

Activité d'apprentissage S-MATH-789

# Projet de Modélisation et d'Implémentation “Simulator of a Cassette Deck”

Enseignants: Tom Mens

Année Académique 2018-2019  
Faculté des Sciences, Université de Mons

Dernière mise à jour: 31 octobre 2018

## Résumé

L'activité d'apprentissage (AA) S-INFO-852 « Projet de modélisation logicielle » 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. Ce travail compte pour **50%** de la note finale de l'unité d'enseignement (UE) « Modélisation logicielle ». **Un projet non rendu implique automatiquement un échec de l'UE.**

Le projet consiste en la réalisation d'un logiciel comprenant une interface utilisateur graphique 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 2.5 ou supérieur. L'implémentation doit être réalisée en Java 8 ou supérieur 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 4.6** ou supérieur est obligatoire pour vérifier que l'application développée correspond aux besoins énoncés et ne contient pas de bogue.

## Table des matières

<b>1</b>	<b>Cahier des charges</b>	<b>3</b>
1.1	Functionalities . . . . .	4
1.2	Product Family Features . . . . .	5
1.3	Graphical User Interface . . . . .	5
<b>2</b>	<b>Modélisation</b>	<b>6</b>
2.1	Use case model . . . . .	6
2.2	Interaction overview diagram . . . . .	7
2.3	Statechart model . . . . .	7
2.4	Diagramme de classes . . . . .	8
2.5	Diagramme de séquences . . . . .	8
<b>3</b>	<b>Implémentation</b>	<b>9</b>
3.1	Bibliothèques recommandées et imposées . . . . .	9
3.2	Outils . . . . .	9
<b>4</b>	<b>Livrables et échéances</b>	<b>11</b>
4.1	Critères de recevabilité . . . . .	11
4.2	Modélisation . . . . .	11
4.3	Implémentation . . . . .	12

# 1 Cahier des charges

*L'énoncé 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.*

The goal of this projet is to design and develop a product family of “cassette deck” simulators. Figure 1 provides some photos of such devices, with varying functionality. The configurable simulator should have the form of a software application consisting of a visual user interface that is controlled by a statechart running behind the scenes.



A single cassette deck player and recorder.



An auto-reverse single cassette deck player and recorder.










A dual deck cassette player and recorder.

FIGURE 1 – Examples of cassette decks.



## 1.1 Functionalities

An audio “cassette” is a magnetic tape that is wound around 2 wheels. While playing, the tape unwinds from the first wheel and winds on the other wheel. Cassettes may have different capacities (typically, 45, 60, 90 and 120 minutes).




The behaviour of a basic cassette deck is quite simple. The device can be turned on or off using a power button . An eject button  allows to open and close the cassette holder to insert or remove a cassette from the player. If a cassette is inserted and the cassette holder is closed, one can use buttons to play , pause  or stop  the music recorded on the cassette. Playing the music is achieved technically by placing a magnetic head on the tape to interpret the magnetic signals stored on it. The tape can be wound or unwound at high speed by using a fast forward button  or fast rewind button . While doing this, the magnetic head needs to be disengaged to allow the tape to move faster without breaking it.

A cassette deck typically provides a visual counter (a positive integer value) that indicates the relative position of the magnetic tape. This counter can be manually reset to 0 using a specific button, or it will be automatically reset to 0 whenever the tape reaches the end. The counter should automatically increase its value when the tape is playing or fast forwarding, and decrease its value when the tape is rewinding.

Some cassette decks may have a built-in speaker, allowing to play the music directly. Other devices simply have one or more output channels to some external device that is capable of playing audio. In presence of a built-in speaker, the cassette deck will feature a turn knob to adjust the audio volume (ranging from 0 to 10 in discrete steps), as well as a turn knob to adjust the stereo balance (ranging from L :-10 to R :+10 in discrete steps) between the Left and the Right speakers.

The cassette deck has sensors to detect if the beginning or the end of the magnetic band has been reached. When the tape is playing or fast forwarding, it will stop automatically when the end of the tape is reached. Similarly, when the tape is rewinding, it will stop automatically when the beginning of the tape is reached. More sophisticated cassette decks may also have the possibility to forward to the next or rewind to the previous song using  and  buttons. (This is achieved by detecting “pauses” of at least 2 seconds on the magnetic type, which will be interpreted as the end of a song.)

