Modélisation Logicielle: Travail individuel
Framework de distributeur de boissons

Enseignants: Professeur Tom Mens, Mathieu Goeminne

Année Académique 2015-2016
Année Préparatoire au Master en Sciences Informatiques
à horaire décalé à Charleroi
Faculté des Sciences, Université de Mons

Dernière mise à jour: 18 septembre 2015

Table des matières

1 Cahier des charges

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). Ce travail compte pour **50%** de la note finale de l'unité d'enseignement (UE) US-M1-INFO60-016-C « Modélisation logicielle ».

Le travail consiste en la réalisation d'un logiciel comprennant une interface utilisateur graphique pour l'énoncé décrit dans la section 1.1. Le travail est décomposé en deux phases : la phase de *modélisation* (qui compte pour un tiers de la note de l'AA) et la phase d'*implémentation* (qui compte pour deux tiers de la note de l'AA). La modélisation doit être faite avec le langage de modélisation UML 2. L'implémentation doit être faite en Java 7 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.

1.1 Énoncé

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.

Le but du projet consiste à modéliser et à développer un *framework* générique permettant de visualiser et de simuler au moins 3 différents types de distributeurs de nourriture ou de boisson (par exemple, une variété de friandises, du café et d'autres boissons chaudes, des boissons froides en canette ou en bouteille ; voir figure 4). L'application devra permettre à l'utilisateur de simuler chaque type de distributeur, en tenant compte des spécificités de chacun. Lors de l'exécution de l'application, il devra être possible de changer le type de distributeur sans devoir redémarrer l'application.

FIGURE 1 – Boissons froides

FIGURE 2 – Boissons chaudes

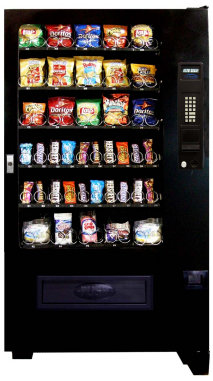

FIGURE 3 – Friandises

FIGURE 4 – Quelques exemples de distributeurs.

En ce qui concerne la modélisation, le framework doit prendre en charge au moins **trois types de distributeurs** :

— Un **distributeur de friandises** (chocolats, chips, ...) ;
— Un **distributeur de boissons froides**, nécessitant un circuit de refroidissement ;
— Un **distributeur de boissons chaudes** (café, thé, chocolat chaud, soupe, ...). Ceci nécessite plusieurs sous-systèmes :

— Un système d'alimentation d'eau (disponible en quantité infinie, mais pouvant être désactivé temporairement), qui doit être chauffée avant de l'ajouter à la boisson commandée.
— Un système permettant de préciser la quantité de sucre à ajouter à certaines boissons. Certaines boissons, telles que la soupe, ne peut pas être sucrée. Si du sucre est demandé, une cuillère sera mise dans le gobelet.
— Un système d'avertissement permettant d'informer l'utilisateur quand la boisson est prête (car la préparation d'une boisson chaude prend toujours un certain temps.)

Pour chaque type de distributeur, un **système de gestion de stocks** est nécessaire, permettant de tenir la comptabilité des produits disponibles. Ce système peut être développé de manière générique, avec une instanciation différente selon le type de distributeur souhaité :

— Pour les distributeurs de friandises ou de boissons froides, la quantité de produits disponibles de chaque type doit être gérée.
— Pour les distributeurs de boissons chaudes, le système doit gérer la quantité de poudre disponible pour chaque type de produit (sucre, thé, café, soupe, chocolat chaud), ainsi que la quantité de gobelets et de cuillères.

Pour **sélectionner le produit désiré,** nous supposons que le distributeur possède un bouton spécifique par type de produit vendu. En appuyant sur ce bouton, le prix ou l'indisponibilité de ce produit sera affiché. Dans le cas d'un distributeur de boissons chaudes, deux boutons supplémentaires seront présents pour augmenter ou diminuer la quantité de sucre.

