

Activité d'apprentissage S-MATH-789 à Mons

# Projet de Modélisation et d'Implémentation *Simulateur d'une radio numérique (DAB+)*

Enseignants: Tom Mens, Gauvain Devillez

Année Académique 2019-2020  
Faculté des Sciences, Université de Mons

Dernière mise à jour: 4 février 2020

## Résumé

L'activité d'apprentissage (AA) S-MATH-789 « Projet de modélisation et d'implémentation » consiste en un travail de modélisation et de programmation logicielle réalisé en groupe d'un ou de deux étudiant(s). Par défaut, les groupes sont constitués d'un seul étudiant.

Le projet consiste en la réalisation d'un logiciel contrôlé par un statechart et comprenant une interface utilisateur graphique (GUI), pour l'énoncé décrit dans la section 1. Le travail est constitué de deux phases : la phase de *modélisation* comptant pour un tiers de la note de l'AA, et la phase d'*implémentation* comptant pour deux tiers de la note de l'AA. La modélisation doit être réalisée avec le langage de modélisation UML. L'implémentation doit être réalisée en Java et doit utiliser des design patterns (dont au minimum le *State design pattern* et le *Singleton design pattern*).

L'utilisation des tests unitaires avec JUnit est **obligatoire** pour vérifier que l'application développée correspond aux besoins énoncés et ne contient pas de bogue. Nous suivons le principe "*if it isn't tested, it doesn't exist*".

## Table des matières

<b>1</b>	<b>Cahier des charges</b>	<b>3</b>
1.1	Functionalities . . . . .	4
1.2	Required features . . . . .	4
1.3	Optional features . . . . .	5
<b>2</b>	<b>Modélisation</b>	<b>6</b>
2.1	Use case model . . . . .	6
2.2	Interaction overview diagram . . . . .	6
2.3	Statechart model . . . . .	6
2.4	Diagramme de classes . . . . .	6
2.5	Diagramme de séquences . . . . .	7
<b>3</b>	<b>Implémentation</b>	<b>8</b>
3.1	Langages et bibliothèques imposées . . . . .	8
3.2	Bibliothèques recommandées . . . . .	8
3.3	Outils . . . . .	9
<b>4</b>	<b>Livrables et échéances</b>	<b>10</b>
4.1	Critères de recevabilité . . . . .	10
4.2	Modélisation . . . . .	10
4.3	Implémentation . . . . .	11

# 1 Cahier des charges

*L'énoncé (en anglais) qui suit est volontairement lacunaire sur le moyen de concevoir le logiciel. À vous de réaliser une modélisation en UML et une implémentation en Java qui soient respectueuses des principes du cours. N'hésitez pas à demander des précisions aux enseignants qui, en tant que « clients » demandeurs de l'application, pourront éclaircir certains points restés ambigus. Si vous rencontrez des incohérences dans le cahier des charges, veuillez en informer les enseignants. L'énoncé proposé ci-dessous sera un requis minimal pour le travail. Toute réalisation d'une fonctionnalité supplémentaire, approuvée par les enseignants, sera considérée en bonus.*

Background : Digital audio broadcasting (DAB) is a digital radio standard for broadcasting digital audio radio services. The DAB+ standard, released in July 2007 is now used in many European countries, including Belgium, and supersedes the older DAB standard. Figure 1 provides some photos of digital radios, with varying functionalities and user interfaces.



FIGURE 1 – Examples of DAB+ radio players.

## 1.1 Functionalities

The goal of this project is to design and develop a configurable simulator for a *product family* of digital radio players. The focus of the project is on the functionality and visual user interface of a configurable radio player. The actual supported standard (DAB+) is of little importance. The configurable simulator should have the form of a software application consisting of a user interface whose execution, after configuration, **MUST be controlled by a statechart that is running behind the scenes**.


Each radio player belonging to the product family will contain required features and optional features. Upon launching, the application will present a **configuration screen** allowing the user to configure a specific DAB+ radio player by selecting the optional features that should be included. Once configured, the application will launch a **simulation screen** presenting the actual simulation of the radio player containing all required features as well as all selected optional features. During the simulation (i.e., without needing to quit the application) it should always be possible to return to the configuration screen to deactivate some of the optional features and activate other ones.

