Génie Logiciel (US-B2-SCMATH-014-M)
Projet de modélisation et d'implémentation (S-MATH-789)
Simulateur pour un distributeur de billets de train

Enseignants: Professeur Tom Mens, David Hauweele

Année Académique 2017-2018
Bachelier en Sciences Mathématiques
Faculté des Sciences, Université de Mons

Dernière mise à jour: 20 septembre 2017

Table des matières

Résumé

L'unité d'enseignement (UE) US-B2-SCMATH-014-M "Génie Logiciel" est composée de deux activités d'apprentissage (AA), chacune comptant pour **50%** de la note finale de l'unité d'enseignement (UE). Un projet non rendu implique automatiquement un échec de l'UE.

1. La note de l'AA S-INFO-013 "Génie Logiciel" sera obtenue sur base des résultats d'un examen écrit.
2. L'activité d'apprentissage (AA) S-MATH-789 "Projet de modélisation et d'implémentation" consiste en un travail de conception/modélisation d'un système logiciel en UML et de l'implémentation/programmation de ce système en Java, conforme aux modèles UML. Le travail consiste en la réalisation d'un système logiciel avec interface utilisateur graphique pour l'énoncé décrit dans la section 1.

Le projet se décompose en deux phases successives :

— La phase de *modélisation* compte pour **un tiers** (33%) de la note de l'AA S-MATH-789. La modélisation doit être fait avec le langage de modélisation UML 2.5 ou supérieur. Le comportement du système doit être modélisé à l'aide d'un ou plusieurs *statechart* (diagramme d'états hiérarchiques) exécutables.

— La phase d'*implémentation* compte pour **deux tiers** (67%) de la note de l'AA S-MATH-789. L'implémentation doit être faite 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.

Chaque étudiant travaillera de manière individuelle (ou en groupe de deux avec l'accord de l'enseignant) sur le projet. Toute tentative de travailler ensemble avec d'autres personnes, et tout soupçon de plagiat, par exemple en copiant du code source depuis Internet ou ailleurs sans le mentionner et sans en informer les enseignants peut mener à un échec du cours ou de l'année d'études.

Le travail sera à rendre sur la plateforme Moodle (`https://moodle.umons.ac.be`) avant les deux échéances imposées (une pour chaque phase). Les dates de remise sont très strictes. L'étudiant sera sanctionné par une note de 0/20 pour cette phase de l'AA si la date de remise n'est pas respectée.

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.

FIGURE 1 – Exemple d'un distributeur de billets de train

Le but du projet consiste à modéliser et à développer un **simulateur** avec interface graphique réaliste d'une **famille de distributeurs de billets de train**. Voir figure 1 pour un exemple d'un distributeur de billets. L'application devra permettre à l'utilisateur de configurer un distributeur spécifique et d'interagir avec son simulateur pour effectuer des activités proposées par le distributeur. Lors de l'exécution de l'application, il devra être possible de changer le type de distributeur sans devoir redémarrer l'application.

1.1 Composants

Afin de fonctionner correctement, chaque distributeur supporté par le simulateur a besoin de plusieurs composants. Certains de ces composants sont obligatoires, d'autres sont optionnels en fonction de la configuration choisie. Les composants **obligatoires** sont :

Controller Le *contrôleur* est le noyau du système. Il contient la logique principale du système. Il contrôle et interagit avec tous les autres composants.

Screen L'écran permet d'afficher diverses informations à l'utilisateur. L'écran peut être tactile ou non. Si l'écran n'est pas tactile, le composant optionnel **Keyboard** est nécessaire pour permettre l'interaction avec l'utilisateur.

CardReader Le lecteur de carte permet à un utilisateur d'insérer sa carte bancaire (Bancontact ou Maestro) ou sa carte de crédit (Visa ou American Express) pour effectuer le paiement.

Printer L'imprimante permet d'imprimer les billets de train, ainsi que les reçus de paiements.

ReceptionSlot La fente de réception permet à l'utilisateur de récupérer les billets et reçus.

Les composants **optionnels** sont :

Keyboard Les distributeurs qui ne sont pas dotés d'un écran tactile possèdent toujours un clavier permettant d'introduire les données. Cependant, les distributeurs disposant d'un écran tactile peuvent également être équipés un clavier physique.

