MASTER EN SCIENCES INFORMATIQUES

Développement d'un support pédagogique à la programmation orientée-objet avec leJOS

Van den Nest, Christopher

Award date:
2011

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Faculté d'informatique
Année académique 2010 - 2011

Développement d'un support pédagogique à la programmation orientée-objet avec leJOS

Christopher Van den Nest

Mémoire présenté en vue de l'obtention du grade de master en Sciences Informatiques

Résumé

Le firmware leJOS permet de créer des programmes en Java afin de contrôler des robots LEGO Mindstorms. Ce mémoire présente deux projets développés en vue de créer un support pédagogique ludique et attractif pour les étudiants en sciences informatiques. Le premier projet propose une série de cas simples dans lesquels un certain nombre de concepts de la programmation orientée-objet sont expliqués et mis en pratique. Le deuxième projet développe un cas plus complexe qui utilise un grand nombre des concepts illustrés dans les différents cas du premier projet.

mots-clés : firmware leJOS, LEGO Mindstorms, support pédagogique, programmation orientée-objet, concepts, projets.

Abstract

The leJOS firmware allows one to write programs in Java in order to control LEGO Mindstorms robots. This master thesis contains two projects developed to create a fun and attractive educational support for students in computer sciences. The former provides several simple exercises in which a lot of concepts of the object-oriented programming are explained and used. The latter is a more complex exercise that uses most of the concepts provided in the former.

keywords : leJOS firmware, LEGO Mindstorms, educational support, object-oriented programming, concepts, projects.

Remerciements

J'adresse mes remerciements aux professeurs de la Faculté d'informatique grâce auxquels j'ai pu réaliser ce parcours universitaire. Plus particulièrement, je tiens à remercier M. Patrick Heymans, mon promoteur et MM. Germain Saval et Maxime Cordy, ses assistants qui m'ont guidé dans l'élaboration de ce mémoire

Merci également à ma famille, à mes amis et à mes collègues pour le soutien qu'ils m'ont apporté.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

A l'heure actuelle, il arrive régulièrement que certains étudiants éprouvent des difficultés à suivre des cours dans lesquels de nombreux concepts abstraits sont enseignés. Ces cours sont souvent dispensés via une approche traditionnelle consistant en une interaction limitée entre le professeur et les étudiants. Il s'agit d'un problème qui donne lieu à de nombreux débats entre professeurs et spécialistes sur l'approche à employer pour permettre aux étudiants d'avoir plus de facilité à appréhender une matière donnée.

Il existe un certain nombre de théories qui traitent de ce problème et qui tentent d'y apporter une solution. Chacune fournit une explication sur la manière dont il faudrait enseigner des concepts aux étudiants en se basant même parfois sur diverses théories concernant le fonctionnement de la mémoire. Plusieurs travaux ont été réalisés dans le même but.

Dans le cadre de ce mémoire, nous nous focalisons sur l'apprentissage de la programmation orientée-objet (OO). Il y a également débat concernant l'enseignement de celle-ci et nous nous sommes basés sur l'un de ces débats (Bruce, 2005) pour déterminer les principales approches pédagogiques à l'enseignement de concepts OO dans un cursus en sciences informatiques, par exemple durant le premier cours de programmation de ce cursus. Dans le cadre de cet exemple, nous en avons dénombré trois. Il s'agit des approches « Object-First » (i.e. enseigner les bases de la programmation à l'aide de concepts OO), « Object-Last » (i.e. enseigner les concepts OO vers la fin du premier cours) et « Non-OO » (i.e. enseigner un autre paradigme durant le premier cours).

Plusieurs outils pédagogiques ont été créés depuis de nombreuses années afin d'aider les étudiants à mieux assimiler les concepts propres à un paradigme de programmation donné. Un des premiers d'entre eux est Karel le

Robot. Il s'agit d'un outil qui a été créé pour enseigner les concepts de la programmation procédurale d'une façon différente, en proposant aux étudiants de programmer un robot afin qu'il effectue des tâches au sein d'un monde virtuel. Cela permet aux étudiants de visualiser le résultat de leur travail.

Un certain nombre d'outils pédagogiques ont également été développés pour l'apprentissage de concepts propres au paradigme OO. Certains de ces outils sont des portages de Karel le Robot ou se basent sur les portages de celui-ci.

L'objectif de ce mémoire est de proposer une solution pour aider les étudiants à mieux comprendre les différents concepts introduits durant un cours de programmation OO et d'avoir plus de facilité à les assimiler. La finalité de cette solution est de permettre d'illustrer ces concepts dans un tel cours et de pouvoir l'utiliser dans le cadre de travaux pratiques s'y afférant.

Afin d'atteindre l'objectif que nous nous sommes fixés, nous avons développé un support pédagogique à la programmation orientée-objet. Pour nous aider à rendre l'apprentissage des concepts OO ludique et plus concret, nous nous sommes inspirés du fonctionnement des outils pédagogiques tels que Karel le Robot. Pour ce faire, nous avons choisi de développer ce support en nous servant du firmware leJOS afin de programmer un robot LEGO Mindstorms NXT 2.0 pour lui faire effectuer diverses tâches à la manière de ce qui se fait sur des outils tels que Karel le Robot.

Pour développer ce support, nous nous sommes également basés sur le cours de Conception et de Programmation Orientée-Objet (CPOO) (Heymans, 2010). Ce cours suit le livre de Barbara Liskov « Program Development in Java » (Liskov et Guttag, 2001) qui nous sert de référence. Bien qu'il ait été élaboré sur base du cours de CPOO, notre support peut parfaitement être utilisé dans n'importe quel cours de programmation OO pour illustrer les concepts OO que le cours de CPOO aborde.

Le public auquel ce mémoire s'adresse est un public averti c'est-à-dire ayant déjà une certaine connaissance du paradigme impératif et possédant quelques notions au niveau de la conception et de la programmation OO. Les exercices qui s'y trouvent permettent, d'une part, aux professeurs d'illustrer des concepts OO dans leur cours, et d'autre part, aux étudiants de cours de programmation OO de les revoir et de les mettre en pratique.

Ce support pédagogique se distingue de la plupart des outils que nous avons listés dans l'état de l'art parce qu'il fournit des explications détaillées et complètes sur les différents concepts OO qu'il illustre. De plus, au lieu de travailler avec un monde virtuel, les étudiants travaillent sur quelque chose de tangible, ce qui peut susciter encore plus d'intérêt de leur part.

Le support a aussi été conçu de telle manière qu'il soit facile pour eux de retrouver des informations utiles concernant les concepts OO qu'ils doivent utiliser. En effet, il contient diverses explications détaillées sur ces concepts ainsi que des rappels provenant notamment du livre de Barbara Liskov.

Outre l'introduction, ce mémoire compte cinq chapitres. Le chapitre 2 propose un état de l'art des théories de l'apprentissage, des approches pédagogiques à l'apprentissage de la programmation et d'outils pédagogiques permettant d'apprendre, pour la plupart, la programmation OO. Le chapitre 3 fournit des explications sur les outils utilisés pour réaliser notre support pédagogique, à savoir le firmware leJOS et un robot LEGO Mindstorms NXT 2.0.

Le chapitre 4 constitue la première partie de notre support pédagogique. Il contient une dizaine d'exercices permettant de pratiquer et d'illustrer un certain nombre de concepts OO. Ces exercices sont composés d'un énoncé, de liens vers le cours qui nous a servi de référence pour les concepts OO abordés dans ces exercices, d'une liste d'erreurs que les étudiants pourraient être amenés à commettre ainsi que d'une solution détaillée.

Le chapitre 5, qui concerne la deuxième partie de notre support pédagogique, propose quant à lui un projet de plus grande envergure constitué d'un seul exercice dans lequel un grand nombre des concepts OO abordés au chapitre 4 peuvent être pratiqués.

Enfin, le chapitre 6 conclut ce mémoire en proposant un résumé des contributions que nous avons apportées, une analyse critique des résultats obtenus en indiquant les avantages et inconvénients du support pédagogique que nous avons créé ainsi que quelques idées et pistes pour toute personne souhaitant poursuivre les recherches entamées dans ce mémoire.

Chapitre 2

Etat de l'art

Dans son article intitulé « Controversy on how to teach CS 1 : a discussion on the SIGCSE-members mailing list » (Bruce, 2005), Kim Bruce nous fournit une base permettant d'établir un état de l'art des approches et des outils pédagogiques existants servant à l'apprentissage de la programmation orientée-objet. Au début de l'année 2004, sur la mailing list de la SIGCSE [1], un débat a éclaté concernant la pédagogie utilisée dans les cours d'introduction à la programmation utilisant le Java comme langage de base. Plusieurs professeurs y ont pris part.

L'auteur de cet article, Kim Bruce, a résumé les arguments des deux camps qui se sont formés lorsque ce débat a eu lieu. Ayant également participé à ces discussions, il y propose, entre autres, sa manière de voir les choses. Il estime que les professeurs des cours d'introduction à la programmation disposent de trois approches pédagogiques différentes ayant comme finalité l'apprentissage de la programmation OO (Bruce, 2005, pages 6-7). Il s'agit des approches « Object-late », « Object-early » et « Non-OO ». Ce constat sert de base à l'établissement de notre état de l'art.

L'état de l'art est découpé en trois parties. La première a pour but de lister et de décrire des théories de l'apprentissage qui sont employées dans l'enseignement, notamment pour enseigner la programmation OO. Dans la deuxième partie, des explications sont fournies sur les trois approches pédagogiques citées par Kim Bruce dans le cadre du premier cours de programmation d'un cursus par exemple en sciences informatiques. Enfin, la

1. Special Interest Group on Computer Science Education - dispose d'un espace sur lequel les professeurs en sciences informatiques peuvent donner leur opinion sur la manière dont les cours sont organisés, dispensés, etc.

troisième partie a pour objectif de lister et de décrire différents outils pédagogiques pouvant être utilisés par les professeurs comme support de leur cours. Certains de ces outils sont d'ailleurs parfois considérés comme des approches à part entière.

2.1 Théories de l'apprentissage

Les théories de l'apprentissage constituent des approches pédagogiques pouvant être utilisées dans de nombreux domaines. Depuis que l'enseignement existe, plusieurs approches ont été testées afin de trouver la meilleure manière d'enseigner des concepts aux étudiants. Cette section a pour objectif de lister un certain nombre de ces théories de l'apprentissage ainsi que de les décrire brièvement. L'objectif est de déterminer lesquelles de ces théories sont les plus pertinentes dans le cadre de l'élaboration de notre support pédagogique.

2.1.1 Comportementalisme

Le comportementalisme est une théorie de l'apprentissage « basée sur les changements observables dans le comportement » et « se focalise sur un nouveau modèle comportementaliste répété jusqu'à ce qu'il devienne automatique » (Mergel, 1998, page 2).

Dans le comportementalisme, l'esprit est vu « comme étant une boîte noire » et l'apprentissage est considéré comme se faisant à l'aide de stimuli externes (Mödritscher, 2006, page 4). Un exemple pouvant être donné est celui d'un chien auquel son maître donne un biscuit à la fin de sa promenade. A force de répéter cette action, le chien va assimiler le fait de recevoir un biscuit au fait qu'il vient d'aller faire une promenade. Au fur et à mesure que le temps passe, celui-ci va donc commencer à se manifester auprès de son maître pour recevoir son dû à la fin de chaque promenade (aboiement si le maître l'a oublié, etc.).

Cette théorie suppose que l'information n'est pas traitée au niveau de l'esprit mais que tout fonctionne par conditionnement (modèle « stimulus - réponse »).

2.1.2 Cognitivisme

Le cognitivisme est une théorie de l'apprentissage pour laquelle « l'apprentissage est un processus interne qui implique la mémoire, la pensée, la réflexion, l'abstraction, la motivation et la méta-cognition » (Mödritscher, 2006, page 5).

Cette théorie repose sur un certain nombre de concepts. L'un d'entre eux stipule qu'une « information ayant du sens est plus facile à apprendre et à retenir » qu'une qui n'en a pas (Mergel, 1998, page 7).

Cependant il est quand même possible de retenir une information difficile à comprendre ou qui n'a pas vraiment de sens. Cela peut être fait à l'aide d'un moyen mnémotechnique. La mnémotechnique est par exemple utilisée pour enseigner aux étudiants apprenant le français différentes conjonctions de coordination à l'aide de la phrase « mais où est donc Ornicar ? ». Ce concept peut également être utilisé pour apprendre les caractères chinois.

Un autre concept que nous pouvons mentionner se base sur le fait que l'étude d'un étudiant peut être améliorée si celui-ci s'exerce de manière régulière. Cela va lui permettre de mémoriser plus facilement ce qu'il est en train d'étudier. Autrement dit, un étudiant en science informatique qui pratique régulièrement les concepts qui lui ont été enseignés va être plus à même de retenir ceux-ci grâce à sa régularité dans son étude (Mergel, 1998, page 7).

2.1.3 Constructivisme

Le constructivisme est une théorie de l'apprentissage « basée sur la prémisse que nous construisons tous notre propre perspective du monde, à travers des expériences individuelles et un schéma » et « se focalise sur la préparation de l'étudiant à la résolution de problèmes dans des situations ambigües »(Mergel, 1998, page 2).

Cette théorie, qui est devenue dominante ces dernières années, suppose donc que l'étudiant construit sa connaissance à partir de ce qu'il connaît déjà. De ce fait, le résultat de la construction de la connaissance par l'étudiant dépend de l'état actuel de sa connaissance. L'étudiant qui a mal assimilé un certain nombre de concepts va continuer à enrichir sa connaissance sur base de concepts mal compris. C'est pourquoi il est important selon cette théorie de mettre l'accent sur une bonne construction de la connaissance par les étudiants.

Les approches plus traditionnelles ne se préoccupent pas vraiment de ce processus de construction de la connaissance contrairement aux approches

pouvant être assimilées à du constructivisme. Celles-ci s'en préoccupent, en effet, dès le début. Pour cette raison, elles sont considérées comme ayant plus de chance de permettre aux étudiants de bien comprendre ce qui leur est enseigné (Ben-ari, 2008, page 45).

Afin qu'un étudiant apprenne quelque chose, il est nécessaire qu'il construise un modèle mental. S'il s'agit d'apprendre à utiliser un environnement de développement, le modèle mental que l'étudiant va se construire va devoir, entre autres, répertorier l'utilité des différentes fonctionnalités offertes par cet environnement. Comme expliqué un peu plus haut, la manière dont ce modèle va être construit va dépendre de la connaissance que l'étudiant avait déjà mais aussi en réalité d'autres critères tels que sa personnalité. Il faut également prendre en compte le fait que les étudiants n'apprennent pas tous les choses de la même manière.

Selon cette théorie, le rôle du professeur est de s'assurer que l'étudiant construise un bon modèle mental, un modèle mental qui va lui permettre d'utiliser sa connaissance non seulemement dans des cas connus mais aussi lorsqu'il rencontrera un nouveau cas (Ben-ari, 2008, pages 48-49).

2.1.4 Avantages et désavantages de ces trois premières théories

Chacune de ces trois théories de l'apprentissage a des avantages et des désavantages. Ceux-ci vont être analysés dans cette sous-section en s'inspirant fortement de (Mergel, 1998).

Tout d'abord, la figure 2.1, qui vient de (Mergel, 1998, page 10), permet de voir l'évolution des théories de l'apprentissage du comportementalisme (Behaviorism) au constructivisme (Constructivism) en passant par le cognitivisme (Cognitivism).

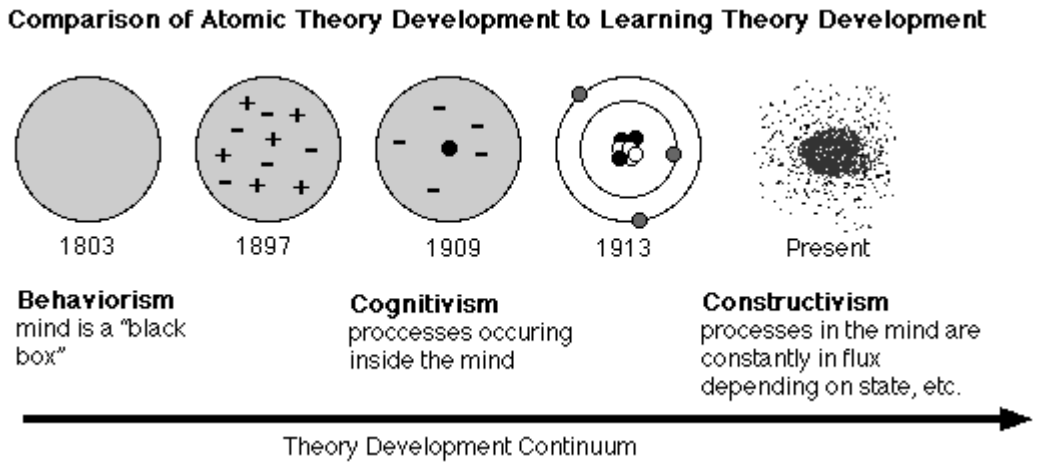

FIGURE 2.1: Du comportementalisme au constructivisme (Mergel, 1998, page 10)

Sur cette figure, nous pouvons voir l'idée principale de chacune de ces théories. Le comportementalisme y est décrit comme voyant l'esprit comme une « boîte noire ». Dans le cognitivisme, par contre, on y parle de « traitements se produisant à l'intérieur de l'esprit » alors que dans le constructivisme, ces traitements sont vus comme étant « dans un état de perpétuel changement ».

Un avantage du comportementalisme est le fait qu'il permette à l'étudiant d'avoir des automatismes face à une situation donnée. Cependant cette forme de conditionnement peut également devenir un désavantage pour l'étudiant. En effet, une personne qui a été conditionnée pour résoudre un problème apparaissant d'une certaine manière ne pourra probablement pas le résoudre s'il survient dans une autre situation.

Concernant le cognitivisme, « l'objectif est d'entraîner les étudiants à effectuer une tâche de la même manière afin de garder une certaine cohérence » (Mergel, 1998, page 20). En entreprise, par exemple, on va établir des procédures afin de déterminer toutes les étapes à effectuer lors de chaque développement d'application et comment elles devront être effectuées afin que tout le monde puisse se comprendre (e.g. écrire les documents en anglais, quel langage utiliser pour effectuer l'analyse (UML, ...)). Ces tâches pourraient cependant peut-être mieux être effectuées d'une autre manière que celle décrite dans ces procédures.

Enfin, comme expliqué plus haut, le constructivisme vise à permettre aux étudiants de construire leur connaissance à partir de ce qu'ils connaissent

déjà. L'avantage de cette théorie est donc qu'ils vont être capables de s'adapter à une nouvelle situation. Ils ne pourront cependant s'adapter que si la construction de leur connaissance a été faite à partir de bases correctes.

2.1.5 Visualisation

Un certain nombre de travaux se sont attachés à la visualisation. Bien que la plupart de ceux-ci dépassent largement le domaine de l'informatique, ceux que nous avons considérés concernaient la visualisation informatique. La visualisation y est décrite comme « pouvant illustrer graphiquement divers concepts en sciences informatiques » (Naps *et al.*, 2002, page 131). La visualisation ne présente cependant un intérêt que si elle permet « d'engager les étudiants dans une activité d'apprentissage active » sans quoi elle ne dispose « d'aucune valeur éducationnelle » (Naps *et al.*, 2002, page 131).

La programmation est un domaine dans lequel on dénombre beaucoup de concepts abstraits. Les étudiants confrontés à ces concepts abstraits ont souvent du mal à bien les comprendre. La visualisation repose sur des représentations graphiques afin d'aider les étudiants à comprendre ces concepts plus facilement. Ils peuvent en effet voir ce qu'ils font et interagir avec ces représentations graphiques.

Comme décrit auparavant, il est cependant important que les étudiants soient actifs en utilisant la visualisation car des études ont montré que « les étudiants engagés activement avec la technologie de la visualisation ont constamment été meilleurs » que ceux qui ont été amenés à l'expérimenter uniquement de manière passive (Naps *et al.*, 2002, page 131). C'est un point dont il faut tenir compte lorsque l'on adapte son cours afin qu'il puisse utiliser la visualisation dans les meilleures conditions possibles.

L'interprétation peut également poser un problème pour les étudiants. Il faut s'assurer que l'interprétation qu'ils vont donner aux représentations graphiques soit correcte. Cela demande de parfois fournir aux étudiants un moyen de comprendre celles-ci en mettant à leur disposition un texte expliquant ce qu'elles sont censées représenter. Cela va permettre de réduire le risque de mécompréhension.

L'utilisation de la visualisation doit en outre dépendre du niveau des étudiants. Ceux qui, par exemple, commencent leurs études en sciences informatiques risquent d'éprouver quelques difficultés s'ils sont confrontés à beaucoup de détails à la fois. C'est pourquoi il vaut mieux commencer par leur faire utiliser des représentations graphiques relativement simples. L'uti-

lisation de supports connus tels que des bandes dessinées peut également être d'une grande aide (Naps *et al.*, 2002, page 132).

D'autres points peuvent également être inclus tels qu'offrir la possibilité aux étudiants d'avoir accès à un historique des étapes effectuées, leur fournir les informations nécessaires pour qu'ils puissent savoir si leur algorithme est performant, leur permettre de « créer leur propre visualisation », etc (Naps *et al.*, 2002, pages 132-133).

2.1.6 Edutainment

Comme pour la visualisation, un grand nombre de travaux se sont attachés à l'edutainment. Le concept de l'edutainment est de permettre aux étudiants d'apprendre tout en s'amusant. Ce mot constitue une contraction des mots anglais « educational » et « entertainment ».

Il a été constaté au fil du temps qu'apprendre en s'amusant permet d'apprendre plus facilement et plus rapidement. L'interactivité joue également un très grand rôle dans l'apprentissage. Un certain nombre de jeux, notamment pour enfants, sont d'ailleurs basés sur ce concept. L'interactivité avec, par exemple, un ordinateur peut aider les étudiants à « enrichir leur connaissance et en même temps à intégrer de manière pratique ce qui leur a été enseigné » (Atmatzidou *et al.*, 2008, page 24).

Les chercheurs qui ont adopté « la méthode Edutainment sont arrivés à des résultats encourageants ». Certains indiquent également que cela permet « non seulement aux étudiants de comprendre les concepts appris aux cours de chaque leçon mais aussi de les intégrer dans leur propre structure de connaissance en tant qu'outils et en tant qu'éléments constructifs qui pourraient avoir une utilité future » (Atmatzidou *et al.*, 2008, page 24).

L'etudainment peut être utilisé pour favoriser le travail en équipe en se basant soit sur la collaboration avec les autres soit sur l'esprit de compétition. Cela requiert entre autres d'être capable de « diviser de manière équitable les tâches parmi tous les membres d'une équipe » ainsi que d'être capable de « communiquer de manière efficace » (Atmatzidou *et al.*, 2008, page 25-26).

2.2 Approches d'apprentissage de la programmation

Les trois approches d'apprentissage de la programmation orientée-objet telles que mentionnées dans (Bruce, 2005) sont, chacune, décrites dans une sous-section différente. Pour les décrire brièvement, faisons l'hypothèse qu'elles sont utilisées pour dispenser le premier cours de programmation des étudiants en sciences informatiques. Tout comme pour les théories de l'apprentissage, le but de cette section est de nous aider à déterminer l'approche qui nous semble être la plus adéquate dans le cadre de l'élaboration de notre support pédagogique.

2.2.1 Approche « Object-late »

L'approche « Object-late » voire « Object-last » est une approche pédagogique utilisée par des professeurs estimant qu'il est primordial que les étudiants maîtrisent d'abord certains concepts procéduraux importants (e.g. les conditions « if else », les boucles « while », etc.) avant de s'attaquer aux concepts propres à la programmation OO. Ces concepts sont néanmoins abordés lors du premier cours de programmation mais assez tardivement, par exemple lors des dix dernières heures de cours (d'où le nom de l'approche) (Bruce, 2005, pages 3, 7).

