Laboratoire de Java

2ème informatique de gestion et 2ème informatique et systèmes (industrielle et réseaux) 2019-2020

Claude Vilvens– Christophe Charlet -
Sébastien Calmant – Jean-Marc Wagner

Projet "Inpres Harbour"

Contexte de développement

1. Préambule

L'Unité d'Enseignement " **Développement orienté objets et multitâche** " (gestion: 9 ECTS (105 h) / indus & réseaux: 8 ECTS (94 h)) se structure en trois Activités d'apprentissage de la manière suivante :

- ♦ AA: Programmation du multi tâche léger- Threads (gestion: 30h (29%) / indus & réseaux: 19h (20%))
- ♦ AA: Programmation orientée objet Unix et Windows - Java (gestion: 30h (29%) / indus & réseaux: 30h (32%))
- ♦ AA: Programmation orientée objet Windows- C# (gestion: 45h (42%) / indus & réseaux: 45h (48%))

Les travaux de programmation réseaux présentés ici constituent la description technique des travaux de laboratoire de l'AA " **Programmation orientée objet Unix et Windows - Java** ".

2. Le projet "Inpres-Harbour"

Le thème de ce laboratoire de programmation Java est la gestion élémentaire d'un port de plaisance et pêche. Il s'agit essentiellement de régler les déplacements des bateaux dans la rade du port, rade dont l'entrée ne laisse passer qu'un bateau à la fois. La gestion des points d'amarrages (à des pontons ou à des quais) est du ressort de la capitainerie tandis que les entrées sont gouvernées par un poste de contrôle situé près du phare situé à l'extrémité de la jetée est. Le personnel de ces organes de contrôles utilise dans les deux cas une application Java, respectivement **Appli_Capitainerie** et **Applic_Phare**.

Le développement de ces applications fera intervenir :

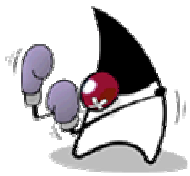
- la conception de classes et interfaces répartis dans des packages structurés;
- les bases de la programmation Java (notamment les packages, interfaces et classes abstraites, exceptions);
- les interfaces graphiques principalement Swing (notamment les JComboBox, JList et JTable);
- les classes utilitaires (containers comme Vector, LinkedList et Hashtable, StringTokenizer, Date-Calendar-DateFormat-Timezone);
- l'utilisation de la ligne de commande élémentaire (javac, java, jar);
- les flux et plus particulièrement les techniques de sérialisation sur fichier, l'utilisation de fichiers Properties et la lecture/écriture de fichiers textes;
- l'utilisation d'une librairie de communication réseau élémentaire (simple échange de chaînes de caractères) disponible sous forme d'un jar;
- les Java Beans avec les chaînes d'événements propriétaires et de type "propriété liée";
- la portabilité Windows-Unix (exécution d'une application Java sur une autre plate-forme, éventuellement à distance avec un serveur X-Window);
- une première approche des threads.

3. Les objectifs et les outils

L'ensemble des travaux proposés ici est à réaliser en utilisant l'environnement de développement **NetBeans version 8.***, basé sur le **JDK 1.8**. Comme éditeur de texte plat, on pourra utiliser **JEdit 4.2** tandis **Xming 6.9** sera le serveur X-Window privilégié (si il est nécessaire de l'utiliser).

Le dossier est, *à priori*, à réaliser **par équipe de deux étudiants** mais **peut aussi être présenté par un étudiant seul** qui le désirerait, avec des avantages et des inconvénients qui s'échangent selon le choix posé.

Un petit conseil : ***lisez bien l'ensemble de l'énoncé*** avant de concevoir (d'abord) ou de programmer (après) une seule ligne ;-). Ceci vous permettra non seulement d'avoir une vision globale du projet mais aussi de déjà remarquer des traitements communs à des points différents des applications. Prévoyez une schématisation des diverses classes (peut-être des diagrammes de classes UML ?) et élaborez d'abord "sur papier" (donc sans programmer directement) les divers scénarios correspondant aux fonctionnalités demandées.



4. Les règles d'évaluation

1) L'évaluation établissant la note de l'AA de " Programmation orientée objet Unix et Windows - Java" est réalisée de la manière suivante :

- **théorie** : un examen écrit en juin 2020, sur base des notes de cours ("Java I: programmation de base") et de listes de questions de théorie à préparer, listes fournies au fur et à mesure; il sera coté sur 20;
- **laboratoire** : **2** évaluations (une fin avril, **non remédiable**, et une durant la 1ère session, remédiable), chacune cotée sur 20; la **moyenne pondérée de ces 2 cotes** (poids respectifs de **35%** et **65%**) fournit une note de laboratoire sur 20;
- note finale : **moyenne de la note de théorie (poids de 5/10) et de la note de laboratoire (poids de 5/10)**.

Cette procédure est d'application tant en 1ère qu'en 2ème session.

2) *Dans le cas où les travaux sont présentés par une équipe de deux étudiants*, chacun d'entre eux doit être capable d'expliquer et de justifier l'intégralité du travail (pas seulement les parties du travail sur lesquelles il aurait plus particulièrement travaillé).

3) En 2ème session, un **report de note** est possible séparément pour **la note de laboratoire** ainsi que pour **la note de théorie** **pour des notes supérieures ou égales à 10/20**.
Toutes les évaluations (théorie ou laboratoire) ayant des **notes inférieures à 10/20** sont **à représenter dans leur intégralité**.

La première partie des travaux de programmation réseaux sera **évaluée** par l'un des professeurs du laboratoire **à partir de la semaine du 20 avril 2020** (avec rentrée d'un dossier papier - le délai est à *respecter* impérativement).

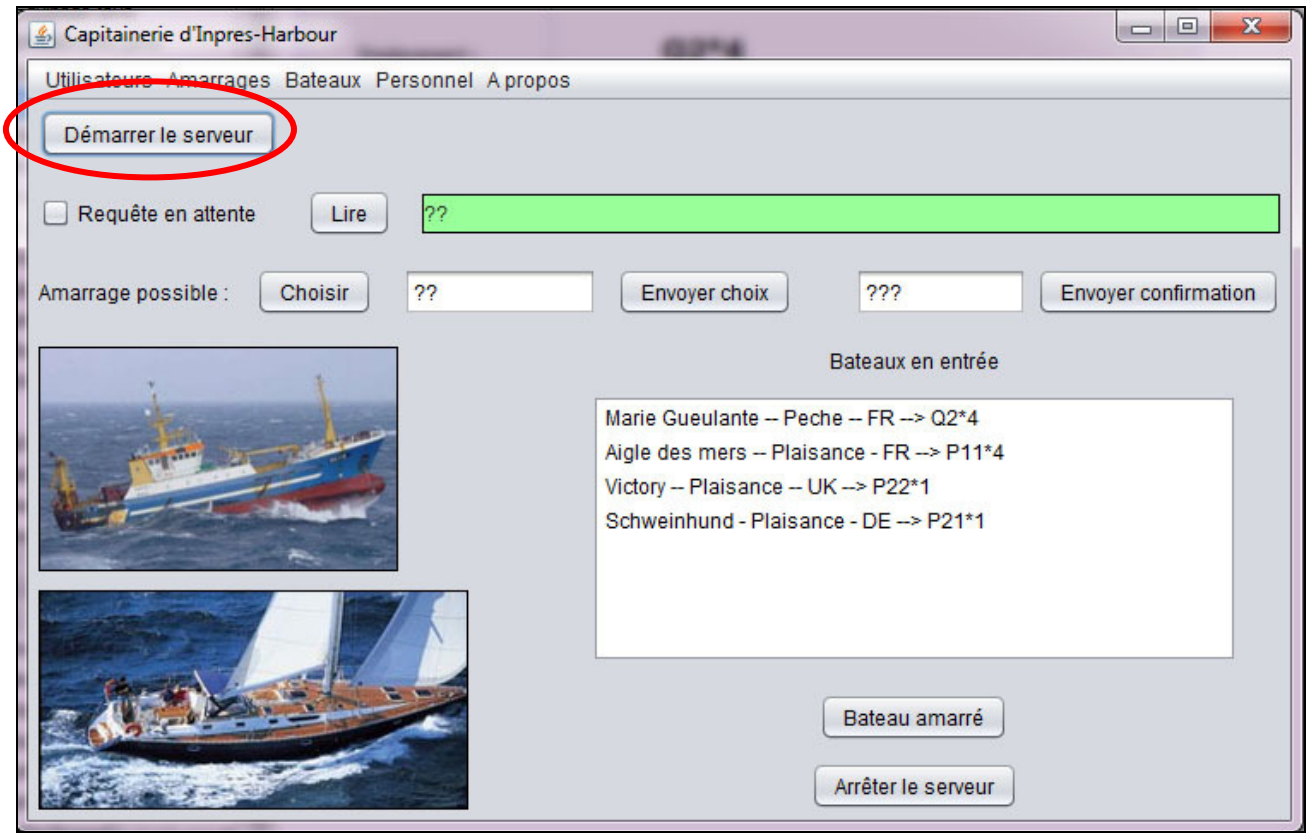
La deuxième partie sera **évaluée** lors de l'examen de laboratoire en juin 2020 (avec rentrée d'un dossier papier).

