IN201 : PhotOwl

Author : Christophe Garion
Public : SUPAERO 2A
Date :

Résumé

Ce projet en équipe a pour but de concevoir, implanter, tester et documenter une application « réelle » afin de mettre en pratique les concepts vus en cours. Le projet de cette année consiste en la création d'une application permettant de gérer une bibliothèque d'images avec un système d'étiquettes avancé et la possibilité d'appliquer des transformations sur les images.

1 Présentation du projet

Il existe de nos jours de nombreux logiciels permettant de gérer une bibliothèque d'images ou de photos. Les logiciels les plus récents permettent non seulement de gérer les photos dans des hiérarchies de répertoires, mais également de leur associer des étiquettes (*tags*) qui permettent ensuite de rechercher plus facilement une photo donnée. Enfin, certains d'entre eux proposent également des fonctionnalités propres aux logiciels de retouche d'images, comme l'application d'un flou gaussien, etc.
Nous allons nous intéresser ici à la conception et la construction d'un logiciel de gestion d'images en développant trois fonctionnalités particulières :

- la classification des images grâce à une ontologie
- l'application de transformations « élémentaires » sur une image
- la sauvegarde des informations sur l'ontologie et les transformations appliquées à des images sous divers formats

2 Classification d'images

Le système le plus simple pour classer des images (ou des fichiers de façon générale) est d'utiliser une hiérarchie de répertoires. Cette classification ne permet pas par contre de classer une image suivant plusieurs critères. Par exemple, supposons qu'en vacances dans les Vosges (si si, ça se fait...), j'ai photographié un darou (appelation locale du fameux dahu [15]). Je souhaite alors classer cette image sous différents concepts : « vacances », « Vosges », « animal », « mythe ». Une hiérarchie de répertoires contenant des répertoires représentant ces différents concepts nous obligerait à dupliquer l'image ou à créer de nombreux liens (cf. figure 1).

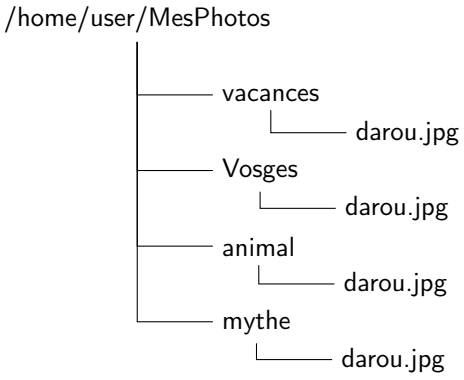


FIGURE 1 – Une hiérarchie de répertoires permettant de classer l'image du darou

Pour pallier ce problème, on utilise maintenant des systèmes d'étiquettes, qui permettent d'associer des mots-clés à une image (plus généralement à un fichier). Des formats de métadonnées, comme IPTC [16] pour le format JPEG ou nativement pour le format PNG [11, 18] (même s'il n'y a pas de standard défini dans ce cas), permettent même d'embarquer ces informations dans les fichiers images. Nous nous intéresserons ici à des étiquettes qui ne seront pas embarquées dans les fichiers, mais gérées par l'application.
Jusqu'à présent, les applications de gestion de bibliothèques d'images ne permettent pas[1] de lier les étiquettes, i.e. d'établir un lien sémantique entre plusieurs étiquettes (inclusion, utilisation, etc.). Par exemple, les étiquettes « bipède » et

1. À ma connaissance...

« quadrupède » pourraient être considérées comme des sous-types d'une étiquette « animal ». L'ensemble des relations existant entre différents concepts s'appelle une *ontologie* [17]. Les ontologies sont de plus en plus utilisées actuellement pour représenter un corpus de connaissances, en particulier dans le domaine du web sémantique [19].
Nous utiliserons des ontologies ne s'intéressant qu'aux relations de taxonomie, i.e. les relations d'inclusion entre concepts. Ces relations peuvent être représentées sous la forme d'un arbre (cf. figure 2 pour un exemple).

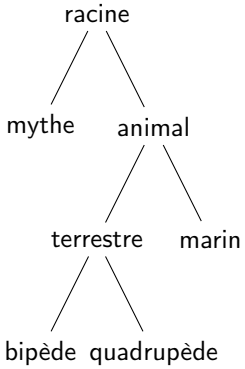


FIGURE 2 – Un exemple d'ontologie sur les étiquettes

Le fait d'avoir une ontologie des étiquettes va nous permettre, à partir d'une image donnée, de trouver les images qui sont sémantiquement proches. Pour cela, on peut utiliser la mesure de Wu-Palmer qui définissent dans [21] une mesure entre deux étiquettes E_1 et E_2 par :

$$Sim(C_1, C_2) = \frac{2 \times depth(C)}{depth(C_1) + depth(C_2)}$$

où :

- C est le plus petit généralisant de C_1 et de C_2, i.e. l'étiquette dont le nœud est le plus profond à avoir à la fois C_1 et C_2 dans des sous-arbres ;
- $depth(C)$ est la plus petite distance entre C et la racine ;
- $depth(C_1)$ est la plus petite distance entre C_1 et la racine en passant par C ;
- $depth(C_2)$ est la plus petite distance entre C_1 et la racine en passant par C.

Plus la mesure entre deux étiquettes est proche de 1, plus les concepts représentés par les étiquettes sont proches.
Par exemple, en utilisant la figure 2, on peut en déduire que :

- la mesure de similitude entre l'étiquette « quadrupède » et l'étiquette « bipède » est de $\frac{2}{3}$, donc les étiquettes sont sémantiquement proches ;
- la mesure de similitude entre l'étiquette « quadrupède » et l'étiquette « marin » est de $\frac{2}{5}$, donc les étiquettes sont sémantiquement proches, mais moins que dans le cas précédent ;
- la mesure de similitude entre l'étiquette « quadrupède » et l'étiquette « mythe » est de 0, donc les étiquettes ne sont pas proches (les concepts sont très différents).

3 Application de transformations sur les photos

NB : si vous cherchez plus d'informations, les notions abordées dans cette section sont développées dans le document [22] qui sera accessible en ligne lorsqu'il sera finalisé.
On veut également pouvoir appliquer des transformations sur les images de la bibliothèque. Pour cela, on suppose que les images dont on dispose peuvent être representées via une matrice de pixels de dimension $m \times n$. Chaque pixel a une couleur qui est représentée par trois composantes (classiquement des nombres entiers compris entre 0 et 255), dites RGB, qui représentent chacune une couleur : rouge, vert et bleu.

3.1 Rotations et symétries

Une des transformations les plus simples que l'on puisse réaliser est d'appliquer une rotation ou une symétrie sur une image. Il suffit alors d'appliquer une transformation élémentaire sur la matrice représentant l'image. Nous ne nous étendrons pas plus sur le sujet.

