

Projet L052 J2ME : TicTacToe



Sommaire

Introduction.....	2
1. Technologies utilisées.....	3
1.1. J2ME.....	3
1.2. RMS.....	4
1.3. Bluetooth.....	4
1.3.1. Description.....	4
1.3.2. Schémas de connexion.....	5
2. Analyse.....	7
2.1. Analyse des besoins :.....	7
2.1.1. Le jeu.....	7
2.1.2. L'affichage.....	7
2.1.3. La liaison bluetooth.....	7
2.1.4. Le son.....	7
2.1.5. Le menu.....	7
2.2. Diagramme UML théorique.....	8
3. Implémentation.....	10
3.1.1. Le jeu.....	10
3.1.2. L'affichage.....	10
3.1.3. La liaison bluetooth.....	11
3.1.4. Stockage et gestion des scores.....	13
3.1.5. Le son.....	13
3.1.6. Le menu.....	14
3.2. Diagramme UML d'implémentation.....	14
3.3. Problèmes rencontrés.....	15
Conclusion.....	17

Introduction

L'objectif de ce projet est d'écrire un programme J2ME qui permette à deux personnes d'utiliser un appareil (PDA, téléphone portable) pour jouer au tic-tac-toe par l'intermédiaire d'une communication bluetooth.

L'interface graphique de l'application devait présenter les différentes cases du jeu avec des croix ou des ronds pour indiquer les différents choix des adversaires.



Fig. 1 : Screenshot pendant la phase de jeu.

Comme on peut l'apercevoir sur la figure 1, chaque joueur peut connaître son score en regardant la zone de texte défilant qui se situe en haut de l'écran du téléphone. Nous avons aussi mis en miniature un rond ou une croix, afin que le joueur ne puisse pas se tromper d'avatar.

A la fin de la partie, une mélodie signale à chaque joueur si il a gagné ou perdu, le résultat de la partie précédente est inscrit sur la zone de texte défilant.

1. Technologies utilisées

1.1. J2ME

Java 2 Micro Edition est une architecture technique dont le but est de fournir un socle de développement aux applications embarquées. L'intérêt étant de proposer toute la puissance d'un langage tel que Java associé aux services proposés par une version bridée du Framework J2SE : J2ME.

Les terminaux n'ayant pas les mêmes capacités en terme de ressources que les ordinateurs de bureau classiques (mémoire, disque et puissance de calcul), la solution passe par la fourniture d'un environnement allégé afin de s'adapter aux différentes contraintes d'exécution. Cependant, comment faire en sorte d'intégrer la diversité de l'offre à un socle technique dont la cible n'est pas définie à priori ? La solution proposée par J2ME consiste à regrouper par catégories certaines familles de produits tout en proposant la possibilité d'implémenter des routines spécifiques à un terminal donné.

MIDP est un profil pour les appareils mobiles utilisant la configuration CLDC, comme les téléphones mobiles. Le profil MIDP précise les fonctionnalités comme l'usage de l'interface client, la persistance de stockage, la mise en réseau, et l'application modèle. Le noyau central de la mise en oeuvre de la version J2ME de Nokia est composé de la configuration CLDC et du profil MIDP.

MIDP définit les principaux packages suivant :

- la gestion du cycle de vie de la midlet `javax.microedition.midlet`
- une interface graphique de haut niveau (usage recommandé) à base d'écrans simples contenant des formulaires, listes, etc. : `javax.microedition.lcdui`
- une interface graphique de bas niveau définissant un canevas générique `javax.microedition.lcdui.Canvas` et package `javax.microedition.lcdui.game`
- la gestion des entrées/sortie `javax.microedition.io` et la possibilité d'activer une midlet sur un évènement `javax.microedition.io.PushRegistry`
- la gestion d'un espace de donnée organisée sous forme de base de donnée simple `javax.microedition.rms`
- la gestion de players pour les contenus multimédia
- quelques opérations sur la manipulation de certificats
- des threads et des timers.

1.2. RMS

Avec MIDP, le mécanisme pour la persistance des données est appelé RMS (Record Management System). Il permet le stockage de données et leur accès ultérieur.

