IN201 : Projet – RobOwl

Author : Christophe Garion <garion@isae.fr>
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 à la création d'une application permettant de simuler une équipe de robots pompiers.

1 Présentation du projet

L'organisation de secours lors d'une catastrophe majeure est un problème important. En particulier, les acteurs présents sur le lieu de la catastrophe peuvent être de différentes nationalités, peuvent être des humains, des robots etc. On cherche donc actuellement à développer des moyens informatiques permettant de faciliter et de guider le travail des sauveteurs sur le terrain (voir par exemple [1]).
Dans ce travail, nous allons plus modestement simuler le travail d'un groupe de robots pompiers (cf. figure 1) chargés d'éteindre un ou plusieurs incendies. Les caractéristiques du projet sont détaillées dans ce qui suit.

FIGURE 1 – Un robot pompier araignée OLE développé par l'université de Magdeburg-Stendal

1.1 La carte

La carte du terrain d'intervention est représentée par un *graphe* dont les nœuds représentent des points particuliers sur la carte et les arcs les chemins possibles entre les nœuds.
Les nœuds relient par une longitude et une latitude et il existe différents types de nœuds : des nœuds de base permettant de représenter un chemin particulier, des nœuds représentant des stations dans lesquelles les robots peuvent recharger leurs batteries et leur réservoir d'eau, des nœuds représentant des incendies.
Les arcs sont caractérisés par deux nœuds, et une valeur représentant la distance entre les deux nœuds. Il existe également plusieurs types d'arc représentant le type de terrain à traverser : terrain plat, terrain escarpé, terrain inondé etc.
Un exemple de carte vous est présenté sur la figure 2.

1.2 Créer une carte à partir de données OpenStreeMap

On va créer les graphes représentant les cartes à partir de données réelles. Le projet OpenStreetMap [2] est un projet collaboratif ayant pour but de créer une carte éditable du monde. On peut en particulier sélectionner une région particulière

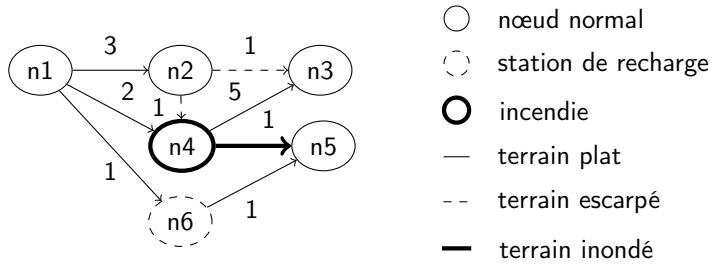

FIGURE 2 – Une carte minimale représentée sous forme de graphe

sur le globe et exporter les données OpenStreetMap la concernant (bâtiments, routes etc.) au format OSM, un format dérivé de XML[3].
XML 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.
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 libre 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>
  <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>
```

Dans le format OSM, on trouvera trois types d'éléments (cf. http://wiki.openstreetmap.org/wiki/OSM_XML) :

- `node` qui représente un nœud contenant l'identifiant du nœud, ses coordonnées GPS ainsi que d'autres informations ;
- `way` qui permet de lier des nœuds entre eux pour faire des chemins ;
- `relations` (que nous n'utiliserons pas) qui permet de représenter des informations géographiques avec des nœuds et de chemins pour représenter par exemple des rues.

Le format OSM est assez complet et nous vous donnerons un programme permettant de simplifier une représentation OSM en ne conservant que ce qui est nécessaire pour le projet.
On devra pouvoir également importer des cartes sous d'autres formats, comme par exemple un format texte simple qui pourrait être le suivant :