Présentation du fonctionnement et des applications du port

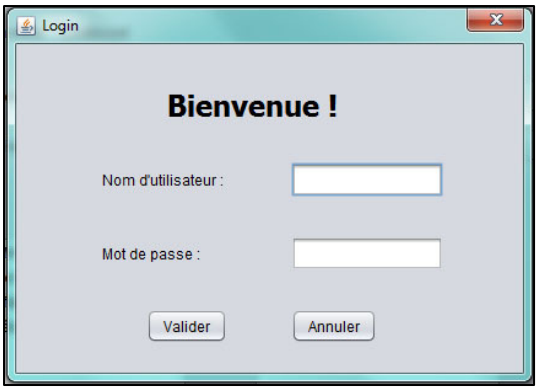
I. Les applications du port

Le contexte est donc ici celui d'un port, Inpres-Harbour, située en Boursoulavie (pays voisin de la Batracie) et dédié à des activités de plaisance et de pêche. Les navires qui y transitent sont de faible tonnage (quelques dizaines de tonneaux). Schématiquement, Inpres-Harbour ressemble à ceci :

Au fond de la rade, la capitainerie qui gouverne le port utilise une application **Applic_Capitainerie** qui ressemble à ceci :

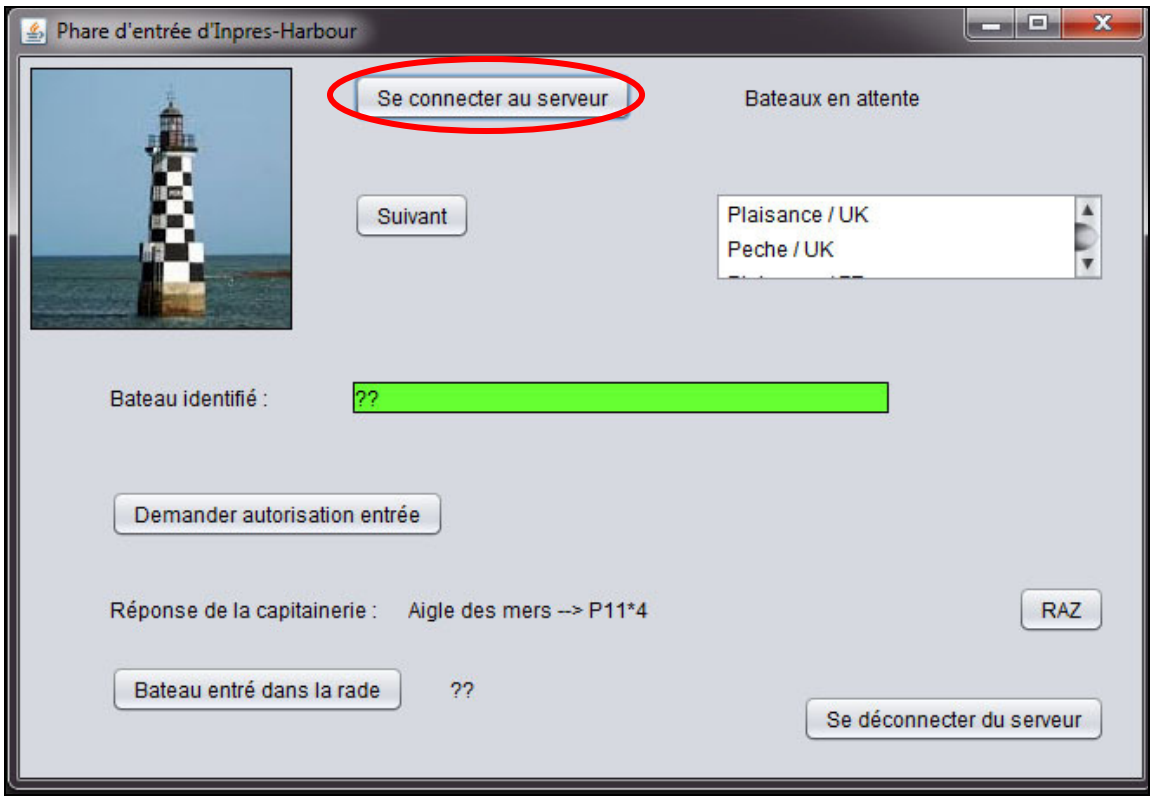


Cette application est fonctionnelle dès que l'utilisateur, qui a du se faire reconnaître par une procédure classique de login-password :



fait démarrer le serveur TCP que l'application contient.

Près du phare situé à l'entrée de la rade du port se trouve un poste de contrôle utilisant une application **Applic_Phare** ressemblant à ceci :



Cette application est fonctionnelle dès que l'utilisateur (qui a du se faire reconnaître par une procédure classique de login-password) fait démarrer le client TCP que l'application contient, client qui se connecte sur le serveur de l'application de la capitainerie.

II. Fonctionnement sommaire du port

Quand un navire parvient aux abords du port, il contacte par radio le poste d'entrée situé près du phare situé à l'entrée pour signaler qu'il est en attente. Cette demande se résume à fournir un type de bateau (plaisance ou pêche) et un pavillon, c'est-à-dire une nationalité, libellée selon le code international comme par exemple FR, BE, GB, DE, NL (voir liste sur http://www.iso.org/iso/fr/french_country_names_and_code_elements). Comme les britanniques sont un peu particuliers, ils leur arrivent d'utiliser "UK" au lieu de "GB". Cette demande est simplement mémorisée dans une boîte de liste (qui, pour la simulation, sera alimentée par une chaîne de Java Beans - voir plus loin). Les demandes seront traitées dans l'ordre d'arrivée de la demande.

La description précise des communications entre le phare d'entrée et la capitainerie sera fournie plus loin (paragraphe 9.3). Qu'il nous suffise de savoir à ce stade que l'opérateur du phare d'entrée va contacter la capitainerie pour vérifier l'existence d'une place d'amarrage libre. Si cet emplacement existe, le navire est autorisé à entrer dans le port pour s'amarrer à l'emplacement désiré. Le capitaine du navire n'a alors plus qu'à se rendre physiquement à la capitainerie pour les formalités d'enregistrement.

Evaluation 1

Dossier :
- ♦ schéma UML (diagramme de classe statique) des classes non graphiques de l'application;
- ♦ explication et code correspondant à un login;
- ♦ explication et code correspondant à l'enregistrement des informations relatives à un bateau qui vient de s'amarrer.

I. Les classes et interfaces Java

1. Les classes de base

Il nous faut donc tout d'abord définir les classes qui vont matérialiser nos bateaux et leur équipage, ainsi que les infrastructures du port (essentiellement les quais et les pontons).

1.1 Les bateaux et leur équipages

Comme nous entendons donner à nos applications le maximum de généricité, nous n'allons pas définir brutalement une classe Bateau mais successivement :

- ♦ un interface **AvecHumains** : caractérise tout élément physique qui héberge ou contient des Humains : un navire, un avion, un bureau, un parlement, une classe, …; une seule méthode y est déclarée : int getNombreHumains()
- ♦ une classe abstraite **MoyenDeTransport** : implémente AvecHumains et caractérise tout élément physique qui transporte des passagers; une variable membre y représente la forme d'énergie utilisée (essence, diesel, kérosène, vapeur, …)
- ♦ une classe abstraite **MoyenDeTransportSurEau** : hérite de MoyenDeTransport et caractérise tout moyen de transport dans l'élément aquatique; une variable membre y représente le fait d'être submersible ou pas (n'oublions pas les sous-marins ! même si cela ne servira pas souvent ici)
- ♦ un interface **AUnIdentifiant** : caractérise tout élément que l'on peut référencer au moyen d'un identifiant unique (un prof de SGBD parlerait de clé primaire); une seule méthode y est déclarée : String getIdentifiant()
- ♦ une classe **Bateau** (ah, quand même !) : hérite de MoyenDeTransportSurEau et comporte divers renseignements comme : le nom, le port d'attache, le tonnage, la longueur et le pavillon (c'est-à-dire sa nationalité); cette classe implémente AUnIdentifiant : l'identifiant est le nom du bateau complété par son port d'attache; un bateau référencie toujours un équipage (voir ci-dessous)
- ♦ deux classes dérivées de Bateau : **BateauPlaisance** (avec le type de permis nécessaire : plaisance option côtière, plaisance extension hauturière, …) et **BateauPeche** (qui a un type : thonier, morutier, crevettier, …)

Toute instanciation d'un Bateau qui ne désigne pas explicitement son nom et/ou son port d'attache donnera lieu au lancement d'une exception **ShipWithoutIdentificationException**.