The simulator should also provide a **failure panel** allowing to simulate breakdowns or failures in certain components or functionalities provided by the radio player, *while the application is running*. The application should be robust in presence of such failures. An *incomplete* list of examples of failures is given below :

- The radio player is **not able to tune to a selected radio station**, either due to an external problem because no radio signal is received, or due to an internal problem with the built-in tuner. If the radio is able to receive both DAB+ and FM signals, and the **radio station is not available through one of these signals**, then the radio player will **automatically try to switch to an alternative signal** (e.g., from DAB+ to FM or conversely, or from a particular frequency to another one **corresponding to the same radio station**).
- One of the radio buttons or knobs is not working.
- **The built-in speaker is not producing any sound.**
- The displayed **time or date is out of sync**. In that case, some **functionality** (e.g. time/date/alarm) may **not be available**.
- The **display is not working**. In that case, most of the functionalities of the radio can still be used normally, with the exception that no information will be displayed on the screen.

## 1.2 Required features

The following **required features** must be present for all digital radios within the product family :

**Activation** A power button  allows to switch the radio player on or off. This button is accompanied by an indicator LED light to visualise the power status of the radio (on or off). Even in the off state, the radio player will still keep track of the time, and the optional alarm functionality, if present, will be able to automatically turn on the radio.

**Station selection** A turn knob allows the user to switch from one available radio station to the next one for which it is receiving a signal of sufficient quality.

**Built-in speaker** A built-in speaker is integrated within each radio player. It comes with a volume turn button to increase or decrease the volume of the sound. The volume can range from 0 to 20 in discrete steps.

**Display** A digital radio contains a LED screen that can display different types of information. The type of information that will be displayed may depend on the size of the screen. Some radios will only have a textual display (allowing to display alphanumeric characters), of 1 or multiple lines. Other radios will have a colour display (allowing to display images or icons as well). Depending on the type of radio, screen type and display resolution, the following information can be displayed :

- The name of the radio station.
- The channel number of the radio station.
- Textual information emitted by the radio station, such as the name of current radio programme, or the song currently being played and the artist name.
- Images emitted by the radio station, such as the logo of the radio station, or a photo of the current radio programme or its presenter, or a picture of the current music being played.
- The current time.

— Any other information that is relevant for implementing other required or optional functionalities. The user should be able to select which information to be displayed by the screen.

**Time and date** Current time (and optionally date, depending on the size and type of the display) will be displayed continuously when the radio player is inactive. However, to preserve energy, the user can change the settings so that nothing is displayed on the screen when the radio player is inactive. When the radio player is active, one of the possible display modes will allow to display time (and date). The user should also be able to change the time and date.

**Station presets** Each radio has a fixed amount of memory that can be used to store the available radio stations. Some of these stored radio stations can be accessed immediately through a limited number of preset buttons (the number can vary depending on the type of radio, but should always be between 3 and 10). Pushing the button (when the radio is active) will select the radio station and start playing. If a given radio station is playing, and a preset button is kept pressed during at least 3 seconds, that radio station will be assigned to that preset button.

### 1.3 Optional features

The following **optional features** are not present in all radio players of the product family. Their activation may be selected in the configuration screen of the application to enable their functionality. Activating a particular functionality may impact the user interface and require additional visual components (e.g., extra buttons).

**Autotune functionality** A button allows the user to autotune the radio stations. Autotune will automatically search for all available radio stations, and will store them in its internal memory in the order they were found, until all memory locations are filled or until no more radio stations are found. The user can then browse through any of these stations and decide to play that radio station, or to store it in one of the radio's station presets (see above). There is also the possibility to automatically store all memorised radio stations in the preset numbers. Note however that, in practice, more stations are available than can be stored in the preset station numbers.

**Secondary speaker** Some radios have two built-in speakers, allowing to output stereo sound. In that case, an additional stereo balance turn button should be present to control the output volume between the left and right speaker. The balance should range from L :-10 to R :+10 in discrete steps.