Une troisième fonctionnalité à modéliser concerne le **monnayeur**. Chaque distributeur possède une fente qui accepte des pièces de monnaie, mais les types de pièces acceptées peuvent varier d'un distributeur à un autre. (Par exemple, certains distributeurs n'accepteront pas les pièces en dessous de 10 cents.) Le distributeur doit rendre des pièces (dans un conteneur spécifiquement prévu pour cela) si le montant total des pièces introduites dépassent le prix du produit.
Certains distributeurs accepteront également des cartes (qui seront soit des cartes bancaires comme Bancontact, ou des cartes à puce rechargeables spécifiquement prévues pour le distributeur).
Consigne : Pour gérer les pièces de monnaie disponibles dans le distributeur, le système de gestion de stocks pourra être exploité.

L'**interface graphique** du simulateur doit contenir plusieurs composants :

— Un écran de configuration doit permettre de choisir le type de distributeur désiré. Cet écran doit également permettre de préciser le type de monnayeur (et les types de pièces acceptées), ainsi que les types de produits vendus par le distributeur. Cet écran de configuration doit apparaître lors du démarrage de l'application, et doit également être accessible lors de l'exécution d'un distributeur spécifique, afin de pouvoir changer le type de distributeur dynamiquement sans devoir relancer l'application.
— Une fois le type de distributeur choisi, un deuxième écran de configuration permettra de préciser (le cas échéant) : la quantité de produits (ou de poudres) de chaque type, le prix de chaque produit, l'emplacement de chaque type de produit, et le nombre de pièces de monnaie de chaque type disponible dans le distributeur (afin de permettre d'échanger la monnaie).
— Après validation de la configuration du distributeur, un écran de simulation du distributeur apparaîtra. Cet écran sera composé de plusieurs panneaux :
  — un panneau d'affichage permettant d'afficher des informations à l'utilisateur
  — une fente permettant d'introduire des pièces de monnaie
  — le cas échéant, une fente permettant d'introduire une carte bancaire ou une carte à puce
  — un panneau contenant les boutons permettant d'interagir avec le distributeur ; ce panneau doit également contenir un bouton permettant d'annuler la commande et de récupérer son argent.
  — un conteneur dans lequel l'utilisateur peut récupérer des pièces correspondant au solde non utilisé. Ce conteneur est toujours ouvert.
  — un conteneur dans lequel le produit commandé peut être récupéré par l'utilisateur. Ce conteneur peut être ouvert ou fermé, et ne peut contenir qu'une seule boisson. Pour les boissons chaudes, le système mettra un gobelet dans le conteneur après la commande de boisson, et le conteneur s'ouvrira dès que la boisson est prête. Si la boisson chaude contient du sucre, une cuillère sera également placée dans le gobelet après que la boisson et le sucre aient été versés.

Lors de la modélisation, les **statecharts** se prêtent très bien à la spécification du comportement du distributeur et de ses composants (comme le monnayeur).

Lors de l'implémentation, prévoyez un **système de journalisation** (logging) permettant de vérifier les détails de toute transaction effectuée (par exemple, le type de produit choisi, la quantité de sucre demandée, le montant introduit, le type de produit reçu, la quantité de stock disponible après la transaction).

Lors de la modélisation et de l'implémentation, utilisez le principe de **programmation défensive**, en utilisant un système de **gestion d'exceptions** et des **tests unitaires**. Pensez à gérer (et à tester !) les nombreux cas problématiques. À titre illustratif, voici une liste *non exhaustive* de problèmes potentiels à considérer :

- certains types de pièces de monnaie ne sont plus disponibles dans le distributeur ;
- le distributeur ne rend pas (ou ne peut plus rendre) la monnaie ;
- une pièce de monnaie est coincée dans le monnayeur ;
- une pièce de monnaie insérée n'est pas reconnue par le monnayeur ;
- un produit (ou gobelet) est coincé dans le distributeur ;
- l'utilisateur a oublié de retirer son gobelet ;
- tous les produits d'un certain type sont vendus ;
- indisponibilité de sucre, des gobelets ou des cuillères ;
- l'alimentation d'eau est désactivée ;
- le système de refroidissement ou de réchauffement ne fonctionne plus ;
- la carte bancaire ou la carte à puce est illisible ;
- le distributeur ne rend pas le produit désiré ;
- le prix du produit rendu ne correspond pas à la monnaie introduite ;
- la monnaie rendue par le distributeur ne correspond pas à celle introduite et au prix du produit demandé ;
- le conteneur de boissons ne peut plus s'ouvrir ou fermer ;
- le distributeur ne devrait jamais proposer de mettre du sucre dans la soupe ;
- ...