1.2 L'équipage des bateaux

Les marins sont représentés par une instance de la classe **Marin** qui mémorise sa fonction (capitaine, second, bosco, maître mécanicien, etc). Il a aussi, bien sûr, un nom, un prénom et une date de naissance (simplement sous forme d'une chaîne de caractères), ce qui est repris dans la classe mère **Humain**. La classe Marin implémente aussi l'interface AUnIdentifiant : l'identifiant est la concaténation des nom, prénom et date de naissance.

Toute instanciation d'un Marin qui ne désigne pas explicitement son nom et son prénom donnera lieu au lancement d'une exception **SailorWithoutIdentificationException**.

La classe **Equipage** regroupe évidemment un ensemble de marins. Cependant, la capitainerie et le second (si il existe) sont des variables membres distinctes du reste de l'équipage.

1.3 Les amarrages

Quand les bateaux sont admis dans la rade du port, ils vont s'amarrer en un point d'amarrage qui peut être un quai ou un ponton simple ou double. On désignera en général par la classe abstraite **Amarrage** ces différentes possibilités : il implémente AUnIdentifiant et sa seule méthode propre est int getCapacite(), pour indiquer le nombre de places disponibles pour des bateaux moyens (de 15 m maximum).
Pour implémenter cette classe abstraite, on aura les classes :

- **Quai** : les bateaux se rangent le long d'un quai; la méthode

MoyenDeTransportEau[] getListe()

fournit la liste des bateaux amarrés;

- **Ponton** : il s'agit d'une plate-forme flottante qui coulisse le long de piquets selon la marée; les bateaux se rangent perpendiculairement au ponton, séparés par des pontons auxiliaires appelés des catways; la méthode

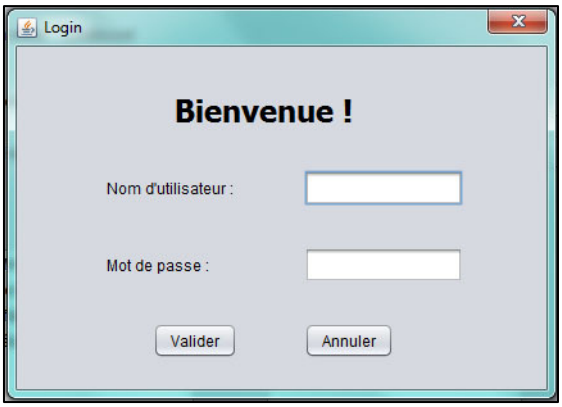
MoyenDeTransportEau[] getListe(int cote)

fournit la liste des bateaux amarrés sur le côté 1 ou 2.

II. Les premières fonctionnalités d'Applic_Capitainerie

2. Le login à la capitainerie

Au préalable, un responsable du port qui veut diriger les opérations depuis la capitainerie doit se faire authentifier sur base d'un habituel login (concaténation de nom et prénom)-password; provisoirement, cela se fera en utilisant **une hashtable statique** en mémoire (clé=login, valeur=mot de passe) :



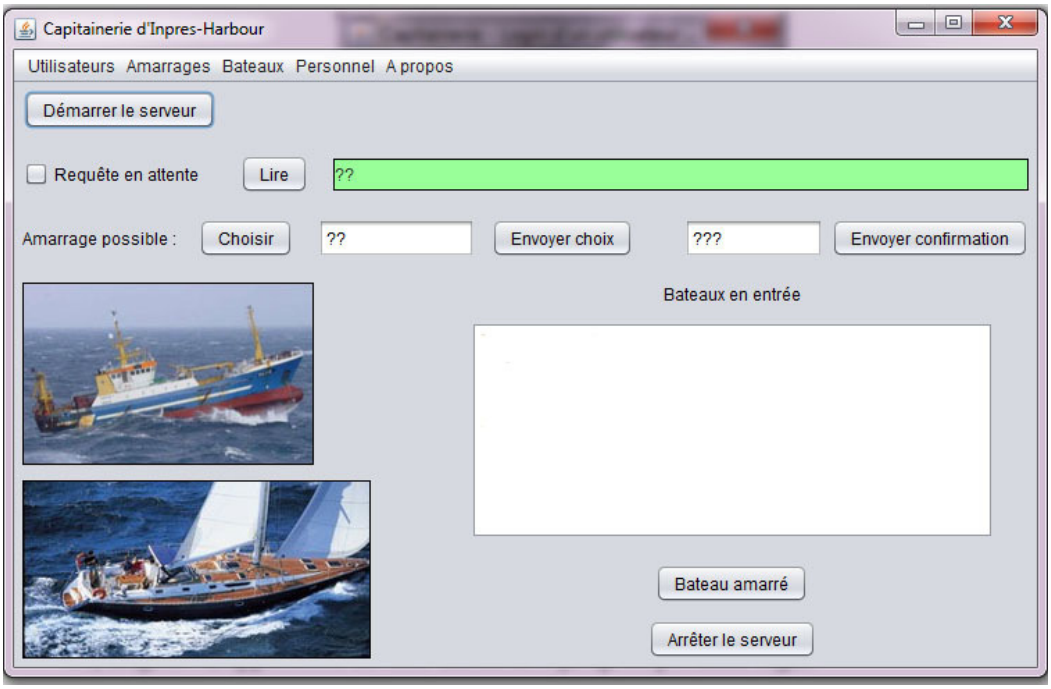
avec une boîte de dialogue d'erreur **DialogErreur** :

qui correspond bien sûr à une exception instance de **LoginException**.

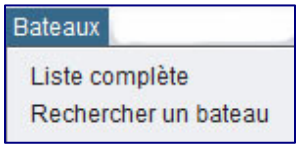
Une fois cette phase réussie, le responsable qui utilise l'application entre dans l'application. Il serait de bon ton que son nom apparaisse dans la barre de titre de la fenêtre de l'application.

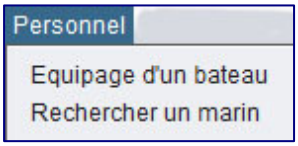
3. Quelques fonctionnalités de base

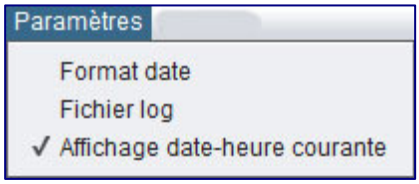
En cas de succès, on obtient donc une fenêtre du ce type:



La structure des menus d'Applic_Capitainerie sera du type suivant:







et

On peut déjà implémenter quelques fonctionnalités simples …

3.1 Le menu "Utilisateurs"

Il comporte notamment les items :

3.1a) Login : Utilisable seulement après un logout et re-présente la boîte de dialogue d'authentification.

3.1b) Logout : L'application devient inutilisable jusqu'au prochain login.

3.1c) Nouveau : au moyen d'une boîte de dialogue, l'utilisateur courant peut créer un nouveau couple (login, password).

Tous les comptes (login, password) sont mémorisés dans une hashtable (pour la 2ème évaluation, elle sera sérialisée).

3.2 Le menu "A propos"

Il comporte les items :

3.2a) Auteurs : Ce choix affiche simplement une boîte de dialogue avec la date d'entreprise de l'application, les prénom et nom de son(ses) auteur(s) ainsi peut-être qu'une photo de ce(s) dernier(s) ;-)

3.2b) Aide : Présente dans une JTextArea une mini-mode d'emploi de l'application.