CoinSlot Certains distributeurs permettent de payer en utilisant des pièces de monnaie. Une fente permet d'insérer des pièces. Le distributeur peut ouvrir ou fermer la fente au moment opportun. Le composant **ReceptionSlot** permet de rendre des pièces si le montant inséré excède le montant à payer.

BillSlot Certains distributeurs permettent de payer en utilisant des billets de monnaie. Une fente permet d'insérer les billets. Le distributeur peut ouvrir ou fermer la fente au moment opportun. Le composant **ReceptionSlot** permet de rendre des billets de monnaie si le montant inséré excède le montant à payer.

CodeScanner Certains distributeurs possèdent un scanneur permettant de lire des codes barres ou des codes QR. Ce scanneur est nécessaire pour réaliser la fonctionnalité d'abonnement décrite dans la Section 1.3.

1.2 Interface graphique

L'**interface graphique** du simulateur doit prévoir au moins deux types d'écrans :

Fenêtre de configuration Lors du démarrage ou à l'aide d'un menu accessible pendant l'exécution du simulateur, la fenêtre de configuration doit permettre de préciser le type de distributeur qu'on souhaite simuler. La configuration permet d'effectuer les choix suivants :

- Un écran tactile ou la combination d'un écran normal avec un **Keyboard**.
- La présence ou non d'un **CoinSlot**. Si un **CoinSlot** est choisi, on a aussi le choix entre la présence ou non d'un **BillSlot**. (Un distributeur qui possède un **BillSlot** doit toujours avoir un **CoinSlot**.)
- La présence ou non d'un **CodeScanner**. Uniquement les appareils ayant ce scanneur donneront accès à la fonctionnalité d'abonnement décrite dans la Section 1.3.

Fenêtre de simulation Après validation de la configuration, la fenêtre de simulation apparaîtra pour le type de distributeur de billets choisi. Cette fenêtre doit permettre à l'utilisateur de simuler les interactions avec le distributeur de billets de train.

Panneau de gestion de pannes Un panneau (ou onglet ou fenêtre ou menu) doit être fourni dans la fenêtre de simulation pour simuler la gestion de pannes. Ce panneau doit permettre de mettre en panne ou de réactiver certains composants du distributeur, pendant que la simulation s'exécute. Une liste non exhaustive des pannes qui devraient être gérées est :

- **CardReader** en panne, empêchant le paiement par carte bancaire.
- Carte coincée dans le lecteur de carte, mettant tout le distributeur en mode hors service, et demandant l'intervention d'un technicien.
- **CodeScanner** en panne, empêchant certaines fonctionnalités pour le renouvellement des abonnements (voir ci-dessous).
- **CoinSlot** ou **BillSlot** en panne, empêchant le paiement par monnaie.
- **Printer** en panne, mettant tout le distributeur en mode hors service, et demandant l'intervention d'un technicien.

1.3 Fonctionnalités

Au niveau des fonctionnalités, le distributeur de billets doit toujours permettre à l'utilisateur d'effectuer les activités suivantes :

Vérifier l'horaire des trains de la journée. Pour vérifier l'horaire des trains, l'utilisateur peut choisir une gare et demander de voir soit la liste de toutes les arrivées dans la gare, ou tous les départs de la gare. Par défaut, les arrivées et départs sont affichés à partir de l'heure courante. L'utilisateur peut changer cette heure pour voir les départs et arrivées plus tard dans la journée. Alternativement, l'utilisateur peut préciser une source et une destination, et demander de voir tous les trains entre ces deux stations, à partir de l'heure actuelle. De nouveau, l'utilisateur peut changer cette heure pour voir les trains plus tard dans la journée.

Acheter des billets. Pour acheter des billets, l'utilisateur doit d'abord préciser la source et la destination. Par défaut, le prix d'un billet standard aller/retour en deuxième classe lui sera proposé pour le jour même. S'il le souhaite, l'utilisateur peut changer les paramètres du billet : première ou deuxième classe, aller simple ou aller/retour, nombre de billets, jour de départ, type de billet (standard, senior :+65, junior :-26, enfant :-12, billets spéciaux pour excursions, billets vélo, ...), ainsi que des réductions éventuelles (familles nombreuses, personne à mobilité reduite, ...). À chaque modification, le prix sera mis à jour automatiquement. Si l'utilisateur accepte la proposition, il confirme son choix, et procède au **paiement**. À tout moment avant le paiement, l'utilisateur peut annuler la transaction.