3.2 Filtres

On peut également appliquer des opérations de filtrage sur une image. Nous ne considérerons ici que des opérations utilisant des filtres linéaires qui permettent par définition d'utiliser la convolution.
Une opération de filtrage utilisera un *noyau de convolution* H qui est une matrice de taille $(2p+1) \times (2q+1)$ à éléments dans $\mathbb{R}$ qui représente le filtre utilisé.
On remarquera que le noyau de convolution est de taille impaire. On suppose que l'indexation du noyau de convolution est centrée, i.e. que les colonnes (resp. les lignes) sont indexées de $-p$ à p (resp. de $-q$ à q).
Pour un pixel de « coordonnées » (x, y) dans la matrice, l'application du filtre reviendra à réaliser l'opération suivante sur chacune de ses composantes RGB ($V(x, y)$ représente la valeur sur une composante RGB d'un pixel de coordonnées (x, y) et V_H la valeur obtenue après application de l'opération de filtrage) :

$$V_H(x, y) = \sum_{i=-p}^{p} \sum_{j=-q}^{q} V(x - i, y - j) \times H(i, j)$$

Se pose alors le problème de l'application du filtre aux frontières de l'image (lorsque $x \notin \{p + 1, \dots, m - p - 1\}$ ou $y \notin \{q + 1, \dots, n - q - 1\}$). On peut considérer que les pixels voisins n'existant pas dans l'image ont des valeurs RGB nulles ou identiques à celles du pixel « réel » le plus proche. Évidemment, ces approximations introduisent des défauts de cohérence des images, mais nous les ignorerons.
Le premier filtre que l'on pourra utiliser est un filtre moyenneur, qui permet de « flouter » une image. Son noyau de convolution est défini par :

$$H_M = \frac{1}{(2p+1) \times (2q+1)} \times [1]_{(2p+1)\times(2q+1)}$$

où $[1]_{(2p+1)\times(2q+1)}$ représente la matrice de taille $(2p+1) \times (2q+1)$ ne comportant que des 1.
Dans ce cas, plus la taille du noyau est grande, plus l'effet de flou sera important.
La figure 3 présente une image originale et l'image obtenue après application d'un filtre moyenneur [2].

(a) Image originale (b) Image floutée

FIGURE 3 – Application d'un filtre moyenneur

Le deuxième filtre que l'on pourra utiliser est un filtre dérivateur, qui permet de détecter les contours d'une image. Le noyau d'un filtre dérivateur dit de Sobel est défini par les deux matrices suivantes :

$$H_{Dx} = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

2. Vous pourrez télécharger le sujet du BE ainsi que les images pour mieux voir l'effet des filtres.

$$H_{Dy} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

qui représentent respectivement la « dérivée » horizontale et la « dérivée » verticale de l'image. On utilise alors la formule :

$$V_H(x,y) = \sqrt{V_{Hx}(x,y)^2 + V_{Hy}(x,y)^2}$$

pour calculer les valeurs RGB en un point de l'image filtrée (il se peut lors de ces calculs que l'on soit obligé de plafonner les résultats à 255 pour ne pas « déborder » sur une autre couleur).
La figure 4 présente quelques applications de filtres dérivateurs construits à partir de ce qui précède.

4 Sauvegarde des étiquettes, de l'ontologie et des transformations

On souhaite pouvoir sauvegarder les ontologies d'étiquettes ainsi que les transformations appliquées à une image. Pour cela, il faut disposer d'un format permettant de définir les ontologies et les transformations. Nous allons nous intéresser ici à un format de sauvegarde utilisant le langage XML.

4.1 Présentation de XML

XML [20] est un langage de marquage qui permet d'ajouter du contenu sémantique à un fichier contenant du texte. Il est facilement lisible par un humain et par une machine, ce qui en fait son intérêt. Nous ferons lors du cours un TP utilisant XML pour vous permettre de le manipuler en dehors du projet.
Un fichier XML représente un *arbre* contenant des *éléments* délimités par des *balises* auxquelles on peut ajouter des *attributs*. Par exemple, le fichier XML suivant représente un ensemble de livres avec leurs titres, leurs auteurs, leur date de parution :

```
<books>
  <book title="Espagnolo Facilo" date="2013">
    <author firstname="Ausias" name="Gamisans"/>
  </book>
  <book title="The Java Programming Language" date="2005">
    <author firstname="Ken" name="Arnold"/>
    <author firstname="James" name="Gosling"/>
    <author firstname="David" name="Holmes"/>
  </book>
</books>
```

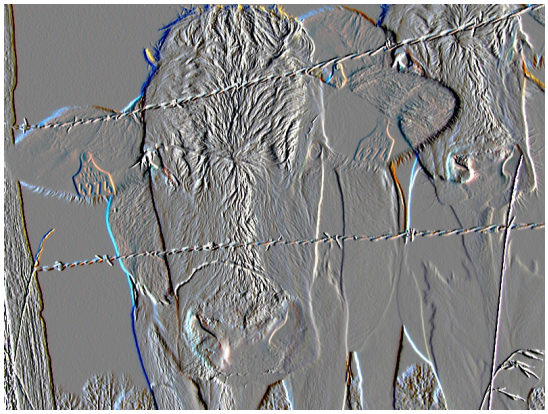
Dans cet exemple :

- l'élément représentant l'ensemble des livres est délimité par la balise `books`. On remarquera que le début d'un élément délimité par une balise `balise` est représenté par `<balise>` et la fin d'un élément par `</balise>`.
- chaque livre est délimité par la balise `book`. Cette balise possède deux attributs dont les clés sont `title` et `date` et dont les valeurs représentent respectivement le titre du livre et sa date de parution. Les valeurs des attributs ne peuvent être que des chaînes de caractères délimitées par `""`.
- pour chaque livre, un ou plusieurs éléments délimités par la balise `author` représente le ou les auteurs du livre. Comme l'élément représentant l'auteur ne contient pas d'information (elles sont en fait contenues dans les attributs), on peut fermer la balise `author` directement : `<author firstname="..."name="..."/>`

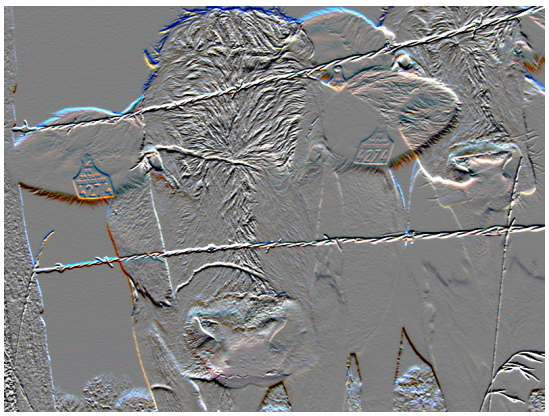
On peut remarquer que l'on aurait pu écrire ce fichier avec les mêmes informations sans utiliser d'attributs :

```
<books>
  <book>
    <title>Espagnolo Facilo</title>
    <date>2013</date>
    <author>
      <firstname>Ausias</firstname>
      <name>Gamisans</name>
    </author>
  </book>
```

(a) Image originale



(b) Utilisation du filtre H_{Dx} seul



(c) Utilisation du filtre H_{Dy} seul

(d) Utilisation des deux filtres avec $V_H(x,y) = \min(255, V_{H_{Dy}}(x,y) + V_{H_{Dy}}(x,y))$

(e) Application du filtre de Sobel

FIGURE 4 – Application de divers filtres dérivateurs

```
<book>
  <title>The Java Programming Language</title>
  <date>2005</date>
  <author>
    <firstname>Ken</firstname>
    <name>Arnold</name>
  </author>
  <author>
    <firstname>James</firstname>
```

```
      <name>Gosling</name>
    </author>
    <author>
      <firstname>David</firstname>
      <name>Holmes</name>
    </author>
  </book>
</books>
```

Attention, les deux fichiers ne sont pas équivalents d'un point de vue XML, même s'ils décrivent la même information. On privilégiera la première représentation.