3.3 Le menu Paramètres

3.3a) Format date : Il permet à l'utilisateur de demander un affichage correct (à son point de vue) de la date courante. On utilise une boîte de dialogue qui comporte

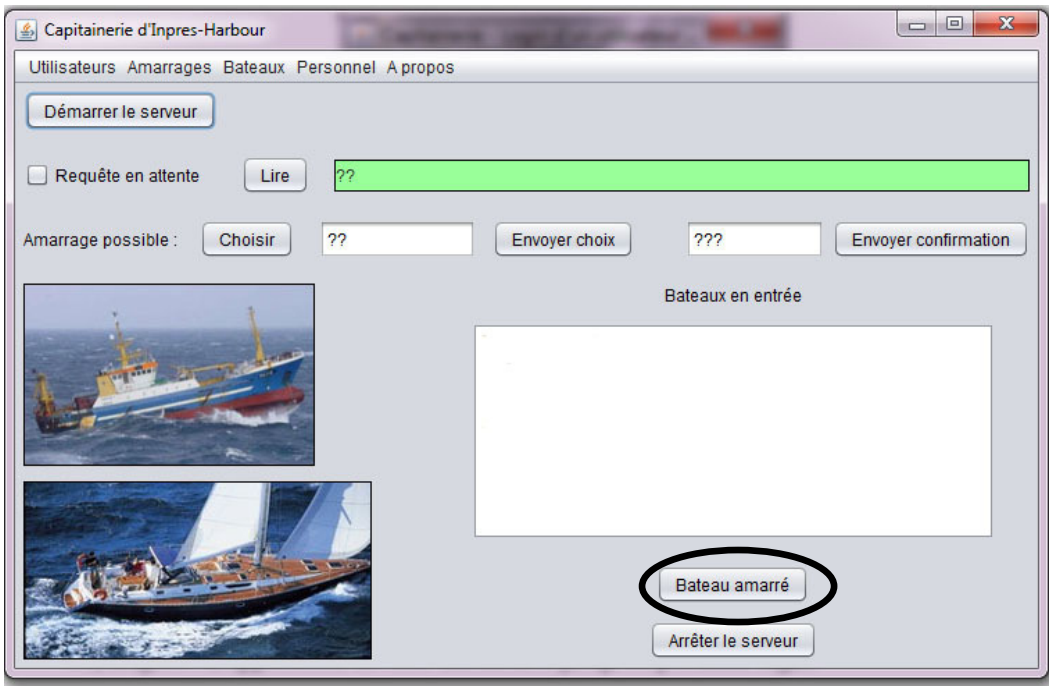
- une boîte combo pour choisir entre France, Royaume Uni, Allemagne, Italie ou U.S.A.;
- une boîte combo pour choisir le format de la date;
- une boîte combo pour choisir le format du temps.

Quand la boîte est refermée, la date courante est utilisée avec les choix ainsi définis pour permettre de construire la chaîne de caractères affichée.

3.3b) Affichage date-heure courante : si cette option est cochée, la date est affichée dans le coin supérieur droit de la fenêtre de l'application (il faudra donc l'ajouter dans le GUI montré ci-dessus).

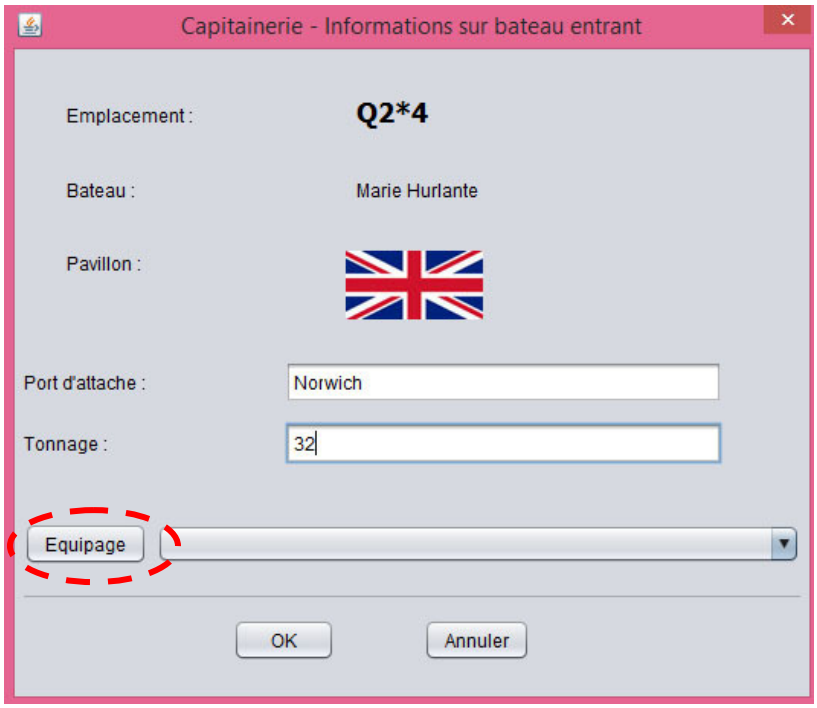
4. Les informations sur un bateau qui vient d'amarrer

Bien que nous n'ayons pas encore installé le processus d'arrivée d'un bateau, nous pouvons déjà créer l'interface graphique qui permet d'encoder les informations concernant un bateau qui vient de s'amarrer dans le port :

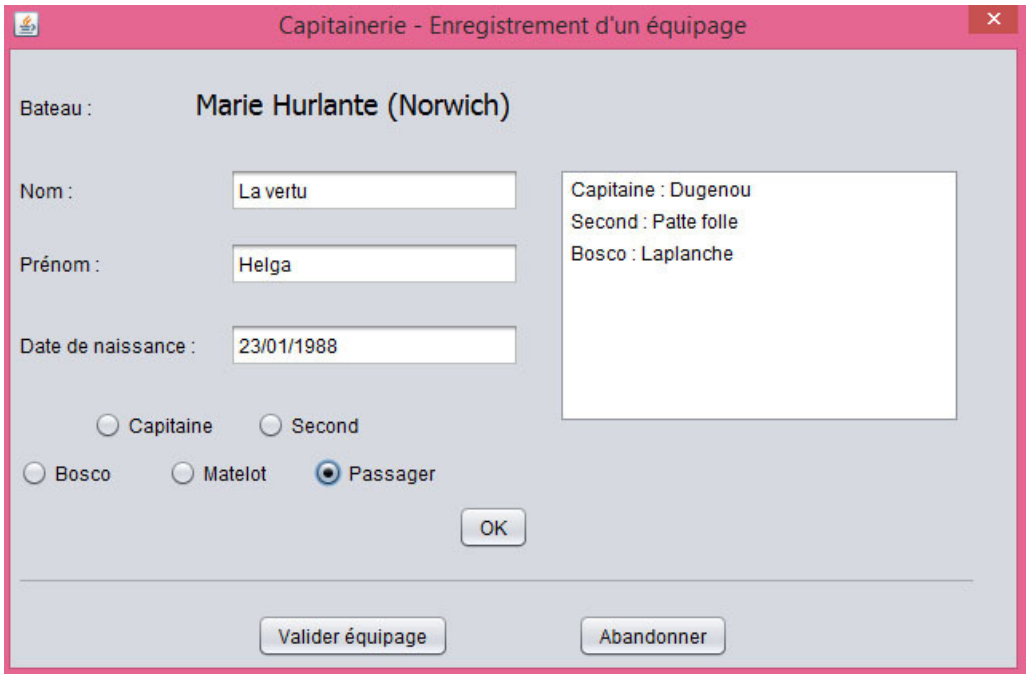


Une fois arrivé (après les opérations d'entrée et de navigation dans le port - voir plus loin),

♦ le capitaine du bateau de plaisance se rend à la capitainerie pour confirmer sa présence, compléter les informations le concernant et décliner les identités de son équipage; l'appui sur le bouton "Bateau amarré" conjugué avec le choix du bateau dans la liste (celle-ci sera provisoirement remplie à la dure dans l'application, en attendant les développements ultérieurs) fait apparaître :



puis



donnant finalement l'équipage complet :

l'ensemble de ces renseignements, avec la date d'arrivée, est destiné à être mémorisé (voir plus loin);

♦ le capitaine d'un bateau de pêche fait la même démarche, mais il n'est pas nécessaire d'enregistrer l'équipage qui est déjà connu.

Ces informations seront effectivement enregistrées dans des fichiers lors de la 2ème partie de ce laboratoire.

5. Portabilité

Pour rappel, afin d'illustrer la portabilité de Java,

l'application demandée doit fonctionner sur une machine **Windows** **et** sur une machine **UNIX** de l'INPRES.

On se servira de la console comme outil de trace/log en affichant les coordonnées des objets créés au moyen du GUI.

Evaluation 2

Dossier :

- étapes manuelles du développement et du déploiement des points 6 et 12;
- fichiers properties utilisés;
- schéma UML (diagramme de classe statique) de la chaîne de beans simulant l'arrivée des bateaux.

III. Développement manuel

6. Un peu de ligne de commande

Imaginons qu'au matin de l'évaluation, une météorite tombe sur votre ordinateur de développement … Résultat : plus de Netbeans et presque plus de projet "Inpres-Harbour" : il ne reste que les fichiers *.java. Il faudrait pourtant faire fonctionner l'application …

Après avoir copié tous les fichiers java dans le répertoire "harbour" (à la racine), on demande depuis la ligne de commande en console

1) de compiler et d'exécuter l'application;
2) de récréer un jar pour l'application puis de l'utiliser pour exécuter l'application.

IV. Persistance des données du projet

7. La persistance des login-passwords

Il s'agit simplement de remplacer la hashtable mémoire des login-passwords par un objet et un fichier properties. On veillera à ce que le fichier properties puise être retrouvé facilement et de manière portable lorsque l'on transportera les applications d'une machine Windows (par exemple) vers une machine Unix (par exemple).