**External audio input** Some radios have an external input source, allowing to receive sound from an external device (e.g., a smartphone or an MP3 player). External input can be received either through a USB input connector, an audio input jack, or wirelessly using Bluetooth.

**External audio output** Some radios have an external output source, allowing to redirect the sound to external speakers. This can be either via Bluetooth, or via an audio-jack (for headphones or to connect to external speakers). Note that Bluetooth cannot be used simultaneously for input and output.

**Automatic date and time** Some radios allow to automatically receive the current date and time, encoded through the received radio signal. If the locally stored time and date is out of sync, the radio will synchronise automatically whenever a radio station is selected and a radio signal is received.

**Alarm functionality** The radio player can be used as a kind of digital clock. An alarm can be activated at a particular time, or deactivated. When the alarm is activated, the digital radio will automatically turn on (if it was turned off) at the programmed time of the alarm, and either start ringing the alarm sound, or start playing the radio station that was selected upon setting the alarm. (If, however, that radio station is not available at the time of the alarm, the normal alarm sound will be used instead.)

**Breaking news** Some radio stations emit news messages or traffic messages from time to time. The digital radio should be able to display such messages if the user desires to do so. Only messages of the currently active radio station will be displayed.

**FM radio** Some digital radios do not only support the DAB+ standard, but also support the analog FM standard. These radios will have a digital radio mode and an analog radio mode, that are very similar, and each allow to store a limited number of radio stations. The main difference is that FM radio stations only emit a very limited amount of digital information (essentially, text messages of limited size).<sup>1</sup>

---

1. Digital data can be encoded and transmitted via FM by shifting the carrier's frequency among a predefined set of frequencies representing

## 2 Modélisation

Lors de la phase de modélisation, vous devez réaliser une **maquette de l'interface graphique** (anglais : *user interface mockup*) pour l'application. Cette maquette permet de montrer à l'utilisateur final à quoi ressemblera le logiciel, avant d'avoir réellement développé les fonctionnalités du logiciel. Les maquettes d'interface utilisateur logicielle peuvent aller de très simples mises en page d'écran dessinées à la main, en passant par des bitmaps réalistes, jusqu'à des interfaces utilisateur semi-fonctionnelles développées dans un outil de développement logiciel, ou en utilisant des logiciels dédiés pour la création d'un UI mockup.

Vous devez également réaliser un **modèle de conception** en utilisant le langage de modélisation *UML 2.5* ou supérieur. Ce modèle doit décrire l'architecture, la structure et le comportement de l'application à réaliser. Le **modèle de conception** doit **au moins** contenir des spécifications des cas d'utilisation (pour définir l'interaction avec l'utilisateur), des diagrammes de classes (pour décrire la structure), des statecharts (machines à états comportementales), des diagrammes de séquence (pour modéliser des scénarios typiques d'interaction entre les différents composants), et un diagramme d'activités (pour modéliser la vue d'ensemble de l'interaction entre les différents cas d'utilisation). L'utilisation de tout autre type de diagrammes UML pour compléter la modélisation de l'application sera considérée en bonus.

### 2.1 Use case model

Le modèle de cas d'utilisation est constitué d'un diagramme des cas d'utilisation. Tous les cas d'utilisation dans ce diagramme doivent obligatoirement être accompagnés d'une **spécification semi-formelle du cas d'utilisation**.

### 2.2 Interaction overview diagram

Un diagramme d'interaction peut être utilisé pour modéliser la vue d'ensemble du comportement de l'application. Ce diagramme précisera dans quel ordre et sous quelles conditions les différents cas d'utilisation du use case model seront exécutés.

### 2.3 Statechart model

Cette partie de la modélisation est **la plus importante**, car elle correspond au comportement principal à réaliser par l'application. Les statecharts modélisés formeront le noyau fonctionnel de l'application à réaliser lors de la phase d'implémentation (cf. Section 3).