1.2 Critères de recevabilité

Cette check-list reprend l'ensemble des consignes à respecter pour la remise du projet de modélisation logicielle. **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 est indiquée sur la plateforme Moodle [1] et dans le présent rapport et devra être respectée à la lettre. *Aucun délai d'aucune sorte ne sera accordé, et aucun retard toléré !* Il vous est conseillé d'uploader des versions préalables avant la version définitive (seule la dernière version reçue avant la date de remise sera évaluée).

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

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

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 les éléments suivants :

1. Pour la partie modélisation :
   - une maquette de l'interface utilisateur en format PDF, portant le nom ML-<noms de famille>-maquette.pdf
   - un rapport de modélisation en format PDF, contenant toutes les images de vos modèles UML accompagnés par une explication textuelle, et portant le nom ML-<noms de famille>-modelisation.pdf

1. https://moodle.umons.ac.be

— un ou plusieurs fichier(s) .sct contenant les statecharts que vous avez modélisés avec l'outil Yakindu Statechart Tools (voir plus loin).

2. Pour la partie implémentation :
   — le rapport d'implémentation en format PDF, portant le nom ML-<noms de famille>-implementation.pdf
   — un mode d'emploi (manuel d'utilisation) en format PDF, incluant des captures d'écran accompagnées d'une explication textuelle, et portant le nom ML-<noms de famille>-manuel.pdf
   — un fichier `jar` **auto-exécutable** indépendant de la plateforme (Windows, Linux, MacOS)
   — le code source (en Java 7 ou supérieur) et tous les fichiers nécessaires à sa compilation et son exécution. Nous devons être capables de compiler ou d'exécuter votre application sans erreur ni exception non gérée ! Tous les attributs et toutes les méthodes de chacune de vos classes et interfaces doivent être commentées grâce aux annotations Javadoc.
   — les tests unitaires (en JUnit 4.6 ou supérieur) et tous les fichiers nécessaires à leur exécution

   Si vous jugez que certaines figures intégrées dans les rapports rendus ne sont pas assez lisibles, vous pouvez toujours ajouter les images originales dans le dossier contenant le rapport.

   Les noms de tous les membres du groupe doivent figurer en page de garde des rapports et du manuel, ainsi que dans la javadoc de chaque fichier de votre implémentation. 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.

1.3 Bibliothèques recommandées et imposées

Swing Pour l'interface graphique (GUI) de votre application, nous recommandons l'utilisation de l'API Swing (qui est basé sur AWT). Plusieurs tutoriels sont disponibles sur Internet. Le livre « Swing, la synthèse » [2] est également un bon point de départ pour comprendre le fonctionnement de cette API.

JUnit Pour les tests unitaires en Java, l'utilisation de JUnit (version 4.6 ou supérieure) est **imposée**. JUnit est un framework pour les tests unitaires en Java. Vous pouvez le trouver sur son site officiel [3] ; il est intégré par défaut dans Eclipse et d'autres environnements de développement Java.
Pour assurer un code de bonne qualité, 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 formellement le comportement que votre application doit avoir au terme du projet. Cela permet de plus d'évaluer votre progression. Une autre bonne pratique est d'utiliser des *tests de régression*. Il s'agit d'écrire un test unitaire qui met en évidence chaque erreur rencontrée, vous n'aurez ainsi pas à comprendre et résoudre deux fois le même problème.

Système de contrôle de versions Nous vous encourageons fortement à utiliser un système de contrôle de versions distribué (par exemple, Git) ou centralisé (par exemple, Subversion) 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 en arrière en cas de problème, etc. Afin de vous assurer de la pérénité 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 [4]. Cependant, cette copie ne doit être accessible, en lecture comme en modification, qu'aux membres du groupe (et éventuellement les enseignants). Les enseignants peuvent, sur demande, mettre à votre disposition un tel dépôt.