4.2 Sauvegarde des associations entre étiquettes et images

Le format XML proposé pour sauvegarder les associations entre étiquettes et image utilise deux éléments principaux : `image` pour définir une image et `tag` pour définir les étiquettes.
`image` possède un attribut `path` pour définir le chemin vers l'image et `name` pour définir un nom à l'image. `tag` n'a pas d'attribut. Un exemple de fichier XML stockant les associations est présenté sur le listing 1.

Listing 1– Un fichier XML décrivant des associations entre images et étiquettes

```
1 <image path="/home/user/MesPhotos/darou.png" name="Le Darou">
2   <tag>animal</tag>
3   <tag>mythe</tag>
4   <tag>vacances</tag>
5 </image>
6 <image path="/home/user/MesPhotos/linux.png" name="Logo Linux">
7   <tag>computer</tag>
8   <tag>linux</tag>
9 </image>
```

4.3 Définition d'ontologies avec XML

Le format XML proposé pour sauvegarder des ontologies est assez simple et utilise deux éléments principaux : `ontology` pour définir une ontologie et `tag` pour définir une étiquette.
`ontology` est un élément simple permettant de regrouper les étiquettes et possède simplement un nom et une date de création comme attributs. `tag` est un élément représentant une étiquette et est caractérisé de même par deux attributs représentant le nom et la date de création de l'étiquette. Un élément `tag` peut englober d'autres éléments `tag` pour représenter la relation de taxonomie de l'ontologie. Les éléments `tag` apparaissant directement sous `ontology` sont par défaut rattachés à la racine de l'ontologie.
Par exemple, le fichier XML représentant l'ontologie de la figure 2 est représenté sur le listing 2.

Listing 2– Le fichier XML décrivant l'ontologie de la figure 2

```
1  <ontology name="mes images" date="2015-09-11 11:00:35">
2    <tag name="mythe" date="2015-09-11 11:00:36"/>
3    <tag name="animal" date="2015-09-11 11:00:36">
4      <tag name="terrestre" date="2015-09-11 11:00:36">
5        <tag name="bipede" date="2015-09-11 11:00:40"/>
6        <tag name="quadrupede" date="2015-09-11 11:00:40"/>
7      </tag>
8        <tag name="marin" date="2015-09-11 11:00:41"/>
9    </tag>
10 </ontology>
```

4.4 Définition de transformations avec XML

Le format XML proposé pour sauvegarder des séquences de transformations est également assez simple. Il s'appuie sur quelques éléments principaux : `transformations`, `symmetry`, `rotation`, `filter`. Le premier élément permet de donner un nom à la séquence de transformation et d'englober ensuite plusieurs éléments représentant des transformations « simples » :

- `symmetry` est un élément possédant un attribut `vertical` qui ne pourra utiliser que les valeurs **`true`** (si il s'agit d'une symétrie verticale) et **`false`** (s'il s'agit d'une symétrie horizontale)
- `rotation` est un élément possédant deux attributs `angle` (donnant la valeur de l'angle de rotation) et `radians` (qui prend les valeurs **`true`** ou **`false`** suivant que l'angle de rotation soit en radians ou en degrés)
- `filter` est un élément possédant deux attributs `p` et `q` pour les dimensions de la matrice de convolution et englobant des éléments `row` représentant chacun les lignes de la matrice. Chaque élément `row` possède soit un attribut `values` qui contient les valeurs des éléments de la ligne séparés par des espaces, soit de multiples éléments `value` contenant les valeurs des éléments de la ligne.

Par exemple, le listing 3 représente une séquence de transformations consistant en une symétrie verticale, une rotation de 10 degrès et l'application de deux filtres.

Listing 3– Un fichier XML décrivant une séquence

```
<transformations name="mes transformations">
  <symmetry vertical="true"/>
  <rotation angle="10" radians="false"/>
  <filter p="1" q="1">
    <row values="1.0 0.0 -1.0"/>
    <row values="2.0 0.0 -2.0"/>
    <row values="1.0 0.0 -1.0"/>
  </filter>
  <filter p="1" q="1">
    <row>
      <value>1.0</value>
      <value>2.0</value>
      <value>1.0</value>
    </row>
    <row>
      <value>0.0</value>
      <value>0.0</value>
      <value>0.0</value>
    </row>
    <row>
      <value>-1.0</value>
      <value>-2.0</value>
      <value>-1.0</value>
    </row>
</transformations>
```

Attention, ce sera à vous de vérifier la cohérence du fichier XML lors de sa lecture (les valeurs sont bien des nombres, le nombre de lignes de la matrice est cohérent avec p et q, ...).

5 Organisation

Le projet est découpé en trois lots de travail (*work packages* ou WP) :

WP1 gestion des étiquettes et de l'ontologie

WP2 application de transformations à une image

WP3 sauvegarde de l'ontologie et des transformations

Lors du projet, vous allez être répartis en équipes de 3 binômes. Chaque équipe projet sera donc composée d'un binôme B1, d'un binôme B2 et d'un binôme B3 sous la responsabilité d'un vacataire de PC (normalement le vacataire de votre PC, mais il y aura des groupes qui seront sur plusieurs PC). Chaque équipe désignera un **responsable d'équipe** qui sera le point de contact privilégié entre les encadrants et l'équipe. Le nom du responsable de chaque équipe sera communiqué au vacataire en charge de l'équipe par mail avant le **12 octobre 2015**.