RMS propose un accès standardisé au système de stockage de la machine dans lequel s'exécute le programme. Il n'impose pas aux constructeurs la façon dont les données doivent être stockées physiquement.

Du fait de la simplicité des mécanismes utilisés, RMS ne définit qu'une seule classe, RecordStore. Cette classe ainsi que les interfaces et les exceptions qui composent RMS sont regroupées dans le package javax.microedition.rms.

Les données sont stockées dans un ensemble d'enregistrements (records). Un enregistrement est un tableau d'octets. Chaque enregistrement possède un identifiant unique nommé recordId qui permet de retrouver un enregistrement particulier.

A chaque fois qu'un ensemble de données est modifié (ajout, modification ou suppression d'un enregistrement), son numéro de version est incrémenté.

Un ensemble de données est associé à un unique ensemble composé d'une ou plusieurs Midlets (Midlet Suite). Un ensemble de données possède un nom composé de 32 caractères maximum.

1.3. Bluetooth



1.3.1. Description

Bluetooth est une nouvelle technologie de transmission sans fil. Son but est de permettre la communication à courte distance entre plusieurs appareils, et sans le moindre câble, en utilisant les ondes radio.

Les appareils compatibles Bluetooth communiquent en utilisant les ondes radio sur les fréquences comprises entre 2400 et 2483.5 MHz.

Le débit théorique est de 1 Mb/s. Tout comme le protocole IP, l'envoi des informations s'effectue par paquets de données entourées de blocs de contrôle. Ces blocs de contrôle permettent la mise en réseau des appareils à distance suffisante, le bon acheminement des données et la correction d'éventuelles erreurs de transmission.

Pour pouvoir communiquer entre eux, les appareils doivent se trouver à une distance maximum de 10 mètres.

Il peut y avoir jusqu'à 8 appareils chaînés pour former un petit réseau (nommé picoréseau ou piconet). Il est cependant possible de connecter entre eux ces piconets pour former des réseaux plus grands nommés scatternets. Certains appareils bluetooth pourront donc servir de passerelle.

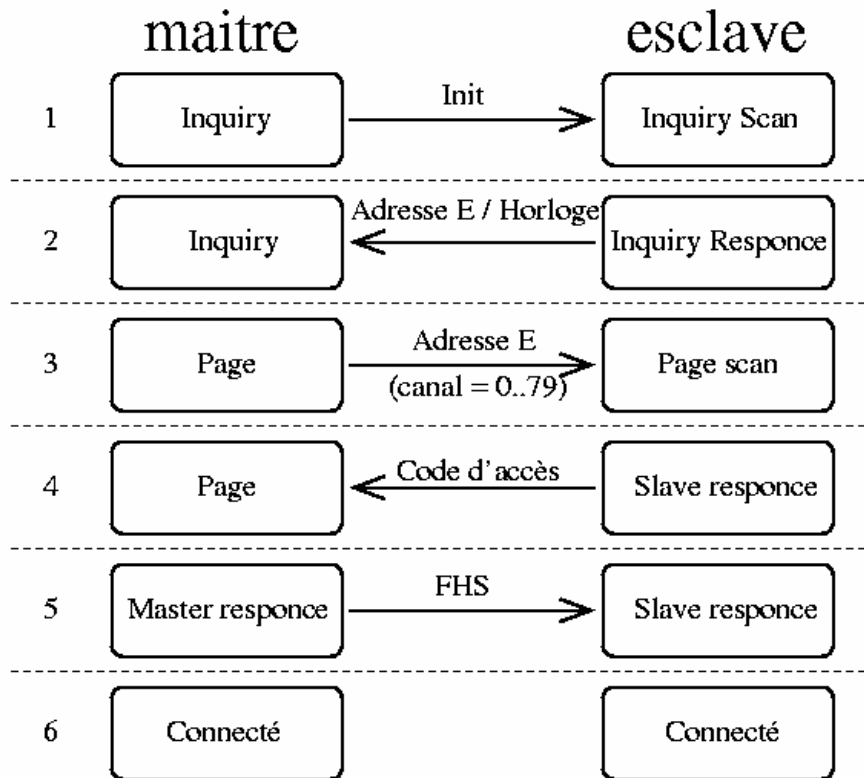
1.3.2. Schémas de connexion

Le plus simple des schémas de connexion est établi lors de la communication entre 2 périphériques bluetooth. Un des deux appareils jouera le rôle de Maître (Master) et l'autre d'esclave (Slave). Le maître est chargé de gérer la communication entre les deux périphériques : c'est lui qui initialise la connexion.