8. La persistance de l'état du port

Toutes les informations d'amarrage doivent être **sérialisées**, de manière à pouvoir retrouver l'application dans son dernier état lors d'un redémarrage (autrement dit, **le port doit mémoriser son état**).

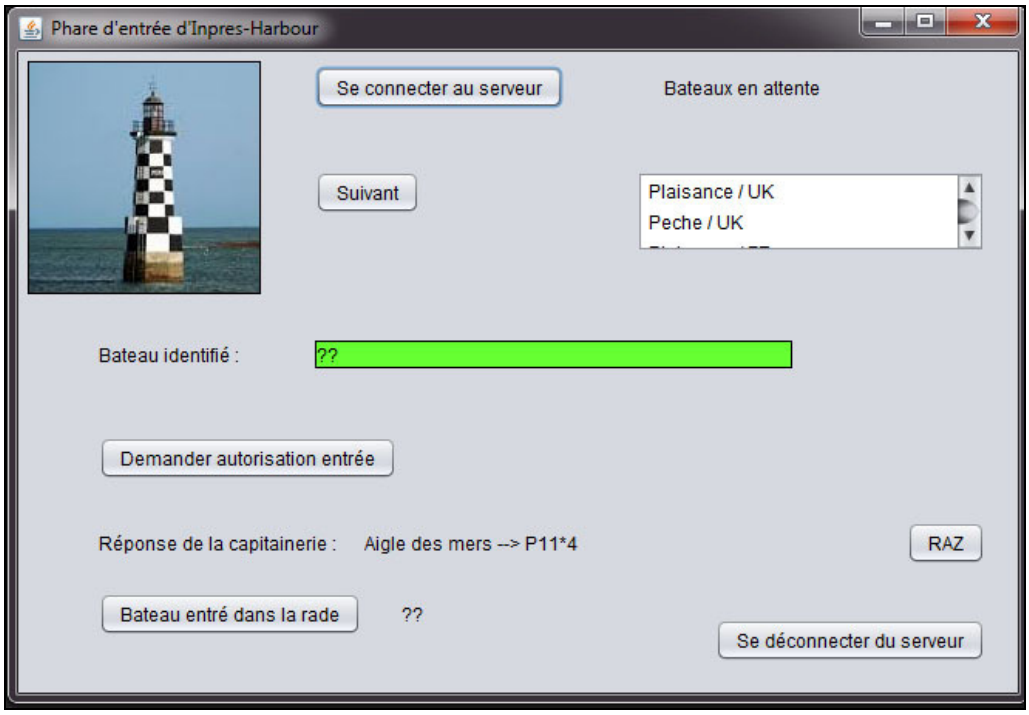
Puisque nous parlons d'amarrages, il est temps de décrire plus précisément le processus d'entrée d'un navire dans le port.

V. Communication entre Applic_Capitainerie et Applic_Phare

9. La communication entre les deux applications

9.1 Implémentation du protocole de communication phare-capitainerie

Il s'agit tout d'abord de créer la deuxième application **Applic_Phare** selon le modèle exposé ci-dessus. Son utilisation sera soumise à la même procédure de login qu'**Applic_Capitainerie**. Pour rappel, son interface ressemble à ceci:



Il s'agit ensuite d'implémenter le dialogue entre les deux applications selon le schéma (le protocole) évoqué ci-dessus. Mais les deux applications doivent communiquer, ce qui réclame une interaction par l'intermédiaire du réseau (en localhost).

9.2 Une librairie de classes basiques de communication réseau

En fait, les deux applications peuvent communiquer entre elles en envoyant des chaînes de caractères. Il s'agit en fait d'une communication réseau TCP/IP, transparente pour le développeur, à priori utilisée sur une seule machine (en "localhost") mais en fait utilisable entre deux machines distinctes.

Pour cela, il est fourni deux classes (appartenant au package **network**) au sein du fichier **BasicStringNetworkLib.jar** (disponible sur l'EV [centre de ressources Vilvens]) :

♦ **NetworkBasicServer** : classe utilisant un thread permettant la réception d'une chaîne de caractères sur un port donné de la machine utilisée; en fait, les chaînes reçues sont mémorisées dans une liste et la méthode getMessage() fournit la première chaîne reçue ou "RIEN" si la liste est vide. Les méthodes principales sont :

public **NetworkBasicServer** (int p, JCheckBox cb);
Constructeur. On fournit le port d'écoute et la checkbox qui sera cochée dans le GUI de l'application si un message entrant est disponible.

public String **getMessage**();
Lecture du message suivant ("RIEN" si il n'y en a pas).

public void **sendMessage**(String m)
Envoi d'un message.

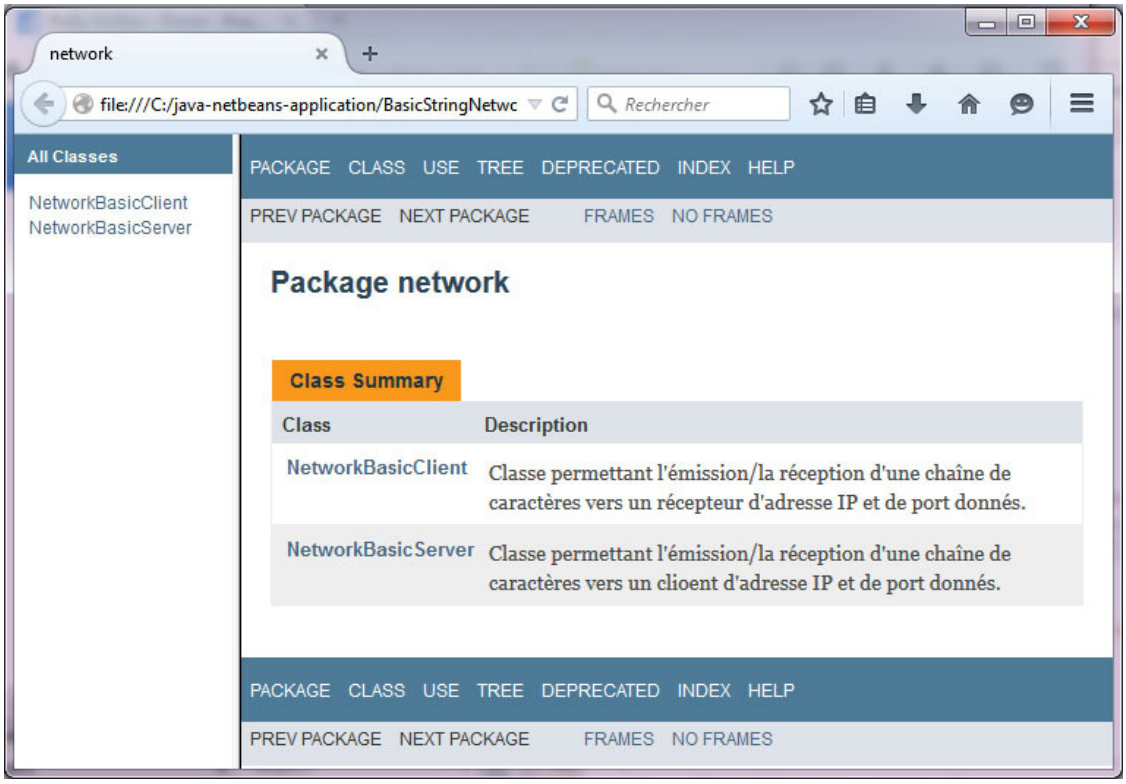
♦ **NetworkBasicClient** : classe utilisant un thread (transparent pour l'utilisateur) permettant l'émission d'une chaîne de caractères vers un récepteur d'adresse IP et de port donnés. Les méthodes principales sont :

public **NetworkBasicClient** (String a, int p);
Constructeur avec connexion.

public String **sendString**(String s)
Envoi d'un message avec attente bloquante de la réponse.

public void **sendStringWithoutWaiting**(String s)
Envoi d'un message sans attente de réponse (simple notification).