Chaque équipe disposera également d'un dépôt Subversion propre. **Le responsable d'équipe sera en charge de déposer les rapports de l'équipe dans le dépôt de l'équipe**.
Pour chaque WP, trois activités sont à mettre en place : conception, implémentation et test. L'activité de test ne consiste pas ici à la création de tests unitaires, car ceux-ci doivent être réalisés par le binôme programmant le WP. Il s'agit des tests de recette permettant de valider le WP. Ces trois activités ne seront pas effectués par le même binôme pour un WP, mais « croisées » pour vous obliger à travailler ensemble (et à bien travailler, sinon vos camarades vont vous le faire savoir...) :

	conception	implantation	test
WP1	B1	B2	B3
WP2	B2	B3	B1
WP3	B3	B1	B2

6 Travail à réaliser

Le but du projet est de concevoir, réaliser et documenter une application disposant d'une interface graphique permettant à un utilisateur de :

- gérer une bibliothèque d'images en associant chaque image à une ou plusieurs étiquette ;
- construire une ontologie des étiquettes et permettre la recherche d'images sémantiquement proches via cette ontologie ;
- appliquer des transformations sur les images (rotation, symétrie, filtre) ;
- sauvegarder l'ontologie des étiquettes et les associations entre images et étiquettes au format XML et pouvoir les reconstruire en lisant les fichiers ;
- sauvegarder des transformations au format XML et pouvoir les rejouer sur une image.

Vous devrez également fournir une version en ligne de commande de l'application (i.e. démarrable non pas via l'interface graphique, mais par un appel à `java` en ligne de commande). Vous choisirez les paramètres utiles pour ces versions (fichier contenant la transformation, image à manipuler etc.) et vous utiliserez le paramètre `args` de type `String[]` de la méthode `main` pour gérer les arguments passés en ligne de commande. Vous pourrez également utiliser les bibliothèques libres Commons CLI [1] ou JCommander [3].

Attention, il ne s'agit pas de proposer une interface textuelle à votre application où l'utilisateur peut créer des choses, mais de pouvoir appliquer directement une transformation à partir de fichiers la paramétrant.
On devra pouvoir par exemple appliquer une transformation définie dans un fichier `trans.xml` sur une image `cow.png` pour obtenir une image `cow-filter.png` de la façon suivante :

```
java photowl -transform trans.xml -input cow.png -output cow-filter.png
```

On devra également pouvoir ajouter une étiquette sur un ensemble de fichiers de la façon suivante :

```
java photowl -add-tag "animal" cow.png cow-filter.png
```

Vous êtes également libres de rajouter des extensions. **Ce n'est pas obligatoire et ces extensions ne seront prises en compte que si la version de base du logiciel est fonctionnelle**. On pourra par exemple :

- pouvoir faire des requêtes complexes sur les étiquettes. Par exemple, trouver toutes les images avec l'étiquette « animal » et l'étiquette « mythe » mais pas l'étiquette « vacances »
- pouvoir sauvegarder des transformations complexes et pas simplement des séquences de transformations (comme par exemple le filtre de Sobel)
- éditer les ontologies via l'IHM par une représentation sous forme de graphe de l'ontologie et du *drag-and-drop* (voir par exemple l'API JGraphX [8])
- ...

Les exigences permettant d'évaluer votre projet sont définies dans ce qui suit.

6.1 Exigences concernant l'analyse du besoin et la conception

[E1] l'analyse du besoin se fera en utilisant des diagrammes de cas d'utilisation et les acteurs seront clairement identifiés (cf. section A).

[E2] la conception du logiciel sera présentée sur un diagramme de classes UML.

[E3] la conception du logiciel permettra de répondre aux besoins exprimés dans la section 1.

[E4] le logiciel sera le plus extensible possible : on pourra facilement ajouter d'autres formats de fichier pour sauvegarder les ontologies ou les transformations, construire des transformations complexes, faire des requêtes complexes sur les étiquettes etc.

[E5] le logiciel sera le plus réutilisable possible : on pourra facilement réutiliser une partie des composants logiciels.

[E6] la documentation plus précise des classes se fera au moyen de documentation Javadoc sur les squelettes des classes.

6.2 Exigences concernant la validation et la vérification du logiciel

[E7] les scénarios de validation de l'application seront présentés au travers de diagrammes de séquence.

[E8] chaque méthode (autre que les méthodes triviales comme les accesseurs et modifieurs simples) sera testée via des tests JUnit.

[E9] chaque classe possédera sa propre classe de test JUnit. Une documentation javadoc minimale sera faite dans la classe de test pour expliquer le but de chaque test.

6.3 Exigences concernant l'implantation du logiciel

[E10] le code de l'application devra pouvoir fonctionner sur une machine du SI sans utiliser nécessairement Eclipse.

[E11] les sources de l'application en elle-même seront disponibles dans un dossier `src` situé à la racine du projet.

[E12] les sources des tests JUnit seront disponibles dans un dossier `tests` situé à la racine du projet.

[E13] les éventuels fichiers de configuration ou d'exemples nécessaires pour des tests et des démonstrations seront placés dans un répertoire `resources` situé à la racine du projet.

[E14] en cas d'utilisation d'une bibliothèque extérieure pour votre projet, celle-ci devra être incluse sous forme d'une archive JAR placée dans le répertoire `lib` situé à la racine du projet. Les licenses et droits d'auteurs éventuels seront respectés et le rapport final fera mention de l'utilisation de cette bibliothèque.

[E15] vous n'avez pas le droit d'utiliser une bibliothèque extérieure au JDK pour les points suivants :

- application d'une transformation sur une image
- gestion basique de l'arbre représentant l'ontologie
- sauvegarde et chargement des fichiers XML

[E16] on devra trouver à la racine de votre projet un fichier texte `README.txt` expliquant comment lancer votre application et la configurer.

[E17] le code fourni sera documenté via Javadoc et indenté correctement. Les contributeurs à une classe seront clairement identifiés via le tag `@author`.

[E18] le code fourni devra correspondre au diagramme de classes de conception final.

[E19] tous les projets seront vérifiés grâce à l'utilitaire JPlag [7].

[E20] chaque classe à développer sera sous la charge d'un élève clairement identifié. Il aura en particulier en charge la gestion des *commits* pour cette classe.

[E21] les exigences **minimales** pour la partie implantation sont les suivantes :

- ajout d'étiquettes sur une image
- gestion des étiquettes via une ontologie et recherche des images sémantiquement proches via l'ontologie
- application de transformations simples (symétrie, rotation, filtre défini par un noyau)
- sauvegarde et chargement des associations images-étiquettes, de l'ontologie et de transformation au format XML

Le respect de ces exigences minimales et des points précédents garantira une note sur la partie implantation correspondant à 80% de la note maximale sur cette partie.

6.4 Exigences concernant la gestion de projet

[E22] un responsable pour chaque équipe sera désigné lors de la première séance de travail sur le projet. Ce responsable sera le point de contact entre le vacataire et l'équipe et sera en charge de déposer les rapports sur LMS.

[E23] le responsable de chaque équipe enverra tous les vendredis (sauf en période de vacances scolaires) au vacataire responsable de son équipe un court mail décrivant les travaux effectués sur le projet durant la semaine.

[E24] lors de la phase d'implantation, des *commits* réguliers seront effectués par les membres de l'équipe. Chaque commit aura un message permettant d'identifier rapidement quel a été le but de l'ajout ou de la modification de code.

[E25] le professeur responsable du cours se réserve le droit de modifier ou compléter le sujet du présent projet de façon unilatérale et parfaitement arbitraire.

