :

- 2.1. Constructeur avec 5 arguments. (0,5 pt)
- 2.2. Constructeur avec 4 arguments. (0,5 pt)
- 2.3. « `toString` » qui retourne une chaîne porteuse d'informations d'un véhicule, cette chaîne aura la forme suivante : (1 pt)
Camion : ID de véhicule : `xxxxxx`, Matricule : `xxxxxx`, Acheté le : `xxxxxx`
Tare : `xxxxxxxxxxxxxxxx` et **Charge** : `xxxxxxxxxxxxxxxxxxxxxxxx`

3- Classe « **Garage** » :

Cette classe a comme canevas :

```
public class Garage {
    private int id ;
    private String libelle ;
    private long longitude, lattitude ;
    private ArrayList<Vehicule> liste;
    public Garage (int id, String libelle, long longitude, long lattitude) {
        .....
    }

    public int searchVehicule(Vehicule V) {
        .....
    }
}
```

```
public Vehicule setVehicule(Vehicule nouveau, int index) {  
.....  
}  
public boolean addVehicule(Vehicule V) {  
.....  
}  
public Vehicule delVehicule(int index) {  
.....  
}  
@Override  
public String toString() {  
.....  
}  
}
```

Compléter cette classe par l'implémentation du code des diverses méthodes :

- 3.1. Constructeur avec arguments qui initialisent tous les attributs. (0,5 pt)
- 3.2. « **searchVehicule** » recherche un véhicule dans la collection et elle retourne son index. S'il est inexistant, elle retourne **-1**. (1 pt)
- 3.3. « **setVehicule** » change un véhicule selon son index et elle retourne le véhicule avant son changement. (1 pt)
- 3.4. « **addVehicule** » qui ajoute un nouvel véhicule à la collection à la condition qu'il ne soit déjà ajouté au paravent, elle retourne **true** si l'ajout est fait avec succès. (1 pt)
- 3.5. « **delVehicule** » supprime un véhicule selon son index et elle retourne le véhicule qui vient d'être supprimé, si l'index est invalide, elle retourne **null**. (1 pt)
- 3.6. « **toString** » retourne une chaîne qui contient la liste des véhicules. (1 pt)

4- Test des classes :

Créer un programme de test permettant d'effectuer les étapes suivantes :

- Créer 2 objets camion ; (0,5 pt)
- Créer un objet garage ; (0,25 pt)
- Ajouter les objets camions au garage ; (0,5 pt)
- Lister tous les véhicules du garage ; (0,25 pt)
- Vérifier si le premier camion existe dans le garage ; (0,5 pt)
- Supprimer le deuxième camion du garage ; (0,25 pt)
- Lister à nouveau tous les véhicules du garage. (0,25 pt)

DOSSIER II : ÉTUDE DE L'APPLICATION DESKTOP SOUS VB.NET

(11 pts)

On veut créer une application Desktop, sous VB.Net, permettant la mise à jour des véhicules. Ces données sont stockées dans la base de données « BD_Demandes » implantée sous SQL Server.

Voici un extrait de cette base de données.

