

Projet réalisé par
Alexandre Toyer
Proposé par
Yann Mathet

Master Informatique Ingénierie de l'Internet

Application d'aide à l'arbitrage
de courses nautiques à l'aide de
visionnages de vidéos

Mot clés : JAVA, Vidéo, Api JMF, Swing, PHP, Ajax



Sommaire

Remerciements	1
Résumé	2
Introduction	3
1. Présentation du projet.....	4
1.1. Contexte : des compétitions difficile à superviser.....	4
1.2. Objectifs : un accès aisé aux vidéos de course pour vérifier les litiges	5
2. Fonctionnalités de l'application	6
2.1. L'application de dépôt des contestations.....	6
2.1.1. Administration : la gestion des courses	6
2.1.2. Utilisation : les coureurs	6
2.2. L'application de traitement des contestations	7
2.2.1. Nécessité d'un lien avec l'application de dépôt.....	7
2.2.2. Sélection de contestation et visionnage des vidéos.....	7
2.2.3. Traitement des vidéos	8
3. Analyses et modélisation	9
3.1. Aperçu de l'existant.....	9
3.2. Considérations techniques	9
3.2.1. Pour le traitement.....	9
3.2.2. Pour le dépôt	10
3.2.3. Le lien entre les applications.....	10
3.3. Recherche d'API vidéo.....	10
3.4. Modélisation	11
3.4.1. Le modèle de données.....	11
3.4.2. La vue et le contrôle.....	12
3.4.3. Abstraction de l'accès aux vidéo	13
3.4.4. La base de données	14
3.4.5. Le modèle de l'application de dépôt des contestations.....	15
4. Réalisations.....	16
4.1. Application Web de dépôt de données.....	16
4.1.1. Programmation.....	16
4.1.1.1. La gestion des courses	16
4.1.1.2. L'utilisation du site par les coureurs.....	18
4.1.1.3. Amélioration de l'ergonomie : Javascript et Ajax	19
4.1.1.4. Génération du XML	22
4.2. Application lourde d'accès ciblé aux vidéos en Java.....	23
4.2.1. Architecture de l'application.....	23
4.2.1.1. Présentation	23
4.2.1.2. Lecture du XML	24
4.2.2. Fonctionnement de JMF	24
4.2.3. Interface graphique en Swing	27
4.2.3.1. Layout et placement des composants	27
4.2.3.2. Lecture des vidéos	31
4.2.3.3. Internationalisation de l'interface	33
4.2.3.4. Affichage de listes de données	35
4.2.4. Amélioration de l'expérience utilisateur.....	36
4.2.4.1. Stockages des préférences de l'utilisateur	36
4.2.4.2. Utilisation des Thread.....	36
4.2.4.3. Traitement et affichage des erreurs	38
Conclusion.....	40

Glossaire	41
Références	42
Annexes	43
Annexe 1 : Diagramme de classes du modèle de l'application.....	43
Annexe 2 : Classe Race	45
Annexe 3 : classe Factory	49
Annexe 4 : Swing : Classe MediaController.....	50

REMERCIEMENTS

J'aimerais remercier mon tuteur Yann Mathet qui a été très disponible durant ce projet et dont les conseils et avis m'ont beaucoup servis.

Je remercie également Vincent Goncalves pour son temps passé pour moi à encoder une vidéo.

RESUME

L'idée de ce projet est née en cherchant un moyen de vérifier le passage des coureurs aux différentes balises dans une compétition nautique. A ce jour, le pointage se fait manuellement ce qui donne lieu à un nombre conséquent de réclamations. L'idée serait de doubler le pointage manuel actuel en plaçant des caméras à chaque balise, pour enlever tout doute lors du débriefing qui suit la course.

Un outil a donc été développé afin de répondre à cette problématique. Il se présente sous la forme de deux applications : un site Internet donnant la possibilité aux coureurs de déposer leurs contestations et un logiciel permettant aux organisateurs de traiter ces contestations en visionnant des vidéos des courses.

La lecture de vidéo en Java restera comme le problème le plus important de ce projet, ayant nécessité beaucoup de temps de recherche et de développement.

INTRODUCTION

L'arbitrage des courses nautiques, particulièrement les courses de planches à voile, est un domaine délicat à gérer en raison du grand nombre de participants à ces événements. En effet les coureurs doivent passer différentes balises au long de chaque course, le rôle des organisateurs étant de contrôler les passages de ces coureurs afin de les valider ou non.

A ce jour le pointage se fait manuellement : un "aboyeur" placé à chaque balise sur un bateau crie le numéro de dossard des coureurs à un second arbitre chargé de consigner les numéros. De plus un enregistrement audio est réalisé. Ce fonctionnement entraîne de très nombreux oublis et donc un lot conséquent de réclamations de la part des coureurs. L'idée serait de doubler le pointage manuel actuel en plaçant des caméras à chaque balise, pour enlever tout doute lors du débriefing qui suit la course.

La problématique était donc de définir un outil permettant le visionnage de chaque passage afin de régler les contestations. Nous commencerons par voir plus en détail les objectifs principaux de ce projet et le contexte dans lequel il s'inscrit, puis nous étudierons les fonctionnalités à implémenter. Nous verrons ensuite la phase d'analyse et de modélisation du problème. Enfin nous terminerons en présentant les réalisations effectuées.

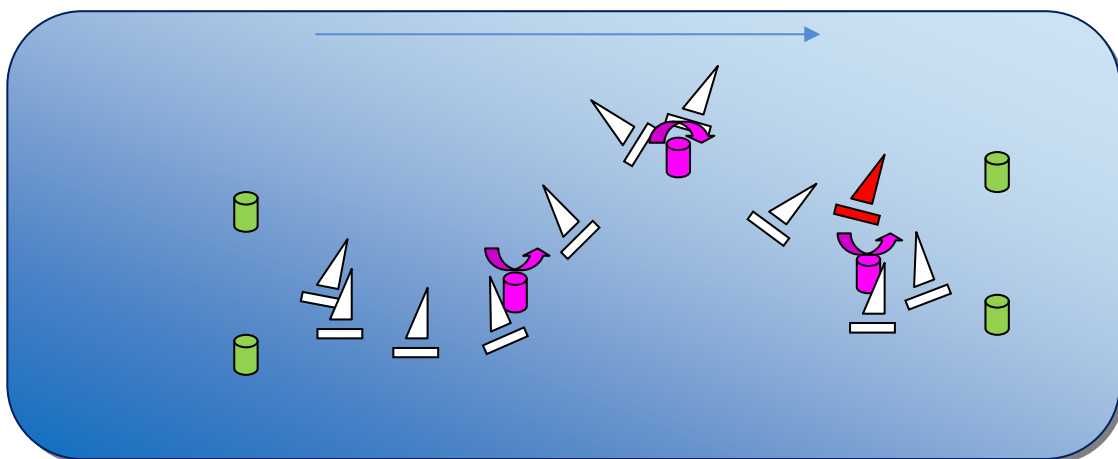
1. PRESENTATION DU PROJET

1.1. Contexte : des compétitions difficile à superviser

Comme énoncé dans l'introduction, ce projet à pour but le développement d'une application pouvant traiter des sections précises de fichiers vidéo. Il a été décidé de mettre en application cette idée avec des courses de planches à voile, qui sont un exemple approprié d'évènements où une application de ce type serait la bienvenue.

En effet dans les courses de ce genre le nombre de participants est très important et se compte souvent en centaines. Le travail des arbitres devient donc compliqué et ils commettent de nombreuses erreurs. Voyons brièvement le déroulement d'une course.

Les coureurs partent en franchissant une balise de départ, souvent composée en réalité de deux balises qui composent ainsi une ligne. Ils doivent ensuite contourner un certain nombre de balises pour enfin passer la ligne d'arrivée. Voici un schéma montrant ce fonctionnement :



On voit sur ce schéma les planches à voiles en blanc, les balises formant les lignes de départ et d'arrivée en vert et les balises intermédiaires en rose. Ces balises possèdent un sens de contournement, représenté ici à l'aide d'une petite flèche les surplombant. On voit que les balises 1 et 3 doivent être contournées par le bas, alors que la balise 2 doit être prise par le haut. On peut voir que tous les coureurs qui contournent actuellement des balises sont bien engagés, sauf le coureur à la voile rouge qui est en train de prendre la balise 3 par le haut alors qu'il devrait la contourner par le bas.

Ce coureur doit donc être disqualifié au regard des règles de la course, puisqu'il n'a pas négocié toutes les balises correctement.

Lors d'une course nautique telle que celle-ci les arbitres sont placés à bord de bateaux à côté de chaque balise et notent les numéros de chaque coureur (présent sur les voiles) contournant correctement les balises, ainsi que ceux les contournant d'une mauvaise façon.

A la fin de la course les coureurs ayant mal négociés certaines balises sont disqualifiés.

Ce fonctionnement semble simple à mettre en place mais le nombre important de participants complique la tâche des arbitres, qui peuvent commettre quelques erreurs.

Ainsi, des coureurs s'estimant disqualifiés à tort peuvent déposer des contestations, s'ils sont sûrs de leur passage. Ces contestations sont difficiles à traiter à l'heure actuelle.

1.2. Objectifs : un accès aisé aux vidéos de course pour vérifier les litiges

Le but principal de ce projet est de proposer une solution pour pallier au problème actuel de l'arbitrage. L'idée est de doubler les contrôles des arbitres à chaque balise par un enregistrement vidéo. Une application serait ensuite développée permettant de retrouver rapidement le passage d'un coureur lorsqu'il dépose une contestation. Des outils classiques de navigation dans la vidéo seraient fournis afin de contrôler efficacement le passage du coureur à la balise. Il serait ainsi possible de faire un ralenti, d'avancer ou de reculer dans la vidéo, ou encore de faire un arrêt sur image.

Pour déposer leurs contestations à la fin des courses les coureurs utiliseraient des bornes informatiques pour se connecter à une seconde application dédiée à recueillir ces contestations. On peut imaginer que cette application soit également accessible de par Internet. Il leur suffirait d'indiquer, pour les balises pour lesquelles ils ont été disqualifiés, leur heure de passage estimée et d'enregistrer la contestation. De plus en plus de coureurs sont équipés de GPS qui leur permettent de reconstituer leur parcours et fournissent les temps de passage à chaque balise.

La réalisation de cette application de dépôt de contestation était un objectif secondaire du projet, qui n'apparaissait pas dans l'énoncé mais dont le développement nous à néanmoins semblé présenter différents intérêts sur lesquels nous reviendront par la suite.

2. FONCTIONNALITES DE L'APPLICATION

2.1. L'application de dépôt des contestations

2.1.1. Administration : la gestion des courses

L'administration de l'application sera effectuée par le comité de course et devra permettre de gérer plusieurs courses à la fois.

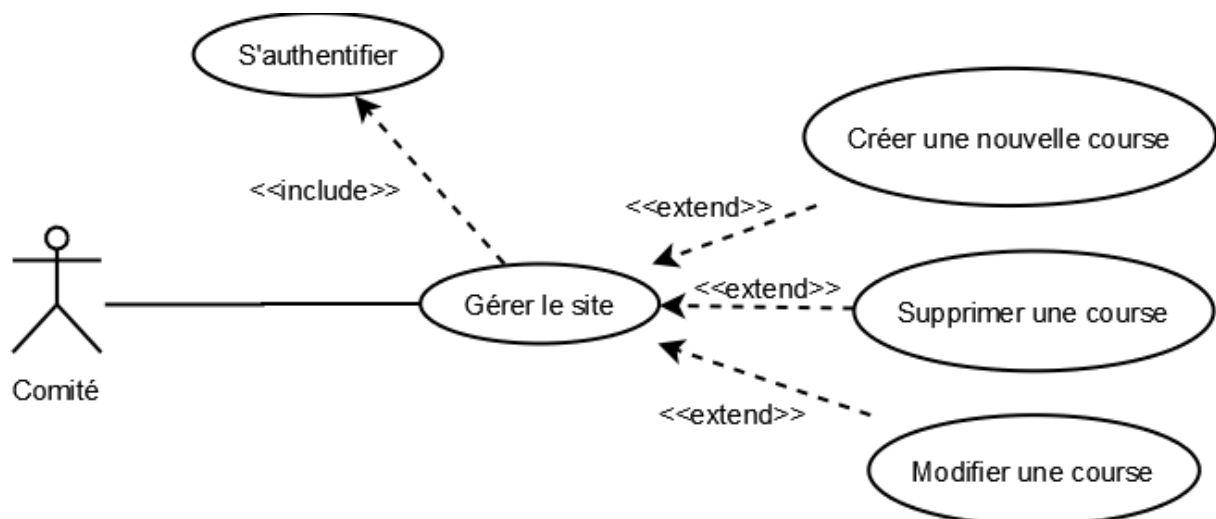
A la fin d'une course le comité de course créera sur l'application une nouvelle entrée pour la course. Cette entrée devra comporter les propriétés suivantes :

- Un nom
- Une date
- Une heure de départ exacte
- La date de fin de validité pour la création de nouvelles contestations

Le comité devra ensuite, pour chaque balise de la course saisir le nom des coureurs inscrits au départ et n'étant pas passés correctement par cette balise.

Il sera également possible de supprimer une course ou de la modifier : supprimer ou ajouter une balise, modifier le numéro d'une balise.

Voici un diagramme de cas d'utilisation résumant ces fonctionnalités :



2.1.2. Utilisation : les coureurs

Le rôle des coureurs dans l'ensemble du projet est assez succinct. Comme nous l'avons déjà évoqué, ils devront simplement se connecter à l'application pour saisir leurs contestations.

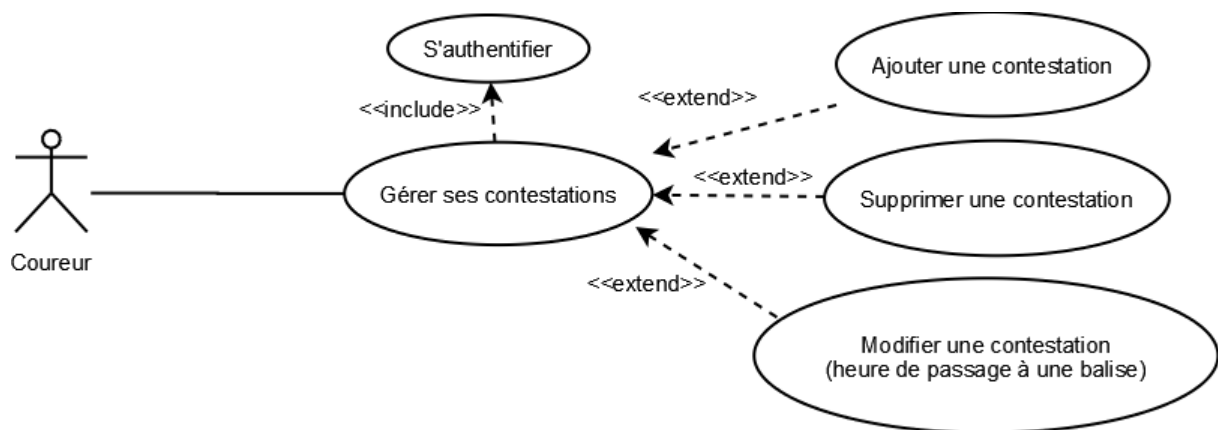
Pour enregistrer une contestation un coureur devra se connecter au site et s'identifier avec son numéro de licence et son nom. Un système d'authentification plus sécurisé serait

trop contraignant, à la fois pour le comité et pour les coureurs. Il nécessiterait un temps trop important pour la création et la distribution de mots de passe personnalisés.

Une fois identifié un coureur verra apparaître les courses pour lesquelles il a été disqualifié à certaines balises. Après avoir sélectionné l'une de ces courses il verra la liste des balises auxquelles il n'est pas correctement passé. Il devra sélectionner la balise sur laquelle il désire placer une contestation et saisir l'heure estimée de son passage à cette balise.

Une fois une contestation effectuée il ne sera pas possible pour un coureur d'en créer une nouvelle pour la même balise. Il pourra cependant retirer une contestation ou modifier l'heure estimée de son passage.