Le statechart à fournir doit tenir compte du fait qu'il s'agit d'une famille de plusieurs "digital radio player". Le comportement de chacun peut varier, selon les features qui seront activés ou désactivés.

Il est obligatoire de modéliser les **statecharts** avec l'outil Yakindu Statechart Tools <sup>2</sup>. Cet outil fournit un générateur de code Java qui peut être utilisé (mais l'utilisation de ce générateur n'est pas obligatoire). Le(s) statechart(s) fourni(s) doit(vent) être exécutable(s) avec le simulateur fourni par Yakindu Statechart Tools.

### 2.4 Diagramme de classes

Le diagramme de classes doit représenter tous les composants du système (configurateur, simulateur et failure panel) à réaliser, ainsi que tous les concepts nécessaires pour réaliser les fonctionnalités. Le diagramme de classe servira comme base principale pour la phase d'implémentation. L'implémentation en Java doit être la plus proche possible du diagramme de classes mais certaines différences peuvent être nécessaires. Par exemple, à cause de contraintes techniques ou d'erreurs dans la modélisation.

La modélisation doit respecter les principes orientés objets. Par exemple, il faut suivre une approche modulaire (en utilisant une bonne structuration en packages, et en séparant l'interface utilisateur de la logique métier et de l'accès

---

digits – for example one frequency can represent a binary 1 and a second can represent binary 0. This modulation technique is known as frequency-shift keying (FSK).

2. Téléchargeable sur [www.statecharts.org](http://www.statecharts.org)

aux données), éviter des god class et data class, et distribuer la responsabilité entre les différentes classes. Il faut aussi utiliser la spécialisation, les classes abstraites et les interfaces judicieusement. Les classes doivent préciser leurs opérations principales. Il n'est pas nécessaire de préciser les setter et getter des attributs. Les associations, compositions et agrégations doivent être précisées avec leur multiplicité.

## **2.5 Diagramme de séquences**

Des diagrammes de séquences doivent être utilisés pour modéliser les interactions entre les différents composants du distributeur, ainsi que pour formaliser le comportement décrit informellement par les cas d'utilisation.

Vous devez utiliser les “fragments combinés” dans les diagrammes de séquence pour modéliser les scénarios des cas problématiques, ainsi que du comportement nécessitant une interaction non triviale entre plusieurs objets.

## 3 Implémentation

### 3.1 Langages et bibliothèques imposées

**Java** Votre projet logiciel sera implémenté en utilisant le langage de programmation Java 8 ou supérieur.

L'utilisation de **design patterns** dans votre code est **obligatoire**. Vous devez respecter le principe de la **programmation défensive**, et utiliser le système de **gestion d'exceptions** de Java. Pensez à gérer (et à tester !) les contraintes de sécurité ainsi que les cas problématiques.

Lors de l'implémentation des statecharts en Java, le *state design pattern* doit être utilisé. Alternativement, vous pouvez utiliser le générateur du code pour les statecharts, fourni par Yakindu Statechart Tools, pour autant que le code obtenu ait le comportement attendu.

**Yakindu** Pour les modèles de **statechart**, l'utilisation de Yakindu Statechart Tools<sup>3</sup> est **obligatoire**. Cet outil permet de spécifier, vérifier, simuler et de générer du code source pour des statecharts.

**JUnit** L'utilisation des **tests unitaires** est **obligatoire**. Pour réaliser les tests unitaires en Java, l'utilisation de JUnit 5 est **imposée**. Vous pouvez le trouver sur son site officiel<sup>4</sup>.

Afin d'assurer une bonne qualité de code, vous devez alterner l'écriture des tests et du code, en utilisant l'approche de *développement dirigé par les tests*. Cette approche permet de définir le comportement que votre application doit avoir au terme du projet. Cela permet de plus de situer où en est votre progression. Une autre bonne pratique consiste à écrire des *tests de régression* : écrivez des tests unitaires qui mettent en évidence chaque erreur rencontrée, vous n'aurez ainsi pas à comprendre et résoudre deux fois le même problème.