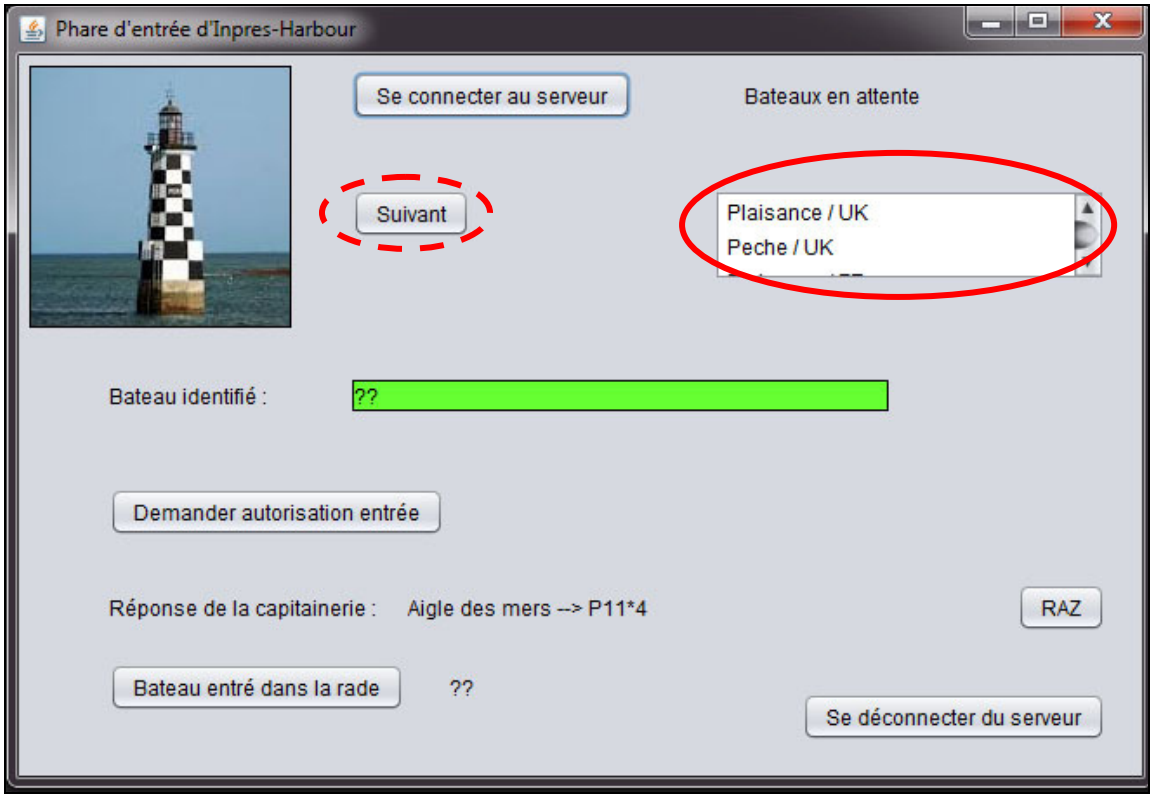
Ces deux classes sont documentées par les javadocs qui se trouvent dans **BasicNetworkDoc.zip** (disponible au même endroit) :



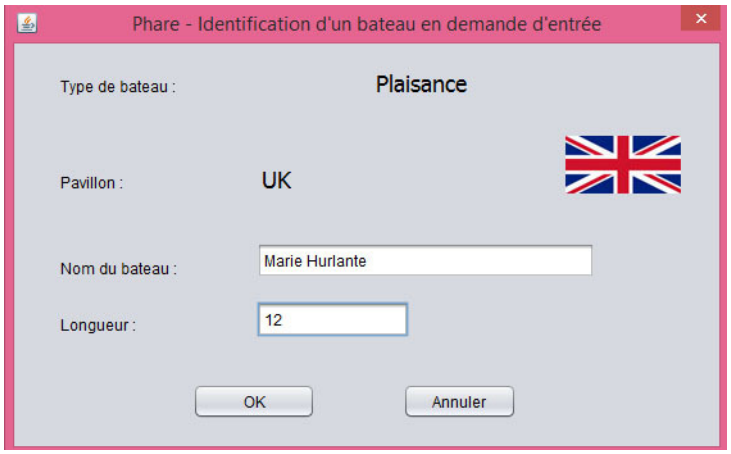
Remarque
Dans cette librairie plus qu'élémentaire, les fins de connexions ne sont pas gérées explicitement, si bien qu'un arrêt brutal d'une application lance une exception, que l'on pourra ignorer ici.

9.3 Description du protocole de communication phare-capitainerie

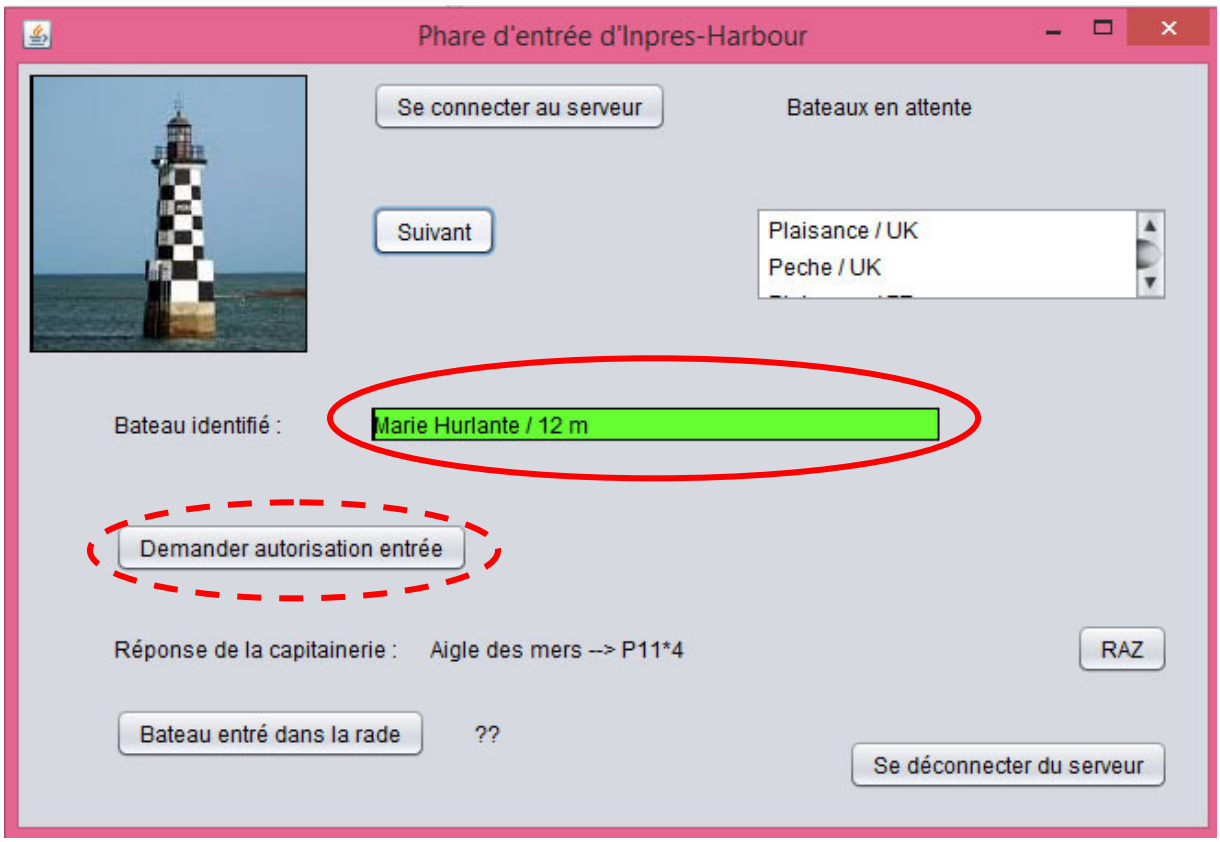
Comme déjà expliqué, quand un navire parvient aux abords du port, il contacte par radio le poste d'entrée situé près du phare situé à l'entrée pour signaler qu'il est en attente. Cette demande se résume à fournir un type de bateau (plaisance ou pêche) et un pavillon. Les demandes seront traitées dans l'ordre d'arrivée de la demande.