6.5 Exigences concernant les livrables

[E26] tous les rapports fournis seront **au format PDF**. Votre vacataire peut se réserver le droit de ne pas corriger votre rapport si ce n'est pas le cas.

[E27] tous les rapports seront déposés dans le dossier `report` du dépôt Subversion de l'équipe dont l'URL est https://eduforge.isae.fr/repos/IN201/GPE/tNUM (où `GPE` est le nom de la PC de rattachement de l'équipe en majuscules et `NUM` son numéro).

[E28] tout le code source de l'application sera déposé sur le dépôt Subversion. Seul le code déposé sur le dépôt sera utilisé pour l'évaluation du projet.

[E29] pour chaque jour de retard, 0.5 points seront retirés sur la note finale.

[E30] le premier rapport à fournir comportera l'analyse du problème et une première conception d'une architecture logicielle permettant de le résoudre. Le nom du rapport sera `rapport-analyse-conception-tXX` ou `XX` est votre numéro d'équipe. Le rapport ne devra pas excéder 15 pages. Le plan du rapport sera le suivant :

1. analyse du problème avec utilisation de use cases ;
2. proposition d'une solution : un diagramme de classes et diagrammes de séquence pour les interactions complexes entre classes ;
3. plan de test : comment valider l'application, quels sont les points durs à tester ;
4. plan de développement : qui fait quoi, en particulier comment seront réparties les responsabilités de chacun en ce qui concerne l'implantation des classes.

Ce rapport est à déposer avant le **19 novembre 2015 23h00**. Dans le même temps, les squelettes des classes Java documentés correspondant au diagramme de classe seront déposés sur le dépôt Subversion.

[E31] le deuxième rapport à fournir est le rapport final. Le nom du rapport sera `rapport-tXX` ou `XX` est votre numéro d'équipe. Le rapport ne devra pas excéder 20 pages. Le plan du rapport sera le suivant :

1. éventuellement, retour sur l'analyse en cas d'erreurs lors du premier rapport ;
2. éventuellement, retour sur le diagramme de classes et/ou les diagrammes de séquence si ceux-ci ont changé depuis le premier rapport ;
3. ce qui fonctionne ;
4. ce qui ne fonctionne pas et éventuellement les causes de ce non-fonctionnement ;
5. conclusion.

Ce rapport est à déposer avant le **11 janvier 2016 23h00**. Dans le même temps, votre vacataire récupérera le code de votre projet depuis votre dépôt Subversion.

[E32] Le code récupéré sur votre dépôt le **15 janvier 2016** sera considéré comme le code définitif de votre BE.

6.6 Exigences concernant la présentation orale

[E33] une présentation orale par équipe de votre projet se déroulera le **15 janvier 2016**. Cette présentation dure 25 minutes et consistera en :

- une ou deux planches présentant ce qui fonctionne, ce qui ne fonctionne pas, les difficultés rencontrées, un bilan personnel du projet ;
- une démonstration de votre logiciel ;

– prévoir également une ou deux planches avec l'architecture de l'application pour pouvoir répondre aux questions éventuelles de votre vacataire.

[E34] un seul élève présentera les transparents lors de l'oral. **Cet élève sera tiré au hasard au moment de la soutenance par le vacataire.**

[E35] vous devez envoyer votre présentation à votre vacataire au format PDF le **13 janvier 2016**.

[E36] le délégué de chaque PC est en charge d'organiser l'ordre de passage des soutenances de sa PC et de l'envoyer le **13 janvier 2016** à son vacataire. On essayera si possible de partager un ordinateur pour éviter de perdre du temps entre chaque présentation.

[E37] le responsable de chaque équipe doit envoyer par mail à son vacataire la répartition des points de travail avant le **13 janvier 2016** (cf. section 11).

7 Récapitulatif des dates importantes

19 novembre 2015 23h00	Analyse et proposition d'une solution de conception
11 janvier 2016 23h00	Réalisation et rapport final
13 janvier 2016	Envoi présentation et corrections éventuelles
13 janvier 2016	Envoi planning (**délégués**)
15 janvier 2016	Oral

8 Logiciel de modélisation UML

Pour faire vos diagrammes UML, vous trouverez de (trop) nombreux outils sur internet. Nous vous conseillons d'utiliser ArgoUML [2], un outil permettant de dessiner facilement des diagrammes UML et de générer le code Java correspondant.

9 Documentation complémentaire

Ce document n'est pas complet. Des documents complémentaires, en particulier des compléments pour la partie implantation, seront disponibles via le site `http://www.tofgarion.net/lectures/IN201`. L'utilisation d'API disponibles et intéressantes à utiliser pour le projet sera illustrée par des exemples sur le site.

10 Rôle des vacataires

Chaque vacataire jouera le rôle du client qui a commandé l'application à développer. Vous pourrez bien sûr contacter par mail votre vacataire ou C. Garion pour des demandes de renseignements précises (**mettre `[IN201]` dans le sujet du mail ainsi qu'un résumé rapide du problème ou de la demande**).

11 Notation

La table 1 présente un barème indicatif pour la notation du projet. La notation se fera par équipe, avec toutefois une pondération possible par binôme si le travail sur une activité sur un WP particulier n'a pas été correctement fait. On peut vous retirer des points, en particulier sur l'oral si celui-ci est trop médiocre.
La partie « Évaluation travail » sur 2 points de la table 1 est une partie de notation que vous devez évaluer. Chaque équipe dispose d'un ensemble de $1,5 \times n$ points, n correspondant au nombre de membres de l'équipe. À vous de répartir ces points sur les différents membres de l'équipe en fonction du travail effectué (un étudiant peut bien sûr avoir plus de 2 points s'il a beaucoup travaillé par rapport aux autres membres de l'équipe).
Le responsable de chaque équipe doit envoyer par mail à son vacataire la répartition des points avant l'oral (cf. section 6.6).
Les parties correspondant à la remise de deux rapports (intermédiaire et final) seront pondérées par les deux rapports.
Vous remarquerez que les parties sont assez équilibrées, ce n'est pas seulement la programmation qui compte...