2.2.2 Approche « Object-early »

L'approche « Object-early » ou « Object-first » est une approche dont l'idée principale est de confronter les étudiants aux concepts de la programmation OO dès le premier cours de programmation. Ils sont donc amenés, dès les premières heures de ce cours, à comprendre voire à manipuler les concepts de base de ce paradigme tels que les classes, l'héritage et l'encapsulation. Cette approche est considérée pour cette raison comme étant relativement complexe car les étudiants sont confrontés à une série de nouveaux concepts très rapidement et en même temps. D'autres concepts peuvent être cités, par exemple la différence entre les types primitifs et les types d'objets ainsi que le concept de mutabilité (e.g. en Java via la classe String dont l'état d'une instance ne peut jamais être modifié) et l'héritage. (Cooper *et al.*, 2003, page 191).

Certains des partisans de cette approche estiment que des concepts tels que la notion de condition, peuvent être expliqués aux étudiants à l'aide

de concepts propres à la programmation OO. Par exemple, concernant la notion de condition, il est possible d'expliquer celle-ci en se servant des interfaces. Dans l'article de Kim Bruce, il mentionne que « la sélection d'alternatives peut être accomplie par le dispatching dynamique lorsque les objets de différentes classes implémentent la même interface et répondent aux mêmes messages » (Bruce, 2005, page 3).

2.2.3 Approche « Non-OO »

La troisième possibilité est de reléguer l'enseignement des concepts de la programmation OO à un cours ultérieur du cursus des étudiants en sciences informatiques. L'objectif serait dès lors d'enseigner les concepts d'un autre paradigme. Il pourrait s'agir d'utiliser un langage comme le C pour enseigner les concepts de la programmation procédurale (Bruce, 2005, pages 1, 7).

2.2.4 Enquête

La Higher Education Academy - Information and Computer Sciences [2] a réalisé, au Royaume-Uni en 2005, une enquête ayant pour objectif de déterminer les langages de programmation enseignés durant le premier cours de programmation ainsi que leur taux d'utilisation. Cette enquête avait également pour but de mettre en évidence les différentes approches pédagogiques utilisées pour les enseigner (Chalk et Fraser, 2008, pages 1-2).

Afin de mener à bien cette enquête, une centaine d'institutions différentes ont été interrogées mais seul un tiers d'entre elles a répondu à celle-ci (52 réponses) (Chalk et Fraser, 2008, page 1). Les résultats concernant la programmation OO montrent que, à l'époque où l'enquête a été réalisée, le Java « était le langage le plus utilisé » lors du premier cours de programmation et ce dans une trentaine d'institutions (Chalk et Fraser, 2008, page 2).

Parmi cette trentaine d'institutions, au niveau des approches pédagogiques, cette enquête a permis de mettre en exergue que l'approche « Object-late » était utilisée par un peu plus de la moitié des répondants. A l'exception d'un petit pourcentage préférant utiliser d'autres approches non mentionnées dans l'enquête, le reste des répondants utilisait l'approche « Object-

2. La Higher Education Academy est une organisation indépendante située au Royaume-Uni. Elle finance plusieurs centres et a, entre autres, pour but d'améliorer les méthodes d'enseignement. L'ICS est un de ces centres. Il est dédié aux études en sciences informatiques.

early » à cette époque. L'enquête révèle également qu'une écrasante majorité des professeurs enseignant le C++ ou le C# lors du premier cours de programmation préférait utiliser l'approche « Object-late ». Il est cependant important de noter que parmi l'ensemble des répondants, ceux enseignant le Java lors du premier cours de programmation étaient beaucoup plus nombreux que ceux enseignant le C++ ou le C# (Chalk et Fraser, 2008, page 3).

2.3 Les outils pédagogiques

Il existe depuis longtemps des outils pédagogiques développés dans le but de permettre à des étudiants d'apprendre les concepts d'un ou plusieurs paradigmes de programmation. En raison des difficultés que peut poser l'apprentissage d'un paradigme, il était en effet déjà conseillé au temps de la programmation procédurale d'user de tels outils pour permettre aux étudiants de pouvoir y faire face avec plus d'aisance (Georgantaki et Retalis, 2007, page 112).

Ce constat semble également concerner la programmation OO. Différentes études révèlent que, lors de son apprentissage, beaucoup de problèmes sont induits par ce paradigme. Une partie de la communauté de chercheurs suggère que certains outils pédagogiques permettent de ne pas se focaliser sur un langage en particulier. L'utilisation de tels outils aurait d'ailleurs un effet positif sur la capacité des étudiants à résoudre des problèmes qui leur seraient posés. Certains chercheurs estiment cependant que les étudiants doivent également être mis en contact avec des outils du monde professionnel durant leur cursus afin de pouvoir s'y habituer avant de rentrer dans le monde du travail. Il s'agirait dès lors d'utiliser les deux types d'outils à bon escient (Georgantaki et Retalis, 2007, page 112).

Dans l'article (Georgantaki et Retalis, 2007), il est mentionné l'existence de différentes catégories d'outils pédagogiques servant à l'apprentissage de la programmation OO. Cette section est découpée en plusieurs sous-sections représentant chacune une catégorie. Chaque sous-section comporte une description de la catégorie à laquelle elle est liée et traite de différents outils pédagogiques qui peuvent y être répertoriés. Certains des outils pédagogiques pouvant être liés à plus d'une catégorie, ils ne seront traités que dans une des catégories auxquelles ils appartiennent.

L'objectif de cette troisième section est de collecter des informations sur un certain nombre d'outils pédagogiques existants dans le but de nous en inspirer pour développer notre support pédagogique.

2.3.1 Les micromondes de programmation

Premier type d'outils pédagogiques, les « micromondes » permettent à un étudiant « de travailler avec un ensemble riche d'objets primitifs pouvant être utilisés pour illustrer les concepts-clés des langages OO » (Bruce, 2005, page 5).

Comme expliqué auparavant, des outils pédagogiques étaient déjà utilisés au temps de la programmation procédurale afin d'enseigner les concepts de ce paradigme d'une autre manière. Parmi ceux-ci, on trouve le micromonde ***Karel le Robot***. Comme son nom le suggère, celui-ci permet à l'utilisateur d'apprendre la programmation procédurale en faisant effectuer des tâches plus ou moins complexes à un robot dénommé Karel dans un monde virtuel. Cet outil utilise un langage à caractère éducatif proche du Pascal. Cet outil pédagogique en a inspiré beaucoup d'autres dont certains sont utilisés afin d'enseigner les concepts de la programmation OO (Georgantaki et Retalis, 2007, page 113).

Ce type d'outils pédagogiques peut être intéressant pour les professeurs car les étudiants sont amenés à faire face à des problèmes plus concrets (par exemple permettre à un robot de détecter et de contourner un obstacle). Le fait qu'ils puissent véritablement visualiser le résultat de ce qu'ils ont programmé dans le micromonde constitue un plus indéniable permettant d'accroître leur motivation et leur attention.

Ces outils proposent en règle générale de programmer dans un langage de programmation qui leur est propre. Cela constitue également un avantage car leur langage est généralement relativement simple et dispose de peu d'instructions comparé aux langages de programmation professionnels. En effet, l'objectif principal de ces outils est de permettre aux étudiants de disposer d'une manière ludique d'apprendre et de bien comprendre différents concepts d'un paradigme donné, dans notre cas du paradigme OO (Xinogalos, 2010, page 512).

Karel++ : ce micromonde est un portage de celui de Karel le Robot. Il « constitue l'entrée de Karel dans l'ère OO » (Georgantaki et Retalis, 2007, page 113) ;

KarelJRobot : celui-ci a été écrit en Java en se basant sur Karel++ (Georgantaki et Retalis, 2007, page 113) ;

JKarelRobot : celui-ci a également été écrit en Java. Il peut être utilisé sur n'importe quelle plateforme disposant d'une machine virtuelle Java. Il n'est lié à aucun langage ou paradigme en particulier (Georgantaki et Retalis, 2007, page 113) ;

Jeroo : également inspiré par le micromonde de Karel le Robot, « l'île de Jeroo » permet, à tout moment, la visualisation de tous les composants et ce à l'aide d'une seule fenêtre. La syntaxe qu'il utilise a pour vocation de permettre aux étudiants, par la suite, d'apprendre le Java ou le C++ sans trop de difficultés. C'est pourquoi elle est fortement proche de celle de ces deux langages de programmation (Georgantaki et Retalis, 2007, page 113). Jeroo fournit également aux étudiants une interface leur permettant de visualiser l'exécution de leur code. Il est d'ailleurs possible d'effectuer celle-ci étape par étape (Sanders et Dorn, 2003, page 202).

Un autre des objectifs poursuivis par cet outil est d'uniquement mettre l'accent sur certains concepts de base de la programmation OO ainsi que sur la manière de contrôler le flux d'une exécution (les conditions if par exemple). C'est pourquoi il n'existe qu'une seule classe dans ce micromonde. Il s'agit de la classe Jeroo. L'étudiant peut librement utiliser des objets de cette classe et faire appel à leurs méthodes mais ne peut point créer d'autres classes (Sanders et Dorn, 2003, pages 201-202).

La figure 2.2 permet de voir l'interface de Jeroo. Il s'agit d'une interface à partir de laquelle nous pouvons visualiser le monde virtuel, les objets qui s'y trouvent ainsi que le code de la fonction main() et des méthodes que nous pouvons ajouter à la classe Jeroo.

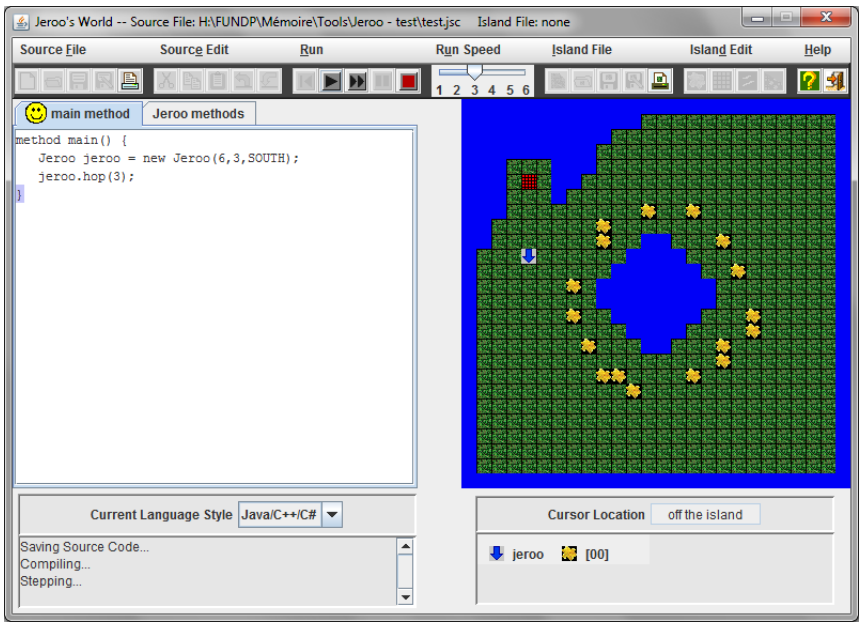

FIGURE 2.2: Jeroo - un exemple

Jeroo a fait l'objet d'une étude de cas dans une université américaine et a été utilisé durant les quatre premières semaines du premier cours de programmation dispensé à des étudiants. Après ces quatre semaines, ils ont effectué une transition vers le Java et l'outil pédagogique BlueJ. Les auteurs de cette étude de cas ont constaté qu'à la fin du semestre, les étudiants qui avaient pris part à cette expérience étaient plus à même de réfléchir à la manière de « décomposer un problème » avant de commencer à programmer car cela se fait de manière plus naturelle au travers des exercices proposés sur Jeroo. Ils estiment qu'au terme de ce cours, « les étudiants ont une meilleure compréhension des structures de contrôle, des objets et des méthodes ». Nous pouvons considérer l'utilisation de Jeroo dans cette étude de cas comme étant une approche à part entière (Sanders et Dorn, 2003, pages 202-203) ;

Alice : cet outil pédagogique se distingue de Karel le Robot par le fait qu'il fournit un micromonde avec un environnement en 3D (Georgantaki et Retalis, 2007, page 113). Ce micromonde met à la disposition des étudiants un certain nombre d'objets et leur permet d'importer d'autres modèles 3D s'ils le souhaitent. Il est d'ailleurs possible d'en trouver sur le site officiel de cet outil [3].

A la figure 2.3, vous pouvez voir une des interfaces de cet outil. Cette interface permet à l'utilisateur de visualiser le monde virtuel d'Alice, de choisir quels objets y ajouter parmi une liste d'objets disponibles et de modifier les propriétés, fonctions et méthodes offertes par cet outil.

3. www.alice.org

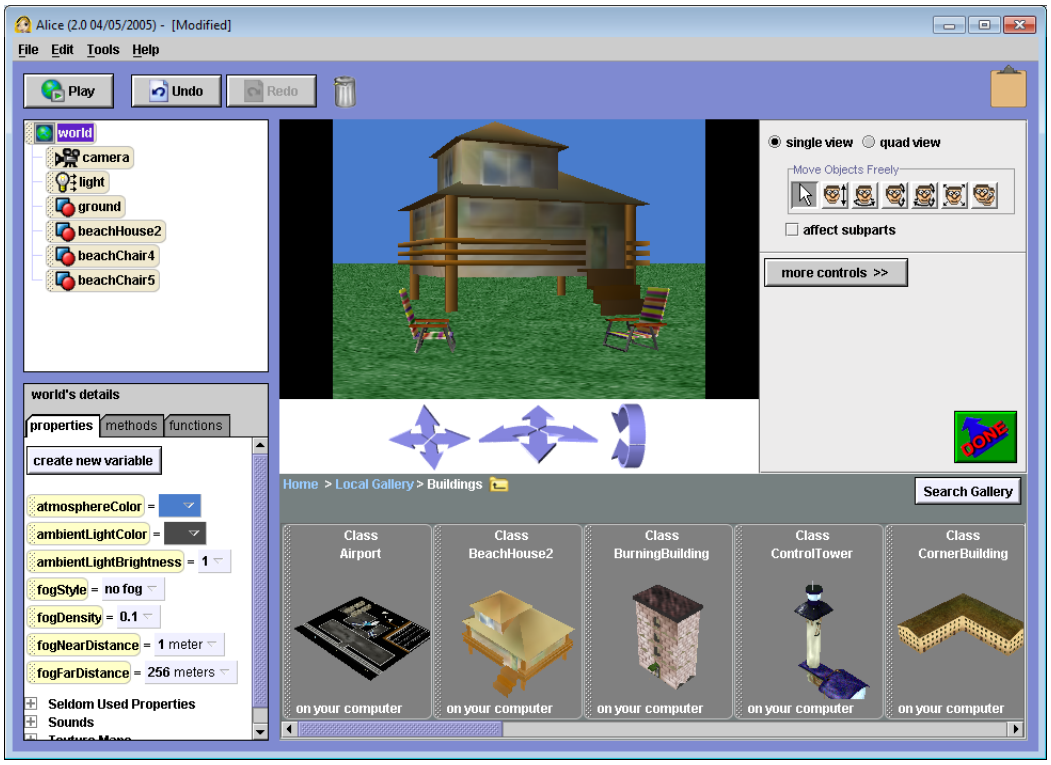

FIGURE 2.3: Alice - l'interface

L'intérêt d'Alice est de permettre aux étudiants de se servir de ces objets pour créer un monde virtuel. Ils peuvent ensuite écrire un programme permettant à ces objets d'interagir entre eux. Dans ce contexte, la 3D est un atout supplémentaire pour permettre aux étudiants de comprendre ce qu'est un objet et de visualiser les conséquences des tâches qu'ils leur font faire. A l'aide de cet outil, il est possible de mieux comprendre les concepts de base de la programmation OO dont la gestion des événements (Georgantaki et Retalis, 2007, page 113) ;

objectKarel : l'université de Macédoine, en Grèce, a créé cet outil afin, entre autres, de proposer aux étudiants un certain nombre de leçons en ligne. Ces leçons sont accompagnées d'explications sur les concepts qu'elles abordent. Cet outil permet aux professeurs de constater quels sont les points faibles de leurs étudiants, notamment de détecter des incompréhensions, car il enregistre ce qu'ils font. Cela permet aux professeurs de réajuster leur cours en conséquence (Georgantaki et Retalis, 2007, page 113).

Cet outil, qui se base sur Karel++, permet aux étudiants de manipuler un voire plusieurs robots dans un environnement qui est composé notamment

de rues et de divers types d'obstacles. Il dispose de certaines caractéristiques permettant de faciliter l'apprentissage du paradigme OO.

La première de ces caractéristiques est l'éditeur de structure. Les concepteurs de cet éditeur ont conçu celui-ci de telle manière que les étudiants sont amenés à se focaliser sur l'aspect structurel de leur programme plutôt que sur la syntaxe du langage qu'ils utilisent. Pour ce faire, un menu est mis à leur disposition afin qu'ils puissent y sélectionner diverses actions dont par exemple la déclaration d'une classe. Ce menu leur permet également de communiquer avec les objets par l'intermédiaire de messages.

La deuxième caractéristique d'objectKarel est la manière dont les étudiants peuvent y voir comment leur programme se comporte à travers leurs robots. La première possibilité qui leur est offerte est d'exécuter leur programme normalement c'est-à-dire en visualisant les robots en train d'effectuer les différentes actions programmées. La deuxième possibilité est d'exécuter leur programme dans un mode grâce auquel il leur est possible de visualiser les différentes instructions exécutées lentement une par une. Ils peuvent dans ce cas voir les conséquences apportées par chacune de ces instructions dans leur micromonde. La troisième et dernière possibilité disponible est d'exécuter leur programme étape par étape. Dans ce dernier mode, ils peuvent choisir à quel moment passer à l'instruction suivante. L'intérêt de ces deux derniers modes est de fournir également aux étudiants des explications sur les différentes instructions exécutées.

La troisième caractéristique concerne la détection des erreurs et la manière dont elles sont expliquées. En effet, lorsqu'une erreur est détectée, objectKarel ne se contente pas d'indiquer celle-ci mais explique également la raison pour laquelle il s'agit d'une erreur. Cela permet aux étudiants de mieux comprendre comment les éviter à l'avenir.

Pour terminer, la quatrième caractéristique concerne les « e-lessons » ou leçons en ligne. Les concepteurs de celles-ci les ont conçues de telle manière que les étudiants soient amenés à utiliser les concepts OO tels que l'utilisation d'objets dès que possible (Satratzemi *et al.*, 2003).

2.3.2 Environnements de développement éducatifs

Egalement conçus dans le but de faciliter l'apprentissage d'un ou de plusieurs types de paradigme via un environnement de développement, les environnements de développement éducatifs constituent un deuxième type d'outils pédagogiques. En plus d'offrir aux étudiants la possibilité de développer une application, ils mettent à la disposition de ceux-ci une série

de fonctionnalités et d'aides qui vont leur permettre de s'améliorer tout en la développant. Etant donné leur vocation pédagogique, ces outils ne sont en général pas aussi complets que les environnements de développement professionnels.

Au fil des années, un certain nombre d'environnements de développement éducatifs ont été conçus afin de permettre aux étudiants d'apprendre les concepts de la programmation OO. Voici une liste reprenant certains de ces environnements :

BlueJ : BlueJ est un des environnements de développement éducatifs les plus connus (Georgantaki et Retalis, 2007, page 114). Il propose une interface relativement simple permettant de visualiser les différentes classes que l'on crée sous la forme d'un diagramme. Ce diagramme, qui pourrait être apparenté à un diagramme de classes UML, offre la possibilité d'indiquer les relations d'héritage entre les classes, permet de créer des packages et permet également d'indiquer par quelle(s) classe(s) une classe est utilisée.

L'implémentation d'une classe (ajout, modification, suppression de méthodes, d'attributs, ...) peut être effectuée en double-cliquant sur celle-ci afin de faire apparaître une fenêtre prévue à cet effet. Cette fenêtre offre également la possibilité de compiler le code de la classe qu'elle contient.

BlueJ offre aussi d'autres fonctionnalités telles qu'un debugger intégré et la possibilité de créer des instances des classes qui ont été développées. L'utilisateur peut interagir avec ces instances afin d'exécuter l'une ou l'autre des méthodes qu'elles contiennent.

Cet outil pédagogique utilise le Java comme langage de programmation et ses créateurs l'ont conçu pour qu'il supporte une approche pédagogique de type « Object-first ». Cela peut être constaté très rapidement car lorsque l'on utilise cet outil, on est directement amené à utiliser les concepts de base du paradigme OO. Ils ont également éliminé des aspects pouvant entraîner une certaine confusion auprès des débutants comme le fait d'être obligé de créer une classe disposant d'une méthode main() pour exécuter son programme (Van Haaster et Hagan, 2004, page 1).

La figure 2.4 permet d'avoir un aperçu de l'interface principale de BlueJ. Cette interface propose de visualiser, à l'aide d'un diagramme de classes simplifié, les différentes classes que l'utilisateur choisit d'y créer. Il peut également instancier ses classes à partir de cette interface, accéder au code de chaque classe, etc.

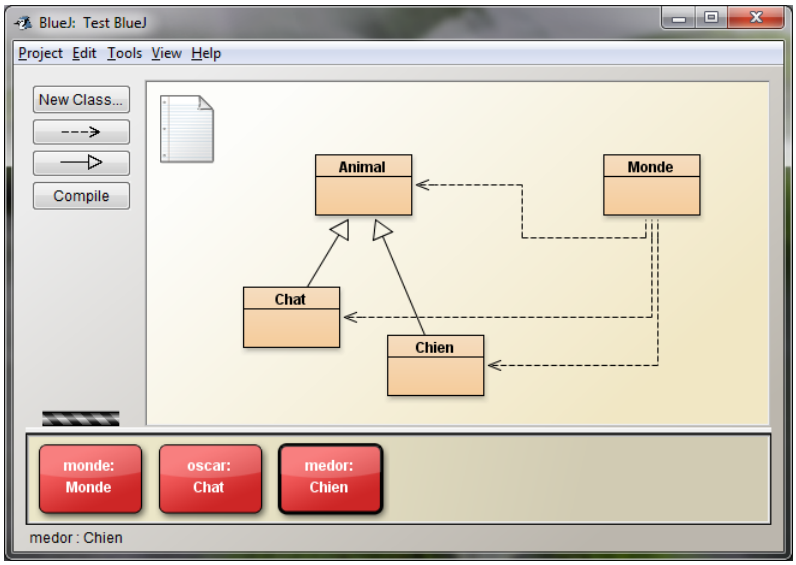

FIGURE 2.4: BlueJ - interface principale

Greenfoot : Greenfoot est un outil qui est considéré comme combinant les avantages des environnements de développement éducatifs ainsi que ceux des micromondes. Il se base à la fois sur BlueJ et sur un micromonde comme Karel Le Robot. Il a été conçu de telle manière qu'il soit possible pour les étudiants de visualiser les classes qu'ils développent ainsi que les relations d'héritage entre celles-ci. L'interface permet également de visualiser les différentes instances créées par les étudiants (Henriksen et Kölling, 2004, pages 4-5).

Tout comme BlueJ, cet outil propose un éditeur auquel l'étudiant peut accéder en double-cliquant sur la représentation de la classe dont il souhaite commencer voire continuer l'implémentation. A partir de cet éditeur, il peut compiler son code, y mettre des points d'arrêt pour la phase de debugging, générer la documentation javadoc, etc.

La figure 2.5 permet de voir l'interface principale de Greenfoot. Elle montre le monde virtuel ainsi que la hiérarchie des différentes classes créées par l'utilisateur. Le monde virtuel, l'ours, les feuilles et les pierres que l'on peut apercevoir sur cette figure sont des instances de chacune des classes présentes dans cette hiérarchie.