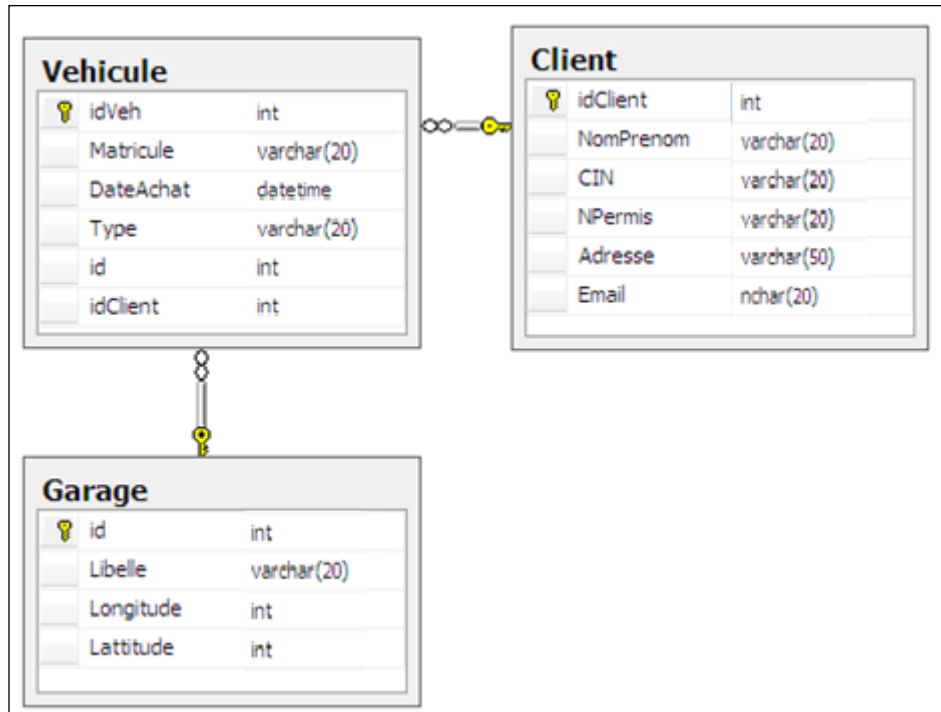


Figure 3 : Extrait du MLDR

L'IHM de gestion des véhicules et ses propriétés sont données ci-après :

L'interface graphique, intitulée "Gestion des véhicules", permet de saisir les données d'un véhicule. Elle comprend :

- Champs de saisie pour : ID véhicule, Matricule, Date Achat (samedi 13 février 2016), Type, Client (rabie amine), Garage (Garage1).
- Boutons d'action : Enregistrer, Supprimer, Modifier.
- Tableau "Liste des véhicules" :

	idveh	Matricule	Type	Libelle	NomPrenom
▶	1	224433 B 1	Voiture	Garage 1	rabie amine
	2	554411 A 1	Moto	Garage 1	rabie sami
*					

Figure 4 : IHM de gestion des véhicules

CONTRÔLE	PROPRIÉTÉ	VALEUR
Bouton de commande	Name	cmdEnregistrer
	Text	Enregistrer
	Name	cmdSupprimer
	Text	Supprimer
	Name	cmdModifier
	Text	Modifier
TextBox	Name	txtID
	Name	txtMat
DateTimePicker	Name	dateAchat
ComboBox	Name	cbTypes
	Name	cbClients
	Name	cbGarages
DataGridView	Name	Table

N.B : Dans cette partie, vous pouvez utiliser le mode connecté ou le mode non connecté.

Rappel : la chaîne de connexion est la suivante :

initial catalog = BD_Demandes; data source=PC_Demande; Integrated Security=true ;

- 1- Selon le mode choisi, préciser et initialiser les objets de connexion à cette base de données. (1,5 pt)
- 2- Créer les procédures suivantes : (3 pts)

```
Public Class Form1
    ' remplir le comboBox cbTypes par les 3 chaînes "Camion", "Voiture" et "Moto"
    Sub chargerTypes ()
        .....
    End Sub

    ' remplir le comboBox cbClients depuis la table client (champ NomPrenom)
    Sub chargerclients ()
        .....
    End Sub

    ' remplir le comboBox cbGarages depuis la table garage (champ libelle)
    Sub chargerGarages ()
        .....
    End Sub

    ' lister tous les véhicules dans l'objet DataGridView « Table »
    ' les champs à afficher sont :idveh, Matricule, Type, Libelle et NomPrenom
    Sub lister ()
        .....
    End Sub
End Class
```

- 3- Donner le code de la fonction « **formValide ()** » qui retourne **true** si le formulaire est valide et **false** dans le cas échéant. Un formulaire est considéré valide si : (1 pt)
 - ✓ Le champ **txtID** est non vide et numérique ;
 - ✓ Un Client est sélectionné dans le combobox «**cbClients**».
 - ✓ Un garage est sélectionné dans le combobox «**cbGarages**».
- 4- Donner le code de la fonction **cleExiste ()** qui retourne **true** si l' ID véhicule saisie existe déjà dans la base de données et **false** dans le cas contraire. (1 pt)

- 5- Donner le code de la procédure événementielle **cmdEnregistrer_Click** qui permet d'ajouter un nouveau véhicule à la table « **vehicule** », cet ajout est fait si le formulaire est valide et la clé primaire est unique. (1,5 pt)

```
Private Sub cmdEnregistrer_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
cmdEnregistrer.Click
.....
End Sub
```