Afin de vous montrer le fonctionnement du protocole Bluetooth lors d'une connexion nous allons prendre un exemple simpliste. Soit deux puces Bluetooth notées M et E. Ici M jouera le rôle du maître et E celui de l'esclave. Le maître et l'esclave sont au début de notre exemple dans un état dit passif. La puce M va initialiser un lien avec la puce E.

L'état passif pour une puce Bluetooth signifie principalement une consommation d'énergie réduite. Dans un état passif il existe plusieurs sous états. Ceux-ci servent durant l'établissement d'une connexion.

Le schéma suivant permet de mieux comprendre le mécanisme de connexion :



Au début du processus, le maître M doit se trouver dans le sous état " Inquiry " et l'esclave E dans l'état " Inquiry scan " :

1. Etant dans l'état " Inquiry ", M envoie un signal pour prévenir E qu'il souhaite initialiser une connexion. E se trouve alors dans l'état " inquiry scan ".
2. Si E se trouve à portée et qu'il est dans l'état " Inquiry scan ", il passe alors dans le sous état " Inquiry response " puis répond effectivement au maître. La réponse de E comporte entre autre son adresse ainsi que des informations sur son horloge.
3. Une fois que E a envoyé sa réponse, il passe dans l'état " Page Scan ". Il se met ensuite en attente d'un message comportant sa propre adresse sur un des 80 canaux existants. Lorsque M reçoit le message réponse de E, celui-ci passe dans l'état " page ". C'est à dire que M stocke les informations reçues (pagination). Ces informations permettent à

M d'avoir conscience de la présence de E. Lorsque M souhaite poursuivre le processus de connexion, celui-ci renvoie un message réponse en y plaçant l'adresse de E. Ce message est renvoyé plusieurs fois sur tous les canaux.

4. Lorsque E voit une réponse à son nom arriver, il se place dans le sous état " Slave response " puis renvoie un message - réponse à M en y joignant son code d'accès.
5. De son côté, M une fois ce code d'accès récupéré, se place alors dans un état " Master response " et renvoie un paquet de type FHS à E. Ce paquet de type FHS (Frequency Hopping Synchronisation) permet à E de se synchroniser avec M.
6. Une fois ce dernier message envoyé, M passe dans l'état " connecté ". De même, lorsque E reçoit ce message il passe aussi dans l'état "connecté".

Ici l'état "connecté" n'est pas un sous état. Pour vérifier que la connexion s'est bien passée, le maître envoie un paquet de type POLL et attend en retour n'importe quel type de paquet. Si une connexion s'est effectivement bien passée, l'esclave est synchronisé avec son maître et se trouve sur le bon canal de communication.

2. Analyse

2.1. Analyse des besoins :

2.1.1. Le jeu

Le jeu du morpion ayant des règles définies, il suffit de programmer les phases du jeu afin de les respecter et de ne pas permettre de triche ou de mauvaise saisie. Il serait intéressant de pouvoir mémoriser les scores par adversaire, afin de ne pas repartir de 0 à chaque partie, et de créer une durée de vie de jeu plus longue.

2.1.2. L'affichage

L'affichage doit être clair et attractif, et doit permettre une bonne lisibilité de l'écran. On pourra utiliser des images et éventuellement un fond d'écran. Par ailleurs, les supports d'utilisation ayant des formats différents, les images devront être adaptables à un affichage plus ou moins grand.

2.1.3. La liaison bluetooth

La liaison bluetooth est une des contraintes de ce projet, elle devra permettre un échange de données et ne pas être perturbée par un autre client/serveur qui chercherait à interférer avec une partie en cours. Pour cela il serait intéressant de ne jamais rompre le lien qui lie le client au serveur. En cas de rupture de la liaison, le mobile devra quitter le jeu après avoir sauvegardé ses données.