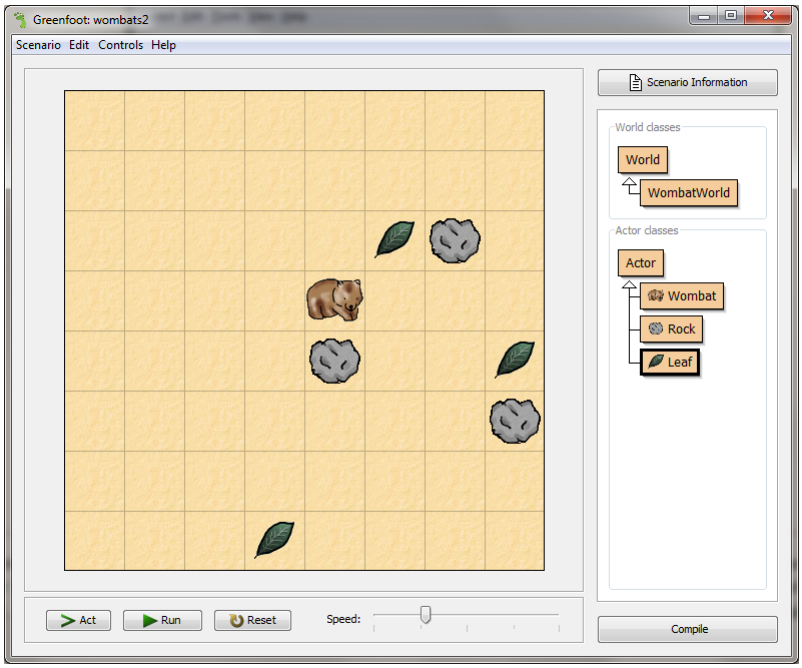

FIGURE 2.5: Greenfoot - interface principale

2.3.3 Environnements de jeu

Les environnements de jeu permettent aux étudiants de programmer en mettant l'emphase sur l'aspect ludique de ce qu'ils sont en train de faire. Il existe, par exemple, un environnement dénommé Robocode. Il s'agit d'un environnement dans lequel des tanks s'affrontent. Il propose aux étudiants de prendre en main le destin d'un tank en particulier. Ils peuvent programmer des bouts de code en Java afin de déterminer comment ce tank doit réagir face à une situation donnée (autrement dit, face à un événement). Par exemple, ils peuvent déterminer comment le tank doit réagir lorsqu'il détecte un autre tank. Le défi est d'arriver à gérer suffisamment bien les différents types d'événement susceptibles de se produire afin que son tank soit le dernier en jeu.

Cet outil permet aux étudiants de manipuler plusieurs concepts de base et avancés de la programmation OO. On peut dire qu'il met notamment l'emphase sur les événements et la manière dont il faut les gérer (Georgantaki et Retalis, 2007, page 117).

2.4 Synthèse

2.4.1 Tableau récapitulatif

Dans le tableau 2.1, nous proposons un récapitulatif des différents micromondes que nous avons décrits dans notre état de l'art. Pour remplir ce tableau en indiquant l'approche et la théorie de l'apprentissage utilisées par chacun de ces outils, nous nous sommes basés sur la description de ceux-ci.

	Comport.	Cognit.	Construct.	Visual.	Edut.	O-F	O-L	Non-OO
Karel le Robot				x				x
Karel++				x		x		
KarelJRobot				x		x		
JKarelRobot				x		x		
Jeroo			x	x		x		
Alice				x	x	x		
objectKarel		x	x	x		x		
BlueJ				x		x		
Greenfoot			x	x	x	x		
Robocode				x	x	x		

TABLE *2.1: Utilisation des approches et théories de l'apprentissage par les différents outils*

Comme vous pouvez le constater, la grande majorité de ces outils utilise, selon nous, l'approche « Object-first » à l'exception de « Karel le Robot » qui est un des premiers micromondes développés avant l'ère du paradigme OO. Nous considérons que tous les outils que nous avons décrits, utilisent également la visualisation.

2.4.2 Réflexion sur le support pédagogique à créer

Le support pédagogique développé dans ce mémoire se base sur l'approche pédagogique « Object-last » et ce, dans le cadre d'un cursus complet et non pas du premier cours comme c'était expliqué dans la première partie de cet état de l'art. Cela signifie que les personnes utilisant ce support sont censées avoir déjà une certaine connaissance dans le domaine de la programmation

impérative avant d'utiliser celui-ci et donc d'avoir déjà eu des cours de programmation.

Outre l'approche pédagogique « Object-last », ce support pédagogique tire profit de concepts provenant de la sous-section concernant les théories de l'apprentissage. Tout d'abord de l'edutainment, le but de ce mémoire étant de développer un support pédagogique ludique. Pour rappel, l'edutainment préconise d'apprendre en s'amusant. Les scénarios créés dans le cadre de ce mémoire l'ont été en se focalisant sur ce principe. De plus, l'aspect recherche est également assez important, les étudiants devant utiliser une API qu'ils ne connaissent pas (leJOS).

Nous nous sommes également inspirés de la visualisation car les étudiants sont amenés à être actifs et à visualiser le produit de leurs efforts. En effet, ils vont par exemple devoir tester comment le robot réagit dans certaines situations en utilisant diverses classes proposées par l'API qu'ils ne connaissent pas.

Ce support pédagogique se base enfin également sur la théorie de l'apprentissage appelée constructivisme, étant donné que nous insistons sur la nécessité d'une bonne connaissance de certains concepts OO avant de passer aux concepts OO plus complexes qui y sont liés.

Chapitre 3

Outils utilisés dans le cadre du projet

Le support pédagogique que nous avons élaboré nécessite l'utilisation de deux outils que nous présentons dans ce chapitre. Il s'agit d'un robot LEGO Mindstorms NXT 2.0 et du firmware leJOS. Ce firmware permet de contrôler le robot à l'aide de programmes réalisés en Java.

3.1 LEGO Mindstorms NXT 2.0

Le LEGO Mindstorms NXT 2.0 est un kit proposé par la société LEGO. Ce kit contient divers composants dont la brique NXT, un certain nombre de pièces à assembler, des capteurs, des moteurs, des câbles et un logiciel. A l'aide de ces divers composants, il est possible de construire un robot, le coeur de celui-ci étant la brique NXT.

La brique NXT est un mini-ordinateur sur lequel on peut exécuter des programmes créés et transférés à l'aide du logiciel fourni par LEGO. Elle dispose de plusieurs ports, certains servant à y connecter les capteurs, d'autres servant à y connecter les moteurs.

Ce logiciel propose des plans pour construire et programmer quatre différents modèles de robot. Ces modèles permettent de construire un robot bipède, un robot sur roues permettant de lancer des balles, un robot ayant l'apparence d'un alligator et un robot immobile qui s'occupe d'effectuer des tris. Pour chacun de ces modèles, LEGO propose des variantes ainsi que des programmes dépendant de celles-ci. Il est également possible de trouver

d'autres modèles sur le site de LEGO ou de se servir de son imagination pour créer son propre modèle.

En plus de permettre de créer et de transférer des programmes ainsi que de fournir des plans pour construire des robots, le logiciel fourni par LEGO peut être utilisé pour envoyer des ordres au robot depuis un ordinateur à l'aide d'une connexion USB ou de la technologie Bluetooth.

Plus d'informations concernant ce kit peuvent être trouvées sur le site officiel de LEGO (en anglais) : http ://mindstorms.lego.com/en-us/default.aspx.

Voici deux figures permettant d'avoir un aperçu des quatre modèles de LEGO Mindstorms NXT 2.0 proposés par LEGO (figure 3.1) ainsi que les différents capteurs disponibles de base (figure 3.2).

FIGURE 3.1: Quelques robots (LEGO, 2011)

FIGURE 3.2: Les différents capteurs de base (LEGO, 2011)

3.2 LeJOS NXJ

Le firmware de base de la brique NXT peut être remplacé par le firmware leJOS NXJ (Lego Java Operating System). Celui-ci offre une API complète permettant de développer des programmes pouvant ensuite être exécutés sur la brique. Ces programmes sont écrits dans le langage de programmation Java. Il propose également une API pour les programmes devant être exécutés sur un ordinateur afin de pouvoir interagir avec la brique. Ces API contiennent des différences notables.

Les commandes permettant de compiler ou d'exécuter (voire transférer sur le robot) les programmes ne sont pas les mêmes selon que le programme soit destiné à être exécuté sur la brique NXT ou sur un ordinateur. Voici une petite liste reprenant les principales commandes :

- nxjc prog.java : permet de compiler le code de ce fichier en vue d'utiliser celui-ci sur une brique NXT. L'API de leJOS utilisée dans prog.java doit être l'API spécifique à la brique NXT ;
- nxj prog : permet de transférer ce programme sur la brique NXT reliée par USB à l'ordinateur sur lequel cette commande est exécutée. Le transfert peut également se faire à l'aide de la technologie Bluetooth pour autant que l'ordinateur et le robot puissent utiliser celle-ci ;
- nxjpcc prog.java : permet de compiler le code de ce fichier en vue d'utiliser celui-ci sur un ordinateur. L'API qui a dû être utilisée dans prog.java doit être l'API pour PC ;
- nxjpc prog : permet d'exécuter ce programme sur un ordinateur.

Dans le cadre de ce mémoire, nous utilisons la version 0.85 de leJOS NXJ sortie en septembre 2009.

Davantage d'informations peuvent être trouvées sur le site officiel de leJOS (en anglais) : http ://lejos.sourceforge.net/. Le firmware peut y être téléchargé et la documentation de l'API consultée.

Chapitre 4

Projet - théorie appliquée pour construire un robot

Ce chapitre constitue la première partie du support pédagogique que nous avons développé. Nous vous y présentons d'abord l'objectif que nous avons cherché à atteindre, la méthode que nous avons employée pour rédiger les différents exercices ainsi qu'une brève description de chacun de ceux-ci à l'aide d'un tableau.

4.1 Objectif

L'objectif que nous cherchons à atteindre dans ce chapitre est de proposer une dizaine de scénarios (i.e. des exercices) permettant à un étudiant de revoir et pratiquer la majorité des concepts OO vus dans le cadre d'un cours de programmation OO. La difficulté de ces scénarios est croissante et les concepts qui y sont abordés sont de plus en plus complexes.

4.2 Méthode

La méthode choisie pour concevoir ce support pédagogique est de découper un scénario en quatre sous-sections. La première sous-section indique quels sont les objectifs que nous poursuivons et propose l'énoncé du scénario. Les objectifs sont plutôt destinés au corps professoral.

La deuxième sous-section concerne les nouveaux points de matière abordés dans ce scénario. Cette sous-section est liée au cours de CPOO dans la

mesure où pour chaque point de matière, nous citons le chapitre dans lequel ce point est abordé.

La troisième sous-section fournit une liste de certaines des erreurs que les étudiants pourraient commettre en tentant de résoudre l'exercice proposé dans le cadre du scénario. Il ne s'agit bien entendu pas d'une liste exhaustive mais des erreurs qui nous semblent les plus probables.

Enfin, la quatrième et dernière sous-section fournit une solution détaillée pour le scénario. Cette solution n'est bien sûr pas la seule et unique solution possible. Cette sous-section propose un maximum d'informations concernant la résolution du scénario et des bouts de code accompagnés de commentaires pour permettre aux étudiants de comprendre comment se servir de certains concepts OO requis dans le cadre de ce scénario.

4.3 Brève description

Le tableau 4.1 présente les différents scénarios de ce chapitre. Chacun de ceux-ci nécessite l'application de concepts OO supplémentaires et la difficulté soit croît, soit reste identique à chaque nouveau scénario. En plus de ces informations, vous pouvez y trouver le nom de chaque scénario ainsi que leur numéro de section.

Scénario	Concepts-Clés	Difficulté	Section
LEGO Mindstorms NXT 2.0	Concepts de base de leJOS	/	4.4
Un robot qui se meut	Classe, objet, encapsulation	Basique	4.5
Un robot qui se meut version 2	Abstraction par spécification, surcharge	Basique	4.6
Un robot détecteur d'obstacles	Héritage	Basique	4.7
Tous des « Object » - RobotEclaireur	Object, override, polymorphisme, exception	Intermédiaire	4.8
Arrays - Vector	Vector, array	Intermédiaire	4.9
ColorList - IntList	Mutabilité, stack et heap, clone(), equals()	Intermédiaire	4.10
Amélioration des types ColorList et IntList	Exception, fonction d'abstraction, invariant de représentation, abstraction par itération, inner class, interface	Complexe	4.11
Adéquation des types ColorList et IntList	Créateur, producteur, mutateur, observateur, programmation défensive, similar()	Complexe	4.12
La classe Robot et ses sous-types revisités	classe abstraite, classe concrète, super-type, sous-type, interface, principe de substitution de Liskov, FA, IR	Complexe	4.13

TABLE 4.1: Résumé des scénarios proposés

L'article « A road map for teaching introductory programming using LEGO mindstorms robots » (Lawhead *et al.*, 2002) nous a servi d'inspiration, en particulier, pour les scénarios « Un robot qui se meut », « Un robot qui se meut version 2 » et « Un robot détecteur d'obstacles ».

4.4 LEGO Mindstorms NXT 2.0 - quelques classes de base

4.4.1 Objectif

L'objectif de ce point-ci est de montrer quelles sont les classes de base permettant de contrôler un LEGO Mindstorms NXT 2.0 à l'aide du firmware leJOS.

La classe **Motor** : cette classe représente un port de moteur. Un LEGO Mindstorms NXT 2.0 en a trois. Ils peuvent être accédés à l'aide des objets A, B et C (représentant chacun un des ports de moteur). Cette classe propose un certain nombre de méthodes dont :
- forward() : mettre un moteur en mode marche avant ce qui permet, par exemple, de faire avancer ou tourner un robot sur roues ;
- backward() : mettre un moteur en mode marche arrière ce qui permet, par exemple, de faire reculer ou tourner un robot sur roues.

La classe **SensorPort** : cette classe représente un port de capteur. Il y a quatre ports de capteur sur le LEGO Mindstorms NXT 2.0. Ils sont accessibles à l'aide de cette classe en utilisant S1, S2, S3 et S4 (représentant chacun un des ports de capteur).

La classe **UltrasonicSensor** : cette classe représente le capteur à ultrasons disponible de base avec le LEGO Mindstorms NXT 2.0. Ce capteur permet au robot de repérer un obstacle se trouvant jusqu'à 250 cm devant lui.

La classe **TouchSensor** : cette classe représente le capteur de contact. Deux capteurs de ce type sont disponibles de base avec le LEGO Mindstorms NXT 2.0. Ce capteur permet, entre autres, de savoir si le robot a touché un obstacle.

La classe **ColorLightSensor** : cette classe représente le capteur de couleurs. Il est, comme les deux types de capteur présentés auparavant, lui aussi disponible de base avec le LEGO Mindstorms NXT 2.0. Ce capteur permet d'effectuer différentes opérations : détecter la couleur d'un objet se trouvant devant lui, détecter la luminosité de l'endroit où le robot se trouve et allumer une lampe en rouge, vert ou bleu.

4.4.2 Remarques

Une description complète de l'API du firmware leJOS est disponible sur le site suivant : http ://lejos.sourceforge.net/. Une telle API démontre déjà l'importance de la modularité dans les programmes OO. On peut déjà constater la présence de différentes classes proposant des services bien distincts et n'étant pas dépendantes les unes des autres.

4.5 Scénario - un robot qui se meut

4.5.1 Objectif

L'objectif de ce scénario est de créer une classe Robot permettant à un robot d'avancer, de reculer, de tourner à gauche et de tourner à droite. Cette classe doit également permettre à un robot d'utiliser son écran LCD. Bien qu'il soit possible d'utiliser les différentes classes vues à la section 4.4, pour des raisons strictement pédagogiques, il est demandé de ne les utiliser que dans la classe Robot. Il est cependant permis de les passer en paramètre à cette classe.

Après avoir terminé le développement de la classe Robot, nous vous proposons de la tester en créant un petit programme permettant à un robot de bouger pendant quelques secondes. Attention cependant aux obstacles.

4.5.2 Matière

Plusieurs points de la matière vue au cours de CPOO doivent déjà être utilisés pour réaliser cet exercice correctement :

- **Abstraction de données** : création d'une **classe** (constructeur, attributs, méthodes, ...) ;
- Utilisation de classes déjà existantes, c'est-à-dire de se servir du principe de **modularité** mais également d'**objets** ;
- La création d'une classe demande aussi de pouvoir appliquer le principe d'**encapsulation**. Cela suppose que l'étudiant ait bien compris l'utilité de la **visibilité** des attributs, des méthodes et des classes. Pour rappel : **public**, **private** et **protected**.

4.5.3 Erreurs possibles

- Créer une classe donnant accès à des attributs censés être « **private** » (à part éventuellement à l'aide d'une méthode) ;
- Obliger l'utilisateur de la classe à devoir construire son robot de manière à ce que le moteur gauche soit par exemple toujours le moteur A et le moteur droit toujours le C.

4.5.4 Solution

Avant de concevoir une classe, il est important de se demander quels services elle va devoir offrir. Dans le cadre de ce scénario, il faut créer une classe permettant à un robot de se mouvoir. Il est également demandé de passer obligatoirement par cette classe pour avoir accès aux moteurs de ce robot. On peut donc déduire les choses suivantes :

- La classe va disposer de deux attributs privés : Motor moteurGauche, Motor moteurDroit ;
- Elle va disposer d'un constructeur permettant à ces attributs d'être assignés. Les moteurs pourraient être passés en paramètre à ce constructeur afin que l'utilisation de cette classe ne dépende pas des ports utilisés par les deux moteurs. Par exemple : le moteur gauche doit être le moteur A, le moteur droit toujours le C ;
- Elle va également disposer des six méthodes publiques suivantes conformément à ce qui a été demandé :
  - avancer() : permet de demander au robot d'avancer en mettant les deux moteurs en mode marche avant ;
  - reculer() : permet de demander au robot de reculer en mettant les deux moteurs en mode marche arrière ;
  - tournerGauche() : permet de demander au robot de tourner à gauche en mettant le moteur gauche en marche arrière et le droit en marche avant ;
  - tournerDroite() : permet de demander au robot de tourner à droite en mettant le moteur droit en marche arrière et le gauche en marche avant ;
  - arreter() : permet de demander au robot de s'arrêter de bouger ;
  - ecrireSurEcranLCD(string message) : permet d'écrire le message passé en paramètre sur l'écran LCD du robot.

```
// partie de la solution en Java
public class Robot
{
    private Motor moteurGauche;
```

```
    private Motor moteurDroit;

    public Robot(Motor mGauche, Motor mDroit){
        moteurGauche = mGauche;
        moteurDroit = mDroit;
    }

    public void avancer() {
        moteurGauche.forward();
        moteurDroit.forward();
    }

    ...
}
```

Cette classe pourrait offrir des services supplémentaires. Elle pourrait par exemple permettre de demander au robot de s'arrêter après avoir avancé pendant x secondes. Ce point sera abordé au scénario suivant (section 4.6).

Après avoir terminé le développement de cette classe, on peut créer un petit programme qui va la tester. Il pourrait proposer de tester chacune des méthodes disponibles (faire avancer le robot, le faire reculer, tourner à gauche, à droite, s'arrêter et écrire sur l'écran LCD « test terminé »). Cela va demander de créer une instance de la classe Robot.

```
public static void main (String[] aArg) throws Exception {
    Robot monRobot = new Robot(Motor.A, Motor.C);
    monRobot.avancer();
    Thread.sleep(2000);
    monRobot.reculer();
    Thread.sleep(1000);
    monRobot.tournerGauche();
    Thread.sleep(1500);
    monRobot.tournerDroite();
    Thread.sleep(500);
    monRobot.ecrireSurEcranLCD("test␣terminé");
    Thread.sleep(2000);
}
```

4.6 Scénario - un robot qui se meut version 2

4.6.1 Objectif

Lors du scénario précédent (section 4.5), une classe Robot permettant à un robot LEGO Mindstorms NXT 2.0 de se mouvoir a été créée. Cette classe permet de contrôler celui-ci en utilisant l'**interface** qu'elle offre, c'est-à-dire le constructeur et toutes les méthodes publiques dont elle dispose. L'**implémentation**, c'est-à-dire comment les méthodes publiques et le constructeur ont été implémentés, est quant à elle cachée.

L'objectif du scénario actuel est de proposer, en plus des méthodes existantes, quatre méthodes supplémentaires permettant au robot de se mouvoir pendant un certain nombre de secondes. Ces quatre méthodes fourniraient à la base le même service que les méthodes avancer, reculer, tournerGauche et tournerDroite mais devraient permettre d'indiquer pendant combien de temps le robot doit avancer, reculer, tourner à gauche ou tourner à droite avant de s'arrêter. Il faudrait cependant respecter la précondition suivante : la valeur du paramètre devrait être supérieure à 0.

```
//REQUIRES: ms > 0
//EFFECTS: le robot avance pendant ms milliseconde(s) et s'arrête.
public void avancer(int ms)
```

Comme dans le scénario précédent (section 4.5), après avoir modifié la classe Robot nous vous demandons de tester les fonctions ajoutées.

4.6.2 Matière

Matière appliquée dans ce scénario :
- L'**abstraction par spécification** (chapitre 1 du cours de CPOO) ;
- La **surcharge** de méthodes (chapitre 2 du cours de CPOO).

4.6.3 Erreurs possibles

- Une erreur possible serait de se contenter de modifier les méthodes existantes en leur ajoutant un paramètre et qu'il ne soit plus possible de les appeler sans celui-ci ;
- Une autre erreur possible serait de ne pas réutiliser les méthodes existantes dans les nouvelles. En effet, une duplication du code rendrait la maintenance de celui-ci plus complexe ;

– Effectuer une surcharge en ne changeant que le type de retour d'une méthode (le programme ne serait pas en mesure de dire quelle méthode il faut utiliser).

4.6.4 Solution

Une solution possible pour ce scénario serait de faire comme suit :
– Pour chaque méthode de mouvement, créer une méthode portant le même nom et disposant d'un paramètre, par exemple : avancer(int ms) ;
– Dans chacune de ces méthodes, commencer par appeler la version sans paramètre. Par exemple dans avancer(int ms), appeler avancer() ;
– Ensuite attendre pendant x millisecondes avant d'effectuer l'opération suivante ;
– Et pour finir, appeler la méthode arreter().
Pour la partie demandant d'attendre pendant x millisecondes avant d'effectuer l'opération suivante, il est conseillé de créer une méthode supplémentaire. Celle-ci pourrait être nommée « pause » et demanderait de fournir en paramètre un entier représentant des millisecondes. Par exemple : pause(int ms). L'objectif de cette méthode serait d'appeler Thread.sleep(ms) et d'éviter que le code soit difficile à maintenir si l'on souhaite à un moment demander au robot d'afficher un message lorsqu'il attend pendant x secondes.

Exemple :

```
public class Robot {
    ...

    //REQUIRES: ms > 0
    //EFFECTS: faire dormir le thread pendant ms millisecondes
    public void pause(int ms) {
        ...
        Thread.sleep(ms);
        ...
    }
}
```

Après avoir terminé de modifier la classe Robot, il est conseillé de tester chacune des nouvelles méthodes.

```
public static void main (String[] aArg) throws Exception {
    Robot monRobot = new Robot(Motor.A, Motor.C);
    // avancer pendant deux secondes et arrêter.
    monRobot.avancer(2000);
```

```
    monRobot.reculer(1000);
    monRobot.tournerGauche(1500);
    monRobot.tournerDroite(500);
}
```

4.7 Scénario - un robot détecteur d'obstacles

4.7.1 Objectif

L'objectif de ce scénario est de créer une classe RobotDetecteur permettant à un robot de détecter un obstacle à l'aide d'un capteur à ultrasons. Il faut donc que cette classe dispose d'une méthode permettant de détecter si un obstacle se trouve à moins de x cm du robot. Il faut également que cette classe permette au robot de se mouvoir, c'est-à-dire d'avancer, de reculer, etc.