Acheter des pass. Des "pass" permettent à un utilisateur d'effectuer plusieurs trajets. Un menu avec trois différents types de pass sera proposé à l'utilisateur : (1) des pass pour 10 trajets entre deux gares prédéfinies ; (2) des pass de 10 trajets sans restriction de trajet ; (3) et des pass permettant de voyager pendant x jours sans restriction de trajet. Après sélection de son choix, l'utilisateur doit fournir les données correspondant à son choix (par exemple, le nombre de jours ou la source et la destination). Il doit aussi fournir son nom, car les pass sont nominatifs. Comme pour les billets, il peut aussi choisir entre première ou deuxième classe, ou bénéficier d'un tarif réduit en fonction de son âge ou d'autres réductions qui s'appliquent. A chaque modification, le prix sera mis à jour automatiquement. Si l'utilisateur accepte la proposition, il confirme son choix, et procède au **paiement**. À tout moment avant le paiement, l'utilisateur peut annuler la transaction.

Acheter des abonnements. Cette dernière fonctionnalité est uniquement offert par les distributeurs dotés d'un **CodeScanner**. Un "abonnement" permet à un utilisateur de faire la navette entre une source et une destination jusqu'à la fin de la période de validité de l'abonnement. Pour acheter un abonnement, l'utilisateur doit fournir son nom et son numéro de registre national, ainsi que la source et la destination, la période de validité (1, 3, 6 ou 12 mois), la classe (première ou deuxième), et des réductions éventuelles. Un prix sera proposé. Si l'utilisateur accepte la proposition, il confirme son choix, et procède au **paiement**. À tout moment avant le paiement, l'utilisateur peut modifier ses paramètres ou annuler la transaction.

Renouveler des abonnements. Un utilisateur peut décider de "renouveler" un abonnement existant, en étendant sa période de validité. Pour ce faire, il scanne le code barre ou code QR affiché sur l'abonnement avec le **CodeScanner**. Si le **CodeScanner** est en panne ou n'est pas capable de lire le code, le système proposera à l'utilisateur d'introduire le numéro de l'abonnement de manière manuelle. Ensuite, l'utilisateur précise la durée de validité de l'extension (1, 3, 6 ou 12 mois). Le système indique le prix à payer. Si l'utilisateur accepte la proposition, il confirme son choix, et procède au **paiement**. À tout moment avant le paiement, l'utilisateur peut modifier ses paramètres ou annuler la transaction.

Paiement. Le mode de paiement dépend des composants disponibles sur le distributeur (**CardReader**, **CoinSlot** et **BillSlot**) ainsi que de leur état (actif ou en panne).

Si le **CardReader** est fonctionnel, l'utilisateur peut introduire sa carte bancaire ou sa carte de crédit. Le système proposera alors de payer le montant indiqué de manière habituelle : l'utilisateur doit introduire le code PIN de sa carte pour confirmer l'achat. Il peut aussi encore décider d'annuler la transaction. Pour des montants en dessous de 5 €, aucun code PIN sera demandé. Dès que la réception du paiement est confirmée par le système, la carte est éjectée.

Si le **CoinSlot** et/ou **BillSlot** est disponible et fonctionnel, l'utilisateur peut introduire des pièces et/ou billets de monnaie pour un montant qui est au moins le prix à payer. Si la machine doit rendre des pièces à l'utilisateur,

elle vérifie si un nombre suffisant de pièces sont disponibles en stock. Sinon, tout l'argent inséré est rendu et l'utilisateur doit choisir un autre mode de paiement ou annuler sa transaction.
Chaque paiement réussi se termine par l'impression des billets demandés, ainsi que l'impression d'un reçu de paiement si l'utilisateur la demande. Les billets, le reçu et (le cas échéant) la monnaie rendue seront déposés dans le **ReceptionSlot**.