Le diagramme de cas d'utilisation suivant résume ces possibilités :



2.2. [L'application de traitement des contestations](#)

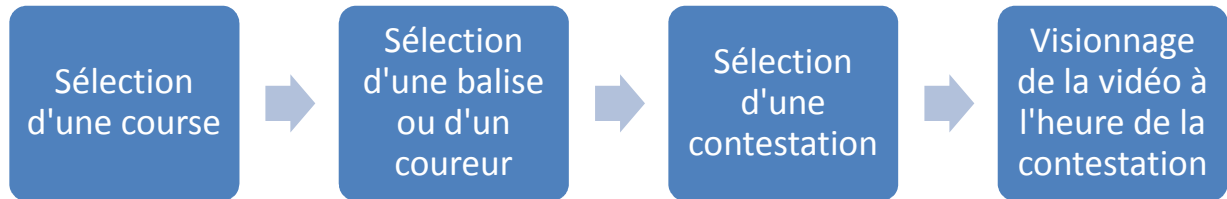
2.2.1. [Nécessité d'un lien avec l'application de dépôt](#)

Un lien sera effectué entre les deux applications, permettant à la seconde application de traiter les contestations enregistrées sur la première par les coureurs.

Ce lien devra faire en sorte qu'il soit possible, à partir de la seconde application, de choisir la course à traiter et d'afficher les contestations détaillées pour cette course.

2.2.2. [Sélection de contestation et visionnage des vidéos](#)

Voici un schéma présentant brièvement et de manière synthétique le cycle d'utilisation de l'application :



L'application de traitement des contestations devra permettre la sélection d'une course parmi une liste de courses disponibles. Deux listes seront ensuite présentées à l'utilisateur : la première regroupant les différentes balises de la course, et la seconde les différents coureurs ayant déposé des contestations.

L'utilisateur devra sélectionner soit une balise, soit un coureur, pour passer à l'écran suivant sur lequel sera présente une liste de contestations. Ces contestations concerneront soit une seule balise et possiblement plusieurs coureurs, soit un seul coureur et possiblement plusieurs balises, selon le choix de l'utilisateur à l'étape précédente.

Après sélection de la contestation à traiter l'utilisateur pourra visionner la vidéo de la balise correspondante. Cette vidéo se lancera à l'heure précisée par le coureur lors du dépôt de sa contestation, plus ou moins un décalage léger saisi par l'utilisateur juste avant le lancement de la vidéo.

L'utilisateur aura ici plusieurs outils lui permettant de naviguer dans la vidéo : avance et recul rapide, pause/reprise, stop, image par image.

L'utilisateur pourra également rechercher directement un utilisateur à partir de son nom et son prénom ou à partir de son numéro de licence.

Il sera proposé à l'utilisateur de spécifier le chemin auquel sont stockées les vidéos des courses. Il pourra de plus choisir la langue de l'application entre l'anglais et le français.

Enfin, l'utilisateur pourra indiquer que le traitement des contestations pour une course est terminé. Cela aura pour conséquence de retirer la course concernée de la liste des courses disponibles au lancement de l'application.

2.2.3. Traitement des vidéos

Le traitement des vidéos n'est pas demandé pour ce projet. On considèrera que les vidéos sont correctement enregistrées lors des courses et qu'elles sont ensuite placées sur l'ordinateur à partir duquel sera exécutée l'application de traitement des contestations.

Les vidéos d'une course seront placées dans un répertoire nommé d'après l'identifiant des courses, et les vidéos correspondantes à chaque balise auront pour nom le nom des balises.

Un fichier XML sera présent dans chaque répertoire afin d'indiquer l'heure de début de chaque vidéo, pour pouvoir synchroniser ensuite la lecture des vidéos avec les heures de contestations données par les coureurs.

3. ANALYSES ET MODELISATION

3.1. Aperçu de l'existant

L'application à réaliser semble assez unique mais on peut cependant retrouver plusieurs principes dans d'autres applications existantes.

La récupération d'informations entre plusieurs applications distribuées est une pratique très courante. Nous pouvons citer ici le protocole RPC (Remote Procedure Call) ou encore les nouveaux Web Services émergents. Ces Web Services sont par exemple utilisés dans de nombreux sites internet proposant des Mashups, c'est-à-dire des sites construits à partir d'informations provenant d'autres sites. L'architecture REST permet une mise en place aisée de cette idée, basée sur les URI. Nous pouvons expliquer cette architecture en prenant pour exemple l'API de www.twitter.com, un site permettant l'envoi entre utilisateurs de courts messages textuels : si à partir d'un programme quelconque on désire récupérer la liste des derniers messages il suffira d'appeler l'adresse :

http://twitter.com/statuses/public_timeline.xml

Pour envoyer un nouveau message à un utilisateur on utilisera :

http://twitter.com/direct_messages/new.format?user=idDeLutilisateur&text=leTexteDuMessage

Un autre aspect de l'application qui est courant est la lecture des vidéos. Il existe plusieurs logiciels classiques de lecture de vidéos, comme Windows Media Player ou VLC Player par exemple. Cependant ces logiciels ne possèdent pas de notions de « sections » de vidéo précises à visionner, tel que l'on voudrait mettre en place pour notre application. On pourra néanmoins s'inspirer des barres de lecture de ces logiciels :



3.2. Considérations techniques

3.2.1. Pour le traitement

L'application de traitement des vidéos sera une application assez complexe à développer. Elle devra pouvoir communiquer avec l'application de dépôt des contestations, et

surtout pouvoir lire des vidéos. De plus elle ne sera utilisée que sur certains postes et ne doit pas être une application accessible d'Internet.

On peut donc choisir de la développer sous la forme d'un client lourd, c'est-à-dire sous la forme d'un logiciel à installer sur les machines clientes.

Ce choix nous permettra de réaliser cette application en langage JAVA, qui est bien indiqué pour réaliser des applications graphiques rapidement et aisément.

Parmi les critères plus scolaires je me dois de préciser que je voulais cette année réaliser un projet en Java, ce langage étant celui le plus étudié durant l'année. De plus j'ai eu l'occasion l'année dernière de faire une application web complexe en PHP durant mon projet et je souhaitais donc changer cette année.

3.2.2. Pour le dépôt

L'application permettant la gestion des courses et le dépôt des contestations devra être accessible d'Internet. Elle utilisera un moyen de stockage des données afin de conserver les informations sur les courses et les contestations.

C'est pourquoi le langage PHP, couplé à une base de données MySQL, paraît approprié à la réalisation d'une telle application.

3.2.3. Le lien entre les applications

Les deux applications doivent pouvoir « communiquer » afin que l'application de traitement des contestations puisse récupérer les informations voulues. La technologie utilisée pour ce lien doit être assez universelle car l'on peut imaginer que ce flux d'informations puisse être traité par une application tierce.

C'est donc tout naturellement la technologie XML qui a été retenue pour réaliser ce lien entre les applications, l'objectif principal de ce langage étant de favoriser l'interopérabilité des systèmes hétérogènes.

3.3. Recherche d'API vidéo

Je me suis dès le début du projet concentré sur la recherche d'une API (Application Programming Interface, bibliothèque de fonctions) pouvant traiter de la vidéo en Java. En effet cette fonctionnalité n'est pas incluse par défaut dans le langage. L'API devra proposer des outils de lecture de vidéo complets et prendre en charge le maximum de formats vidéo possible.

J'ai pour cela parcouru quelques forums et pages de résultats de Google. Il s'est rapidement avéré que le traitement de la vidéo n'était pas l'un des points forts de Java puisque le nombre d'API trouvé fut très faible.

Seules deux API ont résulté de mes recherches : l'API Quicktime for Java et l'API Java Media Framework (JMF) de Sun. Les deux API proposant des services similaires (lecture de vidéo, montage de vidéo, ...) j'ai décidé de réaliser le projet avec l'API JMF, qui étant celle de Sun me paraissait la plus performante. C'est de plus celles dont il est le plus question sur Internet et pour laquelle les informations sont les plus abondantes.

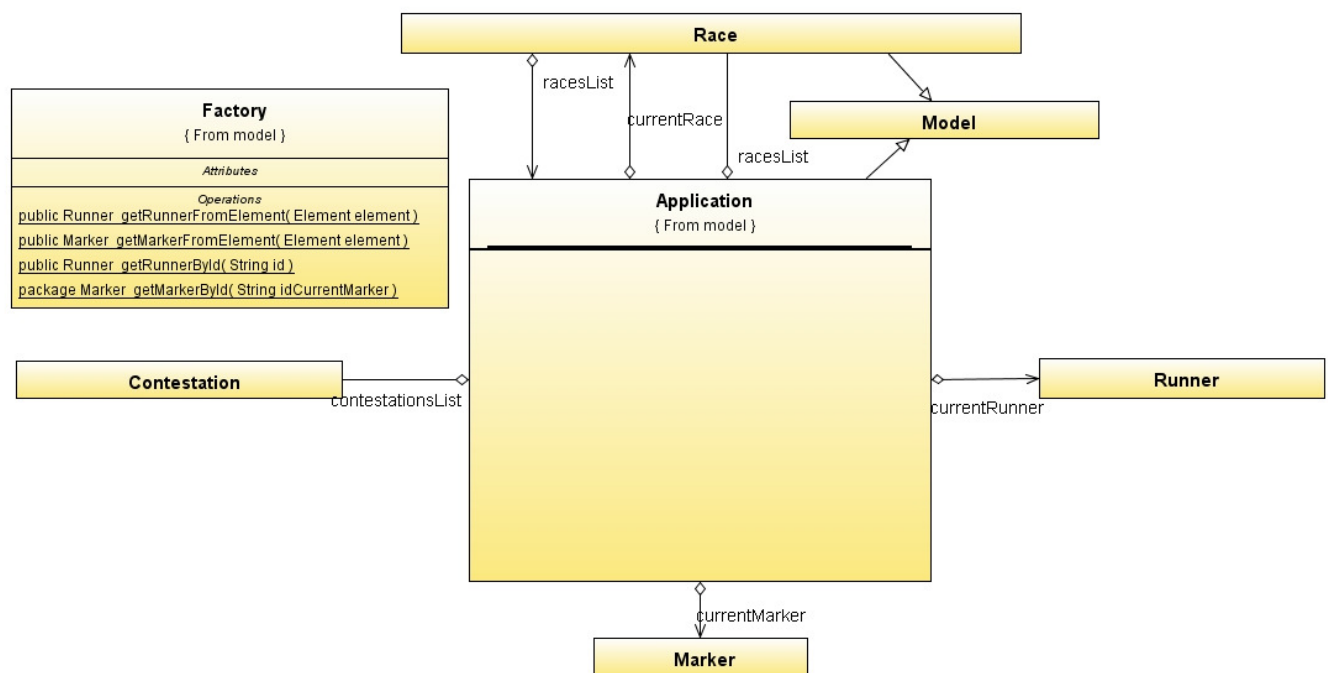
Nous reviendrons plus loin dans ce rapport sur les avantages et les inconvénients de ce choix. L'utilisation de JMF sera vue en partie Réalisations.

3.4. Modélisation

3.4.1. Le modèle de données

Je présenterai dans cette partie la partie modèle de l'application de traitement des contestations, c'est-à-dire les classes permettant de gérer les courses, les balises, les coureurs et l'application sans considération d'interface graphique.

Voici un diagramme de classe synthétique de ces classes, le diagramme complet se trouvant en annexe :



La classe centrale est la classe `Application`, c'est elle qui « organise » les autres classes. Elle possède une liste d'objets `Race`, un objet `Race` représentant la course en cours de traitement, un objet `Runner` représentant le coureur en cours d'affichage si l'utilisateur regarde les contestations par coureur, et la même chose pour un `Marker` représentant la balise courante. Enfin une liste de `Contestation` représente la liste de contestation actuellement affichée si l'utilisateur regarde des contestations.

La classe `Race` représente une course. Elle doit posséder, pour chaque coureur présent dans la course l'ensemble des contestations liées à ce coureur, et pareillement pour les balises.

Ces listes ne sont pas visibles sur le diagramme car je ne savais pas comment les rendre compréhensibles à travers l'UML.

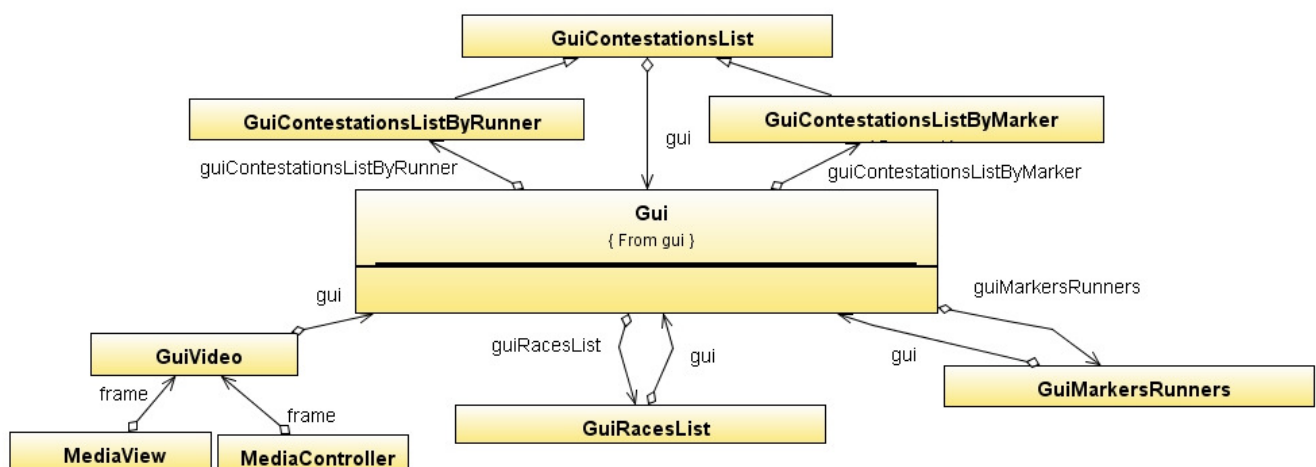
Dans l'implémentation en Java la classe `Race` aura deux attributs définis ainsi :

```
private HashMap<Runner, ArrayList<Contestation>> runnerContestationsMap
private HashMap<Marker, ArrayList<Contestation>> markerContestationsMap
```

Ainsi pour chaque `Runner` il sera facile d'accéder à la liste des contestations correspondantes, idem pour les `Marker`.

3.4.2. La vue et le contrôle

Voici un diagramme synthétique des classes composant la vue et assurant le contrôle de l'application :



La classe `Gui` est ici au centre des classes. C'est elle qui sera la fenêtre principale de l'application sur laquelle viendront se greffer les autres classes.

Nous présentons ici rapidement le rôle de chaque classe. Nous reviendrons plus en détail sur leur implémentation dans la partie consacrée à l'interface graphique.

La classe `GuiRacesList` devra afficher la liste des courses disponibles. Elle utilisera l'objet `Application` afin de récupérer la liste des courses.

La classe `GuiMarkersRunners` affichera pour une course précise la liste des balises et la liste des coureurs ayant déposé des contestations.