Une restriction supplémentaire est imposée dans ce scénario. Comme pour les moteurs, le capteur à ultrasons ne doit pas être directement accessible en-dehors de la classe. Il faut fournir en paramètre du constructeur de celle-ci le port auquel le capteur est relié. Il est cependant permis de créer des méthodes permettant de récupérer des données provenant de ce capteur.

Il faut ensuite créer un programme permettant de tester cette classe. Ce programme doit permettre au robot de prendre une initiative en cas de détection d'un obstacle. Dès son activation, le robot doit aller tout droit. S'il détecte un obstacle à moins de 30 cm, il doit reculer pendant une seconde et ensuite tourner à gauche ou à droite pendant un certain nombre de millisecondes. Il peut ensuite recommencer à avancer si il a le champ libre. Après avoir détecté un certain nombre d'obstacles, il peut s'arrêter.

4.7.2 Matière

La matière appliquée dans le cadre de ce scénario est l'**héritage** (ou **hiérarchie de types**). Cette matière est vue au chapitre 1, au chapitre 2 et au chapitre 5 du cours de CPOO.

4.7.3 Erreurs possibles

– Ne pas réutiliser la classe Robot lors de la création de la classe RobotDetecteur, c'est-à-dire écrire toutes les méthodes dont la classe Robot

dispose déjà à nouveau (avancer, reculer, tournerGauche, tournerDroite, ...) ;
- Ne pas avoir un constructeur permettant de fournir en paramètre le moteur gauche et le moteur droit de la classe Robot ainsi que le port auquel le capteur à ultrasons est relié. Ne pas permettre de fournir ces paramètres à la classe RobotDetecteur aurait le même effet que décrit dans le scénario concernant la classe Robot. L'utilisateur de cette classe serait obligé de construire son robot en tenant compte du fait que le moteur gauche doit toujours être relié au port A, que le capteur à ultrasons doit toujours être relié au port 1, etc ;
- Ne pas utiliser la méthode super() de la bonne manière dans le constructeur de la classe RobotDetecteur, c'est-à-dire par exemple donner en paramètre à cette méthode le moteur droit et ensuite le moteur gauche alors que le constructeur de la classe Robot demande de faire l'inverse.

4.7.4 Solution

Voici une des solutions possibles pour mener à bien ce scénario :
- Créer une nouvelle classe RobotDetecteur qui **hérite** de la classe Robot. Il faut utiliser « **extends** » pour ce faire : « class RobotDetecteur **extends** Robot » ce qui a pour effet de donner aux futurs utilisateurs de la classe RobotDetecteur l'accès aux méthodes auxquelles ils avaient accès via la classe Robot (avancer(), reculer(), ...) ;
- Cette classe doit disposer d'un nouvel attribut privé : UltrasonicSensor capteurUltrasons ;
- Avoir un constructeur permettant de fournir les paramètres suivants : le moteur gauche, le moteur droit et le port auquel le capteur à ultrasons est relié : « RobotDetecteur(Motor gauche, Motor droit, SensorPort portCapteurUltrasons) ». Cela permet aux utilisateurs de cette classe d'être plus libres lorsqu'ils construisent leur robot et relient les moteurs et le capteur à ultrasons à leur port respectif ;
- Le constructeur doit faire appel à la méthode super() et lui fournir en paramètre le moteur gauche et le moteur droit : « super(gauche, droit) » ;
- Le constructeur doit également créer une instance de la classe UltrasonicSensor et l'assigner à l'attribut capteurUltrasons : « capteurUltrasons = new UltrasonicSensor(portCapteurUltrasons) » ;
- Il faut également au moins une méthode permettant de récupérer des informations provenant du capteur à ultrasons. Par exemple : getDistance() qui retournerait un entier et qui appellerait pour ce faire, à l'aide

de capteurUltrasons, la méthode du même nom disponible via la classe UltrasonicSensor.

Voici ce que cela pourrait donner :

```
public class RobotDetecteur extends Robot {
    private UltrasonicSensor capteurUltrasons;

    public RobotDetecteur(Motor gauche, Motor droit,
                          SensorPort portCapteurUltrasons) {
        super(gauche, droit);
        capteurUltrasons = new UltrasonicSensor(portCapteurUltrasons);
    }

    public int getDistance() {
        return capteurUltrasons.getDistance();
    }
}
```

Pour tester cette classe, il faut ensuite créer un programme permettant à un robot de bouger tout en faisant attention aux obstacles qui pourraient se trouver devant lui. Comme expliqué plus haut, il devrait s'arrêter après avoir détecté un certain nombre d'obstacles. Voici à quoi pourrait ressembler ce programme :

```
public static void main (String[] aArg) throws Exception {
    int nbDetections = 4;
    RobotDetecteur monRobot = new RobotDetecteur(Motor.A, Motor.C,
                                  SensorPort.S1);
    monRobot.avancer();
    while(nbDetections > 0) {
        if(monRobot.getDistance() < 30) {
            nbDetections--; // obstacle détecté
            monRobot.reculer(1000);
            monRobot.tournerGauche(1500);
            monRobot.avancer();
        }
    }
}
```

4.8 Scénario - Tous des « Object » - RobotEclaireur

4.8.1 Objectif

L'objectif principal de ce scénario est d'amener l'étudiant à utiliser deux des méthodes de la classe Object, à les redéfinir si nécessaire et à utiliser la notion de polymorphisme. Ce scénario l'amène également à utiliser à nouveau les concepts de l'héritage et de la surcharge de méthodes et lui

propose d'essayer de comprendre comment utiliser le capteur de couleur du LEGO Mindstorms NXT 2.0.

Enoncé :

La classe RobotEclaireur est un type de données permettant à un robot d'utiliser le capteur de couleur du LEGO Mindstorms NXT 2.0 afin d'allumer sa lampe en rouge, en vert ou en bleu.

Voici les particularités que l'on attend de cette nouvelle classe :
- Le robot doit pouvoir se mouvoir ;
- Il doit également pouvoir afficher un message sur son écran LCD ;
- Une méthode doit lui permettre d'allumer sa lampe en rouge, vert ou bleu ;
- Une méthode doit lui permettre d'éteindre sa lampe ;
- Une méthode doit lui permettre d'allumer sa lampe dans une des trois couleurs mentionnées ci-dessus pendant un temps donné.

Dans le cadre de ce scénario, il est demandé de redéfinir la méthode toString() au niveau de la classe Robot, de la classe RobotDetecteur et de la classe RobotEclaireur. Cette méthode devra permettre de retourner le nom de la classe dans laquelle elle se trouve et éventuellement le nom de celle dont elle dérive (e.g. « RobotEclaireur - Robot »).

Voici les principaux objectifs à atteindre au niveau du programme de test :

1. Créer une instance de chacun des trois types mentionnés plus haut ainsi qu'une instance de la classe Object ;
2. Afficher sur l'écran du robot le message retourné par la méthode toString() de chacune de ces quatre instances ;
3. Utiliser la méthode equals(Object) à l'aide de l'instance de la classe Robot et fournir l'une après l'autre les quatre différentes instances en paramètre :

    a) Si la valeur de retour de cette méthode est « true » - alors allumer la lumière verte du robot pendant deux secondes ;

    b) Si celle-ci est « false » - alors allumer la lumière rouge pendant deux secondes.

Quelques questions pouvant être vérifiées à partir du programme de test :

1. Que se passerait-il si l'on assignait l'instance de la classe Robot à la variable contenant celle de la classe Object ?

    a) La valeur de retour de la méthode toString() sur cette variable serait-elle différente que lors de l'appel effectué la première fois ?

b) Si l'on effectue à nouveau le point 3 des principaux objectifs à atteindre, le résultat serait-il différent de la première fois ?

2. Il est possible de caster un Robot en un Object :

   a) Quand est-il possible d'effectuer l'opération inverse ?

   b) Est-il nécessaire d'effectuer une conversion explicite pour caster un Robot en un Object ?

   c) Quid lorsque l'on veut effectuer l'opération inverse ?

4.8.2 Matière

Matière appliquée dans le cadre de ce scénario :
- L'**héritage** ou **hiérarchie de types** : chapitres 1, 2, 5 et 7 du cours de CPOO ;
- **Object**, la classe dont dérive toute classe : chapitre 2 ;
- La **surcharge** de méthode : chapitre 2 ;
- L'**override** de méthodes : chapitre 4 ;
- Le conversion (**transtypage** ou **casting**) d'un type d'objet en un autre (notion de **polymorphisme**) : chapitre 2 ;
- La notion d'**exception** : chapitre 2 pour l'erreur de casting, chapitre 4 pour des informations sur les exceptions en général.

4.8.3 Erreurs possibles

- Ne pas hériter de la classe Robot pour permettre au robot de bouger et de pouvoir afficher des messages sur son écran ;
- L'affectation d'une instance de Robot à une variable de type RobotEclaireur ou RobotDetecteur.

4.8.4 Solution

Cette solution est découpée en trois parties :

1. La première concerne le développement de la classe RobotEclaireur ;
2. La deuxième concerne les objectifs qu'il fallait atteindre dans le programme de test ;
3. La troisième concerne les réponses aux questions posées.

Rappel : Barbara Liskov décrit le polymorphisme comme « généralisant les abstractions afin qu'elles fonctionnent pour beaucoup de types ». Elle explique également que « cela nous permet d'éviter de devoir redéfinir des abstractions quand nous voulons les utiliser pour plusieurs types. » (Liskov et Guttag, 2001, page 190).

Création de la classe RobotEclaireur :

- L'énoncé demande la création d'une classe RobotEclaireur permettant entre autres à un robot de bouger et d'afficher un message sur son écran LCD. Ces fonctionnalités étant déjà disponibles via la classe Robot, il suffit que la nouvelle classe hérite de celle-ci pour qu'il lui soit aussi possible d'offrir ces fonctionnalités : « `public class RobotEclaireur extends Robot` » ;
- Cette classe doit disposer d'un attribut privé : `private ColorLightSensor capteurCouleur;`
- Avoir un constructeur permettant de fournir les paramètres suivants : le moteur gauche, le moteur droit et le port auquel le capteur de couleur est relié : « `RobotDetecteur(Motor gauche, Motor droit, SensorPort portCapteurCouleur)` » ;
- Dans le constructeur, il faut créer une instance de `ColorLightSensor` et l'assigner à `capteurCouleur` ;
- Il faut créer trois méthodes publiques :
  - `allumerCouleur(Colors.Color col)` : permet d'allumer la lampe dans la couleur fournie en paramètre (rouge, vert ou bleu) ;
  - `allumerCouleur(Colors.Color col, int ms)` : idem que `allumerCouleur(Colors.Color col)` mais la lampe s'éteint après ms millisecondes ;
  - `eteindreLampe()` : permet d'éteindre la lampe du capteur de couleur.
- La redéfinition de la méthode toString() (doit être également effectuée dans Robot et dans RobotDetecteur).

Voici ce que ça pourrait donner :

```
public class RobotEclaireur extends Robot {
    private ColorLightSensor capteurCouleur;

    public RobotEclaireur(Motor gauche, Motor droit,
                SensorPort portCapteurCouleur) {
        super(gauche, droit);
        capteurCouleur = new ColorLightSensor(portCapteurCouleur, 0);
    }

    public void allumerCouleur(Colors.Color col) {
        capteurCouleur.setFloodlight(col);
    }

    public void allumerCouleur(Colors.Color col, int ms) {
        capteurCouleur.setFloodlight(col);
```

```
        this.pause(ms);
        eteindreLampe();
    }

    public void eteindreLampe() {
        capteurCouleur.setFloodlight(false);
    }

    public String toString() {
        return "RobotEclaireur␣-␣Robot";
    }
}
```

Programme de test :

Dans le programme de test, il est tout d'abord demandé de créer une instance des trois types de robot différents et une instance d'Object :

```
public static void main (String[] aArg) throws Exception {
    Robot robot = new Robot(Motor.A, Motor.C);
    RobotDetecteur robotDetecteur = new RobotDetecteur(Motor.A,
                            Motor.C, SensorPort.S1);
    RobotEclaireur robotEclaireur = new RobotEclaireur(Motor.A,
                            Motor.C, SensorPort.S2);
    Object object = new Object();
```

Il faut ensuite afficher sur l'écran du robot le résultat de l'appel à la méthode toString() sur chacune des quatre instances créées au point précédent :

```
    ...
    robot.ecrireSurEcranLCD(robot.toString());
    // attendre deux secondes avant de passer à l'instruction suivante
    robot.pause(2000);
    robot.ecrireSurEcranLCD(robotDetecteur.toString());
    robot.pause(2000);
    robot.ecrireSurEcranLCD(robotEclaireur.toString());
    robot.pause(2000);
    robot.ecrireSurEcranLCD(object.toString());
```

Pour terminer, il est demandé d'appeler la méthode equals(Object) de la variable robot en passant chacune des instances créées. Selon la valeur de retour de cette méthode, il faut ensuite allumer la lampe du capteur de couleur en vert ou en rouge pendant deux secondes :

```
    ...
    if (robot.equals(object))
        robotEclaireur.allumerCouleur(Colors.Color.GREEN, 2000);
    else robotEclaireur.allumerCouleur(Colors.Color.RED, 2000);

    if (robot.equals(robot))
        robotEclaireur.allumerCouleur(Colors.Color.GREEN, 2000);
    else robotEclaireur.allumerCouleur(Colors.Color.RED, 2000);
    ...
```

Réponses aux questions posées :

Question 1 :

Si l'on assigne robot à object et que l'on appelle la méthode toString() d'object, celle-ci retournera « Robot » contrairement à la fois précédente car cette fois-ci la variable object contient une instance de la classe Robot et non plus une instance de la classe Object.

```
object = robot;
robot.ecrireSurEcranLCD(object.toString());
```

Si l'on appelle la méthode equals(Object) de robot en fournissant object en paramètre, celle-ci retournera cette fois-ci « true » au lieu de « false » et donc la lumière verte du capteur de couleur sera allumée pendant deux secondes.

```
if (robot.equals(object))
    robotEclaireur.allumerCouleur(Colors.Color.GREEN, 2000);
else robotEclaireur.allumerCouleur(Colors.Color.RED, 2000);
```

Question 2 :

- Il est possible de caster un Robot en un Object. Pour ce faire, il suffit d'assigner une instance de Robot à une variable de type Object ;
- Il est possible d'effectuer l'opération inverse uniquement lorsque la variable de type Object contient une instance de Robot ou d'une de ses classes dérivées ;
- Une conversion implicite suffit pour caster un Robot en un Object (ou e.g. un RobotDetecteur en un Robot) ;
- Lorsque l'on effectue l'opération inverse (caster un Object en un Robot), il faut effectuer une conversion explicite. Il faut donc préciser le type lors de l'assignation du contenu de la variable object à la variable robot.

```
// conversion implicite
object = robot;
// conversion explicite
robot = (Robot) object;

// autre exemple:
object = robotEclaireur;
robot = (Robot) object;
robotEclaireur = (RobotEclaireur) robot;
```

4.9 Scénario - Arrays - Vector

4.9.1 Objectif

L'objectif principal de ce scénario est d'amener l'étudiant à utiliser les arrays et les objets de type Vector. L'étudiant va être également amené à utiliser à nouveau le concept de conversion de type appliqué lors du scénario précédent (section 4.8). Ce scénario va également permettre à l'étudiant d'utiliser deux autres fonctionnalités offertes par le capteur de couleur du LEGO Mindstorms NXT 2.0.

Enoncé :

La classe RobotEclaireur a été développée dans le but de permettre d'allumer la lampe du capteur de couleur en vert, en rouge ou en bleu. Vous devrez compléter cette classe afin qu'elle permette d'utiliser deux autres fonctionnalités du capteur de couleur :
- Une fonctionnalité permettant de détecter la couleur de l'objet se trouvant devant le capteur ;
- Une autre fonctionnalité permettant de détecter le niveau de luminosité de l'endroit où se trouve le robot.

Dans le cadre de ce scénario, le programme à développer doit se servir de cette classe pour détecter la couleur (verte, rouge, ou bleue) de l'objet se trouvant devant le capteur. Dès que le capteur de couleur aura permis de détecter au moins une fois chacune de ces trois couleurs, le programme rejouera dans l'ordre les différentes couleurs détectées (e.g. : rouge, rouge, bleu, rouge, vert) en utilisant sa lampe.

Comme vu au cours de CPOO, la classe Vector permet de stocker des objets de type Object, autrement dit n'importe quel type à part les types primitifs. Vous pouvez vous en servir pour stocker dans l'ordre les différentes couleurs détectées. Cela vous permettra de pouvoir facilement rejouer les différentes couleurs détectées par le capteur de couleur. Dès que les différentes couleurs détectées ont été rejouées, le programme devra stocker le vecteur dans une array de Vector (Vector[]) de taille 2.

Ensuite, le robot devra entrer en mode veille et ne se réactiver que dès qu'il aura détecté que la luminosité est descendue en-dessous de 50% à l'aide de son capteur.

Les objectifs principaux de ce programme sont :

1. De détecter toutes les cinq secondes la couleur de l'objet se trouvant devant le capteur ;

2. Chaque fois qu'une couleur est détectée par le capteur, le programme doit la stocker dans un vecteur ;
3. Dès que les trois types de couleur ont été détectées, le programme doit rejouer, dans l'ordre, les différentes couleurs détectées ;
4. Ensuite le programme doit stocker le vecteur dans une array de Vector de taille 2 ;
5. Après avoir stocké le vecteur dans l'array, le robot doit créer une nouvelle instance de Vector, se mettre en mode veille et attendre que la luminosité détectée soit de moins de 50% avant de recommencer à détecter les couleurs ;
6. Dès que les trois types de couleur ont été détectées à nouveau, le programme devra les rejouer dans l'ordre comme au scénario de la section 4.8 et stocker le vecteur dans la deuxième case disponible de l'array de Vector ;
7. Pour terminer, le programme devra rejouer toutes les couleurs se trouvant dans les deux vecteurs stockés dans l'array.

4.9.2 Matière

Matière appliquée dans le cadre de ce scénario :
- La classe **Vector** : chapitre 2 du cours de CPOO ;
- Les **arrays**.

4.9.3 Erreurs possibles

- Ne pas utiliser de casting lorsque l'on récupère une couleur du vecteur ;
- Utiliser une des cases de l'array de Vector comme si un vecteur y avait déjà été assigné alors que ce n'est pas le cas.

4.9.4 Solution

La première chose à faire est de reprendre la classe RobotEclaireur développée au scénario de la section 4.8 et de lui ajouter deux méthodes :
- `Colors.Color detecterCouleur()` pour détecter la couleur de l'objet se trouvant en face du capteur de couleur ;
- `int detecterNivLuminosite()` pour détecter le niveau actuel de luminosité là où le robot se trouve ;

Voici comment ces ajouts pourraient être implémentés.

Attention : la méthode capteurCouleur.readColor() qui aurait pu s'occuper toute seule de déterminer la couleur de l'objet en face du capteur ne semble pas être compatible avec le capteur de couleur de LEGO Mindstorms NXT 2.0.

```
public class RobotEclaireur extends Robot {
    ...

    Colors.Color detecterCouleur() {
        int[] couleurs = capteurCouleur.getColor();
        Colors.Color coul;

        if ((couleurs[0] == couleurs[1]) &&
            (couleurs[0] == couleurs[2]))
            coul = Colors.Color.NONE;
        else {
            coul = Colors.Color.RED;
            int val = couleurs[0];
            if (val < couleurs[1]) {
                val = couleurs[1];
                coul = Colors.Color.GREEN;
            }
            if (val < couleurs[2]) {
                coul = Colors.Color.BLUE;
            }
        }

        return coul;
    }

    int detecterNivLuminosite() {
        return capteurCouleur.getLightValue();
    }
}
```

La deuxième chose à faire est d'écrire le programme qui utilisera la classe RobotEclaireur pour faire ce qui est demandé dans l'énoncé :

1. Il faut tout d'abord créer les variables suivantes :
    a) Vector vectCol - le vecteur de couleurs ;
    b) Vector[] tabVecteurs - l'array de vecteurs qui doit être de taille 2.

2. Il faut également créer les variables suivantes :
    a) int indiceTabVecteur - pour savoir dans quelle case de tabVecteurs insérer le prochain vecteur ;
    b) int nbDetectDiff - pour connaître le nombre de détection de couleurs différentes (3 max).
3. Il faut ensuite coder de sorte qu'il permette de :
    a) Détecter au moins une fois la couleur bleue, la rouge et la verte (une détection par cinq secondes) ;
    b) Insérer chaque détection de bleu, rouge ou vert dans le vecteur de couleurs ;
    c) Rejouer dans l'ordre toutes les couleurs détectées se trouvant dans le vecteur de couleurs ;
    d) Insérer le vecteur de couleurs dans l'array de vecteurs et attendre de détecter que le niveau de luminosité soit inférieur à 50% avant de recommencer à partir du point (a) avec une nouvelle instance de vecteur ;
    e) De rejouer dans l'ordre le contenu des deux vecteurs de couleurs.

Voici un exemple de ce que ce programme pourrait donner :

```
private static boolean isColorOk(Colors.Color coul) {
    boolean res;

    if (Colors.Color.RED == coul ||
        Colors.Color.GREEN == coul ||
        Colors.Color.BLUE == coul)
        res = true;
    else res = false;

    return res;
}

private static void rejouerCouleurs(RobotEclaireur robot, Vector vect) {
    for (int i = 0; i < vect.size(); i++) {
        robot.allumerCouleur((Colors.Color) vect.elementAt(i), 1000);
    }
}

public static void main (String[] aArg) {
    RobotEclaireur robot = new RobotEclaireur(Motor.A,
                            Motor.C, SensorPort.S2);

    // le vecteur de couleurs
    Vector vectCol = new Vector();
    // l'array de vecteurs de couleurs (taille: 2)
    Vector[] tabVecteurs = new Vector[2];

    int nbDetectDiff;
```

```
    int indiceTabVecteur = 0;

    Colors.Color couleur;
    boolean coulTrouvee;

    int luminosite = 0;
    boolean lumTropElevee = true;

    while(indiceTabVecteur < 2) {

        nbDetectDiff = 0;

        while(nbDetectDiff < 3) {
            couleur = robot.detecterCouleur();

            if (isColorOk(couleur) == true) {
                coulTrouvee = false;
                for (int i = 0; i < vectCol.size() && !coulTrouvee; i++) {
                    if (((Colors.Color) vectCol.elementAt(i)) == couleur)
                        coulTrouvee = true;
                }
                if (!coulTrouvee)
                    nbDetectDiff++;

                vectCol.addElement(couleur);
            }
            robot.pause(5000);
        }

        rejouerCouleurs(robot, vectCol);
        tabVecteurs[indiceTabVecteur] = vectCol;
        vectCol = new Vector();

        indiceTabVecteur++;
        lumTropElevee = true;

        while (lumTropElevee) {
            robot.pause(5000);
            luminosite = robot.detecterNivLuminosite();
            if (luminosite < 50)
                lumTropElevee = false;
        }
    }

    for (int i = 0; i < indiceTabVecteur; i++) {
        rejouerCouleurs(robot, tabVecteurs[i]);
    }
}
```

4.10 Scénario - ColorList - IntList

4.10.1 Objectif

L'objectif principal de ce scénario est de faire pratiquer le concept de mutabilité à l'étudiant. Il va devoir y créer un type mutable et un autre immutable. Dans le cadre de ce scénario, l'étudiant va également être amené :
- à représenter ce qu'il fait au niveau du stack et de la heap ;
- à utiliser des wrapper types afin d'insérer des types primitifs dans un vecteur ;
- à faire attention au principe de clonâge avec le type mutable développé dans le cadre de ce scénario ;
- à faire attention à la méthode equals() au niveau du type immutable qu'il créera.