```
begin_nodes
764715284:43.6139609,1.4673366
191045672:43.6154648,1.4651711
191045673:43.6144569,1.4655994
...
end_nodes
begin_edges
764715284 -> 191045672
764715284 -> 191045673
...
end_edges
```

Attention, les coordonnées GPS des nœuds ne permettent pas de calculer la distance entre ces deux points directement, il faut utiliser une projection de Lambert par exemple.
On pourrait également supposer que la topologie de la carte peut évoluer en cours de simulation. Par exemple, des chemins pourraient devenir impraticables lorsqu'un arbre tombe sur une route.

1.3 Les robots

Les robots sont des « pompiers élémentaires » qui peuvent se déplacer, éteindre un feu et qui connaissent leur position à tout instant. Ils sont également capables de calculer le plus court chemin qu'ils peuvent trouver jusqu'à un point de la carte (en utilisant le graphe représentant la carte). Dès que le robot est sur un nœud correspond à un incendie, il va l'éteindre et cela va prendre un certain temps.
Pour pouvoir déterminer quel est le plus court chemin d'un robot aux différents incendies sur la carte, on peut utiliser un algorithme de *pathfinding*. Ces algorithmes permettent de trouver le plus court chemin dans un graphe. De nombreux exemples se trouvent dans [4], disponible à la bibliothèque et vous pourrez trouver sur Internet de nombreuses applications et exemples (algorithme de Dijkstra, algorithme A*, ...). On essayera d'avoir une application modulaire, qui permette facilement d'implanter un algorithme particulier sans devoir changer toute l'application.
On pourra considérer plusieurs types de robots : robots à chenilles, robots à pattes... Les robots à chenilles pourraient par exemple être incapables de traverser les chemins escarpés, et les robots à pattes pourraient être incapables de traverser les chemins inondé.

1.4 La stratégie de distribution

Pour pouvoir affecter aux robots les tâches qu'ils auront à effectuer (i.e. quels incendies ils devront aller éteindre), nous nous appuyons sur le protocole Contract Net [5]. Dans ce protocole, un manager envoie une demande à plusieurs robots. Chaque robot doit répondre dans un certain temps : soit il refuse la demande, soit il envoie une proposition. Le manager sélectionne ensuite les différentes tâches à affecter aux robots. Pour plus de détails, voir [5].
Dans notre cas, voici un exemple de déroulement du protocole :

1. le manager propose aux robots les incendies à éteindre ;
2. les robots occupés à éteindre un incendie refusent la proposition. Les autres robots calculent leur plus court chemin et renvoient cette valeur au manager ;
3. pour chaque incendie, le manager sélectionne le robot le plus proche pour aller l'éteindre. En cas d'« égalité », une sélection arbitraire pourra être utilisée [1] ;
4. s'il reste des incendies non affectés, le manager attend un certain laps de temps et repropose les incendies restant.

1.5 La simulation

L'objectif de l'application à développer est de fournir un simulateur permettant de visualiser la carte, les incendies et le déplacement des robots. Il y a donc une gestion du temps à réaliser de façon explicite. Deux solutions sont possibles :

- soit on crée un « processus » indépendant pour chaque robot et le déroulement de la simulation pour chaque robot se déroule en parallèle (possible en Java grâce aux *threads*, mais non abordé en cours) ;
- soit on discrétise le problème.

Avec les connaissances que vous avez, la seconde solution est la plus facile à mettre en œuvre. Nous vous proposons donc de discrétiser le problème en utilisant un *simulateur à événements discrets*. Ce simulateur possède une liste ordonnée d'événements datés. A chaque événement est associée une opération à réaliser. Par exemple, un robot se déplaçant peut créer un événement à chaque fois qu'il arrive sur un nœud pour signaler au simulateur qu'il sera dans une autre position à une certaine date. Le simulateur parcourt la liste d'événements et exécute les opérations associées au fur et à mesure.

1.6 L'interface graphique

On devra fournir une interface graphique minimale permettant de voir la simulation se dérouler. Pour cela, on affichera la carte OpenStreetMap utilisée en fond de l'application et on fera se déplacer les robots d'un nœud à un autre. On permettra également à l'utilisateur de créer des incendies pendant la simulation.
Des détails concernant la réalisation d'un interface graphique avec la bibliothèque Swing vous seront présentés lors du cours IN201 et sur le site du cours. Vous pouvez d'ors et déjà parcourir les composants disponibles sur [6].

2 Organisation

Le projet peut être clairement séparé en trois lots de travail (*work packages* ou WP) :

WP1 l'algorithme de *pathfinding*
WP2 les robots, le manager et la simulation
WP3 la création de cartes à partir de données OpenStreetMap

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 à Martine Marlot par mail avant le **24 octobre 2012**. **Le responsable d'équipe sera en charge de déposer les rapports de l'équipe sur LMS**.
Chaque équipe disposera également d'un dépôt Subversion propre.
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 effectuées 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...) :

1. Les plus courageux pourront tenter d'implanter une moulinette de sélection minimisant l'insatisfaction des robots...

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

3 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 :

- lancer une simulation de robots pompiers à partir d'un fichier OSM, texte ou autre format et d'un fichier contenant les positions initiales et caractéristiques des robots et des incendies et observer le déroulement de la simulation sur l'interface graphique ;
- arrêter et reprendre la simulation ;
- sauvegarder l'état d'une simulation dans un fichier ;
- ajouter durant la simulation de nouveaux robots ou incendies.

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 carte, fichier éventuel contenant les positions initiales des robots 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 la bibliothèque libre Commons CLI [7].

Attention, il ne s'agit pas de proposer une interface textuelle à votre application où l'utilisateur peut créer des choses, mais de pouvoir lancer directement une simulation à partir de fichiers la paramétrant. Par exemple, on pourra exécuter (vous pouvez choisir d'autres noms !) :