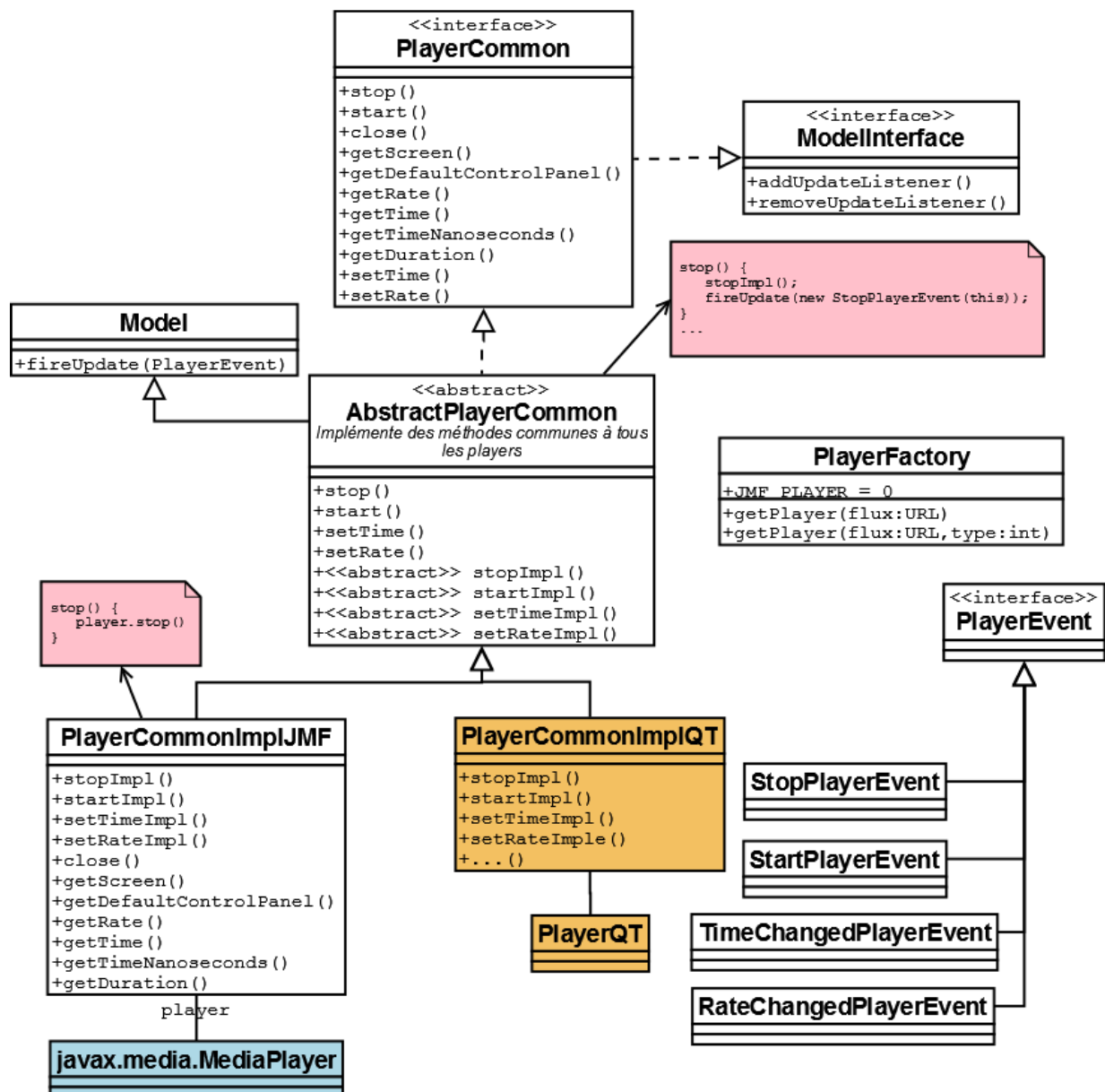
La classe `GuiContestationsList` aura pour rôle d'afficher une liste de contestations. Cette liste affichera soit les contestations pour une balise (classe `GuiContestationsListByMarker`) soit par coureur (classe `GuiContestationsListByRunner`).

Enfin c'est la classe `GuiVideo` qui permettra la lecture des vidéos, en utilisant des objets `MediaView` et `MediaController`.

3.4.3. Abstraction de l'accès aux vidéo

Comme nous le verrons JMF n'est peut-être pas la meilleure solution pour la lecture de vidéo en Java. C'est pourquoi il eût été dommage de se limiter à cette API. Il faudrait, dans l'idéal, utiliser pour le moment JMF mais si des recherches futures permettent de trouver une API plus performante pouvoir l'utiliser en ayant à réécrire le moins de code possible.

C'est pour cela qu'il est intéressant de mettre en place une couche d'abstraction d'accès aux vidéos. Voici un diagramme de classes présentant ce principe :



Les objets manipulés seraient des **PlayerCommon** et non pas des objets propres à chaque API. Cette interface définit toutes les méthodes nécessaires au contrôle d'une vidéo. La classe **AbstractPlayerCommon** permet d'écrire des méthodes qui seront communes à toutes les implémentations de **player**.

Pour utiliser l'API JMF il suffit de développer une sous classe de `AbstractPlayerCommon`, ici appelée `PlayerCommonImplJMF` et de redéfinir les méthodes des classes parentes. Comme indiqué sur le schéma, la méthode `stop()` ne fait par exemple qu'un appel à la méthode `stop()` de son `player`. Ce `player` est simplement un `MediaPlayer` (le `Player` de l'API JMF) construit dans le constructeur de `PlayerCommonImplJMF`.

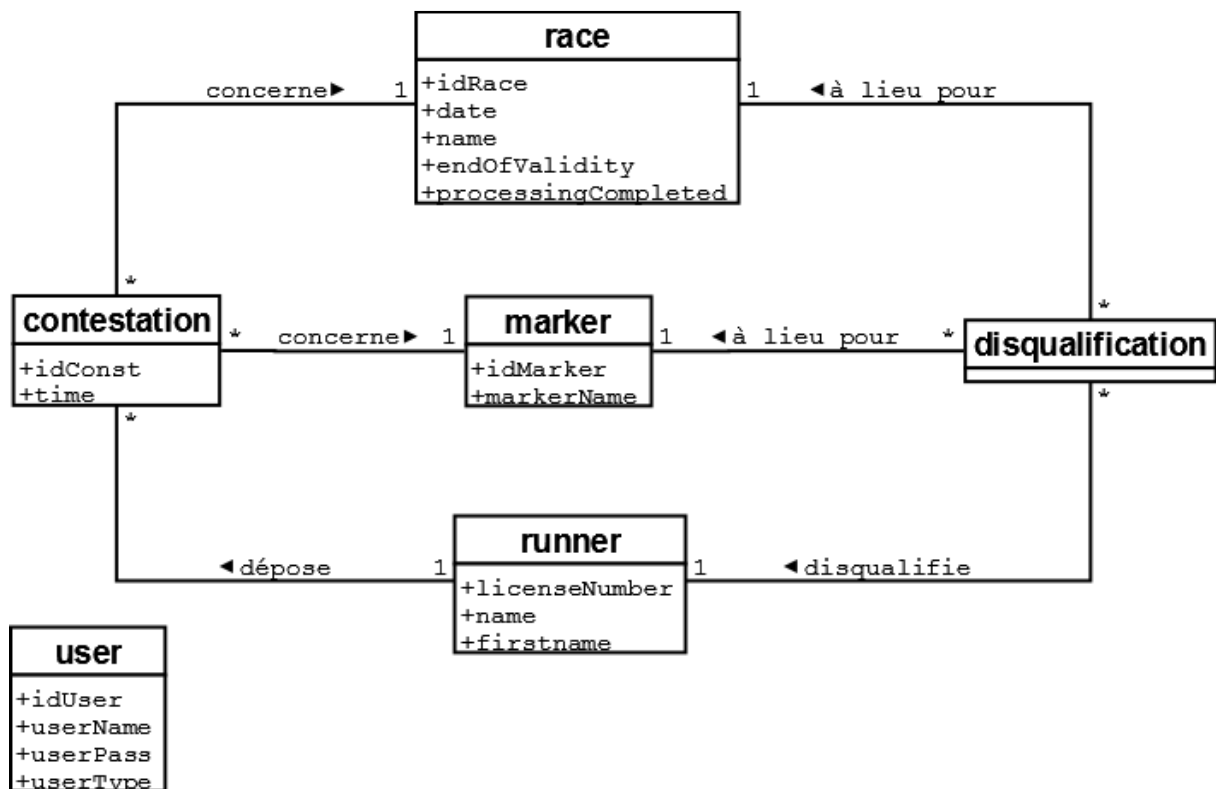
Pour utiliser une nouvelle API il suffit de créer une nouvelle classe héritant d'`AbstractPlayerCommon` et de définir les méthodes requises : la classe `PlayerQT` illustre ce principe sur le diagramme.

Il faut ensuite modifier la classe `PlayerFactory` avec laquelle les lecteurs sont fabriqués afin qu'elle ne renvoie par exemple plus que des `PlayerQT`.

3.4.4. La base de données

Une base de données est indispensable afin de stocker les informations saisies par les différents utilisateurs sur le site Internet.

Voici un diagramme UML représentant la base de données :



Nous pouvons voir au centre du diagramme les trois classes autour desquelles l'application est centrée : les classes `race` (course), `marker` (balise) et `runner` (coureur). La classe `race` possède un nom et une date, ainsi qu'un attribut nommé `endOfValidity` qui représente la date à laquelle le dépôt de contestations par les coureurs ne sera plus possible.

Le dernier attribut de cette classe, `processingCompleted`, est utilisé pour indiquer si le traitement de toutes les contestations pour cette course à été effectué ou non.

La classe `disqualification` sera utilisée pour stocker les disqualifications décidées par le comité de course à la fin de chaque course. C'est pourquoi elle concerne une race, un marker et un runner.

La classe `contestation` est à peu près similaire à la classe `disqualification` sauf qu'elle représente une contestation déposée par un coureur. C'est pourquoi elle possède en plus un attribut `time` utilisé ici pour stocker l'heure estimée par le coureur de son passage à la balise concernée.

Enfin la classe `user` servira ici à stocker les utilisateurs spécifiques de l'application, tel que le comité de course.

Ce diagramme de base de données permet de construire les tables suivantes :

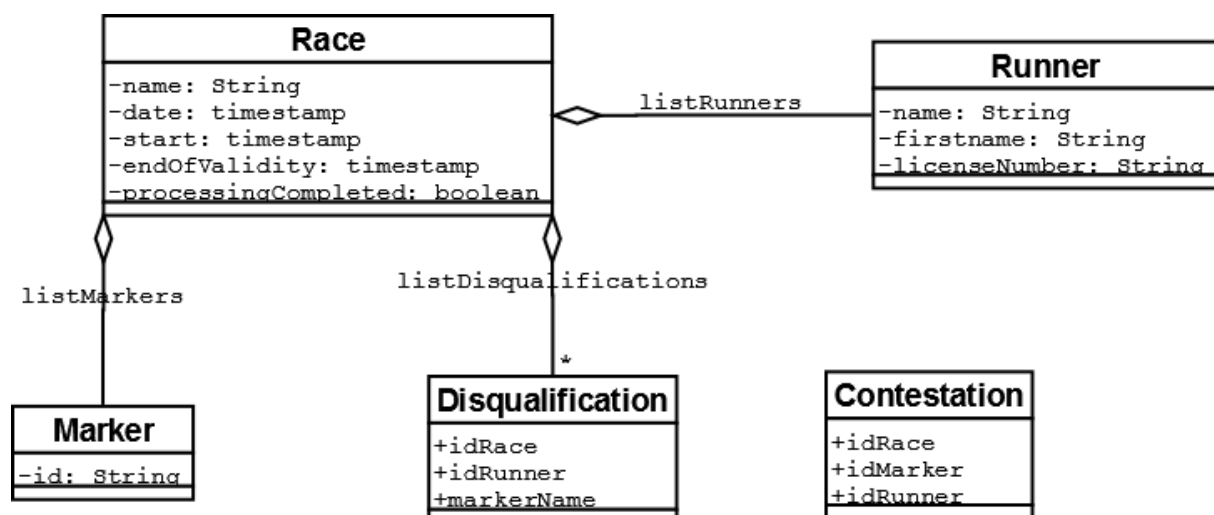
```

race(idRace, name, date, endOfValidity, processingCompleted)
marker(idMarker, markerName)
runner(licenseNumber, name, firstname)
disqualification(idDisqu, idRace, markerName, licenseNumber)
contestation(idConst, time, idRace, runnerLicense, idMarker)

```

3.4.5. Le modèle de l'application de dépôt des contestations

L'application de dépôt des contestations serait elle aussi basée sur l'architecture MVC. Il est donc nécessaire de développer plusieurs classes qui seront le modèle des données. Ce diagramme présente les classes à concevoir :



4. REALISATIONS

4.1. Application Web de dépôt de données

Bien que cela n'était pas demandé dans l'énoncé j'ai décidé de réaliser une petite application Web en PHP pour plusieurs raisons. Tout d'abord cela faisait longtemps que je n'avais pas pu mettre en application ce langage appris durant la première année de Master et j'ai donc voulu « rafraîchir » mes connaissances avec ce site. Cela me servira pour le stage que j'effectue juste après ce projet et qui concerne la programmation objet en PHP 5.

Enfin, je savais que réaliser une brève application web en PHP ne me prendrait pas longtemps et n'empiéterait donc pas beaucoup sur le temps à allouer au reste du projet. Je n'ai consacré que deux jours au développement du site.

C'est pour toutes ces raisons que je me suis limité à une première version légère du site possédant simplement quelques une des fonctionnalités basiques attendues d'une telle application, que nous détaillerons plus en avant dans quelques lignes.

4.1.1. Programmation


4.1.1.1. **La gestion des courses**

Voici un aperçu du formulaire d'ajout de course présenté au comité de course :


Entrez les informations suivantes

Etape 1 - Les informations générales


Nom de la course :

Date (jj/mm/aaaa) : 

Heure de début : H m s

Fin de validité (jj/mm/aaaa) : 

Etape 2 - Les coureurs


Coureurs déjà présents en base de données (pour information) : 

Ajouter nouveaux coureurs disqualifiés :

Nom : Prénom : Numéro de licence :

Nom : Prénom : Numéro de licence :


Etape 3 - Les disqualifications

Ajouter une balise existante : 

Ajouter nouvelles balises :

Nom de la balise :

Numéros de licence des coureurs disqualifiés à cette balise :



Listes des numéros de licence :

On peut voir ici que ce formulaire semble assez complexe mais nous détaillerons son utilisation dans la partie suivante.

On remarque que l'ensemble des informations concernant une course est à saisir sur un seul formulaire et non pas avec une fastidieuse succession de pages.

Lorsque le formulaire est validé, un nouvel objet `Race` est créé. Il se charge alors de créer une liste de `Runner` et de `Marker` avec les informations présentes dans le formulaire. On voit ici que l'utilisateur peut en effet créer directement de nouveaux coureurs ou de nouvelles balises, il n'a pas pour ça à d'abord créer les coureurs ou les balises sur une page puis seulement ensuite à créer la course.

L'objet `Race` crée également une liste d'objets `Disqualification`.

C'est ensuite encore l'objet `Race` qui va enregistrer toutes les informations dans la base de données en bouclant sur toutes les listes qu'il gère. Voici le code permettant ces actions :

```
$values = "SET idRace='".$this->idRace."', name='".$this->name."',
date='".$this->date."', endOfValidity='".$this->endOfValidity."',
processingCompleted='".$this->processingCompleted.'";

if (!$this->update) $requete = "INSERT INTO projet_race {$values}";
else $requete = "UPDATE projet_race {$values} WHERE idRace='".$this->idRace.'";
$base = new MySQL();
$resultat = $base->execute($requete);
if (!$resultat)
```

```

        throw new BdException (mysql_error());

//on parcourt les listes pour enregistrer les objets runner marker et
disqual
foreach($this->listRunners as $key=>$runnerOb) {
    $runnerOb->enregistrer();
}
foreach($this->listMarkers as $key=>$markerOb) {
    $markerOb->enregistrer();
}
foreach($this->listDisqualifications as $key=>$disqOb) {
    $disqOb->enregistrer();
}

```

4.1.1.2. L'utilisation du site par les coureurs

Les coureurs en se connectant voient un tableau semblable à celui-ci :

Course	Date	Action
Defi Wind 800	19/05/2008 - 14H25m08s	Sélectionner
Course de foliiiiiee	22/05/2008 - 11H20m05s	Sélectionner

Ce tableau présente à chaque coureur les courses pour lesquelles le comité l'a disqualifié au passage de certaines balises.

En sélectionnant une course un coureur arrivera sur une page présentant un tableau similaire au suivant :

Course	Balise	Action
Course de foliiiiiee	LaBaliseDeLaMort	Contester
Course de foliiiiiee	LeCasseCroute	Contester
Course de foliiiiiee	OnVoitLeBout	Contester

C'est dans ce tableau qu'il peut voir le détail de ses disqualifications. Ici il a par exemple été disqualifié au passage de trois balises. Il peut décider de contester ces décisions en cliquant sur [Contester](#).