Partie	Détail	Points	Commentaires
Analyse	cas d'utilisation	1,0	diagramme de cas d'utilisation UML et identification correcte des acteurs et du système développé
	total	1,0	
Conception	diagramme de classes	1,5	pertinence du diagramme de classes par rapport au problème, respect des notations UML
	utilisation paquetages	0,5	séparation des différentes parties de l'application en paquetages
	diagrammes de séquences	1,5	diagrammes de séquences permettant d'expliquer les interactions entre les principales classes du système
	réutilisatibilté, extensibilité	1,5	conception d'un système extensible et réutilisable facilement
	total	5,0	
Programmation	exécution sans erreurs	1,0	
	correspondance conception/-programmation	2,0	code produit correspondant à la conception proposée
	qualité du code produit	2,0	
	code ne compilant pas	−5,0	pénalité
	exécution avec erreurs	−2,0	pénalité
	total	5,0	
Tests	tests de validation	1,0	scénarios de test permettant de valider le système du point de vue de l'utilisateur
	tests unitaires JUnit	2,0	tests unitaires sur les classes développées
	exécution avec erreurs	−2,0	pénalité
	total	3,0	
Subversion	utilisation correcte de SVN	1,0	
	total	1,0	
Documentation	rapports	1,0	qualité du rapport écrit
	oral	1,0	qualité des transparents, pertinence des réponses aux questions posées
	javadoc	1,0	
	javadoc manquante	−1,0	pénalité
	total	3,0	
Évaluation travail	évaluation	2,0	cf. commentaires
	total	2,0	

TABLE 1 – Notation détaillée du projet

12 Outils utilisés

Plusieurs outils seront utilisés automatiquement sur votre dépôt afin de permettre une meilleure évaluation de votre projet. Ces outils seront « passés » sur le dépôt au moins une fois par semaine à partir de la date de remise du rapport de conception. Les rapports produits par les outils sont au format HTML et placés dans votre dépôt. Les outils utilisés sont les suivants :

- le compilateur Java du SDK qui va compiler **tous** vos fichiers source situés sous `src` et `tests`. Si la compilation produit une erreur, les autres outils ne seront pas utilisés. Ne laissez donc pas trainer dans votre dépôt des fichiers qui ne compilent pas !
- l'utilitaire `javadoc` du SDK qui va produire la documentation javadoc de vos sources. Cette documentation sera placée dans le répertoire `doc` situé à la racine de votre dépôt. Là encore, attention si vous avez des erreurs (et pas des *warnings*), les autres outils ne seront pas utilisés.
- les classes de tests JUnit seront executées et le résultat de ces tests sera placé dans le répertoire `results/junit`. Pour qu'une classe soit considérée comme une classe de tests, il faut que son nom finisse obligatoirement par `Test`. Si vous avez des tests qui échouent, les outils suivants ne seront pas utilisés.
- Checkstyle [13], un outil permettant de vérifier un certain nombre de règles de « bon codage » (noms des classes commençant par une majuscule, documentation javadoc présente etc.). Ne vous inquiétez pas si vous avez beaucoup d'erreurs, en particulier sur la longueur des lignes, mais cela devrait vous permettre d'améliorer la qualité de votre code. Les résultats de Checkstyle sont placés dans le répertoire `results/checkstyle`.
- FindBugs [12], un outil permettant de trouver d'éventuels erreurs dans votre code (utilisation de références **`null`** etc.). Le résultat produit est placé dans le répertoire `results/findbugs` et est assez exhaustif.
- SvnStat [14], un outil permettant d'obtenir des statistiques sur l'utilisation que votre équipe fait de SVN (pourcentage de *commits* pour chaque membre de l'équipe etc.)

Attention, pour que les outils fonctionnent, il faut que vous respectiez impérativement la structure du dépôt comme précisé dans la section 6.3 (en particulier, les éventuelles bibliothèques extérieures doivent être placées dans le répertoire `lib`).

13 Conseils

Quelques conseils pour ce projet, qui peut paraître au départ assez difficile :

- **organisez-vous en équipe**. En particulier, prévoyez des points de rendez-vous réguliers, définissez clairement les tâches de chacun etc.
- **utilisez des interfaces** pour pouvoir vous abstraire d'un WP que vous ne coderez pas par exemple ;
- **documentez correctement** tout ce que vous faites, car de toute façon, c'est un binôme de votre équipe qui utilisera votre travail...
- soyez **méthodiques**, par exemple ne développez pas des classes à tout va sans les tester. La meilleure solution est d'écrire tout d'abord les tests (avec JUnit par exemple) avant d'écrire les méthodes des classes ;
- développez des classes « génériques » en particulier pour les classes comportant des parties algorithmiques délicates. Même si les premiers algorithmes sont simplifiés, vous pourrez par spécialisation les raffiner par la suite ;
- travaillez réguliérement : si vous attaquez les grosses phases une semaine avant la remise du rapport, vous n'y arriverez pas ;
- soyez synthétiques dans les rapports : il faut que l'essentiel y soit. N'ayez pas peur d'être concis (si c'est clair et complet !). Par exemple, ce n'est pas parce que vous avez 25 cas d'utilisation que votre analyse est bonne...
- travaillez en équipe, il y a du travail pour tout le monde...
- séparez si possible vos classes en paquetages : cela vous permettra de vous partager le travail plus facilement et de faciliter la compréhension de l'application ;
- pour la partie Java, vous pouvez trouver énormément de documentation sur le site d'Oracle : partie documentation de l'API [9] et tutoriels [10] ;
- nettoyez votre dépôt avant de remettre la version finale : vérifiez les résultats produits par les outils décrits dans la section 12, vérifiez que vous n'avez pas de classe inutile qui ne compile pas dans votre dépôt, enlevez les tests qui échouent à cause d'un `fail`, n'oubliez pas votre fichier `README.txt` etc.

– enfin, l'avantage de disposer de vos classes et d'un ordinateur est de pouvoir les comparer de façon automatique d'un projet à l'autre [7]... Ne copiez pas (cela ne veut pas forcément dire de ne pas réfléchir ensemble !), c'est très facilement identifiable et le « camouflage » de sources copiées n'est pas si évident que cela.

A Diagrammes de cas d'utilisation

Dans le cycle de vie d'un produit, la première phase est l'identification des besoins du client. Cette phase est primordiale, car tout le développement du produit va en découler.
L'onjectif d'un produit est évidemment de rendre service à ses utilisateurs. Il faut donc dans un premier temps bien comprendre les désirs et besoins des futurs utilisateurs du système. Dans le cadre d'un système automatisé ou informatique, ces utilisateurs ne sont pas forcément humains : il peut s'agir par exemple d'une base de données devant utiliser une application. Cette phase d'analyse des besoins est très difficile et se fait en plusieurs itérations (« allers-retours » entre maitrise d'œuvre et maitrise d'ouvrage). Il est nécessaire d'utiliser un langage commun pour éviter les ambiguïtés.
Une fois que l'on connaît les besoins des utilisateurs, il faut :

– communiquer ces besoins à l'ensemble des personnes impliquées dans le projet (parties prenantes) ;
– concevoir une implantation opérationnelle y correspondant ;
– vérifier la pleine satisfaction de ces besoins.

Il y a donc nécessité d'utiliser une notation rigoureuse permettant de représenter facilement les besoins de l'utilisateur, de communiquer ces besoins et de préparer la validation du système. Nous allons présenter ici l'utilisation de deux diagrammmes d'UML (*Unified Modeling Language*) [5, 4, 6].
Nous allons utiliser dans ce qui suit un exemple d'un distributeur de billets de banque.