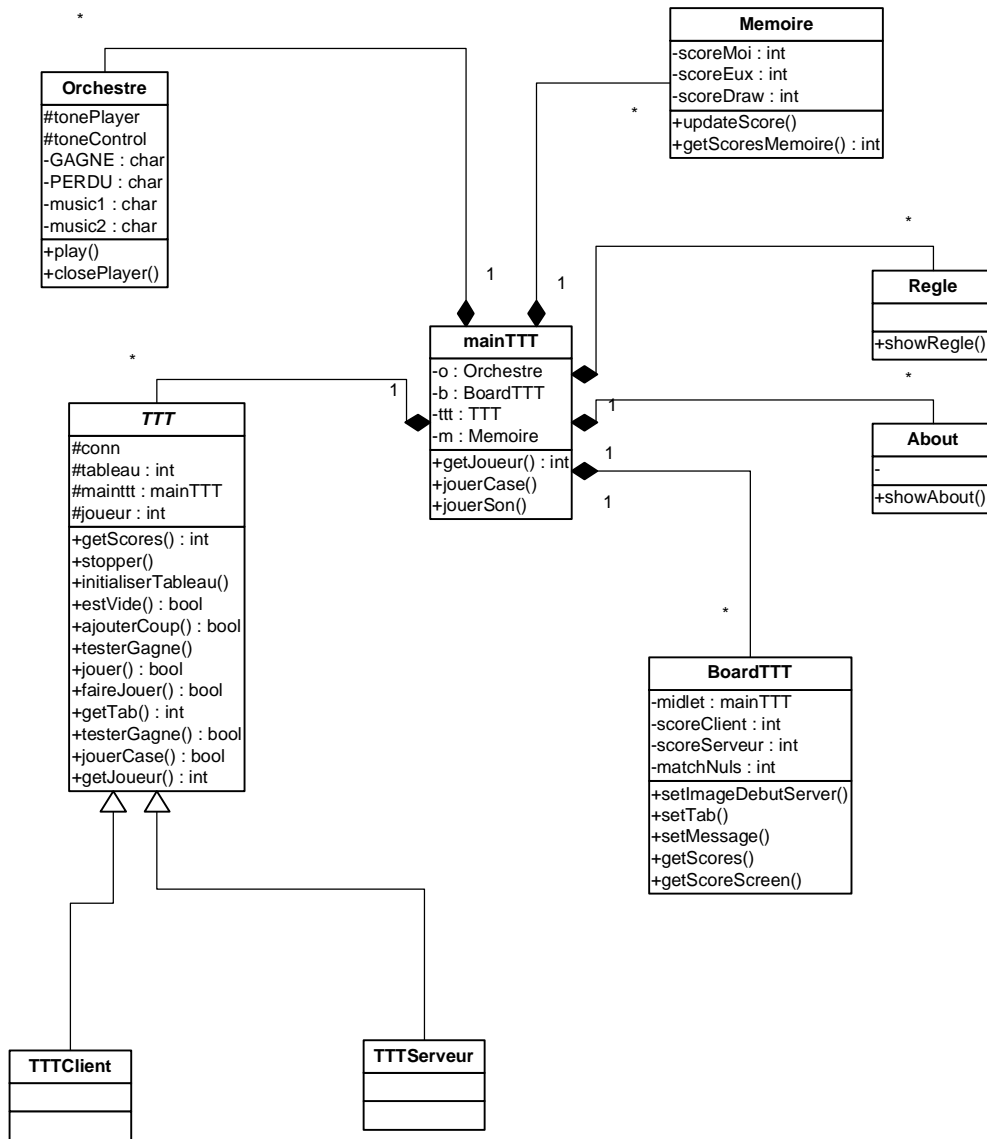
2.1.4. Le son

Dans la mesure du possible, il serait agréable de disposer de quelques effets sonores, lorsque l'on gagne, et lorsque l'on perd. Comme il s'agit de supports mobiles, comme des téléphones portables, la plupart sont équipés de modules sonores permettant de jouer ces mélodies.

2.1.5. Le menu

Le menu permet de définir qui est client, et qui est serveur. Il est nécessaire que les 2 joueurs se mettent d'accord sur qui jouera quel rôle dans la connexion. Ceci est facile à faire car, rappelons le, le bluetooth a une portée de 10 mètres.