En cliquant sur ce lien il arrivera au formulaire suivant :

Entrez les informations suivantes

Course : Course de foliiiiiee

Date : 22/05/2008 - 11H20m05s

Balise : LaBaliseDeLaMort

Heure de votre passage à cette balise : H m s

Il doit ici simplement indiquer l'heure à laquelle il estime être passé correctement à la balise concernée.

4.1.1.3. Amélioration de l'ergonomie : Javascript et Ajax


Comme nous l'avons vu précédemment le formulaire d'ajout de course semble complexe à première vue. Nous revenons ici sur son utilisation et son ergonomie.

Voici d'abord le formulaire présenté à l'utilisateur lors de l'affichage de la page :


Entrez les informations suivantes

Etape 1 - Les informations générales


Nom de la course :

Date (jj/mm/aaaa) : 

Heure de début : H m s


Fin de validité (jj/mm/aaaa) : 

Etape 2 - Les coureurs

Coureurs déjà présents en base de données (pour information) : 

Ajouter nouveaux coureurs disqualifiés :

Etape 3 - Les disqualifications

Ajouter une balise existante : 

Ajouter nouvelles balises :

Ce formulaire se compose de trois parties principales. La première concerne les informations générales de la course. L'utilisateur doit saisir le nom de la course, la date, l'heure exacte de début et la date à laquelle le dépôt de contestations par les coureurs deviendra impossible. Il peut, pour saisir les dates demandées, cliquer sur les boutons présents à droite des champs afin de faire apparaître un calendrier :



La seconde partie du formulaire concerne l'ajout de nouveaux coureurs. Ici l'utilisateur doit saisir les informations sur les coureurs qu'ils souhaitent disqualifier pour certaines balises et qui ne sont pas déjà présents dans l'application. C'est pour cela qu'il lui est présenté une liste rappelant tous les coureurs déjà présents en base de données.

Pour ajouter des coureurs il doit entrer dans le champ approprié le nombre de coureurs qu'il désire ajouter et cliquer sur Ajouter. Des lignes apparaîtront alors dans le formulaire, permettant de renseigner les informations sur les coureurs :

Ajouter nouveaux coureurs disqualifiés :

Nom : Prénom : Numéro de licence :

Nom : Prénom : Numéro de licence :

J'ai utilisé pour cela la fonction Javascript suivante :

```
function addRunnersLines(nbLines, idDiv) {
    //on controle que le nb de lignes a ajouter est raisonnable :
    if(nbLines > 100) {
        alert("Vous ne pouvez pas ajouter plus de 100 lignes.");
        return;
    }

    //on récupère le nombre de lignes déjà présentes
    var elementsTab = document.getElementsByClassName("runnerLine");

    var num = elementsTab.length;
    //on construit le nombre de lignes demandées
    var lines = "";
    for(var i = 0; i<nbLines; i++) {
        //on calcule le numéro de la ligne pour le mettre en id de la
        div (utilisé pour la suppression)
        var numLine = num + i;
        lines += "<div class='runnerLine'
id='runnerLine"+numLine+"'><p>Nom : <input type='text' value=''
name='runnersNames[]' /> Prénom : <input type='text' value=''
name='runnersFirstnames[]' /> Numéro de licence : <input type='text'
value='' name='runnersLicenses[]' /> <input type='button' value='Effacer'
onclick='removeThing(\"runnerLine"+numLine+"\", \"runnersLines\")'
/></p></div>"
    }
}
```

```
}  
//on insère les nouvelles lignes dans la div voulue :  
new Insertion.Bottom($(idDiv), lines);  
}
```

Enfin la troisième partie du formulaire concerne l'ajout de nouvelles balises et l'enregistrement des disqualifications. L'utilisateur doit ici pour chaque balise où des disqualifications ont été décidées saisir la liste des coureurs disqualifiés. Avant de pouvoir faire cela il doit sélectionner les balises voulues, si elles existent déjà dans l'application, ou en créer directement de nouvelles.

Pour ajouter des nouvelles balises l'utilisateur doit saisir le nombre de balises souhaitées dans le champ approprié puis cliquer sur **Ajouter**.

Voici ce qu'il obtient en ajoutant une balise :

Nom :	<input type="text" value="Dercka"/>	Prénom :	<input type="text" value="Michel"/>	Numéro de licence :	<input type="text" value="23244JD"/>	<input type="button" value="Effacer"/>
Nom :	<input type="text" value="Geuht"/>	Prénom :	<input type="text" value="Michka"/>	Numéro de licence :	<input type="text" value="34352KD"/>	<input type="button" value="Effacer"/>

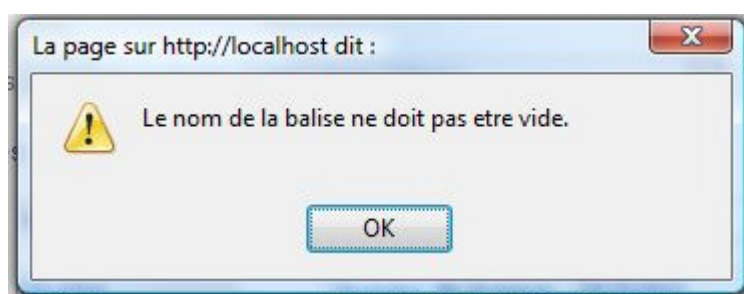
Etape 3 - Les disqualifications

Ajouter une balise existante :

Ajouter nouvelles balises :

Nom de la balise :	<input type="text"/>	<input type="button" value="Effacer"/>
Numéros de licence des coureurs disqualifiés à cette balise :		
<input type="text" value="23244JD Dercka Michel"/>	<input type="button" value="Ajouter"/>	
Listes des numéros de licence :		

Un bloc a été ajouté à la fin du formulaire. Il doit maintenant saisir le nom de la nouvelle balise et ajouter ensuite les coureurs disqualifiés. La liste des coureurs a été construite en partie grâce à une requête Ajax vers le serveur permettant de récupérer la liste des coureurs présents en base de données, et en partie grâce aux nouveaux coureurs ajoutés par l'utilisateur au dessus. Il suffit ensuite de sélectionner un coureur et de l'ajouter. Le nom de la balise doit être saisi au préalable, sinon le message suivant apparaît :



Une fois plusieurs coureurs ajoutés voici le résultat :

Nom de la balise :

Numéros de licence des coureurs disqualifiés à cette balise :

Listes des numéros de licence :

23244JD	<input type="button" value="Effacer"/>
2343Vinc	<input type="button" value="Effacer"/>
SEBVOILE4	<input type="button" value="Effacer"/>
1234XJ	<input type="button" value="Effacer"/>

4.1.1.4. Génération du XML

Le site Internet doit pouvoir, sur demande, générer des flux XML. Ces flux sont de deux types : le premier doit présenter une liste de toutes les courses actuellement disponibles pour traitement, et le second doit présenter les détails de toutes les contestations pour une course précise.

J'ai décidé de suivre l'idée de l'architecture REST reposant sur des requêtes effectuées grâce à des URI. Ainsi la requête suivante produire un résultat spécifique :

...chemin/query.php?type=race&idRace=2&user=comite&pass=passcomite

Ici on demande l'affichage d'un flux XML pour une course particulière (type=race), dont l'identifiant est 2 (idRace=2), et on s'identifie en passant un nom d'utilisateur et un mot de passe (user=comite&pass=passcomite).

Voici un exemple de flux en sortie :

```
<race id='2'>
  <name>Defi Wind 800</name>
  <date>19/05/2008</date>
  <markers>
    <marker id='PhareBreton' />
  </markers>
  <start>
    <hour>14</hour>
    <minute>25</minute>
    <second>08</second>
  </start>
  <contestations>
    <contestationsMarker idMarker='PhareBreton'>
      <contestation>
        <runnerNumber>BALZROCK4</runnerNumber>
        <hour>14</hour>
        <minute>30</minute>
        <second>25</second>
      </contestation>
    </contestationsMarker>
  </contestations>
  <runners>
    <runner id='BALZROCK4'>
      <firstname>Maxime</firstname>
    </runner>
  </runners>
</race>
```

```
        <name>Balzeau</name>
    </runner>
</runners>
</race>
```

Le XML produit pour la requête de l'autre type sera de la forme suivante :

Requête :

...chemin/query.php?type=raceslist&idRace=3&user=comite&pass=passcomite

Résultat :

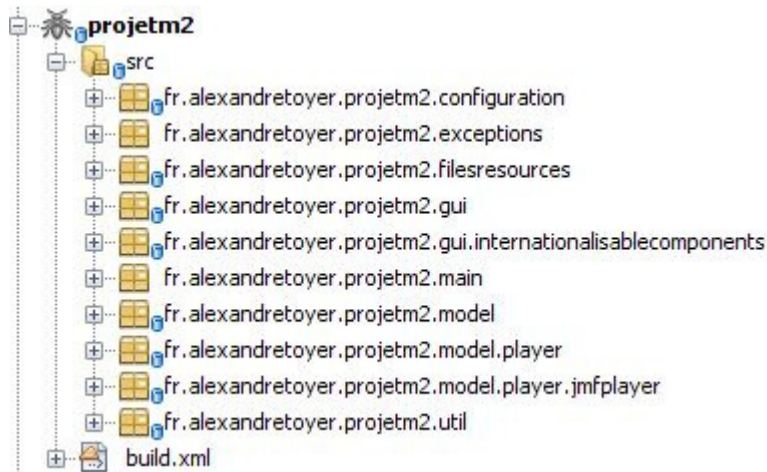
```
<raceslist>
  <race id='2'>
    <name>Defi Wind 800</name>
    <date>19/05/2008</date>
    <start>
      <hour>14</hour>
      <minute>25</minute>
      <second>08</second>
    </start>
  </race>
  <race id='3'>
    <name>Course de foliiiiieeee</name>
    <date>22/05/2008</date>
    <start>
      <hour>11</hour>
      <minute>20</minute>
      <second>05</second>
    </start>
  </race>
</raceslist>
```

4.2. [Application lourde d'accès ciblé aux vidéos en Java](#)

4.2.1. [Architecture de l'application](#)

4.2.1.1. Présentation

L'application Java repose entièrement sur l'architecture Modèle Vue Contrôleur. Je l'ai réalisé avec l'IDE Netbeans, qui a énormément facilité le travail. Voici un aperçu des packages créés :



4.2.1.2. Lecture du XML

Un objet Race se fabrique à partir d'un flux XML. J'ai utilisé pour parser le flux XML retournée par l'application web l'API DOM, qui est facile d'utilisation et assez souple à utiliser. Cela ne posait pas de problème ici, puisque les flux XML que récupèrera l'application seront assez courts et l'inconvénient majeur de DOM est sa lenteur sur le traitement de longs fichiers.

Pour récupérer un flux XML j'utilise le code suivant :

```
//récupération du fichier XML décrivant la course : on ajoute à la
fin de la requete l'id de la course
URL url = new URL("http", Configuration.BASE_SERVER, 80,
Configuration.BASE_QUERY + Configuration.RACE_INFO_QUERY + this.id);

is = url.openStream();
InputStream isc = new InputStream(is);
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document document = db.parse(isc);
```

Un objet `InputStream` est construit à partir du contenu d'une page web, récupéré grâce à un objet `URL`. Cet `InputStream` est ensuite parsé par un objet `DocumentBuilder` afin de fabriquer un objet `Document`.

Lors du parsing du fichier on construit ensuite les objets `Runner` et `Marker` à l'aide des méthodes adéquates de la classe `Factory` (`getRunnerFromElement(Element element) ...`) ainsi que les listes de `Contestation` correspondantes.

4.2.2. Fonctionnement de JMF

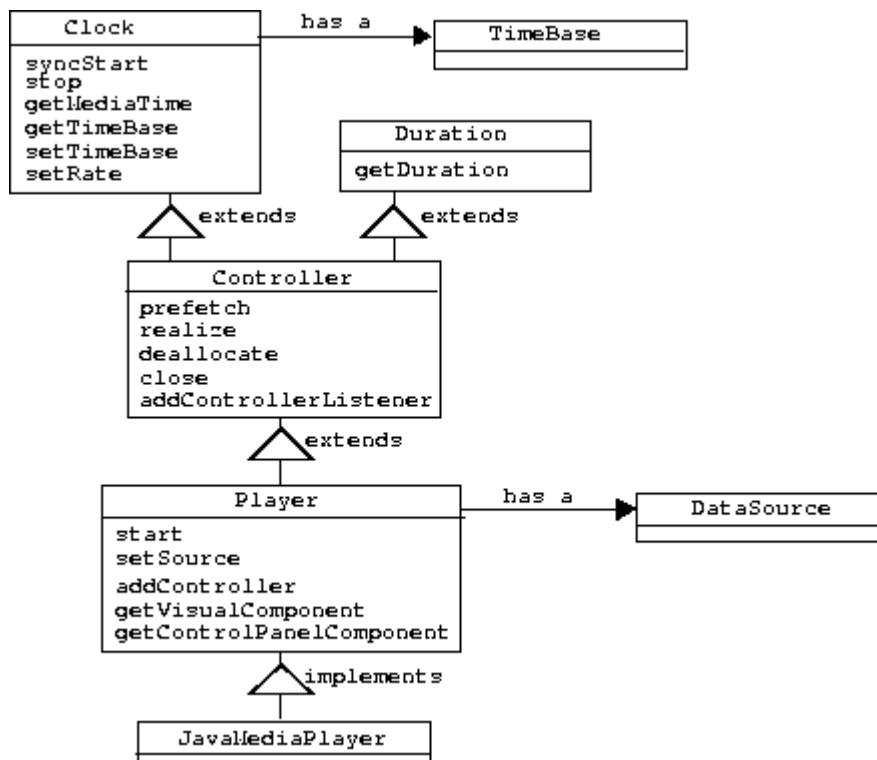
JMF est l'API de traitement de vidéos proposé par Sun, le créateur du langage Java. Cet ensemble de classes n'est pas fourni en standard avec le langage Java et doit être téléchargé et installé séparément.

L'API peut être téléchargée à l'adresse suivante :

<http://java.sun.com/javase/technologies/desktop/media/jmf/2.1.1/download.html>

Je présenterai ici brièvement l'utilisation de JMF pour créer un lecteur capable de lire une vidéo.

Voici un schéma disponible dans la documentation de l'API :



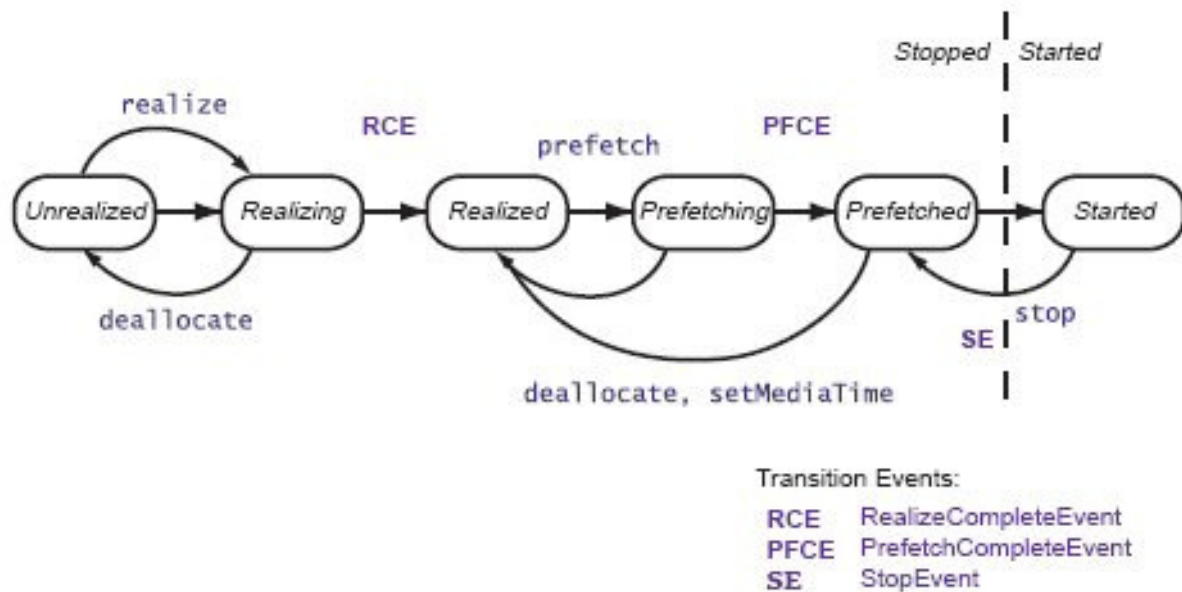
L'objet `MediaPlayer` traite un flux de données temporalisé qu'il lit à partir d'un objet `DataSource`.