- 6- Donner le code de la procédure événementielle **cmdSupprimer_Click** qui permet de supprimer un véhicule sélectionné dans **DataGridView**, de la table « **vehicule** ». (1,5 pt)

```
Private Sub cmdSupprimer_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
cmdSupprimer.Click
.....
End Sub
```

- 7- Donner le code de la procédure **cmdModifier_Click** qui permet de modifier un véhicule affiché dans le formulaire. Cette modification est faite si le formulaire est valide et la clé primaire existe déjà. (1,5 pt)

```
Private Sub cmdModifier_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
cmdModifier.Click
.....
End Sub
```

DOSSIER III : ÉTUDE DE L'INTERFACE CLIENT WEB

(7 pts)

Dans cette partie, on envisage créer un espace Web permettant aux clients de consulter leurs véhicules via leurs identifiants. Un extrait de la base de données est implanté sous MySQL.

Table ▲	Action	Lignes ¹	Type
<input type="checkbox"/> client		2	InnoDB
<input type="checkbox"/> garage		2	InnoDB
<input type="checkbox"/> vehicule		2	InnoDB
3 table(s)	Somme	6	InnoDB

Figure 5 : structure de la base de données « BD_Demandes »

La page Web assurant cette tâche est la suivante :

Matricule	Date Achat	Type	Libelle
55432B1	2016-02-08 01:23:40	Voiture	garage1
55332A4	2015-12-07 02:05:14	Voiture	garage2

Figure 6 : page web pour lister les véhicules d'un client

Cette interface permet au client de lister tous ses véhicules inscrits en saisissant son ID.
Le code à compléter est le suivant :

```
<html>
  <head>
    <style type="text/css">
      #principal{ ... }
      #titre { ... }
      #corps { ... }
      #tab1 {... }
      #tab2 { ... }
      #tab2 th { ... }
      #tab2 td { ... }
    </style>
    <script type="text/javascript">
      function test() {
        .....
      }
    </script>
  </head>
  <body>
    <div id="principal">
      <div id="titre">CONSULTER LA LISTE DE VOS VÉHICULES</div>
      <div id="corps">
        <form method="POST" action="" name="f">
          <table id="tab1">
            <tr><td>Entrez votre ID</td><td><input type="text" name="id"
              placeholder="Entrez votre ID"></td></tr>
            <tr><th><input type="submit" value="Valider" onclick="return
              test();"></th><th><input type="reset" value="Annuler"></th></tr>
          </table>
        </form>
      </div>
    </div>
  </body>
</html>
```



```
<table id="tab2">
<caption>Liste des véhicules</caption>
<tr><th>Matricule</th><th>Date
Achat</th><th>Type</th><th>Libelle</th></tr>
<?php
.....
.....
?>
</table>
</div>
</div>
</body>
```

- 1- Donner le corps de la fonction « **test()** » qui vérifie si l'utilisateur a bien saisi son **ID**. Si c'est le cas, elle retourne **true** sinon retourne **false** dans le cas échéant puis affiche un message d'erreur dans une boîte de dialogue. (2 pts)
- 2- Donner le script PHP permettant d'établir la connexion avec la base de données sachant que : (2 pts)
 - Nom de serveur : **Serv_com**
 - Login : **user**
 - Mot de passe : **1260**
 - Base de données : **BD_Demandes**
- 3- Compléter la partie de script PHP qui permet de vérifier s'il y a un clic sur le bouton de validation. (1 pt)
- 4- Récupérer la liste des véhicules (*spécialement les champs **Matricule, dateAchat, Type et Libelle***) dont **idClient** correspond à l'**ID** saisi par l'utilisateur. Si aucun véhicule ne correspond à l'**ID** saisi, une boîte de dialogue doit apparaître pour signaler ceci. (2 pts)

DOSSIER IV : ÉTUDE DE L'APPLICATION DISTRIBUÉE

(9 pts)

Dans cette partie, on vise étudier une application permettant de synchroniser les différentes stations de travail dans les bureaux de la société avec le serveur se trouvant dans le siège. On opte alors pour l'utilisation de la **RMI (Remote Method Invocation)**.