2.2. Diagramme UML théorique



Pour ce projet, nous voulions bien séparer les différentes couches du jeu, la couche graphique, la couche sonore, la couche mémorisation : Ceci pour permettre une meilleure évolutivité de chacun des composants qui pourraient ne pas être compatibles avec tous les téléphones.

La classe **mainTTT** est la classe midlet instanciée lors du lancement de l'application. Son rôle est d'instancier les différentes classes nécessaires et servir de classe de transition entre les autres classes.

La classe **TTT** est abstraite afin de ne pas être instanciée, elle regroupe les comportements du TicTacToe. Elle est déclinée en deux classes filles : **TTTCClient** et **TTTServeur** qui rajoute la couche réseau nécessaire, et met en place les outils requis en tant que client et en tant que serveur.

La classe **BoardTTT** est l'interface graphique de l'application. Elle regroupe les fonctions d'affichage des phases de jeu.

La classe **Memoire** permettra la mémorisation des scores à chaque fin de partie.

La classe **Orchestre** devra pouvoir jouer des petites mélodies pour rendre le jeu plus convivial et attrayant. Celles-ci seront notamment jouées lorsqu'une partie est finie.

Enfin, les classes **About** et **Regle** permettront l'affichage des écrans « A propos » et « Règles du jeu ».

Le bilan de ces expériences est l'utilisation de méthodes Swing, de fond d'écrans en JPEG et de croix et rond en GIF à fonds transparents. Deux formats d'images sont possibles en fonction de la taille de l'écran.

Nous avons réalisé les images de manière à accentuer le contraste entre le fond et le jeu pour améliorer la lisibilité.

De même, nous souhaitons afficher une icône à côté du nom du projet, il était nécessaire de se rendre dans l'onglet "MIDlets" pour préciser le nom de l'icône `/icons/tictacTooth.png`, le nom du projet `TicTac Tooth` et le nom de la classe principale `tictacTooth.mainTTT`.

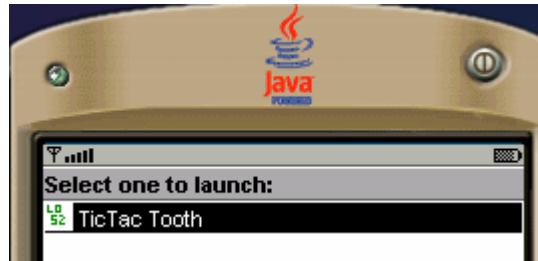
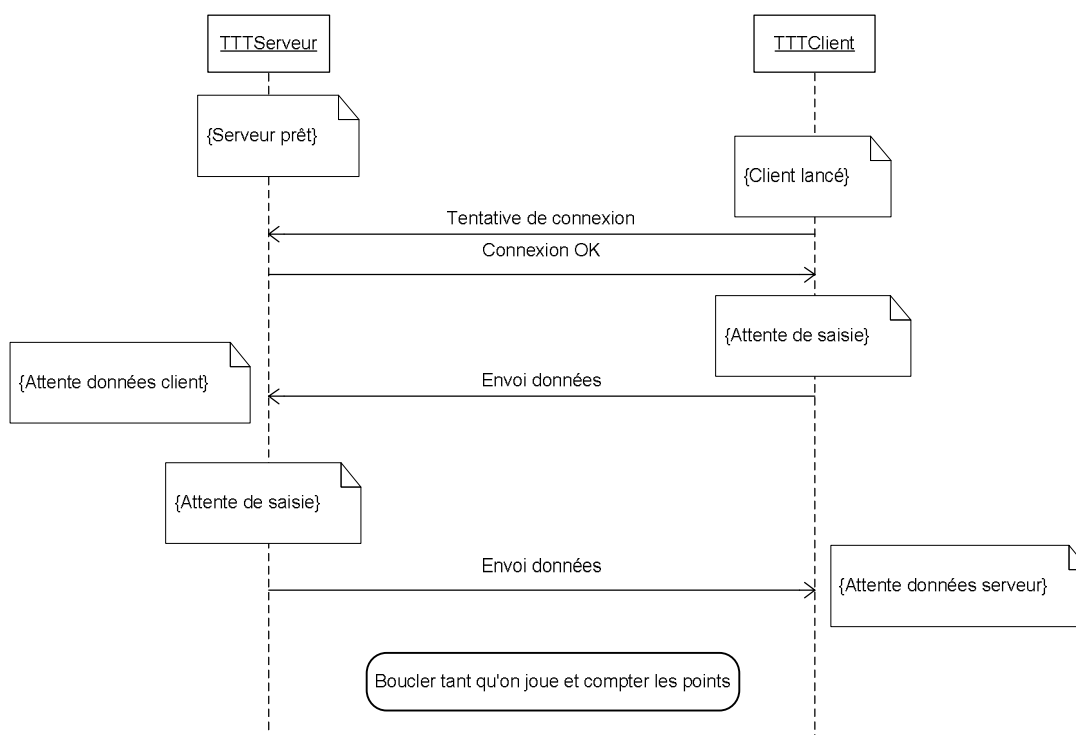