**maven** L'utilisation de Apache maven<sup>5</sup> est **obligatoire** pour la compilation et l'exécution de votre projet et ses tests unitaires. Vous devrez utiliser cet outil pour gérer vos dépendances (notamment à JUnit et à votre système de journalisation), de sorte qu'un `mvn package` suffise à valider, compiler, tester, et packager votre application depuis un nouvel environnement de travail.

### 3.2 Bibliothèques recommandées

**JavaFX** Pour la réalisation de l'**interface graphique** (GUI) de votre application, l'utilisation de JavaFX est fortement recommandée<sup>6</sup>. De nombreux tutoriels sont disponibles sur l'Internet<sup>7</sup>.

Si vous désirez utiliser une autre bibliothèque ou interface graphique (par exemple Swing<sup>8</sup>), il faut demander l'accord des enseignants au préalable.

**Testing with mock objects** Dans la programmation orientée objet, *mock objects* sont des objets simulés qui imitent le comportement d'objets réels de manière contrôlée, le plus souvent dans le cadre de tests unitaires. La principale raison de créer de tels *mock objects* est de pouvoir tester une unité du système logiciel sans avoir à se soucier des modules dépendants. La fonctionnalité de ces dépendances est "simulée" par les *mock objects*. Ceci est particulièrement important si les fonctions simulées sont difficiles ou longues à obtenir (par exemple parce qu'elles impliquent des calculs complexes) ou si le résultat est non déterministe, ou s'il est trop dangereux de les utiliser (par exemple parce que vous ne voulez pas accéder à une base de données externe en production lors de la phase de test). Plusieurs bibliothèques de *mock objects* existent pour Java, compatible avec le framework de tests unitaires JUnit. Par exemple, EasyMock<sup>9</sup>, Mockito<sup>10</sup>, Powermock. Vous êtes encouragés de l'utiliser lors de vos tests.

**Système de contrôle de versions** Nous vous encourageons fortement à utiliser le système de contrôle de versions distribué Git lors du développement. Un tel système facilitera le travail en groupe et vous permettra d'avoir

---

3. [www.statecharts.org](http://www.statecharts.org)

4. <https://junit.org/junit5/> et <https://github.com/junit-team/junit5>

5. <https://maven.apache.org>

6. <https://openjfx.io>

7. <https://openjfx.io/openjfx-docs/>

8. <http://docs.oracle.com/javase/tutorial/uiswing/> et <http://www.tutorialspoint.com/swing/>

9. <http://easymock.org>

10. <https://site.mockito.org>



des backups réguliers de votre travail, de mieux suivre le progrès de votre travail, de retourner à une version précédente en cas de problème, etc. Afin de vous assurer de la pérennité de votre travail et de la facilité à y accéder, nous vous suggérons de placer une copie de votre dépôt sur une plateforme accessible depuis Internet telle que Bitbucket <sup>11</sup>, GitHub <sup>12</sup> ou GitLab <sup>13</sup>. Cependant, cette copie ne doit être accessible, en lecture comme en modification, qu’aux membres du groupe (et éventuellement les enseignants).

**Système de journalisation** Pour faciliter le débogage, nous encourageons l’utilisation d’un système de journalisation, comme la librairie Apache Log4j (version 2) <sup>14</sup> qui est à la fois simple d’utilisation et très complète.

### 3.3 Outils

**Outils de modélisation** Vous pouvez utiliser l’**outil de modélisation UML** de votre choix. L’UMONS possède une licence académique pour Visual Paradigm. Beaucoup d’autres outils commerciaux sont disponibles en version gratuite sous la forme de stand-alone ou de plugin pour un environnement de développement (e.g. pour Eclipse, NetBeans, ou IntelliJ IDEA). Il existe également plusieurs outils open source de modélisation UML. Pour les modèles de **statechart**, l’utilisation de Yakindu Statechart Tools est obligatoire.

**Outils de développement** Vous pouvez choisir librement votre environnement de développement Java (par exemple Eclipse, NetBeans ou IntelliJ IDEA). Une contrainte d’utilisation **essentielle** est que les enseignants qui évalueront l’application doivent pouvoir compiler et exécuter le logiciel et ses tests en utilisant **maven** à partir de la ligne de commande (et sans avoir à installer un environnement de développement Java quelconque).