Since a magnetic tape has two sides, it is technically possible to play music stored on both sides of the cassette. Simple cassette players only have one magnetic head, so the user needs to manually turn around the cassette to be able to play music on the other side of the tape. More sophisticated players provide “auto reverse” functionality have two magnetic heads (one for each side), allowing the cassette deck to play on both sides of the cassette without the user having to manually remove, flip, and re-insert the cassette. This functionality also allows to play music “ad infinitum” : whenever the sensor detects that the end of the magnetic tape is reached, the first magnetic head disengages, the second head engages, and the tape starts playing in the other direction to play the audio recorded on the reverse side of the tape.

Cassette decks may have “recording” functionality, allowing not only to play music, but also to record music. This recording can happen either through a built-in microphone, or through some other input audio channel. In presence of a built-in microphone, the cassette deck will feature a turn button to adjust the volume of the recorded audio, as well as a turn button to adjust the stereo balance of the recorded audio.<sup>1</sup> A specific record button  allows to start recording. To pause or stop recording the pause  or stop button  can be used. While recording (using a specific record button), it is not possible to play, fast rewind or fast forward the cassette. If the end of the tape is reached, recording stops automatically.<sup>2</sup>

Double cassette decks can be seen as a combination of two single cassette decks combined in one device. In addition to all the functionality offered by a single cassette deck, they provide the following additional functionalities :








- Audio can be played simultaneously from both cassettes. A specific slider button allows to specify the volume from either side.
- Audio can be copied from one cassette to the other, by playing music on one cassette and recording music on the other.

1. These turn buttons will be the same as for playing audio through the built-in speaker if the cassette deck provides this feature.




2. With the exception of an “autoreverse” deck that may allow to record on both sides of the tape in one go.

## 1.2 Product Family Features

Any cassette deck should contain at least the following **required features** :

- A power button  to switch the cassette deck on or off.
- A cassette holder that can be opened or closed with the  button to insert and remove a cassette.
- At least one output audio channel to be able to send the audio stored on the cassette to some audio output device.
- A series of buttons to control the cassette :     
- A counter of the relative position of the tape.
- Indicator LED lights to visualise the power status (on or off), the current direction of the tape (forward or reverse), the activity status (e.g., playing, recording, forwarding, ...)

In addition to these required features, the application should allow the user to configure cassette decks allowing any possible combination of the following **optional features** :

- Built-in stereo speakers : This is an output audio channel that is integrated within the cassette deck itself. The speakers can be used to play stereo audio, and can be controlled using volume and balance turn buttons.
- Auto-reverse functionality : A specific button allows to activate this functionality, and an indicator light visualises if the functionality is activated.
- Song detection : This requires specific buttons  and  to forward (resp. rewind) the tape to the next (resp. previous) song.
- Recording functionality : This requires at least one input audio channel, to be able to receive an audio signal to be recorded. It also requires a specific button  to start recording.
- Built-in microphone : this input audio channel is only available if the recording functionality is enabled. It allows one to record the audio received by speaking or singing in the microphone.
- Double-deck functionality : This is basically a cassette deck with two separate cassette holders. As a consequence, some of the cassette controller buttons and indicators need to be provided for each cassette holder.

## 1.3 Graphical User Interface

The application should contain a first **configuration screen** to configure the cassette deck by selecting the optional features one would like to enable. Once configured, the application should launch a second **simulation screen** showing the actual simulation of the cassette deck. During the simulation, it should always be possible to return to the configuration screen to change the available features, in order to simulate another cassette deck, without needing to quit the application.

The simulator should also provide a **failure panel** allowing to disable and re-enable certain functionalities. This allows one to simulate breakdowns or failures of certain components, while the application is running. A non-exhaustive list of examples of failures is given below :

- The cassette holder cannot open or close.
- The cassette is stuck or its tape is broken.
- The recording button cannot be pressed.

## 2 Modélisation

Lors de la phase de modélisation, vous devriez réaliser une **maquette de l'interface graphique** qui sera proposée pour l'application.