Afin d'utiliser les différents concepts mentionnés ci-dessus, l'étudiant va devoir développer un programme similaire à celui développé lors du scénario précédent (section 4.9) c'est-à-dire permettre de rejouer les différentes couleurs détectées par le capteur de couleur et ce dans l'ordre de détection de ces couleurs.

Enoncé :

Dans le cadre de ce scénario, il vous est demandé de créer les deux types de données suivants :

1. ColorList : un type mutable permettant de stocker les couleurs rouges, vertes et bleues ;
2. IntList : un type immutable permettant de stocker des entiers (sous la forme d'Integer). IntList doit être un type immutable parce que nous prévoyons qu'il soit partagé plus tard par différents objets au sein d'un même programme.

Le type mutable ColorList doit disposer des constructeurs et des méthodes suivants :
- `ColorList()` - permet de créer un ColorList vide ;
- `ColorList(Vector)` - permet de créer un ColorList contenant les couleurs disponibles dans le vecteur fourni en paramètre ;
- `void addColor(Colors.Color)` - permet d'ajouter une couleur verte, rouge ou bleue dans ce vecteur de couleurs ;
- `void removeColor(int)` - permet de supprimer la couleur se trouvant à l'indice fourni en paramètre ;
- `Colors.Color getColor(int)` - permet de récupérer la couleur se trouvant à l'indice fourni en paramètre.

- `int getSize()` - permet de récupérer la taille de ColorList ;
- `Vector getVector()` - permet de récupérer les couleurs contenues dans ColorList dans un vecteur.

En ce qui concerne le type immutable IntList, celui-ci doit disposer de constructeurs et de méthodes similaires à ceux disponibles dans ColorList.

Pour chacun de ces deux types de données, redéfinissez les méthodes equals() et/ou clone() si vous pensez que cela est nécessaire en raison de leur nature. Justifiez ensuite pourquoi vous avez choisi de redéfinir ou non ces deux méthodes. Pour justifier la redéfinition ou non de clone(), donnez un exemple en fournissant une représentation du stack et de la heap.

Après avoir terminé d'implémenter ces deux types de données, servez-vous en dans un programme similaire à celui développé lors du scénario précédent (section 4.9). Pour ce faire, vous pouvez réutiliser la classe RobotEclaireur et utiliser le capteur de couleur dont elle dispose pour détecter au moins une fois les couleurs vertes, rouges et bleues. Chaque couleur détectée devra être placée dans un ColorList.

Avant de rejouer les différentes couleurs détectées, attendez que le robot détecte que la luminosité est descendue à moins de 25%. A chaque détection du niveau de la luminosité, insérez le pourcentage détecté dans un IntList. Lorsque les couleurs détectées auront été rejouées, vous devrez indiquer sur l'écran du robot les différents pourcentages de luminosité détectés.

4.10.2 Matière

Matière appliquée dans le cadre de ce scénario :

- Le principe de mutabilité et celui d'immutabilité : voir chapitres 2, 3, 5, 6 et 7 du cours de CPOO ;
- Le principe du stack et de la heap : voir chapitre 2 ;
- La redéfinition de la méthode Object.clone() : voir chapitre 2 ;
- La redéfinition de la méthode Object.equals() : voir chapitre 2.

4.10.3 Erreurs possibles

- Ne pas redéfinir la méthode equals() dans le type immutable afin de permettre à deux objets du même type disposant du même état d'être reconnus comme étant identiques ;
- Ne pas redéfinir la méthode clone() dans le type mutable et donc risquer de voir un partage malencontreux de référence être effectué ;

– Au niveau des constructeurs ColorList(Vector) et IntList(Vector), ne pas copier les éléments contenus dans le vecteur passé en paramètre et là aussi, risquer de voir un partage malencontreux de référence être effectué ;
– Ne pas retourner un nouveau vecteur dans la méthode getVector() de ces deux types. Cela permettrait également de changer l'état de ces deux types depuis l'extérieur.

4.10.4 Solution

Pour commencer, il faut développer un type mutable ColorList et un type immutable IntList. Ces deux types pourraient par exemple disposer d'un attribut privé de type Vector afin de contenir pour l'un des couleurs et pour l'autre des entiers.

Rappel : concernant la mutabilité, « un objet est mutable si son état peut changer », ce qui n'est pas le cas d'un objet immutable (Liskov et Guttag, 2001, page 22).

Voici ce que l'implémentation de la classe ColorList pourrait donner :

```
public class ColorList {
    private Vector vecteur;

    public ColorList() {
        vecteur = new Vector();
    }

    public ColorList(Vector vect) {
        vecteur = new Vector();
        for (int i = 0; i < vect.size(); i++) {
            vecteur.addElement(vect.elementAt(i));
        }
    }

    public void addColor(Colors.Color col) {
        vecteur.addElement(col);
    }

    public void removeColor(int idx) {
        vecteur.removeElementAt(idx);
    }

    public Colors.Color getColor(int idx) {
```

```
        return (Colors.Color) vecteur.elementAt(idx);
    }

    public int getSize() {
        return vecteur.size();
    }

    public Vector getVector() {
        Vector v = new Vector();
        for (int i = 0; i < vecteur.size(); i++) {
            v.addElement(vecteur.elementAt(i));
        }
        return v;
    }
}
```

Dans l'énoncé, il est demandé que la classe ColorList n'accepte que les couleurs vertes, rouges et bleues. Dans l'implémentation ci-dessus, cela n'est pas pris en compte. Afin de prendre cette demande en compte, il faudrait vérifier quelle est la couleur que l'on tente d'insérer. Cette vérification pourrait être effectuée au niveau de la méthode addColor(Colors.Color col).

Le constructeur ColorList(Vector vect) peut lui aussi poser un problème similaire. Il ne permet pas de s'assurer que les couleurs qui se trouvent dans vect sont uniquement des couleurs rouges, vertes et bleues. En réalité, il faudrait même s'assurer que les données qu'il contient sont bien des couleurs.

Afin de permettre à l'utilisateur de la classe ColorList de savoir comment l'utiliser, il faudrait ajouter des spécifications. Pour la méthode addColor(Colors.Color col), la précondition pourrait être que la couleur fournie en paramètre doit être rouge, bleue ou verte. La postcondition serait que la couleur est insérée dans le vecteur de couleurs. Que se passe-t-il si la couleur est jaune ? Cela n'est pas mentionné.

La classe IntList est similaire à la classe ColorList. Il s'agit cependant d'un type immutable, ce qui veut dire que l'état d'une instance ne peut pas changer une fois qu'elle a été créée. Pour ce faire, il faut que toutes les méthodes permettant de changer l'état d'un IntList retournent un autre IntList. Voici comment cela pourrait être implémenté :

```
public class IntList {
    private Vector vecteur;

    public IntList() {
        vecteur = new Vector();
```

```
    }

    public IntList(Vector vect) {
        vecteur = new Vector();
        for (int i = 0; i < vect.size(); i++) {
            vecteur.addElement(vect.elementAt(i));
        }
    }

    public IntList addInteger(int val) {
        IntList intList = new IntList(vecteur);
        intVect.vecteur.addElement(new Integer(val));
        return intVect;
    }

    public IntList removeInteger(int idx) {
        IntList intList = new IntList(vecteur);
        intVect.vecteur.removeElementAt(idx);
        return intVect;
    }

    public int getInteger(int idx) {
        return ((Integer) vecteur.elementAt(idx)).intValue();
    }

    ...
}
```

La manière dont cette classe a été implémentée nous garantit qu'il s'agit bien d'un type immutable. En effet, l'état d'une instance ne peut pas être modifié après sa création par un utilisateur de cette classe. Dans la méthode IntList addInteger(int val), par exemple, on crée un nouvel IntList à partir du IntList dans lequel on se trouve. On ajoute ensuite à ce nouvel IntList l'entier qui a été fourni en paramètre. Cet IntList sera ensuite retourné par cette méthode.

Pour laquelle de ces deux classes faudrait-il redéfinir la méthode equals() ? Normalement, celle-ci devrait être redéfinie dans le type immutable car lorsque l'on compare deux types immutables avec cette méthode, on doit juste se contenter de vérifier que leur état est le même. Pour que la méthode equals() utilisée pour comparer deux IntList retourne vrai, il faudrait que ces deux IntList contiennent les mêmes données et ce dans le même ordre.

Voici comment la méthode equals() pourrait être redéfinie dans IntList :

```
public class IntList {
    ...

    public boolean equals(Object o) {
        if (!(o instanceof IntList))
            return false;
        return equals((IntList) o);
    }

    public boolean equals(IntList intList) {
        if (intList == null)
            return false;
        if (intList.vecteur.size() != this.vecteur.size())
            return false;
        for (int i = 0; i < intList.vecteur.size(); i++) {
            if (this.getInteger(i) != intList.getInteger(i))
                return false;
        }
        return true;
    }
}
```

Comme vous pouvez le constater, nous venons d'implémenter deux méthodes equals() différentes. En réalité, la seule méthode equals() qui redéfinit réellement celle héritée de la classe Object est boolean equals(Object). Elle est donc la seule des deux à être indispensable (le code de boolean equals(IntList) aurait pu se trouver dans la méthode boolean equals(Object)).

Maintenant que nous avons redéfini la méthode equals(), nous sommes sûrs que lorsque l'on comparera deux IntList à l'aide de cette méthode, on comparera réellement leur état et non pas leur référence.

Pour laquelle de ces deux classes faudrait-il redéfinir la méthode clone() ? Celle-ci doit être redéfinie lorsque le fait d'avoir un partage de référence peut être problématique. Il s'agit donc d'une méthode qu'il faut redéfinir dans un type mutable lorsque le risque existe qu'un tel partage se produise. Dans un type immutable, cela ne poserait pas de problème car, par définition, un type immutable ne change pas d'état. Il n'est donc pas grave que deux instances différentes d'un type immutable partagent une même référence à cause d'un clonage.

Afin qu'une classe puisse permettre de cloner ses objets, il faut qu'elle implémente l'interface Cloneable. Pour ce faire, il suffit dans notre cas d'ajouter implements Cloneable à public class ColorList ou à public class IntList.

Dans le cas de ColorList, le risque d'avoir un partage de référence est réel car il s'agit d'un type mutable et cette classe dispose d'un attribut de type Vector. Voici comment la méthode clone() pourrait être redéfinie dans cette classe :

```
public class ColorList implements Cloneable {
    ...

    public Object clone() {
        return new ColorList(vecteur);
    }
}
```

L'implémentation de la méthode clone() est cependant également nécessaire pour IntList. Néanmoins, nous pouvons nous contenter de retourner la référence de l'objet sur lequel cette méthode est appelée. Cela ne présente pas de risque en raison de l'immutabilité de ce type.

Dans le constructeur ColorList(Vector), on s'assure de ne pas avoir de partage de référence en clonant le vecteur passé en paramètre. Dans notre cas, ça fonctionne car le vecteur ne contient que des Colors.Color. Si le vecteur de ColorList pouvait contenir des types mutables, il faudrait faire beaucoup plus attention car bien que nous aurions deux instances de Vector différentes, celles-ci pourraient contenir les mêmes références. Du coup, en changeant l'état d'un de ces ColorList, l'état de l'autre serait également altéré.

Voici un exemple de code effectuant des opérations avec des IntList et des ColorList et qui pourrait poser des problèmes si Object clone() n'avait pas été redéfini dans ColorList :

```
IntList intList = new IntList();
IntList intList2;
IntList intList3;
ColorList colList = new ColorList();
ColorList colList2;

intList = intList.addInteger(2);
intList = intList.addInteger(8);
intList = intList.addInteger(5);
colList.addColor(Colors.Color.GREEN);
colList.addColor(Colors.Color.BLUE);
colList.addColor(Colors.Color.RED);

/* Etat 1 */
```

```
intList2 = intList.addInteger(10);
intList3 = (IntList) intList.clone();

/* Etat 2 */

intList = intList.removeInteger(0);
colList2 = (ColorList) colList.clone();
colList.addColor(Colors.Color.BLUE);
colList2.removeColor(2);

/* Etat 3 */
```

Voici maintenant la représentation du stack et de la heap à chacun des états indiqués dans le code :

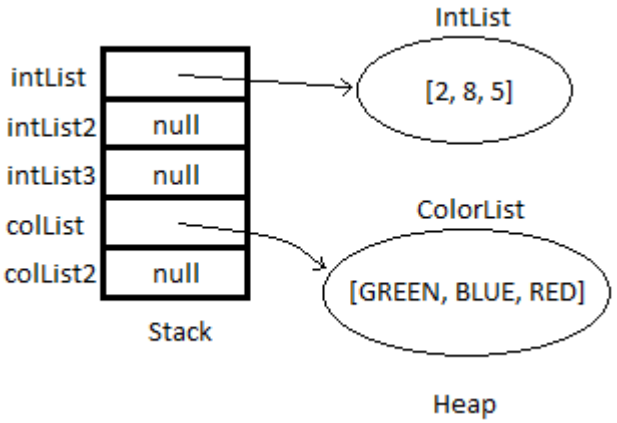

FIGURE 4.1: Stack & heap - état 1

A l'état 1, les variables intList et colList ont été initialisées et disposent chacune de trois valeurs. Les trois autres variables n'ont quant à elles pas encore été initialisées.

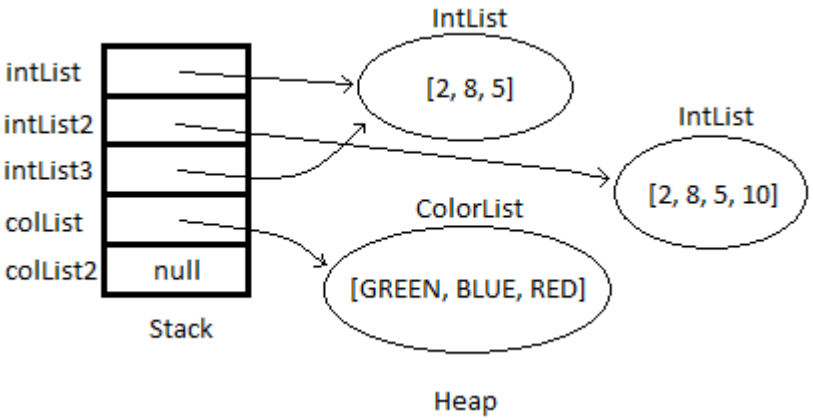

FIGURE 4.2: Stack & heap - état 2

A l'état 2, la variable intList2 reçoit le résultat de l'ajout d'une nouvelle valeur entière au contenu de intList. Quant à la variable intList3, un clone de intList lui est assigné. La méthode Object clone() retournant la référence de l'IntList sur lequel elle est appelée, cela implique un partage de référence. Ce n'est cependant pas grave car IntList est un type immutable. Son état ne peut donc pas changer.

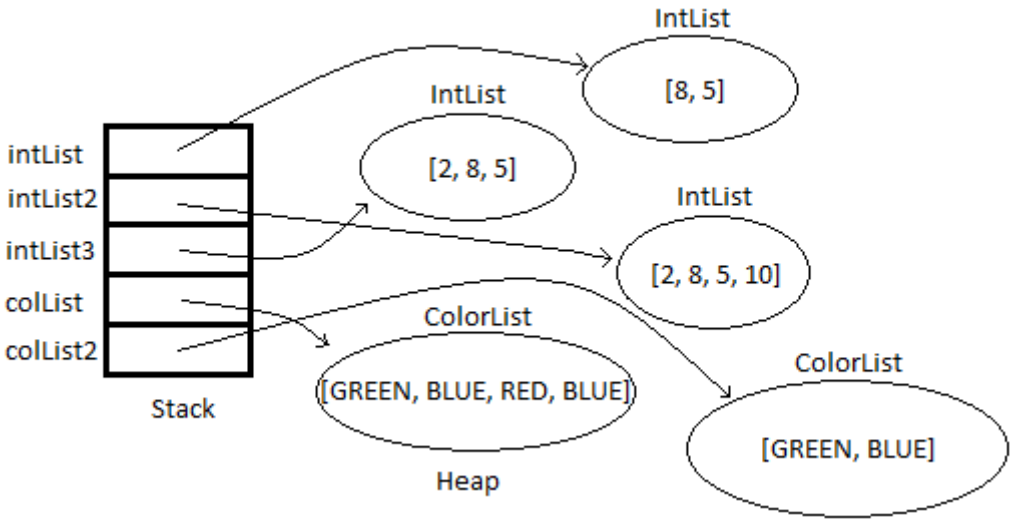

FIGURE 4.3: Stack & heap - état 3

Pour finir, à l'état 3, on commence par enlever une valeur entière de la variable intList. Cela n'a aucune incidence sur la variable intList3 qui avait

été clonée à partir de celle-ci malgré le partage de référence impliqué par le clonâge car IntList est un type immutable.

Ensuite, on assigne un clone de colList à colList2. ColorList étant un type mutable ayant également comme attribut un autre type mutable, une telle opération pourrait faire en sorte qu'il y ait un partage de référence non désiré. Nous avons cependant redéfini Object clone() de telle manière que cela n'arrive pas. C'est pourquoi changer l'état de colList n'a pas d'incidence sur colList2 (et vice versa) contrairement à ce qui se serait passé si Object clone() n'avait pas été correctement redéfini. Si tel avait été le cas, colList et colList2 auraient contenu les valeurs suivantes : [GREEN, BLUE, BLUE].

L'implémentation des classes ColorList et IntList étant terminées, passons au développement du programme demandé dans le cadre de ce scénario. Il s'agit d'un programme semblable à celui développé lors du scénario précédent. Il faut détecter au moins une fois la couleur rouge, la verte et la bleue et ensuite effectuer un rejeu dans l'ordre des différentes couleurs détectées lorsque le niveau de luminosité sera descendu à moins de 25%. A la fin, il faut indiquer sur l'écran du robot toutes les couleurs ainsi que tous les niveaux de luminosité détectés.

Voici un exemple de ce que pourrait donner ce programme :

```
// la méthode isColorOk du scénario peut être réutilisée.
// la méthode rejouerCouleurs doit, elle, être modifiée afin
// d'utiliser un ColorList au lieu d'un Vector.

public static void main (String[] aArg) {
    RobotEclaireur robot = new RobotEclaireur(Motor.A,
                           Motor.C, SensorPort.S2);

    // le vecteur de couleurs
    ColorList colList = new ColorList();
    // le vecteur de luminosité
    IntList lumList = new IntList();

    ...

    while(nbDetectDiff < 3) {
        couleur = robot.detecterCouleur();

        if (isColorOk(couleur) == true) {
            coulTrouvee = false;
            for (int i = 0; i < colList.getSize() && !coulTrouvee; i++) {
                if (colList.getColor(i) == couleur)
                    coulTrouvee = true;
            }
            if (!coulTrouvee)
                nbDetectDiff++;

            colList.addColor(couleur);
        }
        robot.pause(5000);
```

```
    }

    boolean lumTropElevee = true;

    while (lumTropElevee) {
        robot.pause(5000);
        luminosite = robot.detecterNivLuminosite();
        lumList = lumList.addInteger(luminosite);

        if (luminosite < 25)
            lumTropElevee = false;
    }

    rejouerCouleurs(robot, colList);
    System.out.println("Couleurs␣détectées");
    for (int i = 0; i < colList.size(); i++) {
        System.out.println(colList.getColor(i));
    }

    System.out.println("Niveaux␣de␣luminosité␣détectés");
    for (int i = 0; i < lumList.size(); i++) {
        System.out.println(lumList.getInteger(i));
    }
    robot.pause(2000);
}
```

4.11 Scénario - Amélioration des types ColorList et IntList

4.11.1 Objectif

L'objectif de ce scénario est d'amener l'étudiant à améliorer les classes ColorList et IntList afin :

- que ces deux classes possèdent une fonction d'abstraction et un invariant de représentation ;
- de leur fournir un générateur et donc l'implémentation de l'interface Iterator au niveau d'une inner class ;
- que toutes les méthodes qu'elles possèdent disposent de spécifications ;
- d'utiliser les exceptions lorsque les préconditions ne sont pas respectées ;
- de créer plusieurs nouveaux types d'exception.

Ce scénario permet à l'étudiant de pratiquer des concepts de plus en plus avancés du cours de CPOO. Cela va lui permettre de se préparer au cas complexe qu'il sera amené à réaliser plus tard.

Enoncé :

Lors du scénario précédent (section 4.10), vous avez été amené à développer un type mutable ColorList et un type immutable IntList. Certains points pourraient cependant encore poser des problèmes. Par exemple :

- ColorList accepte pour le moment n'importe quel type de couleur ;
- ColorList accepte également que le vecteur qui est fourni en paramètre au constructeur de cette classe, puisse contenir autre chose que des couleurs (Colors.Color).

L'utilisateur de ces classes doit pouvoir être informé de la manière dont elles doivent être utilisées. Il faut dont compléter celles-ci afin qu'elles donnent plus d'information sur ce qu'elles permettent de faire et qu'elles réagissent de manière adéquate en cas de non respect d'une condition.

Dans le cadre de ce scénario, il vous est demandé :

1. d'ajouter dans la classe ColorList une méthode permettant de savoir combien de couleurs différentes sont contenues dans celle-ci ;
2. de décrire la fonction d'abstraction et l'invariant de représentation de chacune de ces deux classes ;
3. de spécifier de manière complète chacun de leurs constructeurs et chacune de leurs méthodes ;
4. de renvoyer une exception lorsque :
   a) le vecteur fourni en paramètre au constructeur de IntList ne contient pas que des int ;
   b) le vecteur fourni en paramètre au constructeur de ColorList ne contient pas que des couleurs rouges, vertes et bleues ;
   c) la couleur fournie en paramètre à la méthode addColor(Colors.Color) n'est pas rouge, verte ou bleue.
5. de créer autant de types d'exception que nécessaire pour répondre aux attentes du point 4 ;
6. de vous assurer que l'implémentation de ces deux classes préserve leur invariant de représentation respectif (comme vu au cours de CPOO) ;
7. de redéfinir la méthode toString() de sorte qu'elle représente l'implémentation de la fonction d'abstraction ;
8. de fournir un générateur à chacune de ces classes ;
9. de modifier le programme créé au scénario précédent afin d'utiliser les exceptions et le générateur de ces deux classes à votre avantage.

4.11.2 Matière

Matière appliquée dans le cadre de ce scénario :
- L'abstraction par spécification : voir chapitre 2 du cours de CPOO ;
- Les exceptions : voir chapitre 4 :
  - Différences entre méthodes partielles et méthodes totales ;
  - Différences entre exceptions vérifiées et exceptions non vérifiées.
- La fonction d'abstraction et l'invariant de représentation : voir chapitre 5.
- L'abstraction par itération, l'inner class et les interfaces : voir chapitres 6 et 7 ;

4.11.3 Erreurs possibles

- Ne pas mettre un commentaire //OVERVIEW au niveau de chacune de ces deux classes afin d'expliquer leur utilité, si elles sont mutables, ... ;
- Confondre fonction d'abstraction et invariant de représentation ;
- Ne pas créer une méthode « repOk() » afin de vérifier que l'invariant de représentation de chacune de ces classes est respecté ;
- Ne pas indiquer, dans la signature de chaque méthode à risque, les exceptions pouvant s'y produire ;
- Confondre méthode partielle et méthode totale (cohérence entre les spécifications et l'implémentation d'une méthode donnée) ;
- Eventuellement de ne pas penser à mettre les nouveaux types d'exception dans un package à part afin de pouvoir les réutiliser dans d'autres programmes.
- Tenter de créer un générateur sans implémenter l'interface Iterator.

4.11.4 Solution

Rappel : Barbara Liskov décrit la fonction d'abstraction comme « capturant la volonté du concepteur de choisir une représentation particulière ». Elle indique qu'on doit y retrouver « les variables d'instance à utiliser et comment elles sont liées à l'objet abstrait qu'elles sont censées représenter. » (Liskov et Guttag, 2001, page 99).