L'objet `Clock` définit les opérations basiques de synchronisation qu'un `Player` utilise pour présenter les données multimédia.

L'objet `Controller` fournit des méthodes pour gérer les ressources du système et charger les données. Il offre de plus un mécanisme de `listener` permettant aux objets implémentant l'interface `ControllerListener` de recevoir des événements particuliers.

L'interface `Player` fournit des méthodes afin de récupérer des composants permettant de visualiser le flux vidéo et d'interagir dessus. Il est possible de définir des composants personnalisés pour remplacer ceux fournis par défaut, qui ont un aspect assez vieillot.

Lors de la création d'un `Player` ce dernier doit passer par plusieurs états avant d'être prêt pour lire le flux vidéo. Ces états sont représentés dans le schéma suivant :



Ils sont au nombre de six et se trouvent dans les petites bulles. Les mots violets écrits sur les flèches correspondent aux évènements levés par le Player lors de ses changements d'états.

On voit que le Player doit passer par de nombreux états avant de pouvoir enfin commencer la lecture et le rendu de la vidéo.

Plusieurs méthodes, définies dans les différentes interfaces et classes parentes, sont disponibles pour agir sur un `MediaPlayer`. Toutes ces méthodes ne peuvent pas être invoquées lors de n'importe quel état du `Player`, et il faut être sûr que le `Player` soit dans un état correct avant d'appeler une méthode dessus. Il est par exemple interdit d'appeler la méthode `setRate()` sur un `Player` lorsqu'il est dans l'état `Unrealized`.

Les méthodes permettant d'agir sur un `Player` sont nombreuses et nous ne les détaillerons pas ici. En voici quelques unes :

- `start()` : permet de démarrer le `Player`
- `stop()` : arrête le `Player`
- `setMediaTime(Time t)` : permet de spécifier le temps de lecture du `Player`
- `setRate(int r)` : permet de spécifier le taux auquel le `Player` doit lire la vidéo. Un `rate` de 2 lira la vidéo deux fois plus vite que la normale. Il est ainsi possible de reculer en indiquant un `rate` négatif.

L'utilisation de JMF ne semble pas trop compliquée à la lecture de la documentation. Cependant cette API étant assez ancienne elle ne permet pas de lire beaucoup de formats vidéo. J'ai ainsi dû chercher longtemps et faire plusieurs essais avant de trouver des vidéos lisibles.

J'ai par la suite trouvé un plugin à JMF, `jffmpeg`, qui permet la lecture de plus de formats vidéo. Je n'ai par contre pas eu le temps de le tester à fond.

Dès le départ du développement il m'a fallu des vidéos pour pouvoir faire des tests. J'avais donc filmé pendant environ une heure des trams passant à l'arrêt de l'université, afin de reproduire le comportement des planches à voiles passant les balises. Cependant leurs passages n'était pas assez fréquent et la vidéo un peu trop semblable tout le long pour s'y retrouver.

J'ai par la suite trouvé sur <http://www.youtube.com> des vidéos de courses de planches à voile filmées à partir de bateaux présents près des balises, qui était à peu près le genre de vidéo que l'application devait lire. Je les ai donc téléchargées et je les ai ré-encodées afin de pouvoir les lire avec JMF.

4.2.3. Interface graphique en Swing

4.2.3.1. **Layout et placement des composants**

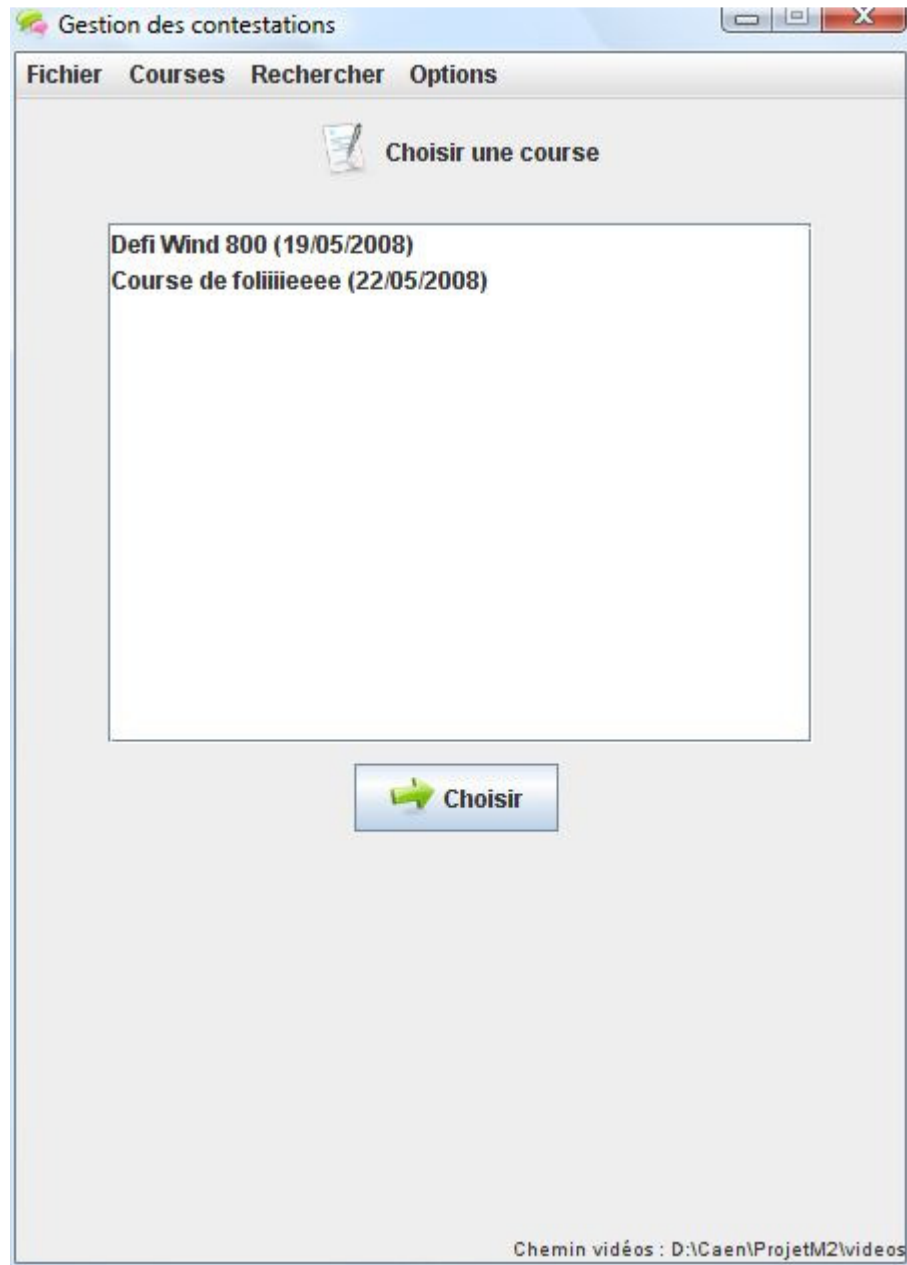
De manière générale j'ai utilisé principalement des composants Box, qui sont des composants assez simples d'utilisation mais offrant une souplesse de mise en forme assez grande.

Ces composants Swing utilisent un layout BorderLayout qui est assez limité en termes de placement, puisque qu'il permet simplement de positionner des éléments soit sur une ligne, soit sur une colonne. Cependant, en utilisant de nombreuses Box et en les imbriquant il est relativement facile d'obtenir rapidement un placement satisfaisant pour l'ensemble des éléments de l'application.

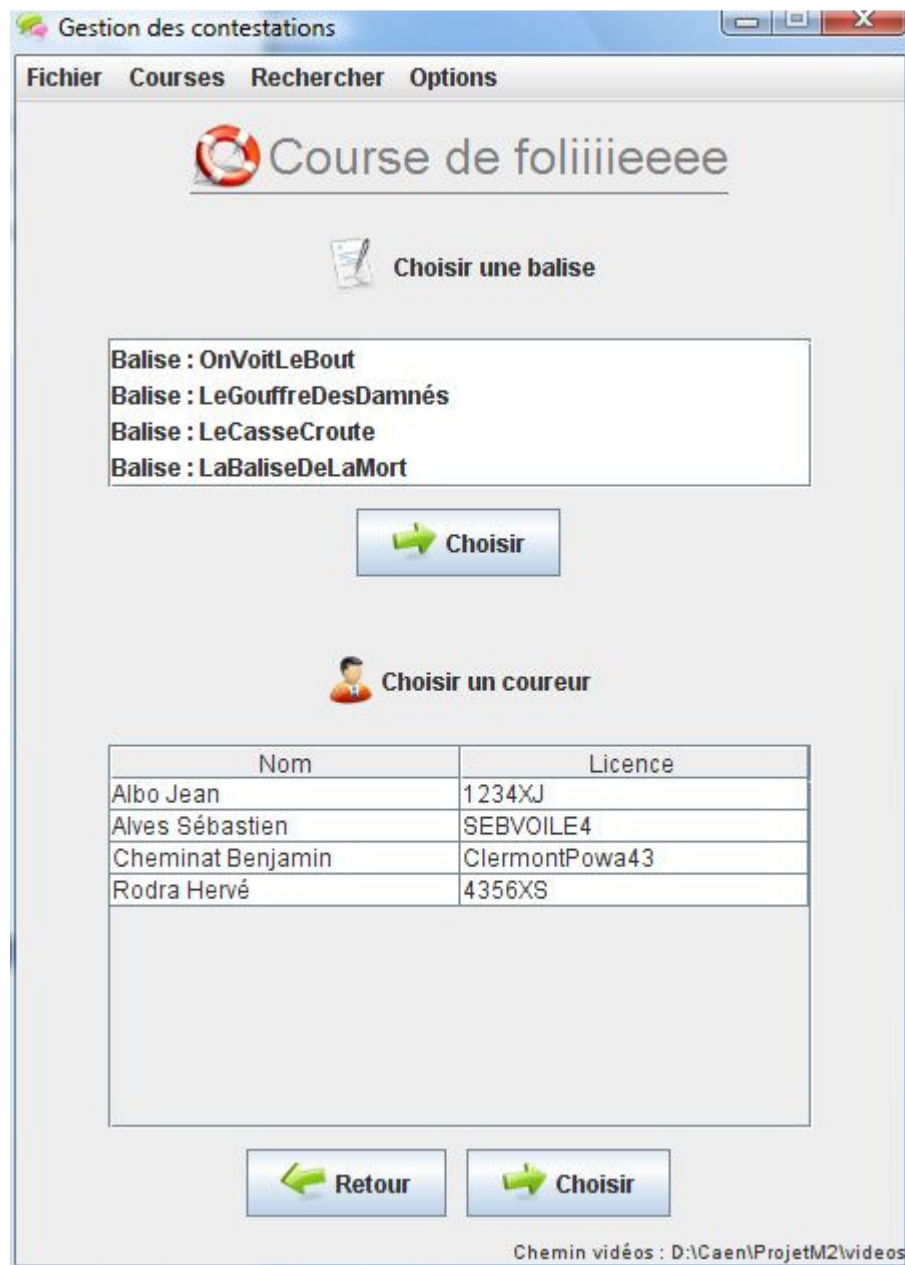
L'application devait ici proposer un système d'enchaînement d'écran. En effet, après avoir sélectionné une course l'utilisateur doit obtenir deux listes, une liste de balises et une liste de coureurs. Enfin, après avoir sélectionné une balise ou un coureur il obtient une liste de contestations.

Voici des captures d'écrans montrant cet enchaînement :

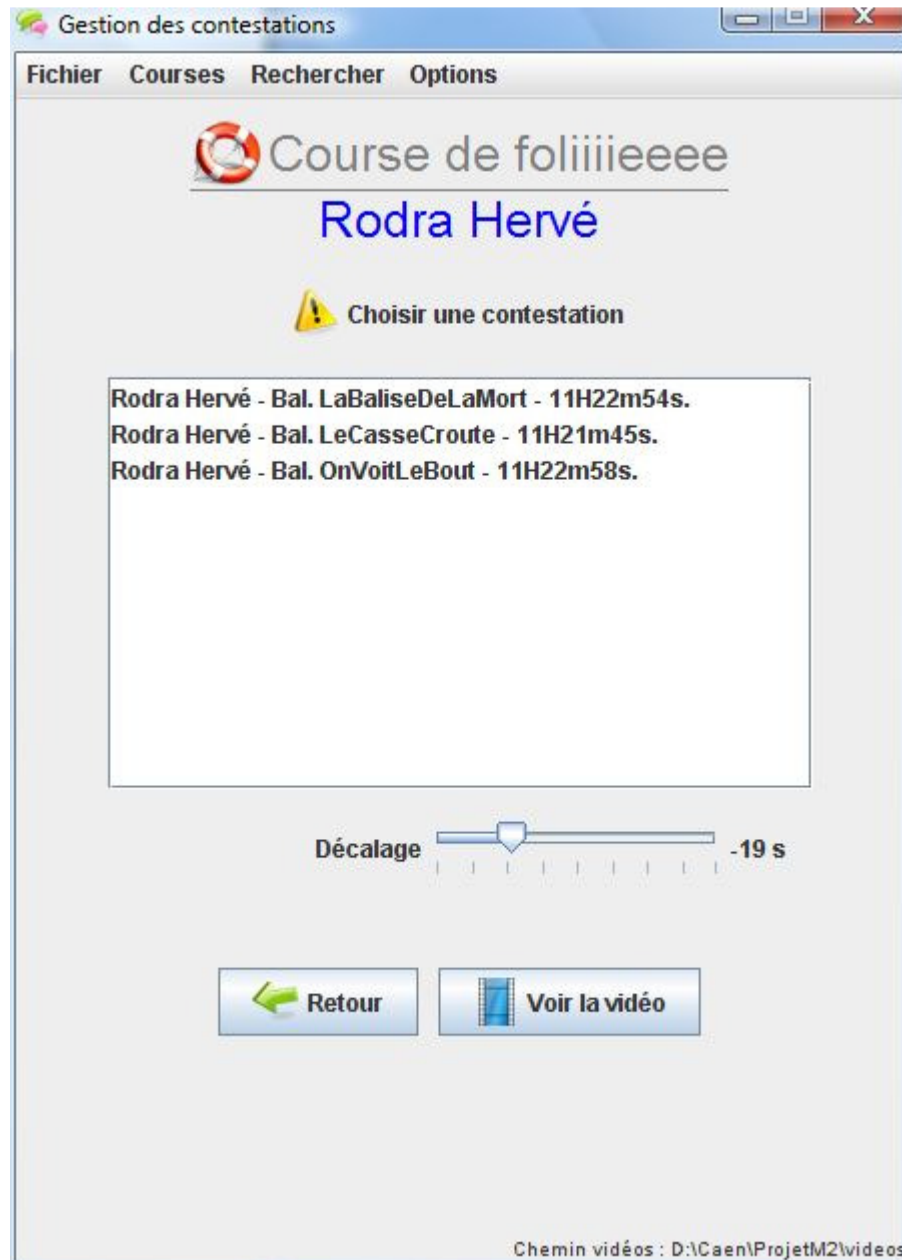
La première fenêtre vue par l'utilisateur :



Le second écran :



La liste des contestations :



Pour implémenter ce fonctionnement j'ai utilisé sur le `JPanel` placé au centre de la `JFrame` principale un `layout CardLayout`. Ce gestionnaire de placement permet de créer plusieurs `card`, sorte de cartes s'empilant les unes sur les autres. Il est ensuite possible d'afficher l'une des `card` créé en l'appelant par le nom choisi lors de sa création.