```
robowl -map supaero-map.osm -config config-init.txt
```

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

3.1 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 algorithmes de plus court chemin, d'autres types de robots, d'autres formats de stockage de cartes etc.

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

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

3.2 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.

3.3 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 éventuel 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 pour les points suivants :

- algorithme de plus court chemin
- simulateur à événement discret

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

[E17] le code fourni sera documenté via Javadoc et indenté correctement.

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

[E19] tous les projets seront vérifiés grâce à l'utilitaire JPlag [8].

[E20] les exigences **minimales** pour la partie implantation sont les suivantes :

- utilisation de l'algorithme de Dijkstra pour le calcul du plus court chemin
- simulateur à événements discrets fonctionnel
- chargement d'une carte depuis un fichier OSM
- chargement d'une situation initiale depuis un fichier
- animation simple de l'interface graphique : pas d'animation intermédiaire pour les déplacements de robots, chaque pas d'animation correspond à l'arrivée d'un événement dans le simulateur, pas de possibilité d'ajout d'incendie ou de robot depuis l'interface graphique etc.

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

3.4 Exigences concernant la gestion de projet

[E21] 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.

[E22] le responsable de chaque équipe enverra tous les vendredis (sauf en période de vacances scolaires) au vacataire responsable de son équipe un mail décrivant les travaux effectués sur le projet durant la semaine.

[E23] le professeur responsable du cours se réserve le droit de modifier ou compléter le sujet du présent projet de façon unilatérale et parfaitement arbitraire.

3.5 Exigences concernant les livrables

[E24] 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.

[E25] tous les rapports seront déposés sur la page LMS du cours : https://http://lms.isae.fr/course/view.php?id=448

[E26] tout le code source de l'application sera déposé sur le dépôt Subversion de l'équipe. Seul le code déposé sur le dépôt sera utilisé pour l'évaluation du projet.

[E27] pour chaque jour de retard, 0.5 points seront retirés sur la note finale.

[E28] 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

Ce rapport est à déposer avant le **25 novembre 2013 23h00**. Dans le même temps, les squelettes des classes Java documentés correspondant au diagramme de classe seront déposés sur le dépôt Subversion.