**Système d’exploitation** Vous pouvez utiliser n’importe quel système d’exploitation pour réaliser votre travail. La seule contrainte est que les livrables (c.-à-d. le code source, les tests unitaires et l’interface graphique) doivent être indépendants de la plate-forme choisie. Le code produit sera testé sur trois systèmes d’exploitation différents (MacOS X, Linux et Windows).

---

11. <https://bitbucket.org/>

12. <https://github.com>

13. <https://gitlab.com>

14. <http://logging.apache.org/log4j/2.x/>

## 4 Livrables et échéances

Deux livrables doivent être rendus. Le premier livrable concerne la partie **modélisation** et doit être déposé le **vendredi 20 mars 2020** au plus tard. Les enseignants inspecteront et approuveront le premier livrable, ou proposeront des améliorations que vous devez intégrer avant d'entamer la phase d'implémentation. Le deuxième livrable concerne la partie **implémentation** et doit être déposé le **vendredi 22 juin 2020** au plus tard.

Nous vous encourageons à bien planifier votre emploi de temps, surtout si vous avez d'autres projets à rendre, car *aucun délai supplémentaire ne sera accordé*. Si le livrable comprend plusieurs fichiers, ceux-ci seront regroupés dans une archive .zip. Ce sera cette archive qui sera rendue sur Moodle. Si le livrable comprend des documents, ceux-ci doivent être au format pdf.

### 4.1 Critères de recevabilité

Cette check-list reprend l'ensemble des consignes à respecter pour la remise du projet. **Le non-respect de ces critères implique la non-recevabilité du projet!** L'étudiant sera sanctionné par une note de 0/20 pour cette phase de l'activité d'apprentissage.

**Respect des échéances** Le projet doit être rendu en deux phases (une pour la modélisation et une pour l'implémentation).

La date de limite des remises devra être respectée à la lettre. *Aucun délai ne sera accordé, et aucun retard toléré.*

Il vous est conseillé d'uploader des versions préalables à la version définitive (seule la dernière version reçue avant la date limite de remise sera évaluée).

**Format d'archive** Votre travail devra être remis sous forme d'une seule archive dont le nom suivra le format suivant :

1. Pour la partie modélisation : ML-<noms de famille >-modelisation
2. Pour la partie implémentation : ML-<noms de famille>-implementation

**Contenu d'archive** Tous les documents rendus doivent commencer par une page de garde indiquant l'intitulé du rapport, les noms des étudiants, et l'année académique. Votre archive devra obligatoirement contenir tous les éléments demandés pour la phase correspondante. Les noms de tous les membres du groupe doivent figurer en page de garde des rapports et du mode d'emploi. Sur la page de garde des rapports et du manuel figureront également leurs intitulés.

**Absence de plagiat** Conformément au règlement universitaire, le plagiat est considéré comme une faute grave.

Chaque groupe travaillera de manière isolée. Toute collaboration entre groupes ou avec un tiers, et tout soupçon de plagiat (par exemple en copiant du code source d'Internet ou d'ailleurs sans le mentionner ou sans respecter la licence et sans en informer les enseignants) sont interdits.

Un outil automatisé sera utilisé pour vérifier la présence du code dupliqué entre les différents projets rendus, ainsi que la présence des morceaux de code copiés d'Internet ou d'une autre source externe sans mention de son origine ou sans respect de la licence logicielle.