Ainsi, lorsque je veux montrer la liste des balises et des coureurs le code est le suivant :

```
public void showMarkersRunners(Race race) {
    //On a cliqué sur une course, on veut maintenant afficher les
    coureurs et les balises pour cette course
    race.setRemainingFromXML();
    application.setCurrentRace(race);

    guiMarkersRunners = new GuiMarkersRunners(this);
    mainPanel.add(guiMarkersRunners,
    Configuration.GUI_MARKERS_RUNNERS_NAME);
}
```

```
cardLayout.show(mainPanel,  
Configuration.GUI_MARKERS_RUNNERS_NAME);  
}
```

Le fonctionnement est semblable pour l’affichage de la liste des courses et de la liste des contestations.

`GuiMarkersRunners`, `GuiContestationsList` et `GuiRacesList` sont des `JPanel` contenant différentes listes et qui sont placés dans des `card` au niveau du `JPanel` principal.

4.2.3.2. Lecture des vidéos

Je présenterai dans cette partie la fenêtre de lecture des vidéos et les classes associées.

Dans la capture d’écran précédente, présentant la liste des contestations, nous pouvions voir un petit `slider` permettant de régler un léger décalage en secondes. Ce décalage est utilisé ici pour lancer la lecture de la vidéo non pas à l’heure du passage estimée par le coureur mais juste avant ou juste après selon le réglage de ce temps de décalage.

Voici une capture d’écran de la fenêtre de lecture :



En haut de la fenêtre des informations générales sont présentes : nom de la course, nom et numéro du coureur, nom de la balise, et en gras l'heure estimée par le coureur pour son passage à la balise.

En dessous se trouve la vidéo en train d'être lue, représentant la balise.

Enfin en bas est présent un panneau de contrôle de la vidéo permettant différentes actions sur celle-ci. Le grand slider permet de suivre l'état d'avancement de la vidéo et surtout de se déplacer à un instant donné. Les boutons du dessous permettent de stopper la vidéo, reculer, mettre en pause ou avancer. A droite se trouve un petit JLabel indiquant l'heure. Ce champ n'indique pas simplement le temps de lecture de la vidéo, par exemple « 2 min » si cela fait deux minutes que la vidéo est commencée, mais il indique l'heure qu'il était au moment exacte de l'image. Ce champ change donc de valeur toutes les secondes. Il est ainsi très facile de se positionner exactement à l'heure estimée par le coureur pour son

passage. J'utilise pour cela le fichier XML présent dans le répertoire des vidéos et indiquant à quelle heure la vidéo a commencé à être enregistrée.

Il aurait été intéressant que le `player` lève des évènements à chaque changement de temps (de seconde par exemple). Il aurait ainsi été simple de capter ces évènements et de les utiliser pour mettre à jour le champ indiquant l'heure et le `slider` indiquant l'avancement de la lecture.

Cependant les `player JMF` ne gère pas ce genre de chose. J'ai ainsi été obligé d'utiliser une autre solution, moins performante. Pour le `slider` j'ai créé un `Thread ThreadMoveSlider` dont la méthode `run()` est la suivante :

```
public void run() {
    while (running) {
        try {
            if (!cont.isSliderMoving()) {
                long percentPlayer = (long) (((long) 100.000000)
* player.getTimeNanoseconds() / (player.getDuration().getNanoseconds()));
                int valueSlider = (int) ((slider.getMaximum() *
percentPlayer) / 100);
                slider.setValue(valueSlider);
                sleep(25);
            }
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}
```

Toutes les 25 ms le thread calcule le positionnement actuel de la vidéo et place le curseur du `slider` à l'endroit correspondant.

Un système similaire est utilisé pour modifier le `JLabel` indiquant l'heure : toutes les 1000 ms le texte du `JLabel` est mis à jour en fonction du temps actuel du `player` récupéré grâce à un appel à `getTime()` sur le `player`.

A l'heure actuelle le défilement image par image n'a pas encore été implémenté. Je rencontre de plus quelques difficultés à faire fonctionner les modes avance et retour rapide. Il semblerait que le temps soit bien affecté par le changement de `rate` mais l'image ne suit pas.

4.2.3.3. Internationalisation de l'interface

J'ai voulu rendre l'application internationalisable, c'est-à-dire pouvoir en modifier la langue, puisque cela peut toujours être utile. De plus cela était intéressant du point de vue de la programmation.

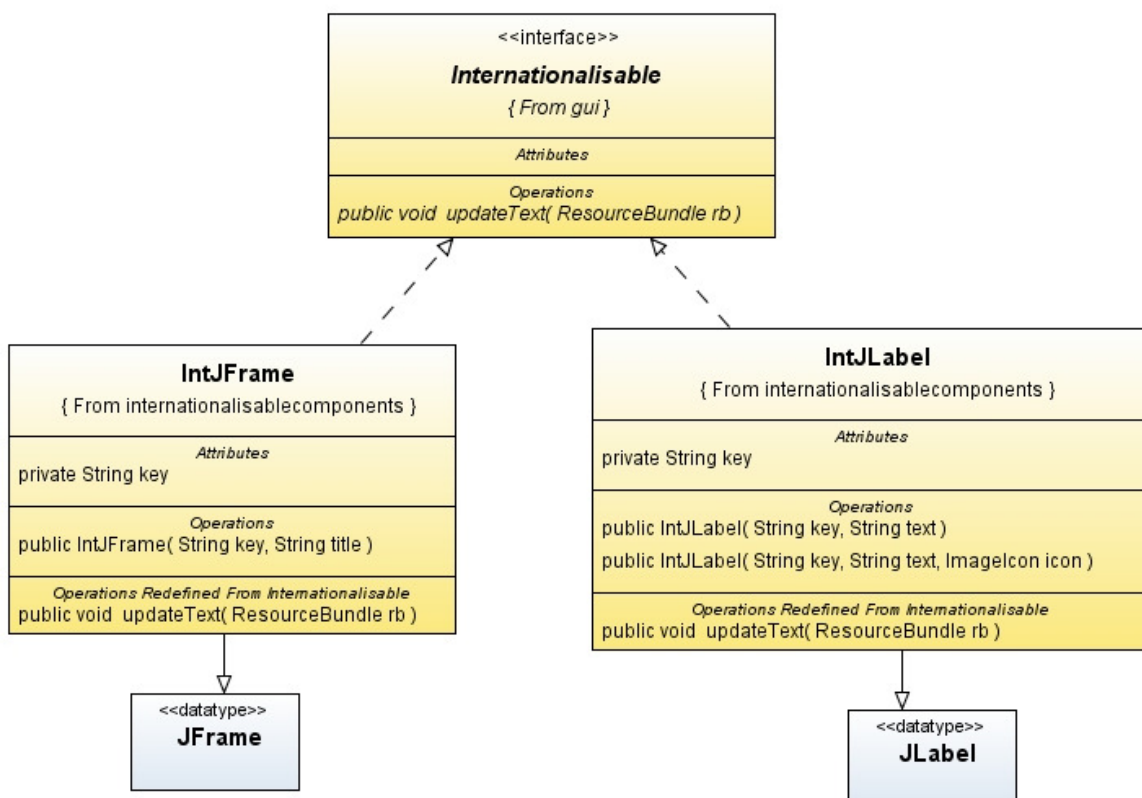
J'avais au départ adopté la solution suivante : j'abonnais chaque composant possédant des éléments internationalisables au composant gérant la langue. Ainsi dès que la langue était changée ce composant levait un évènement qui était capté par tous les composants abonnés. Ces derniers repeignaient alors leurs éléments internationalisables selon la langue en cours dans l'application.

J'ai utilisé pour gérer la langue un `ResourceBundle`, qui est un objet de l'API Java permettant l'utilisation de fichiers `.properties` selon la langue choisie. La langue est représentée dans le code à travers ce que l'on appelle la "Locale". J'ai créé plusieurs fichiers `.properties` (deux pour le moment, un pour le français et un pour l'anglais) dans lesquels j'ai placé des traductions en chaque langue de tous les textes à afficher dans l'application (menus, boutons, labels, etc.).

Pour récupérer un texte précis il suffit d'appeler la méthode `getMessage` sur l'objet `ResourceBundle` en lui passant comme paramètre une clé identifiant le texte à récupérer dans le fichier `.properties`. Selon la `Locale` actuelle de l'application l'objet ira chercher ce texte dans l'un ou l'autre des fichiers.

Pour modifier une langue j'ai ajouté un élément dans le menu Options qui ouvre une petite fenêtre proposant les différentes langues dans une liste déroulante. Lors du clic sur OK la `Locale` de l'application est modifiée selon le choix de l'utilisateur.

La solution de l'abonnement des composants internationalisables ne plaisait pas à mon tuteur qui la trouvait trop fastidieuse à utiliser, puisque pour placer un élément internationalisable il fallait d'abord le créer, mais aussi ajouter une ligne dans la méthode recevant l'évènement de changement afin qu'il se repeigne. Il m'a proposé une solution plus intéressante. Le diagramme suivant la présente partiellement :



On peut en effet créer des nouveaux composants, par exemple des nouveaux `JFrame` ou `JLabel`, héritant des composants swing classiques, en les faisant implémenter une

nouvelle interface, que j'ai appelé `Internationalisable`. Cette interface ne contient qu'une méthode dont la signature est la suivante :

```
public void updateText(ResourceBundle rb);
```

Ainsi chaque composant pourrait se repeindre lui-même dès que l'on appellerait cette méthode. Pour créer un composant internationalisable il suffit de créer un `IntJFrame` par exemple, de l'ajouter à une liste contenant tous les éléments internationalisables et gérer au niveau de la classe gérant la langue. Lorsque cette dernière sera changée, on parcourra la liste pour appeler sur tous les composants leur méthode `updateText()` en leur passant le `ResourceBundle` à utiliser. Le code est ainsi moins dispersé et plus facile à maintenir. Le seul inconvénient est de devoir créer une nouvelle classe pour chaque nouveau type de composant internationalisable.

4.2.3.4. Affichage de listes de données

Comme nous l'avons vu précédemment l'application affiche plusieurs fois des listes. Ces listes sont le plus souvent des `JList` (liste des courses, des balises et des contestations) et une est un `JTable` (la liste des coureurs sur le deuxième écran).

Comme tous les composants Swing les `JList` utilisent un modèle qui leur est propre et affiche les éléments contenus dans ce modèle. Pour créer une liste il faut donc créer un modèle personnalisé, appelé un `ListModel` en Java.

Prenons l'exemple de la liste des courses.

J'ai créé une classe `RacesListAdapter` dérivant de la classe `AbstractListModel` appartenant à l'API de J2SE. Cette classe a pour rôle de faire la correspondance entre le modèle de l'application contenant la liste des courses disponibles et la `JList` permettant de visualiser cette liste.

Elle contient plusieurs méthodes, telles que `getSize()` et `getElementAt()` ayant respectivement comme rôle de retourner la taille de la liste et un élément à une position précise dans la liste. La `JList` utilise ces valeurs retournées pour pouvoir s'afficher. Il suffit donc dans ces méthodes de retrouver les valeurs correspondantes dans la liste des courses et des les renvoyer.

Voici le code de cette classe :

```
class RacesListAdapter extends AbstractListModel implements ChangeListener
{
    ArrayList<Race> racesList;

    RacesListAdapter(Application appli) {
        this.racesList = appli.getRacesList();
        appli.addChangeListener(this);
    }
    public int getSize() {
        return racesList.size();
    }
    public Object getElementAt(int index) {
        return racesList.get(index);
    }
    public void stateChanged(ChangeEvent e) {
```

```
        fireContentsChanged(this, 0, racesList.size());
    }
}
```

`ContestationsListAdapter` et `MarkersListAdapter` fonctionne sur le même principe. Seul `RunnersListAdapter` varie un peu, puisque la vue est un `JTable` et non une `JList` mais l'idée reste la même.

4.2.4. Amélioration de l'expérience utilisateur

4.2.4.1. Stockages des préférences de l'utilisateur

L'application possède plusieurs options de paramétrage : la langue, le chemin d'accès au répertoire des vidéos et le temps de décalage utilisé pour visionner une vidéo. Il serait fastidieux pour l'utilisateur d'avoir à saisir toutes ces informations à chaque nouveau lancement de l'application. L'idéal serait donc de pouvoir enregistrer ces valeurs d'une exécution du programme à l'autre.

C'est ce que permet la classe `Preferences` offerte par l'API standard de Java. Le but de cette classe est en effet de stocker des préférences utilisateurs (ou système) et de pouvoir les retrouver ultérieurement. Ces préférences sont stockées à des endroits variant selon les machines (base de registre, fichiers, ...), mais ce détail n'a pas d'importance, l'utilisation restant toujours la même.

Ainsi à l'ouverture de l'application le programme va rechercher le chemin d'accès aux vidéos et la langue dans les préférences pour que l'utilisateur retrouve les valeurs qu'il avait saisies lors sa dernière utilisation. Cela se fait simplement via un appel tel que celui-ci :

```
prefs.get(nomDuParametre, ValeurParDefautSiNonPrésent) ;
```

La valeur du temps de décalage est récupérée lorsque la liste des contestations doit s'afficher afin de mettre le curseur du `slider` à la bonne valeur.

L'écriture des préférences se fait aussi simplement que la lecture, dès qu'un paramètre est modifié, par l'appel à :

```
prefs.put(nomDuParametre, ValeurDuParametre) ;
```

4.2.4.2. Utilisation des Thread

Dans une interface graphique réalisée en Swing certaines actions effectuées par l'utilisateur peuvent nécessiter un certain temps pour s'exécuter totalement. Tant que l'action n'est pas complètement terminée l'interface reste bloquée dans l'état antérieur au départ de l'action.

Par exemple lors d'un clic sur un bouton effectuant une action demandant un temps important pour s'exécuter (lecture d'un fichier, parsing d'un flux) l'interface restera bloquée, le bouton apparaîtra comme étant encore « enfoncé » alors même que l'utilisateur aura relâché la souris, et plus aucune action ne sera possible, comme fermer la fenêtre par exemple.

Pour remédier à ce problème nous devons utiliser des threads. Le principe est simple : chaque action risquant d'être longue à effectuer sera réalisé dans un thread secondaire. Ainsi lors du lancement de l'action un thread est lancé pour l'exécuter, et le thread principal contenant la fenêtre peut continuer de recevoir des actions.

J'ai utilisé cette solution pour deux actions jusqu'à présent : l'ouverture de la boîte de dialogue permettant de choisir un répertoire pour les vidéos puisque je m'étais rendu compte que cette boîte prenait un certain temps à apparaître, et le parsing du flux XML contenant les détails d'une course.

Voici le code de ce dernier exemple :

La méthode appelée pour afficher la liste des balises et des coureurs de la course :

```
public void showMarkersRunners(Race race) {
    //On a cliqué sur une course, on veut maintenant afficher les
    coureurs et les balises pour cette course
    new ThreadShowMarkersRunners(this, race, application,
    mainPanel, cardLayout, guiMarkersRunners).start();
}
```

Le thread ThreadShowMarkersRunners :

```
class ThreadShowMarkersRunners extends Thread{

    Race race;
    Application application;
    JPanel mainPanel;
    CardLayout cardLayout;
    GuiMarkersRunners guiMarkersRunners;
    Gui gui;

    public ThreadShowMarkersRunners(Gui gui, Race race, Application
    application, JPanel mainPanel, CardLayout cardLayout, GuiMarkersRunners
    guiMarkersRunners) {
        this.gui = gui;
        this.race =race;
        this.application = application;
        this.mainPanel = mainPanel;
        this.cardLayout = cardLayout;
        this.guiMarkersRunners = guiMarkersRunners;
    }

    public void run() {
        race.setRemainingFromXML();
        application.setCurrentRace(race);
        guiMarkersRunners = new GuiMarkersRunners(gui);
        mainPanel.add(guiMarkersRunners,
        Configuration.GUI_MARKERS_RUNNERS_NAME);
        cardLayout.show(mainPanel,
        Configuration.GUI_MARKERS_RUNNERS_NAME);
    }
}
```

On voit que la méthode `run()` de ce thread appelle `setRemaininFromXML()` sur un objet `Race`, qui va servir à remplir l'objet à partir du parsing du flux XML de la course, récupéré sur le site Internet.