A.1 Diagrammes de cas d'utilisation : définition

On peut traduire les exigences des utilisateurs/clients par des fonctionnalités/services du système. Mais souvent, on oublie alors des fonctionnalités importantes ou on en propose des superflues. On va s'attacher à determiner quel doit être le comportement du système pour **chaque utilisateur**. Les différentes façons d'utiliser le système vont être représentées par des **cas d'utilisation**. Ces cas d'utilisations représentent les interactions possibles entre le système et les éléments qui se trouvent à l'extérieur du système : les **acteurs**.

A.2 Acteurs

Il faut dans un premier temps délimiter le système et identifier les différentes entités externes qui interagiront avec lui. Ces entités externes peuvent être des personnes, comme les utilisateurs, mais également d'autres systèmes ou du matériel. On utilise la notion d'acteur pour représenter ces entités externes

Définition. Un acteur représente (c'est une abstraction, i.e. un « *classifier* »)) une entité externe (personne, autre système, ...) jouant un rôle et interagissant avec le système considéré.

La figure 5 représente dans le formalisme UML l'acteur « client » du système distributeur de billets.

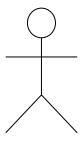


FIGURE 5 – Représentation UML d'un acteur

On peut **spécialiser** un acteur pour définir un acteur ayant un rôle plus précis qu'un acteur déjà présent. La figure 6 représente la spécialisation de l'acteur « client » en acteur « client d'affaires » qui est un client particulier.

A.3 Cas d'utilisation

Du point de vue d'un acteur donné, un cas d'utilisation représente un service, c'est-à-dire une fonction du système qui a une valeur pour l'acteur : calculer un résultat, modifier un état du système etc.

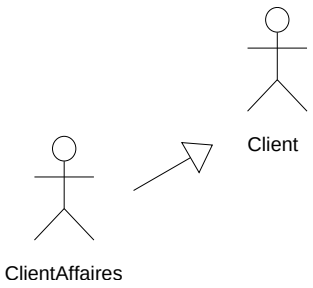


FIGURE 6 – Spécialisation d'un acteur

Définition. Un cas d'utilisation décrit un ensemble de séquences d'actions, y compris des variantes, qu'un système exécute pour produire un résultat tangible pour un acteur. On parle alors de comportement émergent du système (*emergent behaviour*).

Un cas d'utilisation est créé sans spécifier la manière dont il sera implémenté. Par exemple, on peut préciser le comportement d'un distributeur de billets en déterminant la manière dont les utilisateurs dialoguent avec le système sans savoir ce qu'il se passe à l'intérieur du distributeur. Un cas d'utilisation représente donc un ensemble de **scénarios** d'interactions entre le système et des acteurs.
La figure 7 représente le cas d'utilisation « retirer argent ». C'est l'ensemble des scénarios d'utilisation du distributeur de billets correspondant à un retrait d'argent.

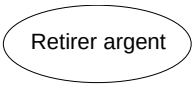


FIGURE 7 – Cas d'utilisation « retirer argent »

A.4 Diagramme de cas d'utilisation

Les acteurs interagissent avec le système via des cas d'utilisation. On représente ces interactions par des **associations** entre acteurs et cas d'utilisation dans un diagramme de cas d'utilisation. Ce diagramme permet également de délimiter le système. Un diagramme d'utilisation possible du distributeur de billets est présenté sur la figure 8. Par exemple, pour retirer de l'argent, les acteurs « client », « banque » et « carte banquaire » interagissent avec le système via le cas d'utilisation « retirer argent ».

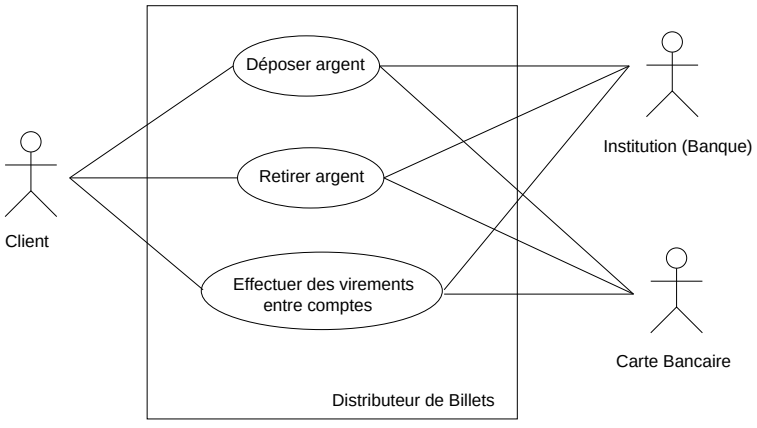


FIGURE 8 – Diagramme d'utilisation du distributeur de billets

A.5 Cas d'utilisation et flots d'événements

Un cas d'utilisation décrit ce que fait le système en prenant résolument un point de vue externe (comportement perceptible par les acteurs). On peut préciser les le comportement d'un cas d'utilisation en décrivant les flots d'événements entre

le système et les acteurs concernés par le cas d'utilisation. Cette description peut être faite via un texte structuré. Une manière systématique de décrire les cas d'utilisation peut être la suivante :

- les **préconditions** du cas d'utilisation qui représentent le contexte avant utilisation du système ;
- les **postconditions** du cas d'utilisation qui représentent l'état du système après le cas d'utilisation ;
- les **échanges** qui sont une suite d'étapes découpées en phase ;
- chaque étape correspond à un événement transmis par un acteur vers le système ou par le système vers un acteur ;
- les **exceptions** qui décrivent les échanges pour les cas exceptionnels :
  - [Serveur non disponible] : l'utilisateur peut appeler l'administrateur système au numéro ...

Par exemple, pour le distributeur de billets, le cas d'utilisation « retirer argent » peut être décrit de la façon suivante :

- **cas d'utilisation** « Retirer argent »
- **préconditions**
  - distributeur en état de marche, dans l'état *init*
- **postconditions**
  - distributeur dans l'état *init*
  - coffre du distributeur débité
  - transaction effectuée
- **étapes**
  - phase d'authentification
    1. le Client insére sa carte [Carte Illisible]
    2. le Distributeur affiche « saisir code »
    3. ...
  - phase de transaction
    1. le Client choisit la transaction « Retrait »
    2. ...
- **exceptions**
  - [Carte Illisible] la carte est ejectée et le distributeur retourne dans l'état *init*

B Diagrammes de séquence

Un cas d'utilisation représente un ensemble de scénarios d'interactions entre le système et les acteurs. Il faut pouvoir représenter ces scénarios. Une première façon de le faire est d'utiliser la langue naturelle. Par exemple, voici un scénario du cas d'utilisation « retirer argent » :

1. le client insère sa carte dans le lecteur ;
2. le distributeur vérifie la validité de la carte ;
3. le distributeur demande le code du client ;
4. le client fournit son code ;
5. le distributeur vérifie le code en utilisant la carte ;
6. le code est valide ;
7. le distributeur demande au client le montant désiré ;
8. le client demande 20 euros ;
9. le distributeur vérifie que le client est autorisé à retirer 20 euros auprès de la banque ;
10. la banque autorise le retrait ;
11. le distributeur fournit 20 euros au client et prévient la banque de la transaction.