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 (en anglais) est illustré ci-dessous pour l'achat d'un billet, en utilisant un distributeur contenant uniquement des composants obligatoires :

Use case name : Pay Ticket
Summary : User pays for and receives a confirmed train ticket.
Actors : User (primary), Bank (secondary)
Assumptions : CardReader is functioning. Printer is functioning. Connection with bank is working.
Preconditions : User has selected a train ticket and confirmed his choice. CardReader is empty.
Basic course of action :

1. Screen informs user to proceed with payment, indicating the price to pay.
2. User inserts bank card in CardReader.
3. System verifies and approves card validity.
4. Screen asks user to enter PIN code.
5. User enters correct PIN code.
6. System sends transaction to bank and receives confirmation.
7. CardReader ejects card.
8. System prints requested ticket(s) and drops them in reception slot.
9. System asks if user needs receipt.
10. If user confirms receipt, system prints receipt and drops it in reception slot.

Postconditions :
— Train ticket and printed receipt (if user requested one) has been dropped in reception slot.
— CardReader is empty.
— User's card or bank account have been debited with train ticket price.
Alternate courses :

1a : User cancels payment transaction. Ticket is not printed.

3a : System rejects and ejects card. Go to step 1.

5a : Users enters wrong PIN code. Go to step 4.

6a : Bank refuses transaction. Ticket is not printed.

2.2 Interaction overview diagram

Un diagramme d'interaction peut être utilisé pour modéliser la vue d'ensemble du comportement d'un distributeur de billets de train. 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

Il est obligatoire de modéliser les modèles de **statechart** avec l'outil Yakindu Statechart Tools [1]. Cet outil fournit un générateur de code Java qui peut être utilisé (mais l'utilisation de ce générateur n'est pas obligatoire).

Au moins un statechart modélisant tout le comportement du contrôleur doit être fourni. Typiquement, ce modèle représentera les états du contrôleur. Selon l'état dans lequel se trouve le contrôleur, il réagira différemment aux événements reçus.

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 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 séquence 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.

1. Téléchargeable sur `www.statecharts.org`

3 Implémentation

Le logiciel sera implémenté en utilisant le langage de programmation *Java 8* ou supérieur. L'utilisation des **tests unitaires** (avec le framework *JUnit*) est obligatoire. Pour la réalisation de l'**interface graphique**, l'utilisation de *JavaFX* est recommandée. Si vous désirez utiliser une autre bibliothèque ou interface graphique (par exemple Swing [2]), il faut demander l'accord des enseignants au préalable.

L'utilisation de **design patterns** dans votre code est fortement recommandée. 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.

Lors de l'implémentation, 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.

3.1 Bibliothèques recommandées et imposées

Base de données Le distributeur dépend fortement d'une *base de données* (au sens large du terme) contenant, entre autres, les déplacements des trains, les abonnements, les types de billets et les différents tarifs. Vous **devez** proposer une approche systématique et élégante pour gérer cette base de données. L'application devant fonctionner sans avoir recours à des services extérieurs, l'utilisation d'une base de données client-serveur (MySQL, par exemple) n'est pas acceptable. Par contre, l'application peut se baser sur des fichiers déployés localement. Nous vous recommandons l'utilisation de SQLite [3]. Le stockage des données dans des fichiers XML est une alternative à considérer.

JavaFX Pour l'interface graphique (GUI) de votre application, nous recommandons l'utilisation de JavaFX. De nombreux tutoriels sont disponibles sur Internet. [4]

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 [5]. 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 [6] 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 [7]. Cependant, cette copie ne doit être accessible, en lecture comme en modification, qu'aux membres du groupe (et éventuellement les enseignants).

2. `http://docs.oracle.com/javase/tutorial/uiswing/` et `http://www.tutorialspoint.com/swing/`
3. `https://www.sqlite.org`
4. Par exemple, `http://www.javafxtutorials.com` et `http://www.tutorialspoint.com/javafx/`
5. `http://www.junit.org/`
6. `https://maven.apache.org`
7. `https://bitbucket.org/`

Système de journalisation Pour faciliter le déboggage, nous encourageons l'utilisation d'un système de journalisation, comme la librairie Apache Log4j (version 2) [8] 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).

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).

8. `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 **lundi 18 décembre 2017** 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 30 mars 2018** 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 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.

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 :
   (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* (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 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 document en format pdf présentant le *mode d'emploi*, expliquant le fonctionnement 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 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.