Quand l'opérateur d'Applic_Phare peut traiter la demande suivante, il appuie sur le bouton "Suivant", ce qui fait apparaître la boite de dialogue

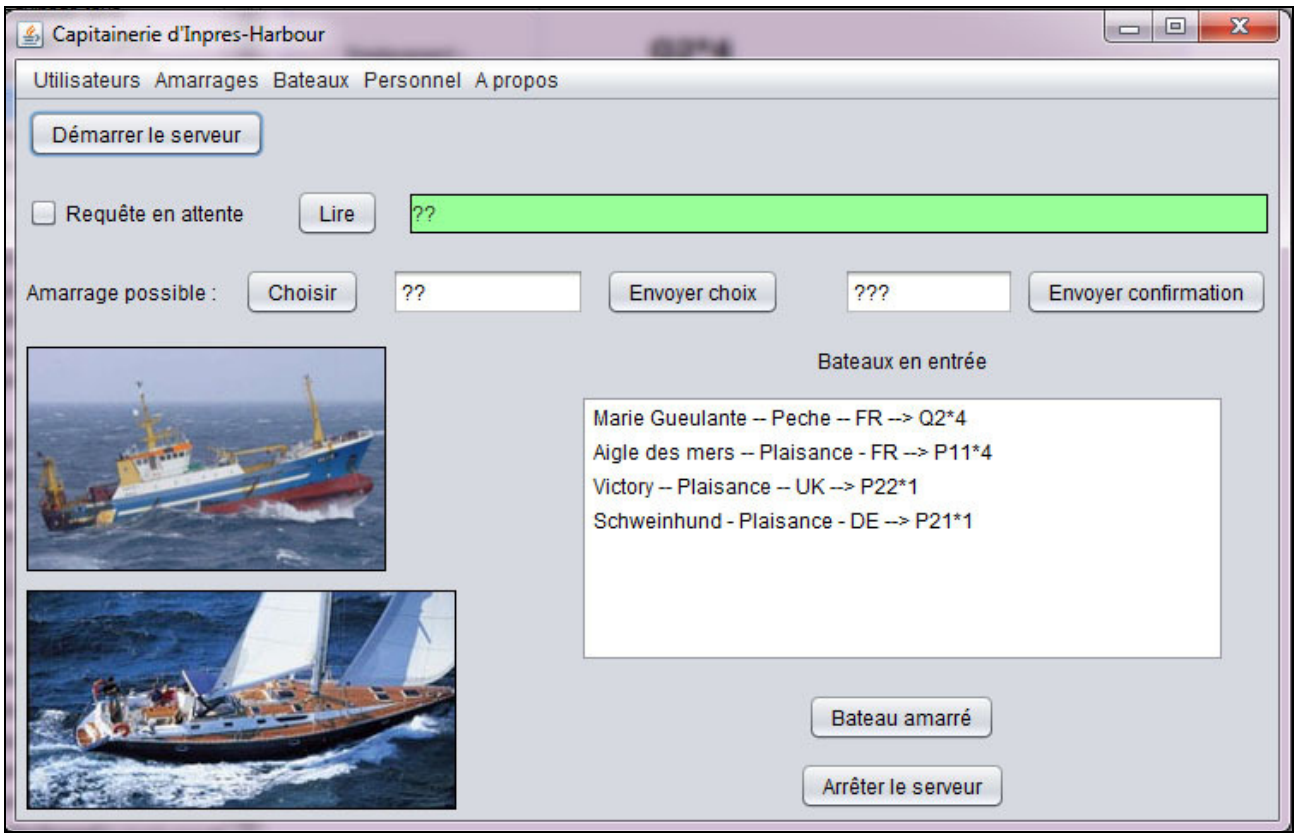


Le capitaine décline donc l'identité de son bateau que l'utilisateur encode (de manière sommaire : nom du bateau et longueur - dans un premier temps, c'est tout ce qui est nécessaire, avec le type de bateau déjà connu, pour accepter ou refuser l'entrée dans le port).



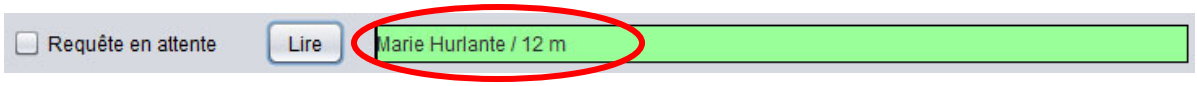
Le responsable du poste d'entrée contacte la capitainerie par la ligne TCP afin de déterminer si une place d'amarrage est disponible, selon le type du bateau (bouton "Demander autorisation entrée"). L'application Applic_Phare reste alors bloquée jusqu'à l'obtention d'une réponse (mesure de sécurité) :

L'Applic_Capitainerie

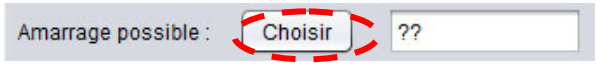


est prévenue qu'une demande est arrivée par le fait que la case à cocher "Requête en attente" est cochée :

Il suffit d'appuyer sur le bouton "Lire" pour prendre connaissance du message :

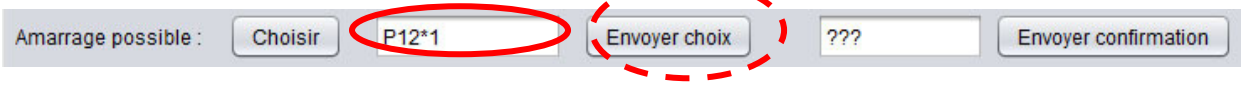


Pour déterminer si un emplacement est libre, on appuie sur le bouton "Choisir"

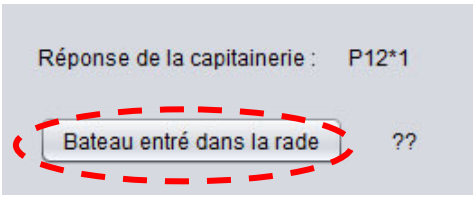


qui fait apparaître, après avoir choisi entre "Plaisance" et "Pêche", l'état des occupations :

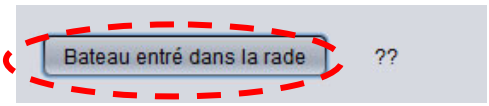
La sélection dans le tableau fait apparaître l'emplacement choisi (ici, le ponton P12*1) et l'appui sur le bouton "Choisir" fait revenir à l'application :



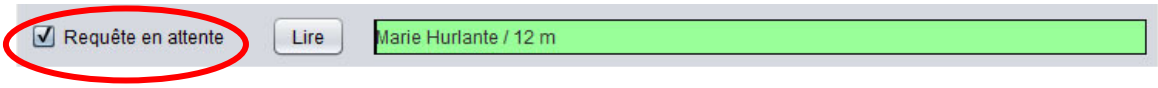
L'appui sur "Envoyer choix" transmet le choix au poste du phare (l'application Applic_Phare se débloque) :



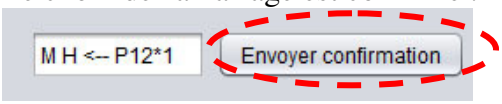
L'opérateur du phare peut transmettre l'autorisation par radio au bateau concerné. Quand le bateau passe devant lui, entrant ainsi effectivement dans la rade, il peut prévenir la capitainerie par l'appui sur "Bateau entré dans la rade".



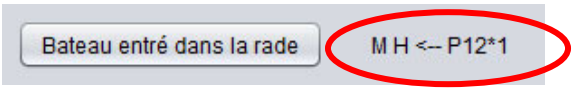
L'Applic_Capitainerie reçoit l'information :



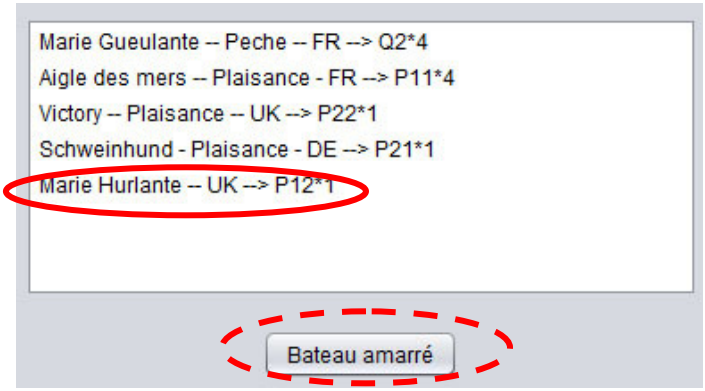
Le choix de l'amarrage est confirmé :



avec côté phare :



et le bateau peut s'y rendre. Dans l'Applic_Capitainerie, le bateau est versé dans la liste "Bateaux en entrée" :



Une fois arrivé, le capitaine du bateau se rend à la capitainerie pour confirmer sa présence, compléter les informations le concernant et décliner les identités de son équipage (pour un bateau de pêche, il n'est pas nécessaire d'enregistrer l'équipage qui est déjà connu). Le point de l'évaluation 1 décrit les détails de l'encodage de ces informations.

Quand un navire désire quitter le port, le capitaine du navire le signale à la capitainerie qui contacte par radio le poste de garde du phare pour lui signaler le départ. Le poste confirmera, toujours par radio, le passage du navire et la date de départ sera enregistrée par la capitainerie. L'amarrage correspondant sera alors libéré.

VI. Les Java Beans

10. La simulation de l'arrivée des bateaux

Tout le scénario développé ci-dessus est très efficace si on dispose des bateaux en attente à l'entrée du port. Cette dernière partie de l'énoncé va permettre de simuler l'apparition de ces bateaux.

10.1 Un thread outil pour les simulations

L'apparition d'un bateau au large sera simulée ici par un tirage de nombres aléatoires : si l'un d'entre eux est multiple d'un nombre de référence, un nouveau bateau en attente sera généré. Afin de ne pas figer l'application de surveillance, un thread assurera cette génération.