On peut utiliser les **diagrammes de séquence** que nous avons vus en cours pour représenter les interactions entre les acteurs et le système de façon chronologique. Le scénario présenté précédemment est représenté sur la figure 9 (on a omis les durées de traitement).
On peut raffiner le diagramme de séquence en commençant à introduire des sous-systèmes dans les diagrammes. Ce n'est pas systématique et ce n'est surtout pas évident : il faut disposer de connaissances « métier » ou de parties du système déjà développées pour pouvoir effectuer ce raffinement. Un raffinement possible du diagramme de séquence précédent est présenté sur la figure 10

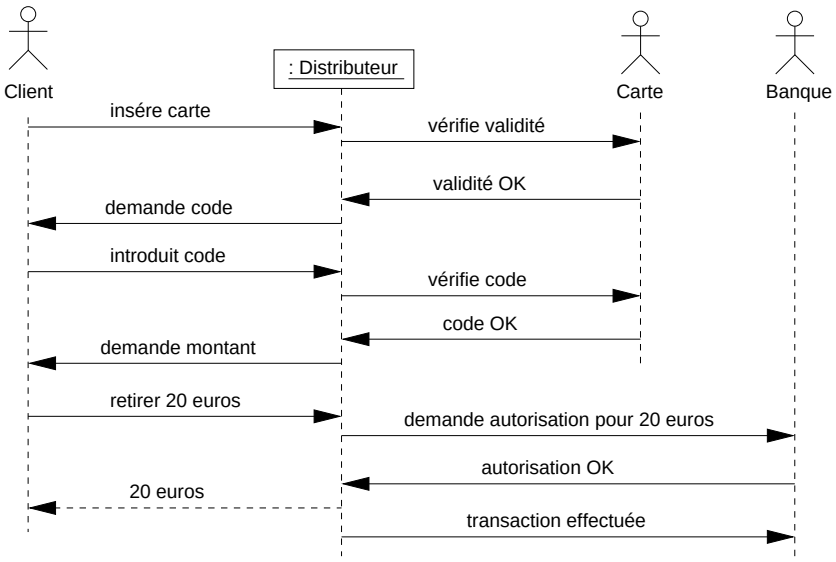


FIGURE 9 – Diagramme de séquence d'un scénario de « retirer argent »

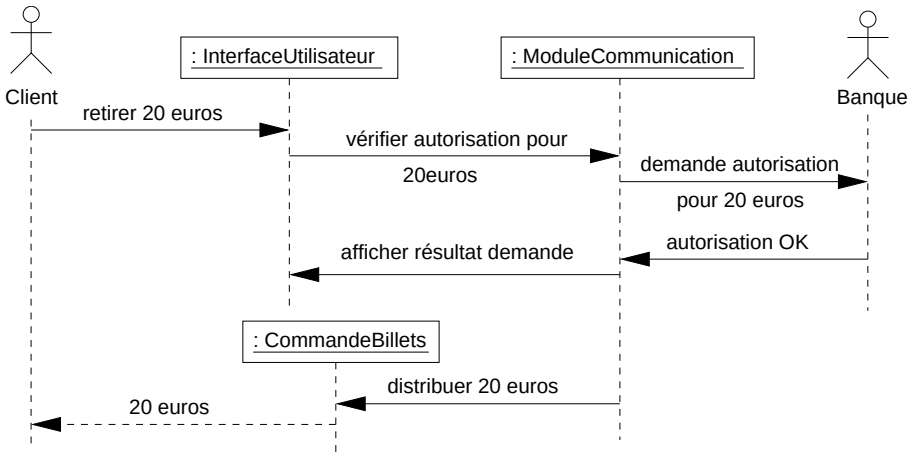


FIGURE 10 – Raffinement d'un diagramme de séquence

Références

[1] Apache Commons. *Commons CLI*. 2012. URL : http://commons.apache.org/cli/.

[2] ArgoUML contributors. *ArgoUML*. 2013. URL : http://http://argouml.tigris.org/.

[3] Cédric Beust. *JCommander*. 2015. URL : https://github.com/cbeust/jcommander.

[4] G. Booch, J. Rumbaugh et I. Jacobson. *The Unified Modeling Language reference manual*. Addison-Wesley, 2004.

[5] G. Booch, J. Rumbaugh et I. Jacobson. *The Unified Modeling Language user guide*. Addison-Wesley, 1998.

[6] M. Fowler. *UML distilled*. Addison-Wesley, 2003.

[7] Institute for Program Structures and Data Organization. *JPlag*. Fakultät für Informatik, Karlsruhe Institut für Technologie. 2013. URL : https://jplag.ipd.kit.edu/.

[8] JGraph Ltd. *JGraphX*. 2105. URL : https://github.com/jgraph/jgraphx.

[9] Oracle. *Java API specifications*. 2013. URL : http://docs.oracle.com/javase/7/docs/api/index.html.

[10] Oracle. *The Java Tutorials*. 2013. URL : http://download.oracle.com/javase/tutorial/.

[11] *PNG (Portable Network Graphics) home page*. URL : http://www.libpng.org/pub/png/.

[12] Bill Pugh et Andrey Loskutov. *FindBugs 3.0*. 2014. URL : http://findbugs.sourceforge.net/.

[13] The Checkstyle development team. *Checkstyle 5.7*. 2014. URL : http://checkstyle.sourceforge.net/.

[14] The SvnStat development team. *SvnStat*. 2014. URL : http://svnstat.sourceforge.net/.

[15] Wikipedia. *Dahu*. URL : http://fr.wikipedia.org/wiki/Dahu.

[16] Wikipedia. *IPTC Information Interchange Model*. URL : http://en.wikipedia.org/wiki/IPTC_Information_Interchange_Model.

[17] Wikipedia. *Ontology*. URL : http://en.wikipedia.org/wiki/Ontology_(computer_science).

[18] Wikipedia. *Portable Network Graphics*. URL : http://en.wikipedia.org/wiki/Portable_Network_Graphics.

[19] Wikipedia. *Semantic web*. URL : http://en.wikipedia.org/wiki/Semantic_web.

[20] Wikipedia contributors. *XML*. 2104. URL : http://en.wikipedia.org/wiki/XML.

[21] Z. Wu et M. Palmer. « Verb semantics and lexical selection ». In : *Proceedings of the 32nd annual meetings of the Association for Computational Linguistics*. 1994, p. 133–138.

[22] Emmanuel Zenou. *Vision par ordinateur, une introduction*. Polycopié ISAE-SUPAERO. 2015.

License CC BY-NC-SA 3.0

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported license (CC BY-NC-SA 3.0)

You are free to Share (copy, distribute and transmite) and to Remix (adapt) this work under the following conditions:

Attribution – You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Noncommercial – You may not use this work for commercial purposes.

Share Alike – If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

See http://creativecommons.org/licenses/by-nc-sa/3.0/ for more details.