4.2.4.3. Traitement et affichage des erreurs

Il n'est rien de plus frustrant dans l'utilisation d'un logiciel que lorsque ce que l'on veut faire ne marche pas sans que l'on sache pourquoi. C'est pour cela que j'ai voulu implémenter une gestion des erreurs, et plus particulièrement afficher des messages explicites à l'utilisateur.

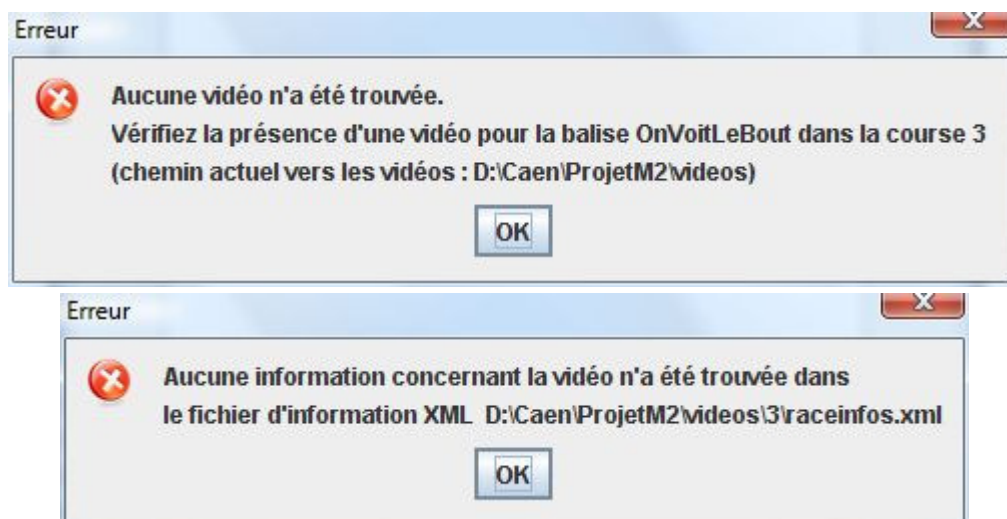
Plusieurs erreurs peuvent survenir dans l'application :

1. Si aucun chemin vers les vidéos n'est trouvé à l'ouverture de l'application (normalement uniquement à la première ouverture de l'application puisque le chemin est ensuite stocké).
2. Si la connexion au site Internet pour récupérer les données XML ne peut pas se faire.
3. Si aucune vidéo ne correspond à la balise pour laquelle on essaie de visionner une contestation.
4. Si aucune information concernant une vidéo n'est présente dans le fichier XML d'informations présent dans le répertoire de la vidéo.

Voici par exemple le message affiché à l'utilisateur dans le cas de l'erreur 2 :



Dans le cas des erreurs 3 et 4 :



Ces derniers messages sont affichés grâce à la méthode `showMessageDialog` de la classe `JOptionPane`. Cette classe Java permet d'afficher des fenêtres d'information à l'utilisateur.

CONCLUSION

L'arbitrage de courses nautiques est souvent difficile à réaliser en raison du grand nombre de coureurs. Il arrive que ceux-ci contestent les décisions des arbitres, et il est actuellement compliqué de répondre à ces contestations. Pour faciliter ce traitement, il était intéressant de mettre en place un système permettant de visionner la course, ne laissant ainsi aucune place au doute.

J'ai d'abord défini les fonctionnalités attendues par les futurs utilisateurs de l'application, avant de réfléchir à leur conception et leur réalisation, à l'aide de différents schémas. J'ai ensuite dû effectuer une étape d'analyse afin d'utiliser par la suite les technologies les plus adéquates. J'ai alors pu réaliser une importante application en Java, s'appuyant sur un petit site utilisant PHP et MySQL.

Ce projet fut une expérience enrichissante de par le nombre de connaissances supplémentaires qu'il m'a apporté en Java, qui est un langage très demandé à l'heure actuelle dans le milieu informatique et qui me servira certainement plus tard. De plus, j'ai pu remettre à jour mes connaissances en PHP et en Ajax, ce qui me sera utile très rapidement.

Je retiendrai le problème des formats des fichiers vidéo et de la lecture de vidéos en général, qui a pesé sur le projet. Il en résulte que le langage Java ne serait peut-être pas dans l'absolu, en dehors du cadre de ce projet, le meilleur choix pour la réalisation d'une application de ce type en raison de ses maigres performances dans le domaine du traitement vidéo.

GLOSSAIRE

PHP : Langage de programmation utilisé dans le développement de sites internet.

Base de données : Ensemble de données se rapportant à un même sujet et gérées par un système de gestion de base de données. On peut interroger le système pour récupérer des données ou en enregistrer de nouvelles.

Java : Langage de programmation permettant le développement de logiciels possédant des interfaces graphiques (fenêtres, boutons, etc.) importantes.

XML : Technologie permettant de structurer des données de façon universelle, compréhensible par de nombreux logiciels.

Format : En informatique le format est la manière utilisée pour représenter des données. Il existe de nombreux formats vidéo.

Thread : Processus léger s'exécutant en parallèle des autres.

REFERENCES

Jérôme Lefèvre, *Intégration du multimédia à Java avec JMF* [En ligne]
<<http://www.supinfo-projects.com/fr/2004/java%5Fjmf/0/>> (consulté le 25 mai 2008)

Sun Microsystems, *The Java Tutorials* [En ligne]
<<http://java.sun.com/docs/books/tutorial/index.html>> (consulté le 25 mai 2008)

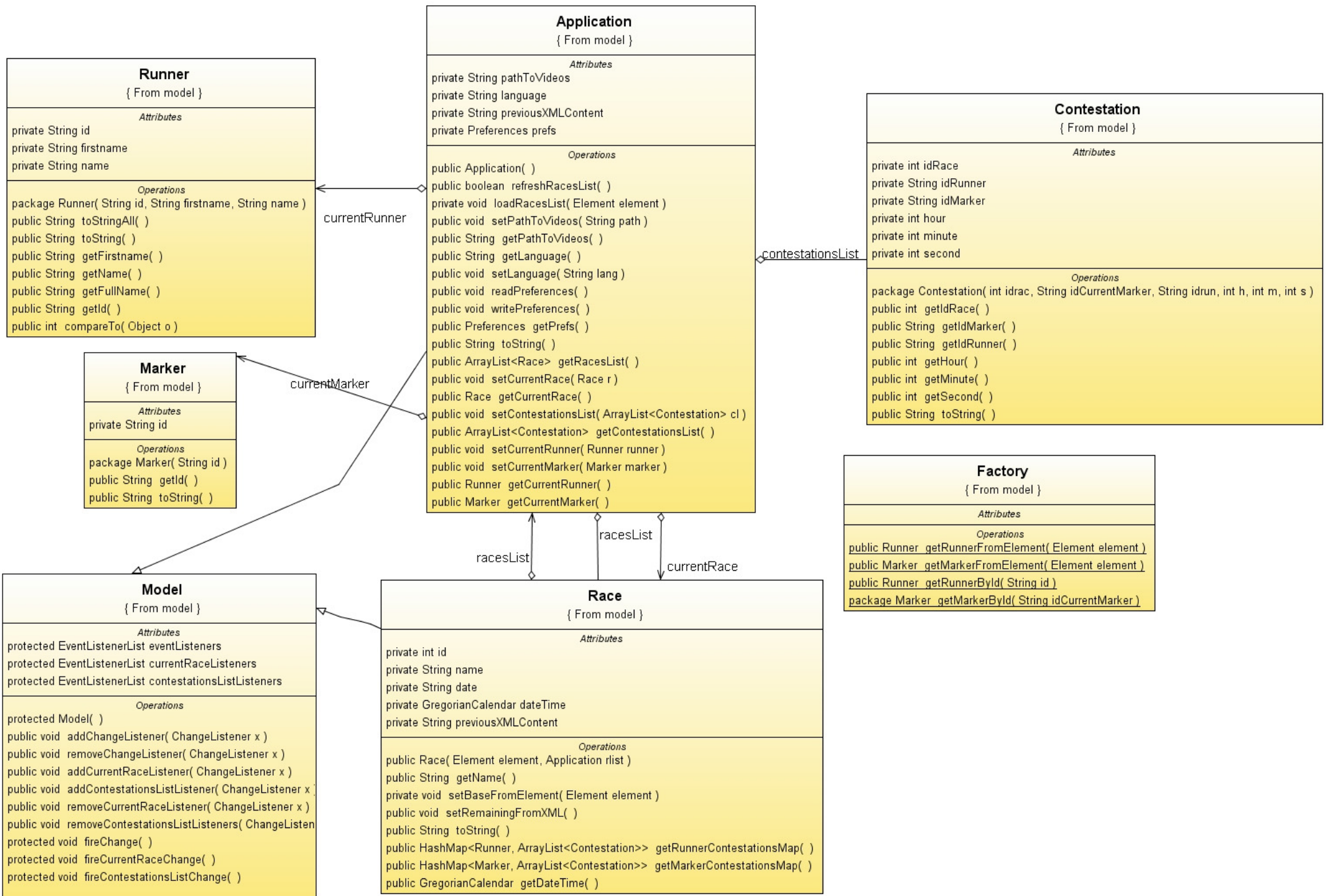
Sun Microsystems, *JMF Programmers Guide* [En ligne]
<<http://java.sun.com/javase/technologies/desktop/media/jmf/1.0/guide/index.html>> (consulté le 25 mai 2008)

Sun Microsystems, *Java API Specification* [En ligne]
<<http://java.sun.com/j2se/1.5.0/docs/api/>> (consulté le 25 mai 2008)

Prototype, *Prototype API Documentation* [En ligne]
<<http://www.prototypejs.org/api>> (consulté le 25 mai 2008)

ANNEXES

Annexe 1 : Diagramme de classes du modèle de l'application



Annexe 2 : Classe Race

```
package fr.alexandretoyer.projetm2.model;

import fr.alexandretoyer.projetm2.configuration.Configuration;
import java.util.*;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;

/**
 *
 * @author Alex
 */
public class Race extends Model {

    private HashMap<Runner, ArrayList<Contestation>> runnerContestationsMap
= new HashMap();
    private HashMap<Marker, ArrayList<Contestation>> markerContestationsMap
= new HashMap();
    private int id;
    private String name;
    private String date;
    private GregorianCalendar dateTime;
    private Application racesList;
    private String previousXMLContent;

    public Race(Element element, Application rlist) {
        racesList = rlist;
        previousXMLContent = "";
        setBaseFromElement(element);
    }

    public String getName() {
        return name;
    }

    /**
     * Permet de terminer le traitement pour une course
     */
    public void terminate() {
        try {
            //on ouvre l'URL permettant de spécifier que le traitement pour
la course est terminé
            URL url = new URL("http", Configuration.BASE_SERVER, 80,
Configuration.BASE_QUERY + Configuration.RACE_TERMINATE_QUERY + this.id);
            url.openStream();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    /**
     * Rempli les champs de base de l'objet : id, name, date, etc. Ne
construit pas les autres champs, comme les listes de coureurs ou de
balises.
     * @param element
     */
}
```

```

    */
    private void setBaseFromElement(Element element) {
        id =
Integer.parseInt(element.getAttribute(Configuration.XML_TAG_RACE_ID));
        name =
element.getElementsByTagName(Configuration.XML_TAG_RACE_NAME).item(0).getTextContent();
        date =
element.getElementsByTagName(Configuration.XML_TAG_RACE_DATE).item(0).getTextContent();

        Element startNode = (Element)
element.getElementsByTagName(Configuration.XML_TAG_RACE_START).item(0);
        int hour =
Integer.parseInt(startNode.getElementsByTagName(Configuration.XML_TAG_RACE_HOUR).item(0).getTextContent());
        int minute =
Integer.parseInt(startNode.getElementsByTagName(Configuration.XML_TAG_RACE_MINUTE).item(0).getTextContent());
        int second =
Integer.parseInt(startNode.getElementsByTagName(Configuration.XML_TAG_RACE_SECOND).item(0).getTextContent());

        String[] dateElements = date.split("/");
        int year = Integer.parseInt(dateElements[2]);
        int month = Integer.parseInt(dateElements[1]);
        int day = Integer.parseInt(dateElements[0]);

        dateTime = new GregorianCalendar(year, month, day, hour, minute,
second);
        System.out.println(dateTime);
    }

    /**
     * Rempli le reste des champs de l'objet en construisant les objets
requis en parsant un fichier XML.
    */
    public void setRemainingFromXML() {
        InputStream is = null;
        try {
            //récupération du fichier XML décrivant la course : on ajoute à
la fin de la requete l'id de la course
            URL url = new URL("http", Configuration.BASE_SERVER, 80,
Configuration.BASE_QUERY + Configuration.RACE_INFO_QUERY + this.id);
            is = url.openStream();
            InputSource isc = new InputSource(is);
            DocumentBuilderFactory dbf =
DocumentBuilderFactory.newInstance();
            DocumentBuilder db = dbf.newDocumentBuilder();
            Document document = db.parse(isc);
            //Si le contenu du fichier XML décrivant la course a changé
depuis le dernier chargement (ou si c'est le premier chargement) :
            if
(!previousXMLContent.equals(document.getDocumentElement().getTextContent())
) {
                previousXMLContent =
document.getDocumentElement().getTextContent();
                //Construction des objets Runner

```

```
        NodeList runnerNodes =
document.getDocumentElement().getElementsByTagName(Configuration.XML_TAG_RU
NNER);
        for (int i = 0; i < runnerNodes.getLength(); i++) {

runnerContestationsMap.put(Factory.getRunnerFromElement((Element)
runnerNodes.item(i)), null);
        }

        //Construction des objets Marker, mais on ne les met pas
dans la map car toutes les balises de la course
//n'ont pas forcément de contestations
        NodeList markerNodes =
document.getDocumentElement().getElementsByTagName(Configuration.XML_TAG_MA
RKER);
        for (int i = 0; i < markerNodes.getLength(); i++) {
Factory.getMarkerFromElement((Element)
markerNodes.item(i));
        }

        //Constructions des objets Contestations et remplissage des
listes
        NodeList contestationMarkerNodes =
document.getDocumentElement().getElementsByTagName(Configuration.XML_TAG_CO
NTESTATION_MARKER);
        //Pour chaque noeud contestationMarker (qui match une
balise et plusieurs contestations) on regarde chaque contestation
        for (int i = 0; i < contestationMarkerNodes.getLength();
i++) {
                String idCurrentMarker = ((Element)
(contestationMarkerNodes.item(i))).getAttribute(Configuration.XML_TAG_CONTE
STATION_MARKER_ID_MARKER);
                NodeList contestationNodes = ((Element)
(contestationMarkerNodes.item(i))).getElementsByTagName(Configuration.XML_T
AG_CONTESTATION);
                //pour chaque contestation
                for (int j = 0; j < contestationNodes.getLength(); j++)
{
                        String idRunner = ((Element)
(contestationNodes.item(j))).getElementsByTagName(Configuration.XML_TAG_CON
TESTATION_RUNNER_NUMBER).item(0).getTextContent();
                        int hour = Integer.parseInt(((Element)
(contestationNodes.item(j))).getElementsByTagName(Configuration.XML_TAG_CON
TESTATION_HOUR).item(0).getTextContent());
                        int minute = Integer.parseInt(((Element)
(contestationNodes.item(j))).getElementsByTagName(Configuration.XML_TAG_CON
TESTATION_MINUTE).item(0).getTextContent());
                        int second = Integer.parseInt(((Element)
(contestationNodes.item(j))).getElementsByTagName(Configuration.XML_TAG_CON
TESTATION_SECOND).item(0).getTextContent());
                        Contestation contestation = new
Contestation(this.id, idCurrentMarker, idRunner,
hour, minute, second);

                        //on récupère le runner ayant l'id du runner ayant
fait la contestation courante
                        Runner runner = Factory.getRunnerById(idRunner);
                        //on récupère la liste des contestations de ce
coureur
```

```

        // if(Configuration.DEBUG_MODE)
System.out.println(" Contestations marker (i) = " + i + ", contestation (j)
= " + j + ",coureur = " + runner);
        ArrayList<Contestation> contestationsList =
runnerContestationsMap.get(runner);
        //s'il a déjà fait des contestations on ajoute la
contestation courante à sa liste
        if (contestationsList != null) {
            contestationsList.add(contestation);
        //sinon on crée la liste en y mettant la
contestation, puis on met la liste dans le HashMap
        } else {
            contestationsList = new ArrayList();
            contestationsList.add(contestation);
            runnerContestationsMap.put(runner,
contestationsList);
        }

        //on fait la même chose avec les balises :
        Marker marker =
Factory.getMarkerById(idCurrentMarker);
        System.out.println("BALISE : " + idCurrentMarker +
" " + marker);
        //si on a pas déjà ajouté la balise à la map on le
fait
        if(!markerContestationsMap.keySet().contains(marker)) {
            markerContestationsMap.put(marker, null);
        }
        //on récupère la liste des contestations de cette
balise
        ArrayList<Contestation> contestationsListMarker =
markerContestationsMap.get(marker);
        //si la balise à déjà des contestations on ajoute
la contestation courante à sa liste
        if (contestationsListMarker != null) {
            contestationsListMarker.add(contestation);
        //sinon on crée la liste en y mettant la
contestation, puis on met la liste dans le HashMap
        } else {
            contestationsListMarker = new ArrayList();
            contestationsListMarker.add(contestation);
            markerContestationsMap.put(marker,
contestationsListMarker);
        }
    }
}

        System.out.println("Fire change :");
        fireChange();
    }
} catch (SAXException ex) {
    ex.printStackTrace();
} catch (IOException ex) {
    ex.printStackTrace();
} catch (ParserConfigurationException ex) {
    ex.printStackTrace();
}
}

public String toString() {