Vous devriez également réaliser un **modèle de conception** en utilisant le langage de modélisation *UML* (version 2.5 ou supérieur). 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.

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 Eclipse ou d'autres environnements de développement. Il existe également plusieurs outils open source de modélisation UML. Pour les modèles de **statechart**, l'utilisation de **Yakindu Statechart Tools** est obligatoire. Cet outil est un plugin pour Eclipse permettant de spécifier, vérifier, simuler et de générer du code source pour des statecharts.

### 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**.

Un exemple est illustré ci-dessous pour l'enregistrement de musique d'une source externe sur une cassette (en supposant un "cassette deck with recording functionality") :

---

**Use case name :** Record Music

**Summary :** User records music from external audio input device on cassette.

**Actors :** User (primary), Audio Input Device (secondary)

**Assumptions :** Cassette deck is connected to audio input device. Both devices and their connection are working correctly.

**Preconditions :** Cassette desk is powered on. Cassette is inserted in cassette holder. Cassette desk is in idle state.

**Basic course of action :**

1. User presses **⏮** button.
2. System rewinds cassette to the beginning of the magnetic tape and stops.
3. System releases **⏮** button.
4. User resets counter to 0.
5. User selects audio input device, and starts playing audio from this device.
6. User presses **▶** of the cassette deck.
7. System engages magnetic head on tape and signals recording status with indicator light.
8. System starts recording on the cassette from the selected input device.
9. User depresses **⏹** to stop recording.
10. System disengages magnetic head from tape and signals idle status with indicator light.

**Postconditions :** Same as preconditions.

**Alternate courses :**

**9a :** System detects that end of tape is reached. System releases **▶** button to stop recording. Go to step 10.

**9b :** [If auto reverse functionality is present and enabled :] System detects that end of tape is reached. System disengages magnetic head, engages magnetic head for opposite side of tape, and continues recording on other side of tape. Go to step 9 or 9a.

---

## 2.2 Interaction overview diagram

Un diagramme d'interaction peut être utilisé pour modéliser la vue d'ensemble du comportement d'un cassette deck. 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). Un exemple simplifié (et délibérément incomplet) d'un statechart sur lequel vous pouvez vous inspirer est fourni dans la figure 2.