Fig. 2 : Affichage initial avec une icône et le nom du projet.

3.1.3. La liaison bluetooth

Pour la communication bluetooth nous utilisons comme base l'exemple de serveur d'écho modifié de manière à ce qu'il n'accepte qu'une connexion à la fois.

Il était nécessaire de changer les paramètres du projet. Ainsi dans "Settings", onglet "API Selection", l'option "Bluetooth/OBEX for J2ME (JSR 82)" doit être cochée.



Pour le serveur, la procédure consiste à :

Récupérer une référence du module BlueTooth

```
localDevice = LocalDevice.getLocalDevice();
```

Rendre ce module accessible

```
localDevice.setDiscoverable( DiscoveryAgent.GIAC );
```

Ouvrir un accès pour notre service

```
notifier = (StreamConnectionNotifier)Connector.open(serverUrl);
```

Attendre un client et le laisser entrer

```
conn = notifier.acceptAndOpen();
```

Créer deux flux (entrée et sortie)

```
input = conn.openInputStream();
```

```
output = conn.openOutputStream();
```

Pour le client (il implémente les fonctionnalités de DiscoveryListener):

Récupérer une référence du module BlueTooth

```
LocalDevice localDevice = LocalDevice.getLocalDevice();
```

On se prépare à chercher un service

```
discoveryAgent = localDevice.getDiscoveryAgent();
```

On lance la recherche de service

```
discoveryAgent.startInquiry(DiscoveryAgent.GIAC, this);
```

Les fonctions suivantes issues de DiscoveryListener permettront de détecter les diverses étapes de connexion et d'y effectuer des opérations :

- *inquiryCompleted*
- *deviceDiscovered*
- *servicesDiscovered*
- *serviceSearchCompleted*

Dans notre cas, nous déclencherons le jeu lorsque la fonction *serviceSearchCompleted* reçoit comme code *SERVICE_SEARCH_COMPLETE*.

Se connecter au serveur détecté

```
conn = (StreamConnection)Connector.open(serviceUrl);
```

Créer deux flux (entrée et sortie)

```
input = conn.openInputStream();
```

```
output = conn.openOutputStream();
```

Lorsque les flux ne sont plus nécessaires, ou lorsqu'une exception est détectée au niveau réseau, alors on ferme les flux :

```
input.close();  
output.close();  
conn.close();
```

Et on quitte.

3.1.4. Stockage et gestion des scores

Plusieurs méthodes permettent de gérer les Records store. *openRecordStore* et *closeRecordStore* permettent respectivement d'ouvrir et de fermer un Record store.

La liste de tous les Record store peut être obtenue par *listRecordStore*. *deleteRecordStore* en supprime un.

Le nombre d'enregistrements dans un Record store est retourné par *getNumRecords*. Les opérations de base sur les enregistrements sont assurées par ces méthodes : *addRecord* (ajout), *deleteRecord* (suppression), *getRecord* (lecture), *setRecord* (modification), *getRecordSize* (taille de l'enregistrement).

Pour notre application, nous voulions stocker les données des scores contre chaque adversaire. Pour cela il aurait été intéressant de récupérer les données dans le répertoire et de les stocker avec les scores correspondants. Hélas, pour des raisons de sécurité cela est impossible, et nous n'avons pas réussi à trouver un identifiant fiable pour la mise en place des scores.