### 4.2 Modélisation

**Livrable.** L'archive contenant le livrable de la phase de modélisation doit contenir trois éléments :

- Un document en format pdf présentant la *maquette de l'interface graphique* qui doit être réaliste et qui doit correspondre aux exigences de l'énoncé. Il ne suffit pas de présenter uniquement des images, le texte accompagnant doit expliquer comment l'interface graphique proposé fonctionne.
- Les fichiers .sct des statecharts, modélisés et exécutables avec l'outil Yakindu Statechart Tools.
- Un document en format pdf contenant le *rapport de modélisation* incluant tous les *diagrammes UML* proposés (y inclus les statecharts), et une *description textuelle* des choix de conception qui ont été pris et des éléments essentiels dans chaque diagramme fourni. Les diagrammes doivent être dessinés avec un outil de modélisation, et doivent être lisibles après impression sur papier en noir et blanc. *Utilisez un fond blanc ou transparent pour tous vos diagrammes et éléments de modélisation.*

Si vous ne parvenez pas à insérer les images de façon lisible dans le rapport, vous pouvez également les joindre à l'archive au format pdf. *Celles-ci doivent cependant quand même se trouver dans le rapport!*

**Exigences de qualité.** Le travail de modélisation sera évalué selon les critères suivants :

1. *Complétude* : La maquette et les diagrammes UML utilisés sont-ils complets ? Couvrent-ils tous les aspects de l'énoncé ? Toutes les exigences (fonctionnelles et non fonctionnelles) sont-elles prises en compte ?
2. *Compréhensibilité* : La maquette de l'interface graphique et les diagrammes sont-ils faciles à comprendre ? Ont-ils le bon niveau de détail ? Pas trop abstrait, pas trop détaillé ?
3. *Exécutabilité* : Les statecharts fournis sont-ils exécutables par le simulateur de Yakindu Statechart Tools, et correspondent-ils au comportement prévu ? *Vous devez faire une simulation de vos statecharts avec Yakindu Statechart Tools afin de vérifier que leur comportement soit correct.*
4. *Style* : Les diagrammes UML sont-ils bien structurés ? Suivent-ils un style de conception orientée objet ? (Par exemple, pour les diagrammes de classes, une bonne utilisation de la généralisation et de l'association entre les classes, l'utilisation des interfaces et des classes abstraites, une description des attributs et des opérations pour chaque classe.)
5. *Exactitude* : Les différents éléments des diagrammes fournis sont-ils utilisés correctement ? Par exemple :
  - (a) Dans le *diagramme de cas d'utilisation*, les acteurs sont-ils correctement définis ? Les notions de généralisation, d'extension et d'inclusion sont-elles correctement mises en œuvre ? Les cas d'utilisation font-ils appel aux points d'extension lorsqu'ils sont nécessaires ? Les conditions d'extension sont-elles présentes ? Y a-t-il une description semi-formelle de chaque cas d'utilisation ?
  - (b) Le *diagramme d'activités* représente-t-il bien la vue d'ensemble des interactions entre les différents cas d'utilisation ?
  - (c) Dans le *diagramme de classes*, les multiplicités sur les associations sont-elles judicieusement utilisées ? L'utilisation de la généralisation, la composition, l'agrégation et l'association est-elle pertinente ? La multiplicité sur les associations est-elle présente et correcte ? Observe-t-on la présence justifiée de certains *design patterns* ?
  - (d) Les *statecharts* sont-ils syntaxiquement et sémantiquement corrects ? Les états, transitions, gardes, événements et actions sont-ils judicieusement utilisés ? Les états modélisés ne sont-ils pas *artificiels* ? Représentent-ils correctement le comportement spécifié dans l'énoncé ? Les transitions représentent-elles fidèlement les différents changements pouvant survenir ? Les états initiaux, finaux et historiques sont-ils correctement utilisés ? Les états composites et concurrents sont-ils correctement utilisés pour améliorer la compréhensibilité du diagramme ?
  - (e) Dans les *diagrammes de séquence*, les opérations appelées correspondent-elles à celles décrites dans le diagramme de classes ? Les objets commencent-ils et finissent-ils leur vie au bon moment ? Les objets communicants entre eux sont-ils connectés ensemble ? Les fragments combinés sont-ils utilisés correctement pour représenter des boucles, des conditions, des exceptions, du parallélisme ?
  - (f) Utilise-t-on les bonnes conventions de nommage ? (Par exemple, évitez l'utilisation du pluriel dans les noms des classes, ne mélangez pas le français et l'anglais, ...)
6. *Cohérence* : Les diagrammes UML sont-ils syntaxiquement et sémantiquement cohérents ? N'y a-t-il pas d'incohérences : (i) dans les diagrammes ; (ii) entre les différents diagrammes ? Les activités dans le diagramme d'activités correspondent-elles aux cas d'utilisation du diagramme de cas d'utilisation ? Les actions dans le diagramme d'états correspondent-elles aux opérations dans le diagramme de classes ? Les événements dans le diagramme d'états correspondent-ils aux événements reçus de l'interface graphique ?