2. V. Berthié et J.-B. Briaud, Swing, la synthèse. Dunod, 2005.
3. http ://www.junit.org/
4. `https://bitbucket.org/`

Logging Comme cela est indiqué dans l'énoncé, vous **devez** utiliser un système de journalisation afin de faciliter le débogage de l'application ainsi que la compréhension de son fonctionnement. Nous vous **recommandons** l'utilisation de Apache Log4j 2[5], qui est à la fois simple d'utilisation et très complet.

Maven Afin de faciliter la compilation, le test et le packaging de votre application, vous **devez** utiliser Maven[6]. Vous devrez également 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, packager et exécuter votre application depuis un nouvel environnement de travail. La plupart des IDE prennent en charge Maven dès la création du projet.

1.4 Outils

Outils de modélisation Les diagrammes UML doivent obligatoirement être réalisés au moyen d'un outil dédié. Bien que vous ayez le libre choix de cet outil, nous vous recommandons Visual Paradigm pour lequel l'UMONS possède une licence académique. Beaucoup d'autres outils commerciaux sont disponibles en version gratuite ≪Community Edition≫ sous la forme de stand-alone ou de plug-in pour Eclipse ou d'autres environnements de développement. Il existe également plusieurs outils open source de modélisation UML.
Pour les modèles de **statechart**, il est **obligatoire** d'utiliser l'outil Yakindu Statechart Tools[7]. Cet outil est un plug-in pour Eclipse permettant de spécifier, vérifier, simuler et de générer du code source pour des statecharts.

Outils de développement Vous pouvez librement choisir votre environnement de développement Java (par exemple Eclipse, NetBeans ou IntelliJ) à condition que les enseignants qui évalueront l'application puissent facilement compiler et exécuter votre logiciel sans devoir installer cet environnement. L'UMONS possède une licence académique pour la version pro d'IntelliJ.

Système d'exploitation Vous pouvez utiliser n'importe quel système d'exploitation pour développer le framework à condition que les livrables (c.-à-d. le code source, les tests unitaires et l'interface graphique) soient indépendants de la plateforme choisie. Le code produit sera testé sur trois systèmes d'exploitation différents (MacOS X, Linux et Windows).

5. `http://logging.apache.org/log4j/2.x/`
6. `https://maven.apache.org/`
7. Téléchargeable sur `www.statecharts.org`

2 Livrables

2.1 Échéances

Deux livrables doivent être rendus :

— Le premier livrable concerne la partie **modélisation** (maquette de l'interface graphique et diagrammes de conception en UML). Il doit être déposé avant les vacances de Noël, le **lundi 14 décembre 2015** au plus tard. Les enseignants inspecteront ce livrable et ils l'approuveront ou proposeront des améliorations que vous devrez intégrer avant d'entamer la phase d'implémentation. Le contenu du livrable ainsi que les exigences de qualité sont décrits dans la section 2.2.

— Le deuxième livrable concerne la partie **implémentation** (code Java, tests unitaires et manuel d'utilisation) et doit être déposé le **vendredi 25 mars 2016** au plus tard. Le contenu du livrable ainsi que les exigences de qualité sont décrits dans la section 2.3.

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** (cf. section 1.2).

2.2 Modélisation

Un **modèle de conception** devra être réalisé sur base du cahier de charges, en utilisant le langage de modélisation *UML 2*. Les étudiants sont libres en ce qui concerne le choix d'un outil de modélisation UML. Au moins les diagrammes *UML 2* suivants doivent être utilisés : les *diagrammes de cas d'utilisation*, les *diagrammes de classes* et les *diagrammes d'états (statecharts)*. Les étudiants sont encouragés à utiliser d'autres types de diagrammes UML pour compléter la modélisation de l'application (par exemple les diagrammes de séquences et diagrammes d'activités).

Pour modéliser le comportement du distributeur et de ses composants (comme le monnayeur), il est obligatoire de modéliser des **statecharts exécutables** avec l'outil Yakindu Statechart Tools.