Désirant néanmoins utiliser cette possibilité, nous avons stocké les scores globaux, c'est à dire le nombre de parties totales gagnées, perdues et de matches nuls.

3.1.5. Le son

Pour les mélodies, nous avons choisi de faire simple, en utilisant la classe *ToneControl* qui permet l'utilisation de mélodies simples.

Initialisation

```
if (tonePlayer == null) {  
    Liaison au module de sonnerie du mobile  
    tonePlayer = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);  
    tonePlayer.realize();  
    toneControl=(ToneControl)tonePlayer.getControl("javax.microedition.media  
        .control.ToneControl");  
}  
tonePlayer.deallocate();
```

Mise en place de la melodie

```
toneControl.setSequence(tune);
```

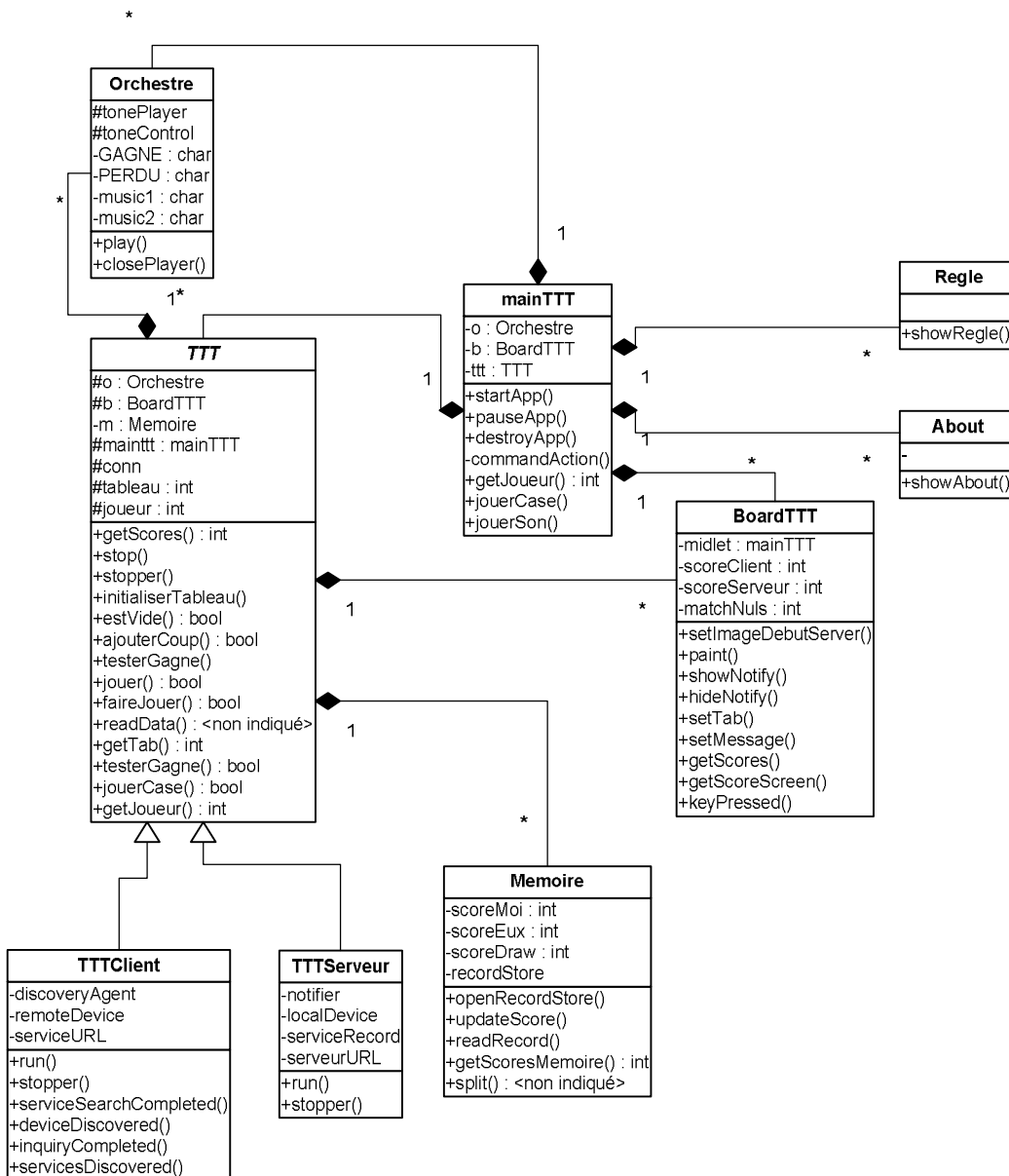
Jouer la melodie

```
tonePlayer.start();
```