Concernant l'invariant de représentation, Barbara Liskov explique que « il est inventé lorsque l'on réfléchit à la manière d'implémenter les constructeurs et les méthodes ». Elle dit aussi que l'invariant de représentation « capture les hypothèses sur lesquelles reposent ces implémentations ». Cela

signifie que l'invariant de représentation « permet de considérer l'implémentation de chaque opération indépendamment des autres » (Liskov et Guttag, 2001, page 99).

Tout d'abord, commençons par commenter le code à l'aide de //OVERVIEW dans les classes ColorList et IntList, c'est-à-dire de fournir une brève description de leur utilité, indiquer si elles sont mutables et fournir leur fonction d'abstraction et leur invariant de représentation.

Voici ce que ça pourrait donner pour ColorList qui doit respecter certaines conditions telles que ne pas contenir autre chose que des couleurs rouges, vertes et bleues :

```
public class ColorList implements Cloneable {
//OVERVIEW: ColorList est un type mutable permettant de stocker
// dans l'ordre d'insertion les couleurs qui lui sont
// fournies à condition qu'il s'agisse de couleurs
// vertes, rouges, ou bleues.
//
// FA(cv) = [cv.vecteur[i].colorValue | 0 <= i < cv.vecteur.size()
//       && colorValue E {GREEN, RED, BLUE}]
//
// IR: cv.vecteur != null &&
// pour tout int i | cv.vecteur[i] est
//  un Colors.Color E {GREEN, RED, BLUE}
    ...
}
```

Voici ce que ça pourrait donner pour IntList :

```
public class IntList implements Cloneable {
//OVERVIEW: IntList est un type immutable permettant de stocker
// dans l'ordre d'insertion les valeurs entières qui lui sont
// fournies.
//
// FA(iv) = [iv.vecteur[i].intValue | 0 <= i < iv.vecteur.size()]
//
// IR: iv.vecteur != null &&
// pour tout int i | iv.vecteur[i] est un Integer != null
    ...
}
```

Avant de spécifier les différents constructeurs et les différentes méthodes, créons d'abord les nouveaux types d'exception. Nous pouvons par exemple créer les types suivants :

- `NotOnlyIntegersException` : dans le cas où l'on fournit au constructeur de IntList un vecteur contenant d'autres objets que des Integer ;
- `NotOnlyColorsException` : dans le cas où l'on fournit au constructeur de ColorList un vecteur contenant d'autres objets que des Colors.Color ;

- `NullValueException` : dans le cas où l'on fournit au constructeur de IntList ou de ColorList un vecteur contenant au moins un objet null ;
- `UnhandledColorException` : dans le cas où le vecteur fourni au constructeur de la classe ColorList contient des couleurs non gérées (c'est-à-dire une autre couleur que rouge, verte ou bleue). Vaut aussi pour la méthode permettant d'ajouter une couleur dans un ColorList.

Voici le code permettant de créer l'exception NotOnlyIntegersException() :

```
public class NotOnlyIntegersException extends RuntimeException {
    public NotOnlyIntegersException() {
        super();
    }

    public NotOnlyIntegersException(String msg) {
        super(msg);
    }
}
```

Nous avons délibérément choisi de créer une exception non vérifiée en faisant en sorte que NotOnlyIntegersException hérite de RuntimeException et non pas d'Exception, auquel cas elle aurait été une exception vérifiée.

Le code des trois autres exceptions étant similaire à celui de NotOnlyIntegersException, il n'a pas été jugé utile de le faire apparaître ici.

Passons maintenant à la spécification et à l'amélioration des constructeurs et des méthodes de ColorList. Profitons-en également pour ajouter la méthode permettant d'indiquer combien de couleurs différentes le ColorList contient (getNumberOfDiffColors()) :

```
public class ColorList {

    ...

    // nombre de couleurs différentes se trouvant dans vecteur.
    private int nbOfDiffColors;

    //EFFECTS: crée un ColorList vide.
    public ColorList() { vecteur = new Vector(); }

    //EFFECTS: crée un ColorList contenant
    // les couleurs contenues dans le vecteur fourni en
    // paramètre (vect). Si vect = null, renvoie une
    // NullPointerException. Si vect contient des éléments null,
    // renvoie NullValueException. Si vect ne contient pas que
    // des Colors.Color, renvoie une NotOnlyColorsException.
    // Si vect contient d'autres couleurs que des couleurs
    // rouges, vertes ou bleues, renvoie UnhandledColorException.
    public ColorList(Vector vect) throws NullPointerException,
        NullValueException, NotOnlyColorsException,
```

```
        UnhandledColorException {
        // vecteur servant à calculer le nombre de couleurs
        // différentes se trouvant dans le vecteur fourni
        // en paramètre.
        Vector vectOfDiffColors = new Vector();
        for (int i = 0; i < vect.size(); i++) {
            if (vect.elementAt(i) == null)
                throw new NullValueException();
            if (!(vect.elementAt(i) instanceof Colors.Color))
                throw new NotOnlyColorsException();
            if (!isColorOk((Colors.Color) vect.elementAt(i)))
                throw new UnhandledColorException();
            if (vectOfDiffColors.lastIndexOf(
                    vect.elementAt(i)) == -1)
                vectOfDiffColors.addElement(vect.elementAt(i));

            vecteur.addElement(vect.elementAt(i));
        }
        nbOfDiffColors = vectOfDiffColors.size();
    }

    //REQUIRES: col != null
    //EFFECTS: retourne vrai si col est une couleur rouge,
    // verte ou bleue. Retourne faux sinon.
    private boolean isColorOk(Colors.Color col) {
        if (Colors.Color.RED == col ||
            Colors.Color.GREEN == col ||
            Colors.Color.BLUE == col)
            return true;
        else return false;
    }

    //EFFECTS: insère la couleur fournie en paramètre dans
    // le ColorList. Si col == null, renvoie une
    // NullValueException. Si col est une autre couleur que
    // vert, rouge ou bleu, renvoie une UnhandledColorException.
    public void addColor(Colors.Color col) throws
        NullValueException, UnhandledColorException {
        if (col == null)
            throw new NullValueException();
        if (!isColorOk(col))
            throw new UnhandledColorException();
        if (vecteur.lastIndexOf(col) == -1)
            nbOfDiffColors++;
        vecteur.addElement(col);
    }

    //EFFECTS: enlève l'élément se trouvant à l'indice idx.
    // Si idx ne se trouve pas dans les bornes du ColorList,
    // renvoie un ArrayIndexOutOfBoundsException.
    public void removeColor(int idx) throws
        ArrayIndexOutOfBoundsException {
        Colors.Color col = (Colors.Color) vecteur.elementAt(idx);
        vecteur.removeElementAt(idx);
        if (vecteur.lastIndexOf(col) == -1)
            nbOfDiffColors--;
    }

    //EFFECTS: retourne le nombre de couleurs différentes que
```

```
    // le ColorList contient.
    public int getNumberOfDiffColors() {
        return nbOfDiffColors;
    }

    ...
}
```

Dans la solution fournie ci-dessus, nous avons décidé de créer une méthode privée isColorOk() afin de s'assurer que la couleur fournie en paramètre est bien rouge, verte ou bleue. Vous pouvez constater que dans la précondition de cette méthode, il est mentionné que col doit être différent de null. Etant donné qu'il s'agit d'une méthode privée de ColorList, on peut raisonnablement penser que le développeur fera attention à bien fournir un Colors.Color non null en paramètre.

Seules quelques méthodes sont disponibles dans ce bout de code mais ce sont celles qui disposent des spécifications les plus complexes de la classe ColorList. Au niveau des signatures de chacune de ces méthodes, nous avons pris soin d'indiquer les exceptions qui pourraient éventuellement être renvoyées. Nous avons également pris soin d'indiquer dans la clause //EFFECTS dans quel cas celles-ci pourraient l'être.

Afin de disposer d'une méthode getNumberOfDiffColors() peu coûteuse, nous avons décidé de créer un nouvel attribut privé nbOfDiffColors. La valeur contenue dans cet attribut permet de savoir combien de couleurs différentes sont stockées dans le ColorList. La valeur de cet attribut est mise à jour dans :

- `ColorList()` : simple initialisation à 0 ;
- `ColorList(Vector)` : initialisation au nombre de couleurs différentes contenues dans le vecteur fourni en paramètre pour autant que son contenu respecte les spécifications de ce constructeur ;
- `addColor(Colors.Color)` : incrémentation d'une unité dans le cas où la couleur fournie en paramètre n'était pas encore présente dans le ColorList. Cela ne se produit que si les spécifications de cette méthode ont été respectées ;
- `removeColor(int)` : décrémentation d'une unité dans le cas où la couleur qui se trouvait à l'indice fourni en paramètre ne peut plus être trouvée ailleurs dans le ColorList. De nouveau, cela ne se produit que si les spécifications de cette méthode ont été respectées.

Ces mises à jour permettent de s'assurer que la valeur de cet attribut privé soit toujours conforme à la réalité. En contrepartie, ces différents constructeurs et méthodes sont désormais plus coûteux qu'auparavant (à l'exception du constructeur par défaut). Cependant le coût d'une utilisation répétée

d'une méthode getNumberOfDiffColors() qui devrait effectuer les vérifications nécessaires lors de chaque appel, serait très certainement bien plus important.

Passons ensuite à la spécification et à l'amélioration des constructeurs et des méthodes de IntList :

```
public class IntList {

    ...

    //EFFECTS: crée un IntList vide.
    public IntList() {
        vecteur = new Vector();
    }

    //EFFECTS: crée un IntList contenant les Integer
    // contenus dans le vecteur fourni en paramètre
    // (vect). Si vect == null, renvoie NullPointerException.
    // Si vect contient des éléments null, renvoie
    // NullValueException. Si vect ne contient pas que des
    // Integer, renvoie NotOnlyIntegersException.
    public IntList(Vector vect) throws NullPointerException,
        NullValueException, NotOnlyIntegersException {
        vecteur = new Vector();
        for (int i = 0; vect.size(); i++) {
            if (vect.elementAt(i) == null)
                throw new NullValueException();
            if (!(vect.elementAt(i) instanceof Integer))
                throw new NotOnlyIntegersException();
            vecteur.addElement(vect.elementAt(i));
        }
    }

    //EFFECTS: retourne un IntList contenant le contenu
    // de l'objet appelant + la valeur entière val fournie
    // en paramètre de cette fonction.
    public IntList addInteger(int val) {
        IntList intVect = new IntList(vecteur);
        intVect.vecteur.addElement(new Integer(val));
        return intVect;
    }

    ...
}
```

Comme pour la solution de ColorList, celle-ci n'est pas complète mais fournit les spécifications des méthodes/constructeurs les plus complexes et les plus intéressants de IntList. A présent, les signatures des différentes méthodes informent également l'utilisateur potentiel sur les exceptions pouvant être renvoyées. Les cas où cela pourrait se produire sont explicités au niveau des spécifications.

Nous pouvons maintenant passer au développement de la méthode repOk() et à la redéfinition de la méthode toString() dans chacune de ces classes

```
public class ColorList {
    ...
    public boolean repOk() {
        if (vecteur == null)
            return false;
        for (int i = 0; i < vecteur.size(); i++) {
            if (vecteur.elementAt(i) == null)
                return false;
            if (!(vecteur.elementAt(i) instanceof Colors.Color))
                return false;
            if (!isColorOk((Colors.Color) vecteur.elementAt(i)))
                return false;
        }
        return true;
    }

    public String toString() {
        if (vecteur.size() == 0)
            return "ColorList: []";
        String res = "ColorList: [";
        for (int i = 0; i < vecteur.size() - 1; i++) {
            res = res + vecteur.elementAt(i) + ", ";
        }
        res = res + vecteur.elementAt(vecteur.size() - 1) + "]";
        return res;
    }
}
```

On peut constater que l'implémentation de la méthode repOk() de ColorList ressemble très fortement au constructeur de ColorList auquel on passe un vecteur en paramètre. C'est logique dans la mesure où dans ce constructeur, on souhaitait également s'assurer que le contenu du vecteur fourni en paramètre était acceptable par rapport à la fonction d'abstraction et à l'invariant de représentation de cette classe, c'est-à-dire :

- disposer d'un vecteur ayant au moins été initialisé ;
- et ne contenant que des couleurs rouges, vertes ou bleues (pas d'objets non initialisés, pas d'objets d'un autre type, pas d'autres couleurs).

Concernant la méthode toString(), sa redéfinition (et donc l'implémentation de la fonction d'abstraction) permet de facilement afficher le contenu du ColorList.

```
public class IntList {
    ...
    public boolean repOk() {
        if (vecteur == null)
            return false;
        for (int i = 0; i < vecteur.size(); i++) {
            if (vecteur.elementAt(i) == null)
                return false;
            if (!(vecteur.elementAt(i) instanceof Integer))
                return false;
        }
        return true;
    }
```

```
    public String toString() {
        if (vecteur.size() == 0)
            return "IntList: []";
        String res = "IntList: [";
        for (int i = 0; i < vecteur.size() - 1; i++) {
            res = res + vecteur.elementAt(i) + ", ";
        }
        res = res + vecteur.elementAt(vecteur.size() - 1) + "]";
        return res;
    }
}
```

Pour les mêmes raisons que ColorList, l'implémentation de l'invariant de représentation du IntList à travers la méthode repOk() est très semblable à l'implémentation du constructeur IntList(Vector).

Nous pouvons maintenant continuer l'amélioration de ces deux classes en créant une inner class dans chacune d'entre elles. Cette inner class va servir de générateur et pour ce faire, elle doit implémenter l'interface Iterator. Il faut également penser à ajouter une méthode publique au ColorList et à l'IntList permettant d'accéder à cet itérateur.

```
public class ColorList {
    ...

    //REQUIRES: l'état du ColorList ne peut pas être changé
    // pendant l'utilisation de cet itérateur.
    //EFFECTS: retourne un générateur permettant de parcourir
    // toutes les données que le ColorList contient.
    public Iterator colors() {
        return new ColorListGen();
    }

    private class ColorListGen implements Iterator {
        private int index;

        ColorListGen() {
            index = 0;
        }

        public boolean hasNext() {
            return (index < vecteur.size());
        }

        public Object next() throws NoSuchElementException {
            if (!hasNext())
                throw new NoSuchElementException();
            return vecteur.elementAt(index++);
        }

        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
}
```

Comme vous pouvez le constater dans la solution fournie ci-dessus, l'inner class ColorListGen implémente bien l'interface Iterator et ses deux méthodes boolean hasNext() et Object next(). Cette interface nécessite également d'implémenter la méthode remove(). Etant donné que nous n'en avons pas besoin, nous l'avons implémentée tout en retournant une UnsupportedOperationException() pour indiquer à l'utilisateur que cette méthode n'est pas supportée. Il est donc maintenant possible de créer le générateur ColorListGen afin de parcourir toutes les couleurs contenues dans un ColorList.

Nous avons développé une nouvelle méthode appelée colors() dans le ColorList. Celle-ci nous permet d'obtenir un générateur. ColorList étant un type mutable, il faut faire attention à ce que l'état du ColorList à partir duquel on a créé un générateur ne change pas durant l'utilisation de ce générateur. Si l'état devait être amené à changer, le programme ne garantirait plus le bon fonctionnement du générateur (d'où la clause //REQUIRES dont cette méthode dispose).

```
public class IntList {
    ...

    //EFFECTS: retourne un générateur permettant de générer toutes
    // les valeurs contenues dans le IntList. Le générateur
    // fournit des Integer.
    public Iterator integers() {
        return new IntListGen();
    }

    private class IntListGen implements Iterator {
        ...
    }
}
```

Le générateur IntListGen étant très sembable au générateur ColorListGen, il n'a pas été jugé nécessaire de fournir son implémentation. Notez cependant que la méthode integers() qui permet de créer un tel générateur ne dispose pas d'une clause //REQUIRES exigeant que l'état du IntList ne soit pas altéré pendant l'utilisation de ce générateur. En effet, l'état du IntList ne peut pas être altéré car il s'agit d'un type immutable.

Terminons maintenant ce scénario en modifiant le programme développé au scénario précédent afin qu'il utilise désormais les améliorations des classes ColorList et IntList. Dans ce programme, la méthode isColorOk() qui avait été utilisée lors des deux scénarios précédents n'est plus nécessaire (en réalité elle se trouve au niveau de ColorList maintenant). En effet, nous avons modifié la classe ColorList afin qu'elle respecte les exigences concernant les couleurs pouvant y être insérées. Si nous tentons d'insérer une mauvaise couleur, la classe ColorList va renvoyer une exception.

```
private static void rejouerCouleurs(RobotEclaireur robot,
    ColorList colList) {
    Iterator colIt = colList.colors();
    while (colIt.hasNext()) {
        robot.allumerCouleur((Colors.Color) cotIt.next(), 1000);
    }
}

public static void main (String[] aArg) {
    RobotEclaireur robot = new RobotEclaireur(Motor.A,
                           Motor.C, SensorPort.S2);

    // le liste de couleurs
    ColorList colList = new ColorList();
    // le liste de luminosité
    IntList lumList = new IntList();

    ...

    while(colList.getNumberOfDiffColors() < 3) {
        couleur = robot.detecterCouleur();

        try {
            colList.addColor(couleur);
        }
        catch (NullValueException npe) {}
        catch (UnhandledColorException uce) {
            System.out.println("Couleur␣incorrecte␣"
                + "détectée:␣" + couleur);
        }

        robot.pause(5000);
    }

    boolean lumTropElevee = true;

    while (lumTropElevee) {
        robot.pause(5000);
        luminosite = robot.detecterNivLuminosite();
        lumList = lumList.addInteger(luminosite);

        if (luminosite < 25)
            lumTropElevee = false;
    }

    rejouerCouleurs(robot, colList);
    System.out.println("Couleurs␣détectées");
    System.out.println(colList.toString());
    System.out.println("Niveaux␣de␣luminosité␣détectés");
    System.out.println(lumList.toString());

    robot.pause(2000);
}
```

Comme vous pouvez le constater dans le bout de code ci-dessus :

– nous utilisons désormais le générateur de ColorList pour effectuer le rejeu des couleurs ;

– nous utilisons également la nouvelle méthode du ColorList « getNumberOfDiffColors() » afin de savoir combien de couleurs différentes celui-ci contient au lieu de passer par une variable externe comme c'était le cas auparavant ;
– c'est le ColorList qui désormais vérifie si la couleur qui lui est fournie peut être acceptée ou non. Si ce n'est pas le cas, le ColorList renvoie une exception que nous allons pouvoir détecter grâce aux instructions try et catch écrites dans ce programme.

La méthode toString() que nous avons redéfinie au niveau du ColorList et du IntList est également désormais utilisée afin d'afficher leur contenu à la fin du programme. Cela est possible grâce à cette redéfinition qui correspond à une implémentation de la fonction d'abstraction de ces deux classes.

4.12 Scénario - Adéquation des types ColorList et IntList

4.12.1 Objectif

L'objectif principal de ce scénario est d'amener l'étudiant à vérifier si les types ColorList et IntList semblent adéquats. Il s'agit également d'une opportunité pour le faire réfléchir sur les types des différentes opérations dont ces types de données disposent, c'est-à-dire :
– des créateurs ;
– des producteurs ;
– des mutateurs ;
– des observateurs.

Ce scénario va également permettre d'aborder le concept de programmation défensive et la définition de la méthode Object.similar() au niveau d'un des deux types.

Enoncé :

Les types ColorList et IntList sont-ils adéquats ? C'est la question à laquelle vous allez devoir tenter de répondre dans ce scénario. L'adéquation d'un type dépendant de son utilisation potentielle, plusieurs réponses sont possibles.

Dans le cadre de ce scénario, il vous est demandé :

1. d'indiquer le type de chacune des opérations disponibles dans ces deux types de données (créateurs, producteurs, mutateurs, observateurs).

Comme vu au cours de CPOO, un type de données doit au moins disposer de trois types d'opérations différentes. Est-ce bien le cas ?

2. de vérifier là où cela vous semble nécessaire (en fonction du type des opérations) si l'invariant de représentation est toujours respecté (repOk()) ;
3. de définir la méthode similar() provenant de la classe Object dans ces types de données.
4. d'indiquer si ces types de données sont adéquats. Si vous pensez qu'ils pourraient être améliorés, comment cela serait-il possible ?

4.12.2 Matière

Voici la matière couverte par ce scénario :
- Les quatre catégories d'opérations (créateur, producteur, mutateur, observateur) : chapitre 5 du cours de CPOO ;
- La programmation défensive (FailureException) : chapitre 4 ;
- Méthode similar() de la classe Object : chapitre 4.

4.12.3 Erreurs possibles

- Se tromper au niveau du type des différentes opérations : par exemple prendre un constructeur auquel on fournit un type qu'il va devoir créer pour un créateur ;
- Ne pas ajouter une vérification sur l'invariant de représentation dans les créateurs, les mutateurs et les producteurs ;
- Ne pas définir la méthode similar() comme comparant l'état d'un objet à celui d'un autre ;
- De ne pas trouver un cas dans lequel l'adéquation de l'un des deux types de données pourrait être remise en question. Par exemple, il n'y a pas de méthode permettant de vider un ColorList ou un IntList d'un coup.

4.12.4 Solution

Faisons d'abord une liste des différentes méthodes de ColorList triées par type d'opération :
- Créateurs :
  - ColorList() ;
  - ColorList(Vector).

- Producteur :
  - Object clone().
- Mutateurs :
  - void addColor(Colors.Color) ;
  - voir removeColor(int).
- Observateurs :
  - Iterator colors() ;
  - int getSize() ;
  - Vector getVector() ;
  - boolean isColorOk() ;
  - boolean repOk() ;
  - String toString().

Vous pouvez constater que le seul producteur dont ColorList dispose est la méthode Object clone() qui permet de créer une instance de ColorList disposant du même état que l'instance sur laquelle cette méthode a été appelée. En effet, ColorList est un type mutable et de ce fait, dispose plutôt de mutateurs permettant à son état d'être changé.

Voici celle de IntList :

- Créateurs :
  - IntList() ;
  - IntList(Vector).
- Producteurs :
  - Object clone() ;
  - IntList addInteger(int) ;
  - IntList removeInteger(int).
- Observateurs :
  - boolean equals(IntList) ;
  - boolean equals(Object) ;
  - ...

IntList est un type immutable contrairement à ColorList. Son état ne pouvant pas changer, il est donc normal qu'il ne dispose d'aucun mutateur. Il fournit cependant divers producteurs permettant d'ajouter ou d'enlever une valeur entière d'une de ses instances.

Ajoutons maintenant la méthode similar() à ColorList :

```
public class ColorList {
    ...

    public boolean similar(Object o) {
        if (!(o instanceof ColorList))
            return false;
```

```
        return similar((ColorList) o);
    }

    public boolean similar(ColorList colList) {
        if (colList == null ||
            colList.vecteur.size() != vecteur.size())
            return false;
        for (int i = 0; i < vecteur.size(); i++) {
            if (vecteur.elementAt(i) !=
                colList.vecteur.elementAt(i))
                return false;
        }
        return true;
    }
}
```

Cette méthode ainsi implémentée va nous permettre de comparer l'état de deux ColorList différents. Dans IntList il n'est pas vraiment nécessaire de l'implémenter car la méthode equals(Object) nous permet déjà de comparer l'état de deux instances de ce type de données en raison de son immutabilité. Si vraiment l'on souhaitait l'implémenter, on pourrait se contenter de retourner le résultat de equals(Object).

Afin de vérifier la non violation de l'invariant de représentation, il faudrait tester, ne fut-ce que provisoirement, le résultat de la méthode repOk() dans tous les créateurs, producteurs et mutateurs. Voici un exemple de la manière dont on pourrait implémenter ces vérifications dans ColorList :

```
public class ColorList {
    ...

    public ColorList(Vector vect) throws ... {
        ...
        if (!repOk())
            throw new FailureException();
    }

    public void addColor(Colors.Color col) throws ... {
        ...
        vecteur.addElement(col);
        if (!repOk())
            throw new FailureException();
    }
```

```
    ...
}
```

Le type de données ColorList semble adéquat car il permet :
- de créer des instances de ce type ;
- d'ajouter une couleur ;
- d'en enlever une en fournissant un indice ;
- et de connaître la taille du ColorList ou de récupérer une couleur dont l'indice est fourni en paramètre.