Puisqu'il s'agit d'une application avec interface graphique, vous devriez également fournir une **maquette de l'interface graphique** qui sera utilisée par l'application. Cette maquette devra être créée au moyen d'un logiciel de votre choix.

Livrable. Ce livrable doit contenir trois éléments :

— une *maquette de l'interface graphique* qui doit être réaliste et qui doit correspondre aux exigences de l'énoncé.

— le *rapport de modélisation* incluant tous les *diagrammes UML* proposés, et une *description textuelle* des choix de conception qui ont été pris ainsi que des éléments essentiels dans chaque diagramme fourni. Les diagrammes doivent être lisibles après impression sur papier en noir et blanc.

— les fichiers `.sct` des statecharts, modélisés et exécutables avec l'outil Yakindu Statechart Tools.

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 ? Les statecharts proposés sont-ils exécutables par le simulateur de Yakindu Statechart Tools ?
2. *Style* : Les diagrammes sont-ils bien structurés ? Suivent-ils un style de conception orienté 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.)
3. *Exactitude* : Les différents éléments d'un diagramme UML 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 ? Fait-on appel aux points d'extension lorsqu'ils sont nécessaires ? Les conditions d'extension sont-elles présentes ? Y a-t-il une description de chaque cas d'utilisation ?
   (b) Dans le *diagramme de classes*, les multiplicités sur les associations sont-elles judicieusement utilisées ? L'utilisation de 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 de *design patterns* lorsque cela est utile ?

(c) Les *statecharts* (spécifiés avec Yakindu Statechart Tools) 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 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 compréhension du diagramme ? *Vous devez faire une simulation de vos statecharts avec Yakindu Statechart Tools afin de vérifier leur comportement correct.*

(d) Utilise-t-on les bonnes conventions de nommage ? (Par exemple, éviter l'utilisation du pluriel dans les noms des classes, ne pas mélanger le français et l'anglais, ...)

4. *Compréhensibilité* : la maquette de l'interface graphique et les diagrammes sont-ils faciles à comprendre ? Ont-ils le bon niveau de détail ? Pas trop abstraits, pas trop détaillés ?
5. *Cohérence* : Les diagrammes UML sont-ils syntaxiquement et sémantiquement corrects et cohérents ? N'y a-t-il pas d'inconsistance : (i) dans les diagrammes ; (ii) entre les différents diagrammes ? 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 ?

2.3 Implémentation

Le logiciel sera implémenté en utilisant le langage de programmation *Java 7* ou supérieur. L'implémentation et l'utilisation de tests unitaires (avec le framework *JUnit 4.6* ou supérieur) sont **obligatoires**. Vous pouvez utiliser la bibliothèque graphique de votre choix, mais l'utilisation de *Swing* (basé sur AWT) est fortement recommandée. Lors de l'implémentation des statecharts en Java, le *state design pattern* doit être utilisé obligatoirement. Plutôt que d'implémenter manuellement ce *state design pattern*, 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. L'utilisation d'autres design patterns est fortement recommandée.

Livrable. Le livrable doit contenir un mode d'emploi (manuel d'utilisation) expliquant le fonctionnement de l'application, un rapport d'implémentation, un fichier .jar auto-exécutable, et une version complète du code source, des tests unitaires, et l'interface graphique. Tous les fichiers nécessaires à la compilation du code Java et JUnit doivent être inclus. Toutes les bibliothèques nécessaires à l'utilisation de l'application doivent être déclarées comme autant de dépendances dans Maven.

Exigences de qualité. Nous exigeons une bonne qualité de code. Le code source en Java ne peut pas contenir d'erreur 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 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 convient d'éviter un style procédural avec des méthodes complexes et beaucoup d'instructions conditionnelles. Les design patterns doivent être utilisés.

— *Documentation* : Le code source doit être documenté au moyen de javadoc et de commentaires, de sorte que le rôle et le fonctionnement des éléments de l'application soient facilement compréhensibles pour le lecteur. La javadoc générée automatiquement à partie du code source devra couvrir l'entièreté des classes, interfaces, attributs et méthodes de votre application.
— *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 et intuitif.
— *Indépendance de la plateforme* : 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 spéciale 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.