3.1.6. Le menu

Pour le menu, nous avons utilisé un objet Form associé à un objet List. Form peut prendre plusieurs types de paramètres afin d'afficher des menus selon divers contextes. On réutilise cette fonctionnalité pour les écrans « A propos » et « Règles du jeu » avec un objet Alert passé en paramètre.

3.2. Diagramme UML d'implémentation



Comme on peut le voir dans ce diagramme, nous avons préféré garder des références aux objets Orchestre, mainTTT, Memoire, BoardTTT directement dans la classe TTT. Nous

avons fait ce choix car il nous facilitait grandement la programmation en réduisant le nombre de getters/setters de l'application.

3.3. Problèmes rencontrés

Dans le Wireless Toolkit, les zones mémoire RMS émulsés sont les mêmes lorsque l'on utilise un seul modèle pour les deux joueurs : il y a alors conflit et seul un des deux a accès à la mémoire. Pour pallier à ce problème il faut utiliser deux modèles différents.

De même, il nous est impossible de quitter le serveur et de le relancer : le serveur fermé reste ouvert et il nous est impossible de le fermer proprement. Après avoir essayé les exemples fournis avec le J2ME, il est apparu que ce problème était récurrent et que nous n'en viendrions pas à bout.

Lors de l'implémentation du menu, l'accès aux parties "A propos" et "Règles du jeu" créait une nouvelle instance du serveur. Le problème majeur était le *commandlistener* qui ne filtrait pas les différents cas... Par exemple lorsqu'on appuyait sur VALID, il validait les 2 menus à la fois ce qui avait pour effet de relancer le serveur.

Pour avoir un affichage plus sympathique, nous avons utilisé les propriétés d'animation et de transparence des GIF. Cependant les images ne doivent pas être trop lourdes et surtout doivent correspondre au type 89a... Ainsi la transparence est correctement affichée mais nous ne sommes parvenus à créer de GIF animés compatibles. Cela nous a permis d'avoir une image derrière la grille de jeu, sans pour autant que les croix/ronds ne viennent la masquer totalement.

J2ME propose lors de l'exécution une liste de téléphone émulsés. Ceux-ci sont assez différents, tant au niveau de la taille de l'écran que du nombre de couleurs. Ainsi nous avons dû créer des images de tailles différentes et ajuster l'affichage en fonction du téléphone utilisé.

Enfin, le fait que J2ME soit un java « allégé » a induit un manque dans les fonctions usuelles. Par exemple, nous avons dû redéfinir une méthode *split* qui se trouve habituellement dans la classe `java.lang.String`



Fig. 3 : Différents cas pour l'affichage en fonction du téléphone utilisé.

Conclusion

Lors de l'analyse et de la mise en place du projet, nous avons à la fois découvert de nombreuses fonctionnalités qui facilitent l'utilisation de médias divers, et nous nous sommes confrontés aux problèmes liés à la légèreté de J2ME par rapport à JAVA. Par ailleurs nous n'avons pas réussi à identifier formellement un adversaire, ce qui a restreint les possibilités de notre application.

Nous sommes toutefois fiers du résultat, car à force d'ajouter quelques détails supplémentaires à de nombreuses reprises, notre application est devenue un produit fiable, certes limité du point de vue de la diffusion à cause des images utilisées (sous copyright), mais bien conçu, graphiquement et architecturalement.

Ce projet pouvait sembler facile, et c'était le cas si l'on se contentait d'un tic-tac-toe basique, sans fonctions avancées. Nous nous sommes fixés des objectifs, sans être sûr de pouvoir tous les réaliser, et nous sommes parvenu, malgré quelques relations humaines tendues (départ d'un membre de l'équipe), à les atteindre.