Nous pourrions cependant nous demander si un utilisateur potentiel de ce type de données ne souhaiterait pas pouvoir par exemple vider le contenu de celui-ci à l'aide d'une méthode. Cette méthode pourrait s'appeler empty() et se contenter de vider le contenu du vecteur du ColorList. Cela vaut aussi pour le IntList.

Une autre méthode qu'un utilisateur potentiel du ColorList pourrait souhaiter serait de permettre d'enlever toutes les occurrences d'une même couleur. A l'aide de cette méthode, il pourrait par exemple supprimer toutes les couleurs rouges de son ColorList.

Enfin, nous pourrions imaginer une méthode permettant de vérifier la présence d'une couleur donnée dans le ColorList. A l'aide de cette méthode, il nous serait par exemple possible de déterminer si oui ou non le ColorList contient au moins une couleur verte.

Voici ce que ça pourrait donner :

```
public class ColorList {
    ...

    //EFFECTS: supprime le contenu du ColorList.
    public void empty() {
        vecteur.clear();
        nbOfDiffColors = 0;
    }

    //EFFECTS: supprime toutes les couleurs dont l'état
    // est le même que la couleur passée en paramètre
    // (même couleur). Si col == null, renvoie
    // NullPointerException
    public void removeColors(Colors.Color col) throws
        NullPointerException {
        int lastIndex = vecteur.lastIndexOf(col);
        if (lastIndex != -1) {
            while (lastIndex != -1) {
                vecteur.removeElementAt(lastIndex);
                lastIndex = vecteur.lastIndexOf(col);
            }
            nbOfDiffColors--;
        }
    }
```

```
    //EFFECTS: renvoie vrai si le ColorList contient une
    // couleur dont l'état est le même que celui de la
    // couleur passée en paramètre. Renvoie faux sinon.
    public boolean containsColor(Colors.Color col) {
        if (vecteur.lastIndexOf(col) != -1)
            return true;
        else return false;
    }
}
```

4.13 Scénario - la classe Robot et ses sous-types revisités

4.13.1 Objectif

L'objectif de ce scénario est de revoir les concepts concernant la hiérarchie de types de données c'est-à-dire :
- Les classes abstraites, les classes concrètes, les super-types et les sous-types ;
- Les interfaces et leurs implémentations ;
- Le principe de substitution de Liskov.

Dans ce scénario, l'étudiant va être amené à créer sa propre hiérarchie de types de données. Celle-ci pourra être ensuite réutilisée dans le cadre du chapitre 5 pour les cas plus complexes qui y seront abordés.

Enoncé :

Les scénarios précédents vous ont amenés à développer une classe Robot et quelques sous-classes (RobotDetecteur, RobotEclaireur). Au fil des scénarios, vous avez été amenés à les améliorer et à développer d'autres classes tout en pratiquant divers concepts de la programmation OO.

Dans le cadre de ce scénario, vous devrez créer les premières classes d'une nouvelle hiérarchie de classes avec comme super-type une classe abstraite Robot et ses deux sous-types RobotNXT et RobotRCX.

Voici une liste des quatre modèles proposés par LEGO (sans compter les variantes) dont vous pouvez vous inspirer pour commencer cette hiérarchie :

1. Le RoboLanceur ;
2. Le Robogator ;
3. Le trieur de couleurs ;
4. Alpha Rex.

Les classes que vous allez créer devront pouvoir être utilisées pour faire bouger le robot (le faire avancer, le faire reculer, ...), pour écrire quelque chose sur son écran LCD, pour permettre d'effectuer des opérations spécifiques à un certain type de robot, etc. En effet, contrairement au RoboLanceur, au Robogator et à Alpha Rex, le trieur de couleurs ne peut, lui, pas bouger. Toutes les classes qui dépendent d'un LEGO Mindstorms NXT devront hériter de la classe RobotNXT ou d'un de ses sous-types.

Attention : ces classes doivent juste offrir une interface permettant par exemple de faire bouger un robot (comme ce fut le cas des classes Robot, RobotDetecteur, ... développées précédemment). Il ne s'agit pas ici de lui donner une intelligence, un comportement. La réalisation complète d'au moins un des modèles listés ci-dessus sera à effectuer au chapitre 5.

Tous les types développés dans le cadre de ce scénario devront scrupuleusement respecter le principe de substitution de Liskov. Chacun de ceux-ci devra, dans la mesure du possible, disposer d'une fonction d'abstraction (ainsi que toString()) et d'un invariant de représentation (ainsi que repOk()). Vous devrez également fournir les spécifications de toutes les méthodes que vous développerez si cela est nécessaire.

Vous devrez aussi créer une ou plusieurs interfaces fournissant des méthodes permettant d'utiliser un ou plusieurs capteurs du LEGO Mindstorms NXT 2.0. Vos futures classes devront implémenter ces interfaces si vous souhaitez qu'elles puissent utiliser lesdits capteurs.

Deux restrictions qui seront également d'application au chapitre 5 :

1. Seuls la classe Robot et ses sous-types ont le droit d'accéder aux moteurs, à l'écran LCD, aux boutons, etc sans passer par une méthode ;
2. Si un super-type dans cette hiérarchie propose déjà d'accéder à un des éléments mentionnés ci-dessus à l'aide d'une méthode, alors ses sous-types devront également passer par cette méthode pour y accéder.

4.13.2 Matière

- Les classes abstraites et les classes concrètes : chapitre 7 du cours de CPOO ;
- Les super-types et les sous-types : chapitre 7 ;
- Les interfaces ainsi que leurs implémentations : chapitre 7 ;
- Le principe de substitution de Liskov : chapitre 7 ;
- La fonction d'abstraction et l'invariant de représentation : chapitre 5 ;

4.13.3 Erreurs possibles

- Une mauvaise décomposition des classes ;
- Le non-respect du principe de substitution de Liskov ;
- Ne pas avoir une spécification complète :
  - des classes (fonction d'abstraction, invariant de représentation, ...) ;
  - des méthodes (les exceptions générées par les méthodes, ...).

4.13.4 Solution

Rappel : Le principe de substitution de Liskov est basé sur trois règles. La première concerne les signatures. Barbara Liskov explique que « les objets d'un sous-type doivent tous avoir les méthodes du super-type, et que les signatures des méthodes d'un sous-type doivent être compatibles avec les signatures des méthodes correspondantes du super-type » (Liskov et Guttag, 2001, page 174).

La deuxième règle concerne les méthodes. Barbara Liskov indique que pour respecter le principe de substitution , « les appels aux méthodes de ce sous-type doivent se comporter comme les appels aux méthodes correspondantes du super-type » (Liskov et Guttag, 2001, page 190). Enfin, la troisième et dernière règle concerne les propriétés. « Le sous-type doit préserver toutes les propriétés qui peuvent être prouvées à propos des objets du super-type » (Liskov et Guttag, 2001, page 175).

La solution fournie ci-dessous n'est pas complète. Il ne s'agit pas non plus de la seule et unique solution mais plutôt d'une des solutions possibles pour résoudre cet exercice.

Nous pouvons d'abord nous demander quelles classes nous allons créer. Comme expliqué dans l'énoncé, il y a au moins quatre modèles proposés par LEGO lorsque l'on se procure un LEGO Mindstorms NXT 2.0 : le RoboLanceur, le Robogator, le trieur de couleurs et l'Alpha Rex. L'énoncé explique également qu'il faut créer un début de hiérarchie avec comme super-type une classe abstraite Robot. Commençons donc par nous demander ce dont cette classe aurait besoin.

Un robot dispose au minimum :
- d'un « calculateur » sur lequel se trouve un écran LCD ;
- d'une batterie ;
- d'un moyen de générer du bruit.

Sur base de cela, nous pouvons définir que notre classe Robot disposera de plusieurs méthodes abstraites qui permettront :

– d'afficher un message sur l'écran LCD ;
– de donner des informations sur la batterie (voltage, si elle est rechargeable, ...) ;
– d'offrir la possibilité de lire des fichiers audio, de faire beeper le calculateur, etc.

Voici à quoi pourrait ressembler notre classe Robot :

```
public abstract class Robot {

    public Robot() {}

    //EFFECTS: écrit sur l'écran du calculateur le message
    // passé en paramètre (avec retour à la ligne).
    public void ecrire(String string) {
        System.out.println(string);
    }

    //EFFECTS: écrit sur l'écran du calculateur le voltage de
    // la batterie et si oui ou non elle est rechargeable.
    public abstract void batteryInfo();

    //EFFECTS: fait beeper le calculateur une fois.
    public abstract void beep();

    //EFFECTS: joue le fichier WAV fourni en paramètre.
    // La valeur retournée représente le nombre de millisecondes
    // pendant lesquelles la musique sera jouée.
    // Renvoie l'exception SampleCannotBePlayedException si une
    // erreur se produit.
    public abstract int jouerMusique(File file);
}
```

Maintenant que la classe Robot a été écrite, nous pouvons réfléchir à ses sous-types directs : RobotNXT et RobotRCX. Ces sous-types doivent implémenter les méthodes abstraites de la classe Robot ainsi que quelques nouvelles méthodes qui leur sont propres. Etant donné que nous ne disposons que d'un LEGO Mindstorms NXT 2.0, nous n'allons implémenter que cette classe.

```
public class RobotNXT extends Robot {
\\OVERVIEW: classe représentant la version NXT
\\ des LEGO Mindstorms.

    public RobotNXT() {}

    //EFFECTS: retourne vrai si le bouton ESCAPE a été pressé,
    // faux sinon.
    public boolean isEscapeButtonPressed() {
        return Button.ESCAPE.isPressed();
    }

    //EFFECTS: retourne vrai si le bouton ENTER a été pressé,
    // faux sinon.
    public boolean isEnterButtonPressed() {
        return Button.ENTER.isPressed();
```

```
    }

    //EFFECTS: retourne vrai si le bouton LEFT a été pressé,
    // faux sinon.
    public boolean isLeftButtonPressed() {
        return Button.LEFT.isPressed();
    }

    //EFFECTS: retourne vrai si le bouton RIGHT a été pressé,
    // faux sinon.
    public boolean isRightButtonPressed() {
        return Button.RIGHT.isPressed();
    }

    public void batteryInfo() {
        ecrire("Batterie␣:");
        ecrire("----------");
        ecrire("Voltage␣:␣" + Battery.getVoltage() + "␣V");
        ecrire("Rechargeable␣:␣" + Battery.isRechargeable());
    }

    public void beep() {
        Sound.beep();
    }

    public int jouerMusique(File file)
                        throws SampleCannotBePlayedException {
        int res = Sound.playSample(file);
        if (res <= 0)
            throw new SampleCannotBePlayedException();
        else return res;
    }
}
```

En plus de l'implémentation des méthodes abstraites de la classe Robot dans RobotNXT, nous avons ajouté quelques méthodes permettant de vérifier si un des boutons du LEGO Mindstorms NXT 2.0 a été pressé.

Dans les différents modèles proposés par LEGO, nous pouvons constater que certains concernent des robots qui peuvent se déplacer et que d'autres concernent des robots immobiles (c'est-à-dire des robots dont les moteurs sont utilisés à d'autres fins).

Cette constatation nous permet de dégager deux sous-types différents :
- Un sous-type permettant à un robot d'utiliser des moteurs pour se déplacer ;
- Un autre sous-type utilisant les moteurs du robot pour faire autre chose.

Nous allons donc pouvoir créer les deux classes suivantes :
- La classe RobotMobile : un robot NXT qui peut se déplacer ;
- La classe Machine : un robot NXT immobile.

Commençons par la classe RobotMobile :

```
public class RobotMobile extends RobotNXT {
```

```
//OVERVIEW: cette classe permet à un robot de disposer de
// deux moteurs pour se mouvoir (avancer, reculer, tourner
// à gauche ou à droite, ...
//
// IR(c): c.moteurGauche != null && c.moteurDroit != null &&
//  c.moteurGauche est le moteur gauche du robot &&
//  c.moteurdroit est le moteur droit du robot.
// A = {avance, recule, tourne_g, tourne_d, arrêt}
// FA: C -> A: si c.moteurGauche = avance && c.moteurDroit = avance
//           => avance
//              si c.moteurGauche = recule && c.moteurDroit = recule
//           => recule
//              si c.moteurGauche = recule && c.moteurDroit = avance
//           => tourne_g
//              si c.moteurGauche = avance && c.moteurDroit = recule
//           => tourne_d
//              si c.moteurGauche = arrêt && c.moteurDroit = arrêt
//           => arrêt

    private Motor moteurGauche;
    private Motor moteurDroit;

    //REQUIRES: gauche est le moteur gauche du robot,
    // droit est le moteur droit du robot.
    //EFFECTS: crée un RobotMobile ayant comme moteur gauche
    // "gauche" et comme moteur droit "droit". Si l'un de ces
    // deux paramètres est null, renvoie une NullPointerException.
    public RobotMobile(Motor gauche, Motor droit) throws
            NullPointerException {
        super(); // pas obligatoire
        if (gauche == null || droit == null)
            throw new NullPointerException();
        moteurGauche = gauche;
        moteurDroit = droit;
    }

    //EFFECTS: fait avancer le robot
    public void avancer() {
        moteurGauche.forward();
        moteurDroit.forward();
    }

    //EFFECTS: fait reculer le robot
    public void reculer() {
        moteurGauche.backward();
        moteurDroit.backward();
    }

    ... (tournerGauche, tournerDroit, stop, pause)


    //EFFECTS: retourne le moteur gauche du robot
    protected Motor getMoteurGauche() {
        return moteurGauche;
    }

    //EFFECTS: retourne le moteur droit du robot
    protected Motor getMoteurDroit() {
        return moteurDroit;
```

```
    }
}
```

La classe RobotMobile est très simple. Elle permet seulement d'effectuer des opérations très basiques. Toutefois, il est important de souligner que pour que la classe fonctionne correctement, il faut que le moteur gauche soit réellement le moteur gauche du robot et que le moteur droit soit également réellement le moteur droit du robot.

Passons maintenant à la classe Machine. Comme indiqué précédemment, cette classe ne doit pas permettre à un robot de se déplacer mais doit plutôt lui permettre d'effectuer certaines opérations (faire fonctionner un moteur, en faire fonctionner un autre, ...).

```
public class Machine extends RobotNXT {
//OVERVIEW: cette classe permet à un robot de disposer de deux
// moteurs de manière indépendante. Un des moteurs peut tourner
// pendant que l'autre est à l'arrêt et vice versa.
//
// IR(c): c.moteur1 != null && c.moteur2 != null

    private Motor moteur1;
    private Motor moteur2;

    //EFFECTS: crée une instance de Machine disposant de
    // deux moteurs m1 et m2. Si l'un de ces deux paramètres est
    // NULL, renvoie une NullPointerException.
    public Machine(Motor m1, Motor m2) throws NullPointerException {
        super(); // pas obligatoire
        if (m1 == null || m2 == null)
            throw new NullPointerException();
        moteur1 = m1;
        moteur2 = m2;
    }

    //EFFECTS: fait fonctionner le moteur numéro 1 en marche avant
    public void avancerMoteur1() {
        moteur1.forward();
    }

    //EFFECTS: fait fonctionner le moteur numéro 1 en marche arrière
    public void reculerMoteur1() {
        moteur1.backward();
    }

    ... (avancerMoteur2, reculerMoteur2, stopMoteur1, stopMoteur2,
        pause, ...)

    //EFFECTS: retourne le moteur numéro 1 du robot
    protected Motor getMoteur1() {
        return moteur1;
    }

    //EFFECTS: retourne le moteur numéro 2 du robot
    protected Motor getMoteur2() {
        return moteur2;
```

```
    }
}
```

Etant donné que la classe Machine ne permet d'utiliser ses moteurs que de manière indépendante, il n'est pas nécessaire de spécifier que le moteur numéro 1 correspond par exemple toujours au moteur le plus à gauche du robot, etc.

Avant de passer aux sous-types de RobotMobile et de Machine, passons un peu de temps afin de créer les interfaces dont ils auront besoin pour pouvoir utiliser les différents capteurs du LEGO Mindstorms NXT 2.0 comme demandé dans l'énoncé. La plupart de ces interfaces doivent offrir une ou plusieurs méthodes permettant d'utiliser ces différents capteurs.

Listons les interfaces qu'il va falloir développer :
- Une interface permettant d'utiliser le capteur à ultrasons : ICaptUltrasons ;
- Une interface permettant d'utiliser le capteur de couleurs : ICaptCouleurs ;
- Une interface permettant d'utiliser un capteur tacticle : ICaptTactile ;
- Et une interface permettant d'utiliser deux capteurs tactiles : IPaireCaptTactiles.

Commençons par développer l'interface ICaptUltrasons :

```
interface ICaptUltrasons {
//OVERVIEW: interface offrant une méthode permettant
// de se servir d'un capteur à ultrasons.

    //EFFECTS: fournit en cm la distance entre le capteur et un objet.
    // Si aucun objet ne se trouve à moins de 255 cm du capteur,
    // retourne 255.
    public int getDistance();
}
```

Passons maintenant à l'interface ICaptCouleurs :

```
interface ICaptCouleurs {
//OVERVIEW: interface offrant plusieurs méthodes permettant
// de se servir d'un capteur de couleurs.

    //EFFECTS: allume la lampe dans la couleur donnée à condition
    // qu'elle puisse être affichée par le capteur. Dans le cas
    // contraire, éteint la lampe.
    public void setFloodlight(Colors.Color col);

    //EFFECTS: allume ou éteint la lampe.
    public void setFloodlight(boolean b);

    //EFFECTS: retourne un tableau avec les valeurs rouge, verte et
    // bleue.
    public int[] getColor();
```

```
    //EFFECTS: retourne la couleur lue par le capteur.
    public Colors.Color readColor();

    //EFFECTS: retourne le pourcentage de lumière perçue par le capteur.
    public int getLightValue();
}
```

Il ne nous reste maintenant plus que les interfaces concernant les capteurs tactiles à écrire. Commençons par ICaptTactile :

```
interface ICaptTactile {
//OVERVIEW: interface offrant une méthode permettant
// de se servir d'un capteur tactile.

    //EFFECTS: indique si oui ou non le capteur tactile est pressé.
    public boolean isPressed();
}
```

L'interface IPaireCaptTactiles offre deux méthodes identiques à celle qu'offre ICaptTactile, une pour chaque capteur :

```
interface IPaireCaptTactiles {
//OVERVIEW: interface offrant deux méthodes permettant
// de se servir de deux capteurs tactiles distincts

    //EFFECTS: indique si oui ou non le capteur tactile
    // numéro 1 est pressé.
    public boolean touchSensor1IsPressed();

    //EFFECTS: indique si oui ou non le capteur tactile
    // numéro 2 est pressé.
    public boolean touchSensor2IsPressed();
}
```

Ceci conclut ce scénario. Cette hiérarchie devra être complétée dans le cadre du chapitre suivant proposant des cas plus complexes.

Chapitre 5

LEGO Mindstorms NXT 2.0 - Projet Alpha Rex

Ce chapitre constitue la deuxième partie de notre support pédagogique. Il s'agit de développer un programme permettant de faire communiquer un ordinateur avec un robot LEGO Mindstorms NXT 2.0, à l'aide de la technologie Bluetooth.

5.1 Objectif

Ce chapitre a pour objectif de proposer un projet de plus grande envergure que les scénarios effectués jusqu'à présent par l'étudiant. Le but de ce projet est de lui permettre de mettre en pratique la plupart des concepts OO vus au chapitre 4). Au dernier scénario de celui-ci, l'étudiant a dû commencer à développer une hiérarchie de robots. Dans le cadre de ce projet, il lui est demandé de réutiliser celle-ci ainsi que les interfaces qu'il a dû créer.

Enoncé :

Au cours du chapitre 4, vous avez été amené à utiliser divers concepts OO afin de mener à bien les différents exercices qui vous ont été proposés. Le projet à réaliser dans le cadre de ce chapitre-ci a pour objectif de vous faire réutiliser la plupart de ces concepts au sein d'un même exercice.

Dans le cadre de ce projet, il vous est demandé de développer un programme en vous basant sur le modèle « Alpha Rex » tel que proposé par LEGO dans son LEGO Mindstorms NXT 2.0. Pour ce faire, vous allez devoir :

- Permettre d'accéder à différentes fonctionnalités proposées par « Alpha Rex » :

  - Permettre d'utiliser deux moteurs afin de pouvoir se mouvoir, lui permettre de lever une jambe, ... ;
  - Permettre d'utiliser un troisième moteur afin de faire bouger ses bras et ses mains ;
  - Etre capable d'utiliser ses différents capteurs.
- Permettre d'effectuer des tests similaires à ceux proposés par LEGO :
  - Test 1 :
    - Faire lever une de ses jambes ;
    - Le faire avancer pendant quelques secondes ;
    - Le faire reculer pendant quelques secondes.
  - Test 2 :
    - Faire balancer ses bras et lui faire ouvrir et fermer ses mains ;
    - Le faire avancer pendant quelques secondes tout en lui faisant toujours balancer ses bras ainsi qu'ouvrir et fermer ses mains ;
    - Faire en sorte qu'il n'utilise plus ses trois moteurs ;
    - Le faire reculer pendant quelques secondes.
  - Test 3 :
    - Faire avancer Alpha Rex dès qu'il détecte la présence de votre main à moins de 100 cm de lui ;
    - Le faire pivoter vers la gauche et tenter de le faire retourner de là où il vient dès qu'il détecte que votre main se trouve à moins de 30 cm de lui.
  - Test 4 :
    - Faire avancer Alpha Rex dès qu'il détecte la présence de votre main à moins de 100 cm de lui et le faire s'arrêter à 30 cm de celle-ci ;
    - Lui faire demander une balle de couleur rouge, verte ou bleue ;
    - Si la balle qui lui est donnée est de la bonne couleur, lui faire allumer son capteur de couleurs dans cette couleur pendant cinq secondes ;
    - Si la balle qui lui est donnée n'est pas de la bonne couleur, lui faire allumer pendant une seconde la lampe dans la couleur demandée et lui faire retester ensuite la couleur de l'objet qu'on lui fournit.

Il vous est ensuite demandé d'offrir une possibilité de contrôler le robot à distance. Pour ce faire, vous allez devoir :

- Créer un programme qui, via une communication Bluetooth, va permettre de donner des ordres préprogrammés au robot (effectuer test 1, test 2, ...) ou de le contrôler (lui demander d'avancer, de s'arrêter, lui faire demander une balle de couleur rouge, demander une balle et lui faire communiquer la couleur de celle-ci, ...) ;
- Permettre au robot de communiquer via Bluetooth avec un ordinateur afin qu'il puisse recevoir des ordres de celui-ci et qu'il puisse lui envoyer des informations (couleur détectée, ...).

Bonus :

Permettre à un ou plusieurs robots de communiquer avec un même ordinateur. Permettre aux robots de se donner des ordres entre eux. Continuer la hiérarchie pour créer les classes nécessaires pour les autres modèles (pour permettre d'utiliser les moteurs, les capteurs, ...).

5.2 Matière

- Les classes abstraites et les classes concrètes : chapitre 7 du cours de CPOO ;
- Les super-types et les sous-types : chapitre 7 ;
- Les interfaces ainsi que leurs implémentations : chapitre 7 ;
- Le principe de substitution de Liskov : chapitre 7 ;
- La fonction d'abstraction et l'invariant de représentation : chapitre 5 ;

5.3 Erreurs possibles

- Une mauvaise décomposition des classes ;
- Le non-respect du principe de substitution de Liskov ;
- Ne pas avoir une spécification complète :
  - des classes (fonction d'abstraction, invariant de représentation, ...) ;
  - des méthodes (les exceptions générées par les méthodes, ...).