Il est fourni pour cela une classe **ThreadRandomGenerator** qui produit un nombre aléatoire dans la fonction qu'il exécute (en fait, sa méthode **run**() héritée de la classe **Thread** de la librairie Java) et "prévient" l'objet qui lui a été passé comme paramètre dans son constructeur, c'est-à-dire qu'il appelle une méthode de cet objet nommée *traiteNombre*(). On peut parler d'un schéma d'"*alarme*". Bien sûr, pour assurer que l'objet en question possède bien cette méthode, il devra implémenter un interface **UtilisateurNombre** qui ne déclare que deux méthodes :

UtilisateurNombre.java

```
package threadsutils;

public interface UtilisateurNombre
{
   String getIdentifiant();
   void traiteNombre(int n);
}
```

Donc, dans notre cas, la fenêtre principale de notre application Applic_Phare (par exemple), implémentera cet interface. Le constructeur du thread :

public **ThreadRandomGenerator**(UtilisateurNombre un, int bi, int bs, int md, int tp) réclame, outre l'objet à "prévenir", les bornes inférieures et supérieures des nombres aléatoires produits, le nombre multiple de déclenchement et le temps de pause entre deux générations (en secondes). En effet, pour que la simulation soit assez aisée à manipuler, on "freine" le déroulement des opérations de génération au moyen de l'appel de la méthode de classe

sleep(<nombre de millisecondes>);

(qui réclame un traitement d'exception).

En résumé :

ThreadRandomGenerator.java

```
package threadsutils;;

/**
 * @author  Charles-Apollon Vilvens
 */

public class ThreadRandomGenerator extends java.lang.Thread
{
   private UtilisateurNombre utilisateurThread;
   private int borneInferieure, borneSuperieure, multipleDeclenchement, tempsPause;
   private int nombreProduit;

   public ThreadRandomGenerator(UtilisateurNombre un, int bi, int bs, int md, int tp)
   {
      utilisateurThread = un; borneInferieure=bi; borneSuperieure=bs;
      multipleDeclenchement = md; tempsPause = tp; nombreProduit = -1;
   }

   public void run()
   {
         Double dr;
         while (true)
         {
                  dr = new Double(borneInferieure +
                           Math.random()*(borneSuperieure - borneInferieure));
                  nombreProduit = dr.intValue();
                  System.out.println(utilisateurThread.getIdentifiant() +
```

```
                    "> nombreProduit = " + nombreProduit);
                if (nombreProduit % multipleDeclenchement == 0)
                {
                   System.out.println(utilisateurThread.getIdentifiant() +
                        "> ---------- !!!!!!! " + nombreProduit + "!!!!");
                   utilisateurThread.traiteNombre(nombreProduit);
                }

                try
                {
                        Thread.sleep(tempsPause*1000);
                }
                catch (InterruptedException e)
                {
                        System.out.println("Erreur de thread interrompu : " + e.getMessage());
                }
        }
   }
}
```

Tout objet qui souhaite utiliser cet outil :

- implémente l'interface UtilisateurNombre ;
- instancie un thread ThreadRandomGenerator (pour réaliser le travail de génération et de test de multiple de déclenchement) et le fait démarrer par **start**().

Lorsque l'UtilisateurNombre attaché au thread est avisé d'une alarme, il lui reste à agir en conséquence, par exemple par un affichage ou par la "mise à feu" d'une chaîne de beans.

10.2 L'application Applic_Phare : les Java Beans

Le mécanisme de détection de l'arrivée de bateaux sera assuré par une chaîne de trois Java Beans :

a) le bean **KindOfBoatBean** est un UtilisateurNombre et lance un ThreadRandomGenerator; lorsqu'une alarme lui est transmise, il génère un deuxième nombre aléatoire pour choisir le type de bateau arrivant :

a1) si le nombre est un multiple de 7, il s'agit d'un bateau de plaisance;
a2) sinon, si le nombre est multiple de 17 : il s'agit d'un bateau de pêche.

Ce bean possède une propriété liée Info (de type String) qui contient le type de bateau détecté ("Plaisance" ou "Peche") et dont le changement de valeur produit donc l'émission d'un PropertyChangeEvent.

b) le bean **BoatBean** est avisé de ce changement de propriété et génère un **BoatEvent** (objet à créer manuellement, ainsi que le listener correspondant à implémenter par le futur bean récepteur) qui contient les infos connues sur l'événement (date heure de réception, type de bateau) avec en plus un pavillon déterminé par un nombre aléatoire;

c) le bean **NotifyBean** est celui qui est prévenu et joue deux rôles :
- il fait apparaître une boîte de dialogue modale qui annonce l'arrivée d'un bateau;
- il met à jour les informations affichées dans la liste de l'Applic_Phare affichée en permanence en ajoutant le bateau en fin de file.

VII. Quelques fonctionnalités supplémentaires

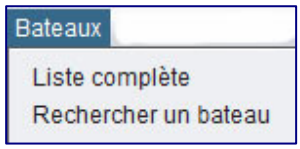
11. Les fonctionnalités additionnelles

On peut à présent ajouter de nouvelles fonctionnalités liées au menu.

11.1 Le menu "amarrages"

Les deux items permettent d'obtenir dans une boîte de dialogue l'occupation du port de plaisance et du port de pêche. En fait, il s'agit simplement des boîtes de dialogue permettant à la capitainerie de choisir un amarrage (voir paragraphe 5) si ce n'est que le bouton "Choisir" se nomme à présent "Ok".

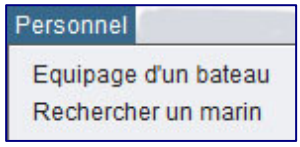
11.2 Le menu "Bateaux"



11.2a) Liste complète : Il s'agit d'obtenir la liste complète des bateaux abrités dans le port (nom du bateau, port d'attache et amarrage), liste idéalement triée par ordre alphabétique sur le nom des bateaux (une idée : la classe ArrayList possède une méthode sort() …).

11.2b) Rechercher un bateau : pour obtenir toutes les informations le concernant (longueur, pavillon, date d'arrivée, peut-être date prévue de départ) et la composition de l'équipage.

11.3 Le menu "Personnel"



11.3a) Equipage d'un bateau : on recherche un bateau par son nom et son port d'attache pour obtenir la liste des membres d'équipage (il s'agit donc d'un sous-ensemble de la requête précédente …)

11.3b) Recherche d'un marin : il s'agit de retrouver toutes les informations concernant un marin désigné par son nom et son prénom.

12. Quelques spécifications supplémentaires

1) Les traces sur la console sont remplacées finalement par un fichier de log; l'utilitaire de log est un JavaBean **FichierLog**; ce bean ne génère pas d'événement et est donc simplement une classe qui écrit et lit dans un fichier texte (Reader et Writer); on utilisera bien sûr un JTextArea pour la visualisation ("Paramètres → Fichier log" dans le menu - point 9.4bci-dessus).

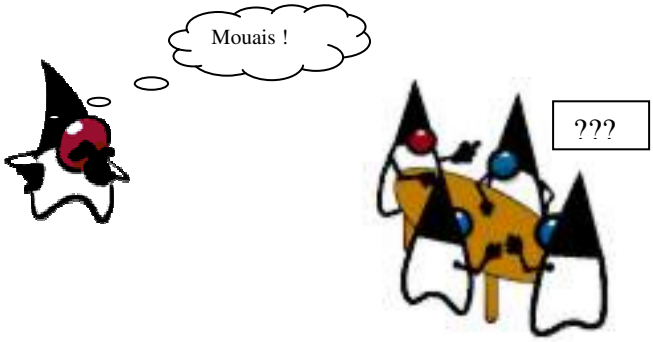
2) On s'en doute, l'application dépend d'un fichier de configuration qui est également un ***fichier properties*** : on y trouve des indications telles que

- le nom du fichiers properties (et d'autres fichiers properties éventuels) et de sérialisation des bateaux arrivés et repartis;
- le port d'écoute pour les communications réseaux;
- les nombres de référence utilisés pour les simulations par tirage aléatoire (7 et 17 dans l'énoncé - on devrait pouvoir moduler);
- le temps de sommeil du ThreadRandomGenerator(1000 ms dans l'exemple);
- le nom du fichier log.

3) Les deux fenêtres principales d'application doivent faire apparaître la date et l'heure de démarrage de l'application (pas la date-heure courante) selon le format fixé par le fichier de configuration.

4) A priori, les développements proposés ci-dessus s'effectuent avec NetBeans 8.* sous Windows. Une fois réalisée, **on vérifiera que l'on sait exécuter l'application depuis une fenêtre DOS (technique des class en jar et java –jar) et on demande ensuite de porter l'application sous UNIX et de la tester sur la machine Sunray** - ou l'inverse.

Bon travail !



s: CV, CC, SC & JMW