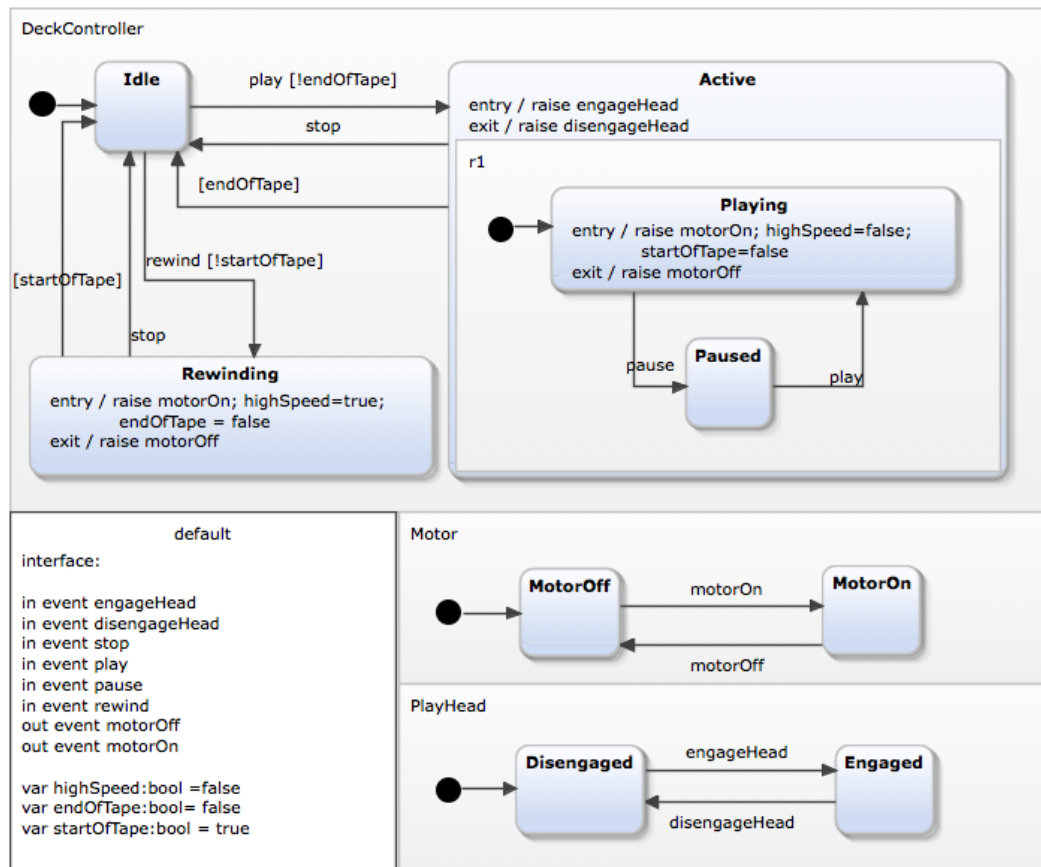


FIGURE 2 – Simplified example of a cassette deck player with limited functionality.

En suivant une approche itérative, trois statecharts doivent être fournis, correspondant à une évolution naturelle de l'application en ajoutant toujours plus de fonctionnalités :

1. Un statechart pour un "cassette deck" contenant uniquement les "required features" de la Section 1.2.
2. Un statechart pour un "cassette deck" contenant tous les "required features" et tous les "optional features" de la Section 1.2, sauf la fonctionnalité "double-deck".
3. Un statechart contenant tous les required et optional features, y inclus la fonctionnalité "double-deck".

Il est obligatoire de modéliser les **statecharts** avec l’outil Yakindu Statechart Tools<sup>3</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) doi(ven)t être exécutable(s) avec le simulateur fourni par Yakindu Statechart Tools.

## 2.4 Diagramme de classes

Le diagramme de classes doit représenter les concepts essentiels du système (configurateur et simulateur) à réaliser, et doit fournir la base pour la phase d’implémentation. Toutes les classes présentes dans ce modèle doivent également faire partie de l’implémentation. (Lors de l’implémentation, d’autres classes “techniques” peuvent encore être rajoutées.)

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



## 3 Implémentation

Le logiciel sera implémenté en utilisant le langage de programmation *Java 8* ou supérieur. L'utilisation des **tests unitaires** est obligatoire. L'utilisation de **design patterns** dans votre code est fortement recommandée. 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.

### 3.1 Bibliothèques recommandées et imposées

**JavaFX** Pour la réalisation de l'**interface graphique** (GUI) de votre application, l'utilisation de *JavaFX* est recommandée. De nombreux tutoriels sont disponibles sur Internet.<sup>4</sup>

Si vous désirez utiliser une autre bibliothèque ou interface graphique (par exemple Swing<sup>5</sup>), il faut demander l'accord des enseignants au préalable.

**JUnit** L'utilisation des **tests unitaires** est obligatoire. Pour réaliser les tests unitaires en Java, l'utilisation de JUnit (version 4.6 ou supérieure) est **imposée**. Vous pouvez le trouver sur son site officiel<sup>6</sup>. JUnit est intégré par défaut dans Eclipse et d'autres environnements de développement pour Java.

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>7</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. La plupart des IDE prennent en charge *maven* dès la création du projet.

**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 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>8</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>9</sup> qui est à la fois simple d'utilisation et très complète.

### 3.2 Outils

**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).

---

4. Par exemple, <http://www.javafx-tutorials.com> et <http://www.tutorialspoint.com/javafx/>

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

6. <http://www.junit.org/>

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

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

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

**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).

## 4 Livrables et échéances

Deux livrables doivent être rendus. Le premier livrable concerne la partie *modélisation* et doit être déposé le **lundi 17 décembre 2018** 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 29 mars 2019** 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 respecter le 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 d'un seul 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* contenant la *maquette de l'interface graphique* qui doit être réaliste et qui doit correspondre aux exigences de l'énoncé.
- 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.*

**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 leur comportement 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 :
- 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 ?
  - Le diagramme d'activités représente-t-il bien la vue d'ensemble des interactions entre les différents cas d'utilisation ?
  - 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 ?
  - 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 ? Reflètent-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 ?
  - 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 ?
  - 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* : 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 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.