```

```

        return name + " (" + date + ")";
    }

    public HashMap<Runner, ArrayList<Contestation>>
getRunnerContestationsMap() {
        return runnerContestationsMap;
    }
    public HashMap<Marker, ArrayList<Contestation>>
getMarkerContestationsMap() {
        return markerContestationsMap;
    }

    public GregorianCalendar getDateTime() {
        return dateTime;
    }
}

```

Annexe 3 : classe Factory

```

package fr.alexandretoyer.projetm2.model;

import fr.alexandretoyer.projetm2.configuration.Configuration;
import java.util.*;
import org.w3c.dom.*;

/**
 * Gère les coureurs et les balises
 * @author Alex
 */
public class Factory {

    /**
     * Contient les Runner déjà instanciés
     */
    static private HashMap<String, Runner> runners = new HashMap();
    /**
     * Contient les Marker déjà instanciés
     */
    static private HashMap<String, Marker> markers = new HashMap();

    /**
     * Méthode factory statique retournant une instance de Runner
     * @param element Element XML à partir duquel sera retournée une
instance de Runner
     * @return Une instance de Runner correspondant à l'élément XML passé
en argument
     */
    static public Runner getRunnerFromElement(Element element) {
        //on récupère l'id du coureur
        String id = element.getAttribute(Configuration.XML_TAG_RUNNER_ID);
        //si la liste des coureurs possède une entrée ayant cet id on
renvoie le coureur correspondant
        if(runners.containsKey(id)) {
            return runners.get(id);
        }
        //sinon on le construit et on l'ajoute à la liste
        else {

```

```

        String firstname =
element.getElementsByTagName(Configuration.XML_TAG_RUNNER_FIRSTNAME).item(0)
).getTextContent();
        String name =
element.getElementsByTagName(Configuration.XML_TAG_RUNNER_NAME).item(0).get
TextContent();
        Runner newRunner = new Runner(id, firstname, name);
        runners.put(id, newRunner);
        return newRunner;
    }

}

/**
 * Méthode factory statique retournant une instance de Marker
 * @param element Element XML à partir duquel sera retournée une
instance de Marker
 * @return Une instance de Marker correspondant à l'élément XML passé
en argument
 */
static public Marker getMarkerFromElement(Element element) {
    //on récupère l'id de la balise
    String id = element.getAttribute(Configuration.XML_TAG_RUNNER_ID);
    //si la liste des balises possède une entrée ayant cet id on
renvoie la balise correspondante
    if(markers.containsKey(id)) {
        return markers.get(id);
    }
    //sinon on la construit et on l'ajoute à la liste
    else {
        Marker newMarker = new Marker(id);
        markers.put(id, newMarker);
        return newMarker;
    }
}

/**
 * Retourne un coureur existant en cherchant son id
 * @param id
 * @return Un coureur présent dans la liste des coureurs déjà
instanciés
 */
static public Runner getRunnerById(String id) {
    return runners.get(id);
}

static Marker getMarkerById(String idCurrentMarker) {
    return markers.get(idCurrentMarker);
}
}

```

Annexe 4 : Swing : Classe MediaController

```

package fr.alexandretoyer.projetm2.gui;

import fr.alexandretoyer.projetm2.configuration.Configuration;
import
fr.alexandretoyer.projetm2.gui.internationalisablecomponents.IntJLabel;
import fr.alexandretoyer.projetm2.model.*;

```

```

import fr.alexandretoyer.projetm2.model.player.PlayerCommon;
import java.awt.*;
import java.awt.event.*;
import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;
import javax.swing.*;
import javax.media.*;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

/**
 *
 * @author Alex
 */
public class MediaController extends JPanel implements ChangeListener,
MouseListener, ActionListener, InterfaceStoppable {

    JSlider slider;
    JButton bStop;
    JButton bRewind;
    JButton bPlay;
    JButton bForward;
    JLabel time;
    JSlider sliderPasParPas;
    PlayerCommon media;
    boolean playing;
    boolean alreadyStarted = false;
    private long startTime;
    // Le panneau de controle (execution, avance rapide)
    //private Component masterControl = null;
    private GuiVideo frame;
    private IntJLabel pap;
    private Race race;
    /**
     * Liste des threads créés par le controller : à arrêter lors de la
     fermeture de la fenetre
     */
    private ArrayList<Thread> threadManaged;
    private boolean sliderMoving = false;
    private Date dateDebutVideo;

    public MediaController(GuiVideo _frame, PlayerCommon _media, long
startT, Race r, Date _dateDebutVideo) {
        System.out.println("-----\nMediaController :
constructeur\n-----");
        threadManaged = new ArrayList();
        media = _media;
        playing = false;
        frame = _frame;
        startTime = startT;
        race = r;
        dateDebutVideo = _dateDebutVideo;

        setLayout(new BorderLayout(this, BorderLayout.Y_AXIS));

        Box all = new Box(BoxLayout.Y_AXIS);

        //Barre de défilement
        Box boxSlider = new Box(BoxLayout.X_AXIS);
        slider = new JSlider();

```

```
slider.setMinimum(0);
slider.setMaximum(10000);
slider.setValue(0);
slider.setPaintLabels(true);
slider.addMouseListener(this);
boxSlider.add(slider);
//on crée le thread qui bougera le slider selon le temps du player
:
Thread moveSlider = new ThreadMoveSlider(media, slider, this);
moveSlider.start();
threadManaged.add(moveSlider);

//Boutons de controle :
Box boxControls = new Box(BoxLayout.X_AXIS);
boxControls.add(Box.createHorizontalGlue());

// --> stop
ImageIcon iconStop = new
ImageIcon(getClass().getClassLoader().getResource(Configuration.ICON_STOP_B
LUE));
bStop = new JButton("", iconStop);
bStop.addMouseListener(this);
bStop.addActionListener(this);
bStop.setPreferredSize(new Dimension(25, 20));
boxControls.add(bStop);

// --> rewind
ImageIcon iconRewind = new
ImageIcon(getClass().getClassLoader().getResource(Configuration.ICON_REWIND
_BLUE));
bRewind = new JButton("", iconRewind);
bRewind.addMouseListener(this);
bRewind.addActionListener(this);
bRewind.setPreferredSize(new Dimension(35, 20));
boxControls.add(Box.createHorizontalStrut(5));
boxControls.add(bRewind);

// --> play/pause
ImageIcon iconPause = new
ImageIcon(getClass().getClassLoader().getResource(Configuration.ICON_PAUSE_
BLUE));
bPlay = new JButton("", iconPause);
bPlay.addMouseListener(this);
bPlay.addActionListener(this);
bPlay.setPreferredSize(new Dimension(60, 35));
boxControls.add(Box.createHorizontalStrut(5));
boxControls.add(bPlay);

// --> forward
ImageIcon iconForward = new
ImageIcon(getClass().getClassLoader().getResource(Configuration.ICON_FORWAR
D_BLUE));
bForward = new JButton("", iconForward);
bForward.addMouseListener(this);
bForward.addActionListener(this);
bForward.setPreferredSize(new Dimension(35, 20));
boxControls.add(Box.createHorizontalStrut(5));
boxControls.add(bForward);

// --> JLabel avec le temps
```

```

        ImageIcon iconTime = new
ImageIcon(getClass().getClassLoader().getResource(Configuration.ICON_CLOCK)
);
        JLabel clock = new JLabel("", iconTime, JLabel.LEFT);
        time = new JLabel("00:00:00");//, 6);
        time.setPreferredSize(new Dimension(50, 50));
        time.setFont(new Font("arial", Font.PLAIN, 11));
        JPanel timePanel = new JPanel();
        timePanel.setLayout(new FlowLayout());
        timePanel.add(clock);
        timePanel.add(time);
        boxControls.add(Box.createHorizontalStrut(3));
        boxControls.add(timePanel);
        //on lance le thread qui va aller modifier le temps dans le field
toutes les secondes :
        Thread changeTimeField = new ThreadChangeTimeField(time, media,
race, dateDebutVideo);
        changeTimeField.start();
        //on l'ajoute à la liste des threads controlés
        threadManaged.add(changeTimeField);

        //Slider pour image par image
        Box boxPasParPas = Box.createHorizontalBox();
        boxPasParPas.add(Box.createHorizontalGlue());
        sliderPasParPas = new JSlider();
        sliderPasParPas.setPreferredSize(new Dimension(50, 25));
        sliderPasParPas.setPaintLabels(true);

        pap = new
IntJLabel(Configuration.KEY_APPLICATION_VIDEO_STEP_BY_STEP,
frame.getGui().getMessage(Configuration.KEY_APPLICATION_VIDEO_STEP_BY_STEP)
);
        frame.getGui().addInternationalisableComponent(pap);
        pap.setForeground(new Color(59, 105, 159));
        boxPasParPas.add(pap);
        boxPasParPas.add(sliderPasParPas);

        //Ajouts des différents éléments à la boîte all
        all.add(Box.createVerticalStrut(10));
        all.add(boxSlider);
        all.add(Box.createVerticalStrut(10));
        all.add(boxControls);
        all.add(Box.createVerticalStrut(10));
        all.add(boxPasParPas);
        all.add(Box.createVerticalStrut(10));

        all.add(Box.createVerticalGlue());

        add(all);

        slider.addChangeListener(this);

        System.out.println("MediaController 1 : AVANT : " +
media.getTime().getNanoseconds());
        System.out.println("MediaController 2 : StartTime en secondes : " +
startTime / 1000);
        System.out.println("MediaController 3 : Durée totale en secondes :
" + media.getDuration().getSeconds());
        Time newTime = new Time(startTime * 1000000);

```

```

        media.setTime(newTime);
        System.out.println("MediaController 4 : newTime : " +
newTime.getSeconds());
        System.out.println("MediaController 5 : APRES : " +
media.getTime().getSeconds());

        changePlaying(true);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == bStop) {
            changePlaying(false);
            media.setTime(new Time(0));
        }
        if (e.getSource() == bRewind) {
            media.setRate(1);
            System.out.println("rewind " + media.setRate(-3.0f));
            changePlaying(true);
        }
        if (e.getSource() == bPlay) {
            if (playing) {
                changePlaying(false);
            } else {
                changePlaying(true);
            }
        }
        if (e.getSource() == bForward) {
            switch ((int) media.getRate()) {
                case 1:
                    media.setRate(3);
                    break;
                case 3:
                    media.setRate(10);
                    break;
                case 10:
                    media.setRate(1);
                    break;
            }
            System.out.println(media.getRate());
            changePlaying(true);
        }
    }

    public void stateChanged(ChangeEvent e) {
        if (e.getSource() == slider && sliderMoving) {
            double ratio = ((double) (slider.getValue() -
slider.getMinimum())) / (slider.getMaximum() - slider.getMinimum());
            System.out.println("_____");
            System.out.println("slider.getValue() = " + slider.getValue() +
" slider.getMinimum() = " + slider.getMinimum() + " slider.getMaximum() = "
+ slider.getMaximum() + "--> ratio : " + ratio);
            double newTime = (long) (media.getDuration().getSeconds() *
ratio);
            System.out.println("new time : " + newTime);
            media.setTime(new Time( (long) (newTime * 1000000000) ));
            System.out.println("Nouveau temps = " +
media.getTime().getSeconds());
            System.out.println("_____");
        }
    }
}

```

```
    }

}

boolean isSliderMoving() {
    return sliderMoving;
}

private void changePlaying(boolean _playing) {
    System.out.println("ChangePlaying : playing : " + _playing);
    ImageIcon icon;
    if (_playing) {
        icon = new
ImageIcon(getClass().getClassLoader().getResource(Configuration.ICON_PAUSE_
BLUE));
        media.start();
    } else {
        icon = new
ImageIcon(getClass().getClassLoader().getResource(Configuration.ICON_PLAY_B
LUE));
        media.stop();
    }
    bPlay.setIcon(icon);
    playing = _playing;
}

public void mouseEntered(MouseEvent e) {
    ImageIcon icon;
    if (e.getSource() == bStop) {
        icon = new
ImageIcon(getClass().getClassLoader().getResource(Configuration.ICON_STOP)
);
        bStop.setIcon(icon);
    }
    if (e.getSource() == bRewind) {
        icon = new
ImageIcon(getClass().getClassLoader().getResource(Configuration.ICON_REWIND
));
        bRewind.setIcon(icon);
    }
    if (e.getSource() == bPlay) {
        //si le media est en lecture on affiche l'icone pause
        if (playing) {
            icon = new
ImageIcon(getClass().getClassLoader().getResource(Configuration.ICON_PAUSE)
);
        } //sinon l'icone lecture
        else {
            icon = new
ImageIcon(getClass().getClassLoader().getResource(Configuration.ICON_PLAY)
);
        }
        bPlay.setIcon(icon);
    }
    if (e.getSource() == bForward) {
        icon = new
ImageIcon(getClass().getClassLoader().getResource(Configuration.ICON_FORWAR
D));
        bForward.setIcon(icon);
    }
}
}
```

```
    public void mouseExited(MouseEvent e) {
        ImageIcon icon;
        if (e.getSource() == bStop) {
            icon = new
ImageIcon(getClass().getClassLoader().getResource(Configuration.ICON_STOP_B
LUE));
            bStop.setIcon(icon);
        }
        if (e.getSource() == bRewind) {
            icon = new
ImageIcon(getClass().getClassLoader().getResource(Configuration.ICON_REWIND
_BLUE));
            bRewind.setIcon(icon);
        }
        if (e.getSource() == bPlay) {
            //si le media est en lecture on affiche l'icone pause
            if (playing) {
                icon = new
ImageIcon(getClass().getClassLoader().getResource(Configuration.ICON_PAUSE_
BLUE));
            } //sinon l'icone play
            else {
                icon = new
ImageIcon(getClass().getClassLoader().getResource(Configuration.ICON_PLAY_B
LUE));
            }
            bPlay.setIcon(icon);
        }
        if (e.getSource() == bForward) {
            icon = new
ImageIcon(getClass().getClassLoader().getResource(Configuration.ICON_FORWAR
D_BLUE));
            bForward.setIcon(icon);
        }
    }

    public void mouseClicked(MouseEvent e) {
    }

    public void mousePressed(MouseEvent e) {
        if (e.getSource() == slider) {
            sliderMoving = true;
        }
    }

    public void mouseReleased(MouseEvent e) {
        if (e.getSource() == slider) {
            sliderMoving = false;
        }
    }

    public void stop() {
        Iterator it = threadManaged.iterator();
        while (it.hasNext()) {
            ((InterfaceThread) it.next()).stopThread();
        }
    }
}
```