Dans le serveur principal, on installe un serveur RMI intégrant une classe présentant des méthodes partagées. Dans les différentes stations de travail, on installe une application client-RMI exploitant les méthodes offertes par le serveur.

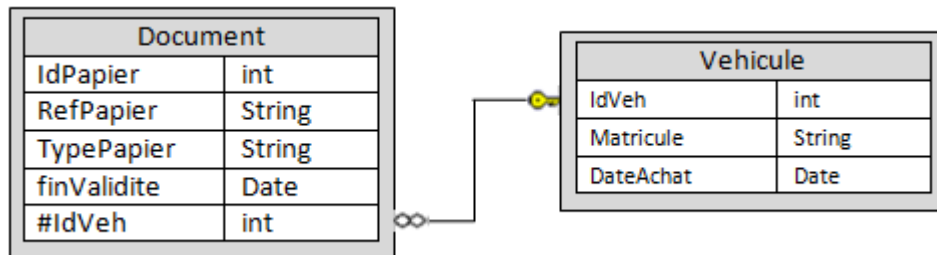


Figure 7 : Extrait de la base de données « BD_Demandes »

Les règles de l'art de la démarche RMI consistent en :

- Développer l'interface de l'objet distant : cette interface doit hériter de la classe « **Remote** » et ses méthodes doivent lever l'exception « **RemoteException** ».
- Implémenter la classe de l'objet distant : cette classe doit implémenter l'interface précédente et doit hériter de la classe « **UnicastRemoteObject** ». Le constructeur de cette classe doit aussi lever l'exception « **RemoteException** ». Tous les objets communiqués via cette interface doivent implémenter l'interface « **Serializable** ».
- Développer la classe présentant le serveur RMI : Dans cette classe, on doit instancier l'objet distant puis le publier dans le serveur de nom « **rmiregistry** ».
- Dans l'application cliente, on crée le « stub » (une instance de l'interface distante publiée dans le serveur de noms) puis on l'utilise pour appeler les méthodes distantes.

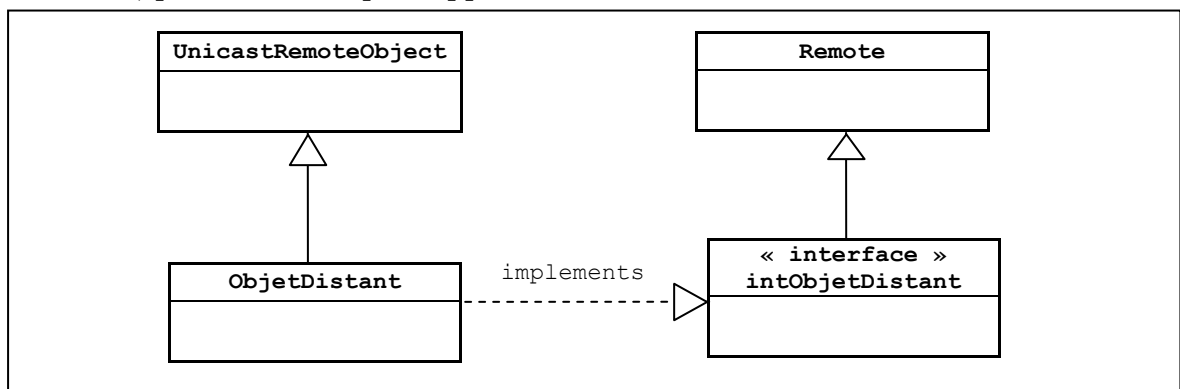


Figure 8 : diagramme de classe

PROGRAMME COTÉ SERVEUR :

L'objet distant doit présenter les traitements suivants :

- ✓ Constructeur sans arguments permet d'établir une connexion à la base de données sous SQL Server. L'URL à utiliser est :

"jdbc:sqlserver://PC_Demande:1433; database = BD_Demandes; IntegratedSecurity=true;"