[E29] 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 **15 janvier 2014 23h00**. Dans le même temps, votre vacataire récupérera le code de votre projet depuis votre dépôt Subversion.

[E30] Le code récupéré sur votre dépôt le **20 janvier 2014** sera considéré comme le code définitif de votre BE.

3.6 Exigences concernant la présentation orale

[E31] une présentation orale par équipe de votre projet se déroulera le **20 janvier 2014**. 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.

[E32] vous devez envoyer votre présentation à votre vacataire au format PDF le le **17 janvier 2014**.

[E33] le délégue de chaque PC est en charge d'organiser l'ordre de passage des soutenances de sa PC et de l'envoyer le **17 janvier 2014** à son vacataire. On essayera si possible de partager un ordinateur pour éviter de perdre du temps entre chaque présentation.

4 Récapitulatif des dates importantes

25 novembre 2013 23h00	Analyse et proposition d'une solution de conception
15 janvier 2014 23h00	Réalisation et rapport final
17 janvier 2014	Envoi présentation et corrections éventuelles
17 janvier 2014	Envoi planning (**délégués**)
20 janvier 2014	Oral

5 Logiciel de modélisation UML

Pour faire vos diagrammes UML, vous trouverez de (trop) nombreux outils. Nous vous conseillons d'utiliser ArgoUML [9], un outil permettant de dessiner facilement des diagrammes UML et de générer le code Java correspondant.

6 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 BE sera illustrée par des exemples sur le site.

Partie	Détail	Points	Commentaires
Analyse	cas d'utilisation	1,5	diagramme de cas d'utilisation UML
	identification acteurs	0,5	identification correcte des acteurs et du système développé
	total	2,0	
Conception	diagramme de classes	2,0	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	2,0	conception d'un système extensible et réutilisable facilement
	total	6,0	
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
	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
	total	3,0	
Javadoc	javadoc rapport conception	1,0	
	javadoc rapport finale	1,0	
	total	2,0	
Rapports et oral	rapports	1,0	qualité du rapport écrit
	oral	1,0	qualité des transparents, pertinence des réponses aux questions posées
	total	2,0	

TABLE 1 – Notation détaillée du BE

7 Rôle des PC(wo)men

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**).

8 Notation

La table 1 présente un barème indicatif pour la notation du BE. 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.**
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...

9 Conseils

Quelques conseils pour ce BE, 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 binôme sur chaque WP, il y a du travail pour deux...
- 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 [10] et tutoriels [11] ;
- enfin, l'avantage de disposer de vos classes et d'un ordinateur est de pouvoir les comparer de façon automatique d'un BE à l'autre [8]... 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*) [12, 13, 14].