### 4.3 Implémentation

**Livrable.** L'archive contenant le livrable de la phase d'implémentation doit contenir :

- Une version complète du code source, des tests unitaires, et de l'interface graphique. Le code et les tests doivent être compilables et exécutables avec **maven** (à partir de la ligne de commande) sur n'importe quel système d'exploitation.
- Un fichier **.jar** auto-exécutable.

- Un document en format pdf contenant le *rapport d'implémentation*, justifiant les choix d'implémentation, les différences par rapport à la modélisation, la présence des design patterns, et les problèmes connus.
- Un lien URL vers **une vidéo** (durée recommandée : entre 3 et 10 minutes) qui sert comme *mode d'emploi*, démontrant les fonctionnalités de l'application. L'audio enregistré dans la vidéo devrait correspondre à une explication orale des fonctionnalités de l'application.

**Exigences de qualité.** Nous exigeons une bonne qualité de code. Le code source en Java ne peut pas contenir d'erreurs de syntaxe ou de compilation. L'exécution du code ne peut pas donner lieu à des échecs ou erreurs. Plus précisément, l'implémentation sera évaluée selon les critères suivants :

- *Tests* : La présence des tests est obligatoire : “*If it isn't tested, it doesn't exist!*” Les tests doivent vérifier si le code correspond aux exigences de l'énoncé. La suite de tests doit être exécutable en une seule fois. L'exécution de la suite de tests ne peut pas donner lieu à des échecs ou des erreurs. Les tests doivent suffisamment couvrir le code développé. Plusieurs scénarios d'utilisation doivent être testés.
- *Complétude* : Toutes les fonctionnalités spécifiées dans l'énoncé doivent être implémentées.
- *Conformité* : L'interface graphique de l'application doit être conforme à la maquette de l'interface utilisateur proposée dans le premier livrable. La structure du code source doit être conforme aux modèles UML proposés dans le premier livrable. Chaque écart entre les modèles et le code source doit être justifié dans le rapport d'implémentation.
- *Style* : Le programme doit suivre les bonnes pratiques de *programmation orientée objet*, en utilisant le mécanisme de typage, l'héritage, le polymorphisme, la liaison tardive, les mécanismes d'abstraction (interfaces et classes abstraites), et l'encapsulation des données. À tout moment, il faut éviter un style procédural avec des méthodes complexes et beaucoup d'instructions conditionnelles.  
Les design patterns doivent être utilisés.
- *Exactitude* : Le programme doit fonctionner correctement dans des circonstances normales.
- *Fiabilité* : Le programme ne doit pas échouer dans des circonstances exceptionnelles (p.e. données erronées, format de données incorrect, problème de réseau, problème de sécurité,...) Afin de réduire les erreurs lors de l'exécution du programme, le programme doit utiliser le mécanisme de gestion d'exceptions.
- *Convivialité* : Le programme doit être facile à utiliser, convivial, fluide et intuitif.
- *Indépendance de la plate-forme* : Le code produit doit être indépendant du système d'exploitation. Le code produit sera testé sur trois systèmes d'exploitation différents (MacOS X, Linux et Windows). Une attention particulière doit être apportée aux problèmes d'encodage des caractères qui rendent les accents illisibles sur certains systèmes d'exploitation. Un autre problème récurrent est l'utilisation des chemins représentant des fichiers : Windows utilise une barre oblique inversée (backslash) tandis que les systèmes dérivés d'Unix utilisent une barre oblique (slash). La constante `File.separator` donne une représentation abstraite du caractère de séparation. Un autre problème récurrent est la façon différente de gérer les retours à la ligne.