- ✓ Une méthode « **vehDisponible** » permettant de sélectionner tous les véhicules disponibles et retourner le résultat sous forme d'une **ArrayList**.
- ✓ Une méthode « **invalidDocs** » permettant de sélectionner les documents dont la fin de validité sera dans une semaine à partir d'aujourd'hui. Le résultat est retourné sous forme d'un **ArrayList**.

```
Public class ObjetDistant extends UnicastRemoteObject implements
intObjetDistant
{
    Private Connection conn;
    .....

    //constructeur
    Public ObjetDistant() throws RemoteException {
        .....
    }

    // La méthode vehDisponible
    Public ArrayList<Vehicule> vehDisponible() throws
RemoteException {
        .....
    }

    // La méthode vehDisponible
    Public ArrayList<Document> invalidDocs() throws
RemoteException {
        .....
    }
}
```

- 1- Créer l'interface « **intObjetDistant** » qui hérite de la classe « **Remote** » ; (1 pt)
- 2- Pour la classe « **ObjetDistant** » donner le code de :
 - Le constructeur. (1 pt)
 - La méthode **vehDisponible()**. (2 pts)
 - La méthode **invalidDocs()**. (2 pts)
- 3- Le code suivant représente la méthode **main()** du serveur :

```
1 public static void main(String[] args) {
2     try {
3         LocateRegistry.createRegistry(1099);
4         ObjetDistant obj = new ObjetDistant();
5         Naming.rebind("rmi://NomServeur:1099/NOBJ", obj);
6     } catch (Exception ex) {
7         System.out.println("Erreur : "+ex.toString());
8     }
9 }
```

Expliquer l'utilité des lignes de codes numéro 3, 4 et 5.

(1,5 pt)

PROGRAMME COTÉ CLIENT :

Chaque client possède une interface graphique permettant la récupération des informations sur les véhicules disponibles (*on se limite dans ce cas à l'affichage de la liste des véhicules disponibles : utilisation de la méthode distante **vehDisponible***). L'interface graphique des clients est la suivante :

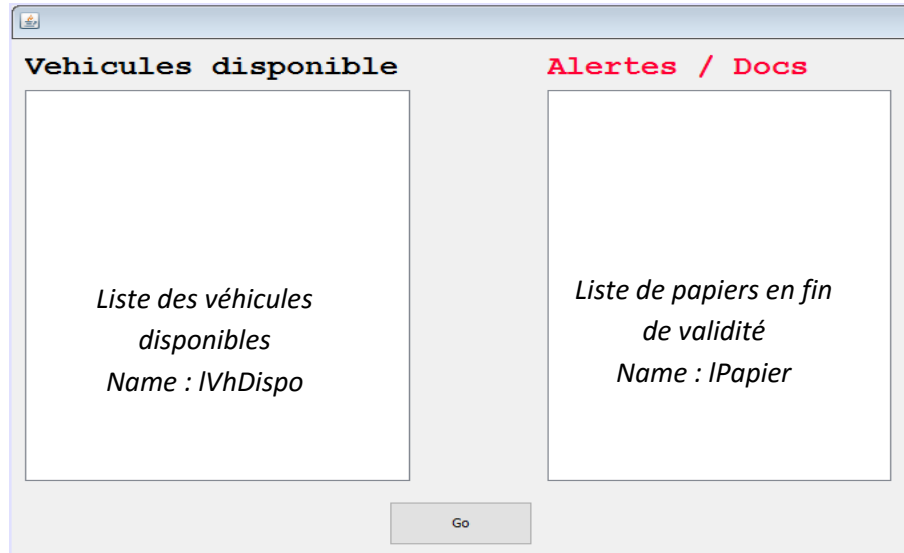


Figure 9 : Interface client

Donner le code de :

- 4- La méthode événementielle de « **JFrame** » lors de son ouverture. (1 pt)
- Utilise la méthode « **Naming.lookup** » pour récupérer la référence de l'objet distant.
 - Stock cette référence dans l'objet « **Obj** » définit comme attribut de la forme **JFrame**.

```
private void formWindowOpened(java.awt.event.WindowEvent evt) {  
    .....  
}
```

- 5- La méthode événementielle du bouton « **Go** » : Permet d'exécuter la méthode « **vehDisponible** » sur le serveur. La valeur retournée (**ArrayList**) doit être affichée dans la liste « **Liste** ». (0,5 pt)

```
private void GoActionPerformed(java.awt.event.ActionEvent evt) {  
    .....  
}
```