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 déterminer 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 3 représente dans le formalisme UML l'acteur « client » du système distributeur de billets.

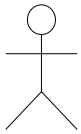

FIGURE 3 – Représentation UML d'un acteur

On peut **spécialiser** un acteur pour définir un acteur ayant un rôle plus précis qu'un acteur déjà présent. La figure 4 représente la spécialisation de l'acteur « client » en acteur « client d'affaires » qui est un client particulier.

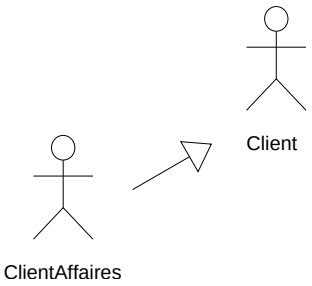

FIGURE 4 – Spécialisation d'un acteur

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.

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 5 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.

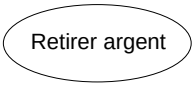

FIGURE 5 – Cas d'utilisation « retirer argent »

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 6. 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 ».

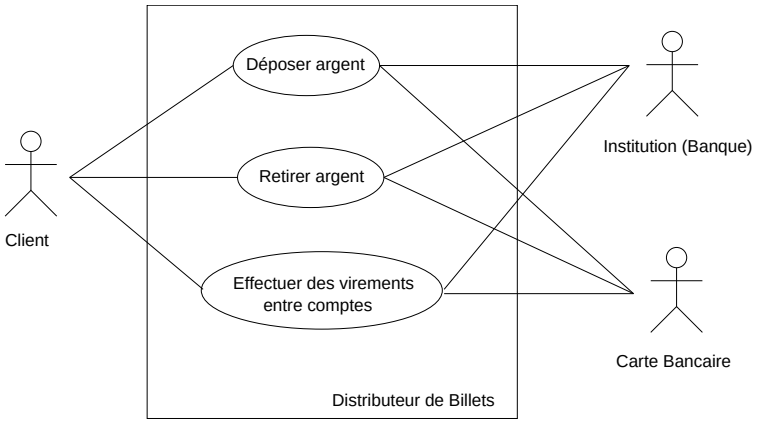

FIGURE 6 – Diagramme d'utilisation du distributeur de billets

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 7 (on a omis les durées de traitement).

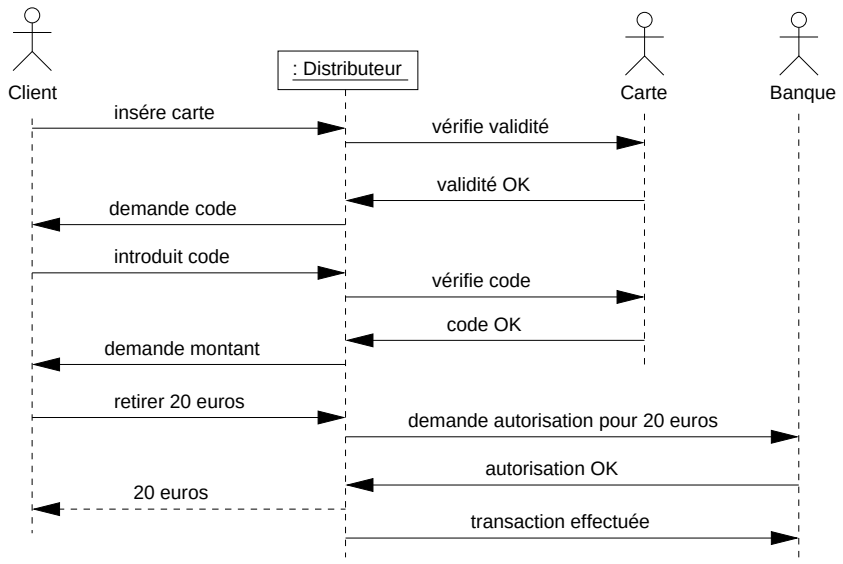

FIGURE 7 – Diagramme de séquence d'un scénario de « retirer argent »

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 8

Références

[1] *RoboCup Rescue*. http://www.robocuprescue.org/. URL : http://www.robocuprescue.org/.

[2] OpenStreetMap contributors. *OpenStreetMap – The Free Wiki World Map*. 2013. URL : http://www.openstreetmap.org/.

[3] Wikipedia contributors. *XML*. 2103. URL : http://en.wikipedia.org/wiki/XML.

[4] T.H. CORMEN et al. *Introduction à l'algorithmique*. 2e éd. Dunod, 2004.

[5] Foundation for Intelligent Physical Agents. *FIPA Contract Net Interaction Protocol specification*. URL : http://www.fipa.org/specs/fipa00029/SC00029H.pdf.

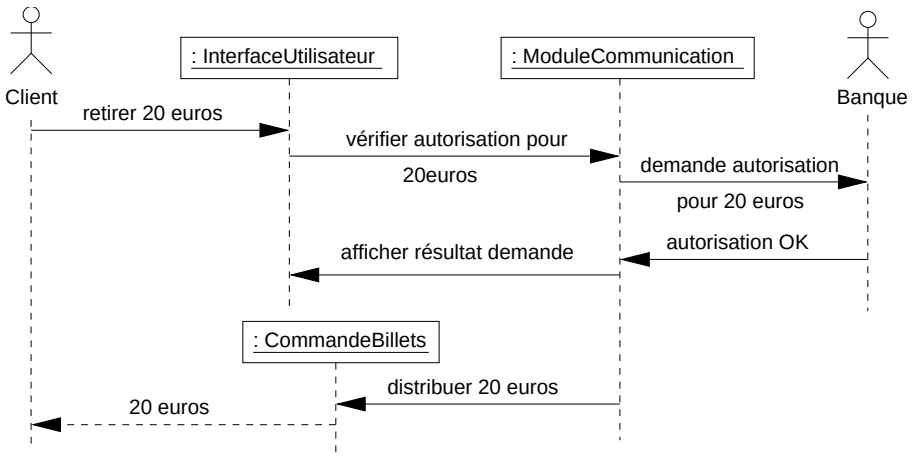

FIGURE 8 – Raffinement d'un diagramme de séquence

[6] ORACLE. *A visual guide to Swing components*. 2013. URL : http://download.oracle.com/javase/tutorial/ui/features/components.html.

[7] APACHE COMMONS. *Commons CLI*. 2012. URL : http://commons.apache.org/cli/.

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

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

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

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

[12] G. BOOCH, J. RUMBAUGH et I. JACOBSON. *The Unified Modeling Language user guide*. Addison-Wesley, 1998.

[13] G. BOOCH, J. RUMBAUGH et I. JACOBSON. *The Unified Modeling Language reference manual*. Addison-Wesley, 2004.

[14] M. FOWLER. *UML distilled*. Addison-Wesley, 2003.

License CC BY-NC-SA 3.0