5.4 Solution

Dans le cadre de ce projet, la solution est cette fois-ci décomposée en plusieurs sous-sections. La première de ces sous-sections contient un diagramme de classes représentant toutes les classes créées au dernier scénario du chapitre 4 ainsi que toutes celles que nous allons créer dans ce projet afin de réaliser l'objectif principal.

Les sous-sections suivantes sont dédiées en général à une des nouvelles classes créées dans le cadre de ce projet. Il s'agit de classes représentant le robot Alpha Rex, de classes offrant la possibilité de le contrôler à distance et de classes permettant au robot et à un ordinateur de communiquer ensemble. Dans ces sous-sections, vous pourrez trouver des bouts de code des classes concernées avec quelques explications sur le ou les concepts OO qui s'y trouvent.

La solution proposée ne constitue qu'une des solutions possibles permettant de résoudre le problème qui vous est posé.

5.4.1 Diagramme de classes

Dans cette sous-section, vous trouverez la figure 5.1 consistant en un diagramme de classes simplifié ainsi qu'une brève description de chacune des nouvelles classes que nous allons créer pour ce projet.

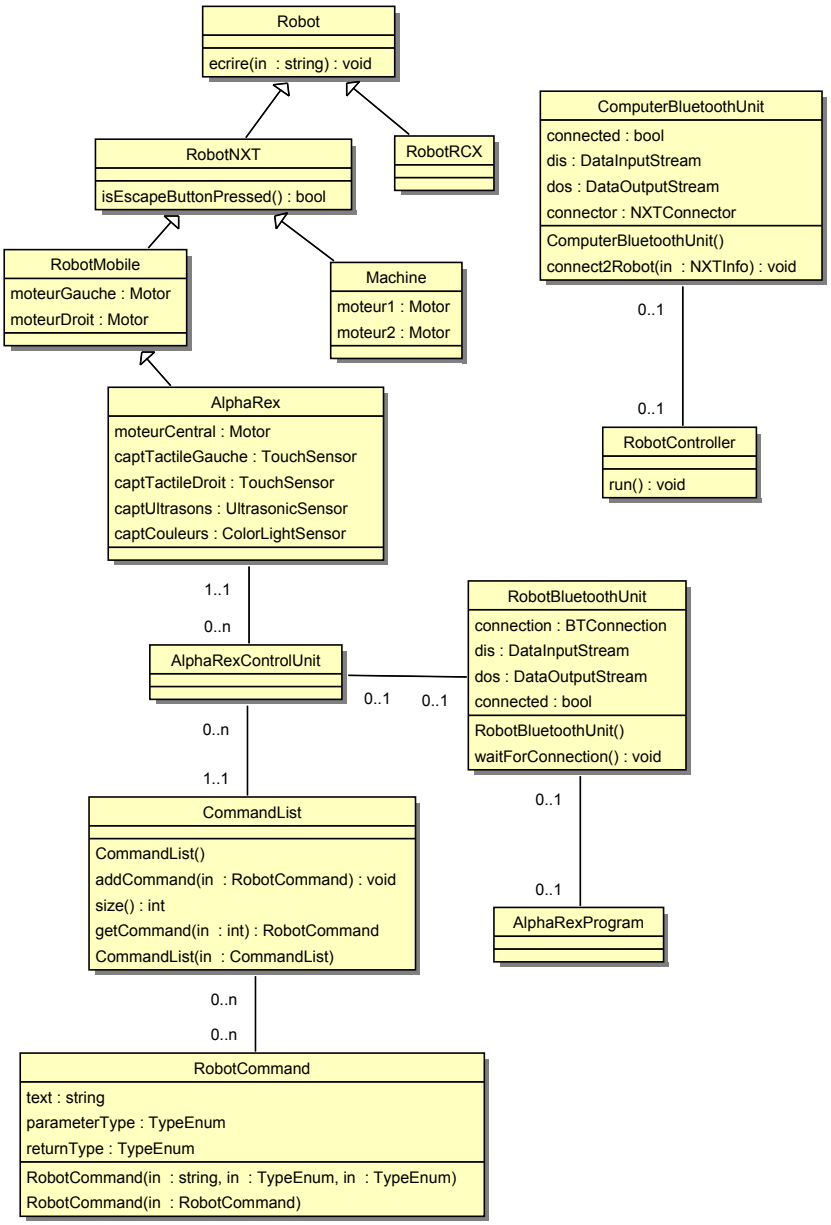

FIGURE 5.1: Diagramme de classes

Voici la liste des nouvelles classes :

1. AlphaRex : cette classe hérite de RobotMobile et représente le modèle Alpha Rex tel que proposé par LEGO. Elle offre la possibilité :

   a) de faire bouger les bras et les mains du robot à l'aide du troisième moteur dont Alpha Rex dispose ;

b) d'utiliser les quatre capteurs du robot ;

c) d'effectuer des opérations propres à un robot bipède (e.g. lever une jambe et ne faire bouger que celle-ci).

2. AlphaRexControlUnit : cette classe constitue un intermédiaire entre la classe AlphaRex et des utilisateurs souhaitant envoyer des commandes au robot. Elle offre la possibilité :

   a) de fournir la liste des différentes commandes qu'elle permet d'effectuer ;

   b) d'exécuter la commande qu'on lui demande d'exécuter ;

   c) parmi ces différentes commandes se trouvent quatre commandes « avancées » que cette classe propose (i.e. les quatre séries de commandes demandées dans l'énoncé (test 1, test 2,...)).

3. RobotBluetoothUnit : cette classe permet au robot de communiquer avec un ordinateur ou un autre robot à l'aide de la technologie Bluetooth. Elle offre la possibilité :

   a) d'attendre une connexion entrante et de mettre un terme à celle-ci ;

   b) d'envoyer et de recevoir des informations de la part de l'appareil qui est connecté au robot.

4. AlphaRexProgram : cette classe dispose de la méthode main et constitue l'entrée du programme que l'on est en train de concevoir dans le cadre de ce projet du côté du robot. Elle a pour objectif :

   a) d'utiliser RobotBluetoothUnit pour attendre une connexion entrante ;

   b) de permettre de communiquer la liste des commandes à l'aide des classes AlphaRexControlUnit et RobotBluetoothUnit ;

   c) d'attendre qu'on lui fournisse un numéro de commande à exécuter et éventuellement retourner le résultat s'il y en a un (e.g. une commande permettant de retourner la couleur détectée par le capteur de couleurs).

5. RobotCommand : cette classe représente une commande pouvant être exécutée sur le robot qui la fournit.

6. CommandList : cette classe permet de stocker la liste des différentes commandes disponibles (RobotCommand). Elle permet entre autres :

   a) d'ajouter et de supprimer une commande (RobotCommand) ;

b) de récupérer le texte propre à une commande ainsi que le type du paramètre à fournir et le type de la valeur de retour.

7. ComputerBluetoothUnit : cette classe permet à un ordinateur de communiquer avec un robot à l'aide de la technologie Bluetooth. Elle offre la possibilité :

   a) de fournir la liste des robots portant un certain identifiant (par exemple « NXT ») ;

   b) de permettre de se connecter à l'un de ces robots ;

   c) d'envoyer et de recevoir des informations de la part de celui-ci.

8. RobotController : cette classe a pour objectif de permettre à l'utilisateur de se connecter à un robot et de lui envoyer des commandes. Elle offre la possibilité :

   a) d'afficher la liste des robots portant un certain identifiant en se servant de ComputerBluetoothUnit ;

   b) de laisser l'utilisateur choisir le robot et d'afficher la liste des commandes dont il peut se servir ;

   c) d'envoyer une commande au robot.

9. RobotControllerProgram : cette classe dispose de la méthode main et constitue l'entrée du programme que l'on est en train de concevoir dans le cadre de ce projet du côté de l'ordinateur. Elle se contente de créer une instance de RobotController et d'exécuter la méthode permettant à l'utilisateur de se connecter à un robot et de choisir les commandes à exécuter.

5.4.2 AlphaRex

La classe AlphaRex est la classe qui représente le modèle du robot bipède Alpha Rex que LEGO propose dans LEGO Mindstorms NXT 2.0. Elle hérite de la classe RobotMobile et doit implémenter les trois interfaces suivantes : ICaptUltrasons, ICaptCouleurs et IPaireCaptTactiles.

Voici un bout de code reprenant la première partie de l'implémentation de cette classe :

```
public class AlphaRex extends RobotMobile implements IPaireCaptTactiles,
    ICaptUltrasons, ICaptCouleurs {
//OVERVIEW: cette classe permet d'avoir accès aux fonctionnalités
// basiques d'un robot bipède de type Alpha Rex (modèle du LEGO
// Mindstorms NXT 2.0).
```

```
//
// IR(c): c.moteurCentral != null && c.captTactileGauche != null
//        && c.captTactileDroit != null && c.captUltrasons != null
//        && c.captCouleurs != null
// A = Etat_CaptTactileGauche X Etat_CaptTactileDroit
//     Etat_LampeCaptCouleurs X MouvementBras X Mouvement
// FA(c): <e1, e2, e3, e4, e5> où
//    e1 = {pressé, pasPressé}
// et e2 = {pressé, pasPressé}
// et e3 = {coulRouge, coulVerte, coulBleue, rien}
// et e4 = {avance, recule, arrêt}
// et e5 = {super.Mouvement, leverEtBougerJambeGauche,
//    leverEtBougerJambeDroite}

    private Motor moteurCentral;
    private TouchSensor captTactileGauche;
    private TouchSensor captTactileDroit;
    private UltrasonicSensor captUltrasons;
    private ColorLightSensor captCouleurs;

    ...
}
```

Le premier concept OO couvert par ce bout de code est le concept de classe. Ce concept permet de créer le type de données « AlphaRex » qui va représenter le robot de type Alpha Rex.

Le deuxième concept couvert par ce bout de code est le concept d'héritage, la classe AlphaRex héritant de la classe RobotMobile. Cela nous permet d'avoir accès aux fonctionnalités offertes par RobotMobile depuis AlphaRex. En effet, RobotMobile est une classe permettant à un robot de se mouvoir et elle gère deux moteurs distincts. Dans AlphaRex, nous rajoutons des attributs propres à un robot bipède de type Alpha Rex i.e. un moteur pour les bras ainsi que les capteurs tactiles gauche et droit, le capteur à ultrasons et le capteur de couleurs.

Le troisième concept couvert par ce bout de code est celui de l'implémentation d'interfaces. Vous pouvez constater que trois interfaces doivent être implémentées pour cette classe. Ces trois interfaces, comme la classe RobotMobile, ont été créées dans le cadre du dernier scénario du chapitre 4. Nous verrons un peu plus loin l'implémentation de quelques méthodes de ces interfaces.

Enfin, le dernier concept couvert par ce bout de code concerne les spécifications propres à la classe AlphaRex. Il s'agit donc d'une brève description de celle-ci ainsi que de son invariant de représentation et de sa fonction d'abstraction.

Voici maintenant un bout de code montrant le constructeur de cette classe avec ses spécifications :

```
...

//REQUIRES: gauche est le moteur gauche du robot,
// droit est le moteur droit du robot, central est le
// moteur contrôlant les bras et les mains du robot.
//EFFECTS: crée un AlphaRex ayant comme moteur gauche
// "gauche", comme moteur droit "droit" et comme moteur
// contrôlant les bras et les mains "central". Si l'un de
// ces trois paramètres est null, renvoie une NullPointerException.
// this_post = <e1, e2, e3, e4, e5> où e1 = pressé et e2 = pressé et
// e3 = rien, e4 = arrêt, e5 = arrêt
public AlphaRex(Motor gauche, Motor droit, Motor central,
        SensorPort capTacG, SensorPort capTacD, SensorPort captUs,
        SensorPort captCoul) throws NullPointerException {
    super(gauche, droit); // obligatoire
    if (central == null)
        throw new NullPointerException();
    moteurCentral = central;

    captTactileGauche = new TouchSensor(capTacG);
    captTactileDroit = new TouchSensor(capTacD);
    captUltrasons = new UltrasonicSensor(captUs);
    captCouleurs = new ColorLightSensor(captCoul, 0);

    this.getMoteurDroit().setPower(45);
    this.getMoteurGauche().setPower(45);

    initialisation();
}

...
```

Dans ce bout de code, le premier concept OO que l'on peut apercevoir est tout simplement celui du constructeur qui permet de créer une instance d'AlphaRex. Etant donné que cette classe hérite de la classe RobotMobile et que le constructeur de celle-ci nécessite deux paramètres, nous devons également faire appel à la méthode super() tout en lui fournissant les deux paramètres requis par le constructeur de RobotMobile (à savoir `gauche` et `droit`).

Le deuxième concept que l'on peut y constater est celui des spécifications. En effet, ce constructeur dispose d'une précondition et d'une postcondition. La postcondition utilise les états indiqués dans la fonction d'abstraction de ce type de données.

Le troisième concept que l'on peut y apercevoir est celui des exceptions. Le constructeur indique qu'il peut renvoyer une exception de type « NullPointerException » dans le cas où la variable `central` n'aurait pas été instanciée. Cette exception peut également être renvoyée par le constructeur de la classe mère dans le cas où les variables `gauche` et `droit` n'auraient, elles non plus, pas été instanciées.

Enfin, le dernier concept OO auquel nous avons affaire dans ce bout de code est celui de la création d'instances pour les quatre différents capteurs. L'instanciation se fait à l'aide du mot-clé `new` et appelle le constructeur de la classe dont on veut créer une instance (= objet).

Le bout de code suivant concerne les interfaces :

```
...

//EFFECTS: this_post = <e1, e2, e3, e4, e5> où
// e3 = coulRouge si col est rouge,
// e3 = coulVerte si col est verte,
// e3 = coulBleue si col est bleue,
// e3 = rien sinon.
public void setFloodlight(Colors.Color col) {
    captCouleurs.setFloodlight(col);
}

...
```

Ce bout de code permet de constater le concept OO d'implémentation d'une méthode d'une interface, en l'occurrence ici de l'interface `ICaptCouleurs`. Les spécifications de cette méthode ont été redéfinies pour y intégrer les états du capteur de couleurs de l'AlphaRex.

Pour terminer avec AlphaRex, voici un dernier bout de code de cette classe :

```
...

//EFFECTS: this_post = <e1, e2, e3, e4, e5> où e1 = pressé
// et e2 = pressé et e5 = avance
public void avancer() {
    initialisation();
    this.getMoteurDroit().rotate(180);
    super.avancer();
}

...
```

Ce bout de code permet de constater le concept OO de redéfinition (override) de la méthode `avancer()` de la classe mère dans la classe fille AlphaRex. Dans cette méthode, nous effectuons quelques instructions avant d'appeler la méthode du même nom au niveau de la classe mère à l'aide du mot-clé `super` (i.e. `super.avancer()`). La postcondition ayant été renforcée (e1 = pressé, e2 = pressé) et la précondition ne l'ayant pas été, nous pouvons considérer que le principe de substitution de Liskov a bien été respecté.

5.4.3 AlphaRexControlUnit

La classe AlphaRexControlUnit a été conçue afin de servir d'intermédiaire entre la classe AlphaRex et un potentiel utilisateur. Elle a pour objectif de gérer les demandes d'exécution de commandes dont elle fournit d'ailleurs la liste. Elle permet de proposer également certaines commandes qui constituent en réalité une série d'actions que AlphaRex va devoir exécuter. Il s'agit des commandes « avancées » demandées par l'énonce (test 1, test 2, ...).

Voici le bout de code correspondant au début de l'implémentation de cette classe :

```
public class AlphaRexControlUnit {
//OVERVIEW: cette classe permet de communiquer avec un AlphaRex.
// Elle fournit les différentes commandes utilisables par
// l'utilisateur et parmi lesquelles se trouvent des commandes
// proposant d'exécuter une série d'actions.
// IR(c) = c.robot != null && commands != null

    private AlphaRex robot;
    private CommandList commands;

    //EFFECTS: crée une instance d'AlphaRexControlUnit.
    // Si ar est null, alors renvoie une NullPointerException.
    public AlphaRexControlUnit(AlphaRex ar)
            throws NullPointerException {
        if (ar == null)
            throw new NullPointerException();
        robot = ar;

        commands = new CommandList();
        remplirCommands();
    }

    ...
```

Ce bout de code couvre à nouveau le concept de classe. Ici, il s'agit donc de la classe AlphaRexControlUnit. Il couvre notamment à nouveau les concepts de spécifications d'une classe (à l'aide notamment d'un invariant de représentation), de spécifications d'une méthode et d'exception.

Le bout de code suivant concerne les commandes « avancées » :

```
    //EFFECTS: le robot lève sa jambe gauche et
    // la fait bouger pendant deux secondes. Ensuite il
    // avance pendant cinq secondes et recule de nouveau
    // pendant cinq secondes.
    private void test1() {
        robot.leverJambeGauche();
        robot.attente(2000);
        robot.avancer();
        robot.attente(5000);
        robot.reculer();
```

```
        robot.attente(5000);
        robot.stop();
    }

    //EFFECTS: le robot commence par rester immobile
    // pendant trois secondes tout en bougeant ses bras.
    // vers l'avant. Il avance ensuite pendant
    // trois secondes, s'arrête, arrête de faire bouger
    // ses bras et attend pendant deux secondes.
    // Pour terminer, il recule pendant trois secondes
    // tout en faisant bouger ses bras vers l'arrière.
    private void test2() {
        robot.bougerBrasVersAvant();
        robot.attente(3000);
        robot.avancer();
        robot.attente(3000);
        robot.stop();
        robot.arreterBras();
        robot.attente(2000);
        robot.reculer();
        robot.bougerBrasVersArriere();
        robot.attente(3000);
        robot.stop();
        robot.arreterBras();
    }
```

Le concept couvert par ce bout de code est de nouveau celui des spécifications des méthodes. Ce bout de code constitue l'implémentation d'une partie du code concernant les commandes « avancées » du robot.

Pour terminer, le bout de code suivant concerne la manière dont les commandes demandées par un utilisateur sont prises en compte :

```
    //EFFECTS: exécute la commande fournie en paramètre
    // avec le paramètre qui lui est associé (si nécessaire).
    // Si numCommande n'est pas connu, retourne null.
    public Object executerCommande(int numCommande, Object parameter) {
        switch (numCommande) {
            case 0: robot.initialisation(); return null;
            case 1: robot.leverJambeGauche(); return null;
            case 2: robot.leverJambeDroite(); return null;
            case 3: robot.avancer(); return null;
            ...
            case 15: return new Integer(robot.getDistance());
            case 16: setFloodlightBool((String) parameter);
                return null;
            case 17: setFloodlightCol(((Integer) parameter).intValue());
                return null;
        ...
        }
    }
```

Cette méthode demande que l'on fournisse un numéro de commande à exécuter ainsi qu'un paramètre (si nécessaire). Elle peut également, selon la commande choisie, fournir une valeur de retour provenant du robot. Ainsi, il est possible de, par exemple, demander au robot de fournir la distance

de l'obstacle se trouvant devant le capteur à ultrasons. Nous pouvons aussi lui demander d'allumer la lampe de son capteur de couleurs en se basant sur le paramètre fourni par l'utilisateur. C'est grâce à cette méthode qu'un utilisateur sur son ordinateur va être capable de donner des ordres à son robot.

5.4.4 RobotCommand & TypeEnum

La classe RobotCommand est une classe représentant une commande qu'il est possible d'exécuter. Elle dispose d'un texte qui peut être montré à l'utilisateur afin qu'il sache à quoi sert cette commande. Elle lui permet également de savoir si cette commande nécessite qu'il lui fournisse un paramètre et si cette commande retourne une valeur de retour.

Tout d'abord voici le code du type énuméré TypeEnum :

```
public enum TypeEnum {
    NO_TYPE,
    INT_TYPE,
    STRING_TYPE
}
```

Ce type énuméré est utilisé pour detérminer le type du paramètre et le type de la valeur de retour d'une commande donnée. NO_TYPE signifie cependant qu'il n'y a pas de paramètre et/ou pas de valeur de retour.

Le code suivant concerne la classe RobotCommand et le début de son implémentation :

```
public class RobotCommand implements Cloneable {
//OVERVIEW: cette classe représente une commande exécutable
// par un robot.
// IR(c): c.text != null && c.parameterType != null
//        && c.returnType != null

    private String text;
    private TypeEnum parameterType;
    private TypeEnum returnType;

    //EFFECTS: crée une instance de RobotCommand.
    // Renvoie une NullPointerException si txt, paramType ou
    // retType == null.
    public RobotCommand(String txt, TypeEnum paramType, TypeEnum retType)
            throws NullPointerException {
        if (txt == null || paramType == null ||
             retType == null)
            throw new NullPointerException();
        text = txt;
        parameterType = paramType;
        returnType = retType;
    }
```

```
//EFFECTS: crée une instance de RobotCommand
// à partir d'une autre. Renvoie une NullPointerException
// si rc == null.
private RobotCommand(RobotCommand rc)
        throws NullPointerException {
    if (rc == null)
        throw new NullPointerException();
    this.text = rc.text;
    this.parameterType = rc.parameterType;
    this.returnType = rc.returnType;
}

...
```

Voici donc le début de l'implémentation de la classe RobotCommand. Comme vous pouvez le constater, elle doit implémenter l'interface Cloneable. Cette interface demande d'implémenter une méthode dénommée clone() qui a pour but de fournir une copie de l'instance sur laquelle cette méthode est appelée.

Afin de faciliter l'implémentation de cette méthode clone(), nous avons créé un constructeur privé acceptant en paramètre une instance de la classe RobotCommand. Cela nous permet de directement copier les attributs de cette instance et de les assigner à la nouvelle instance que nous sommes en train de créer (le clone).

Etant donné que les trois attributs sont des types immutables, nous pouvons les assigner sans nous soucier du fait que les références de ces attributs risquent d'être partagées.

```
...

public Object clone() {
    return new RobotCommand(this);
}

...
```

Voici donc le code réimplémentant la méthode clone() de la classe Object. Elle retourne, comme prévu, un Object et crée une nouvelle instance de RobotCommand à partir de l'instance sur laquelle cette méthode est appelée.

5.4.5 CommandList

CommandList est une classe qui a pour objectif de contenir une liste d'instances de RobotCommand qui lui sont fournies. Il s'agit d'un type mutable.

Voici le bout de code constituant la première partie de son implémentation :

```
public class CommandList implements Cloneable {
//OVERVIEW: CommandList est un type mutable permettant
// de stocker dans l'ordre d'insertion les commandes
// (RobotCommand) qui lui sont fournies.
//
// IR(c): c.listOfCommands != null &&
//  pour tout int i | c.listOfCommands[i] est
//  non null.

    Vector listOfCommands;

    //EFFECTS: crée une instance de CommandList.
    public CommandList() {
        listOfCommands = new Vector();
    }

    ...
```

Que pouvons-nous y constater ? Premièrement, il s'agit d'un type mutable qui doit implémenter l'interface Cloneable. Cela signifie que nous allons à nouveau devoir réimplémenter la méthode clone() de la classe Object.

Deuxièmement, la solution choisie pour implémenter ce type de données est d'utiliser un Vector. De ce fait, nous allons devoir prêter attention à certains détails inhérents à ce type de données.

```
    ...

    //EFFECTS: ajoute une commande à la liste de commandes.
    // Si command == null, renvoie une NullPointerException.
    public void addCommand(RobotCommand command)
            throws NullPointerException {
        if (command == null)
            throw new NullPointerException();
        listOfCommands.addElement(command);
    }

    //EFFECTS: retourne la taille de la liste de commandes.
    public int size() {
        return listOfCommands.size();
    }

    //EFFECTS: retourne la commande située à l'indice idx
    // de la liste de commandes. Si idx se trouve en dehors
    // des bornes de la liste, renvoie une
    // ArrayIndexOutOfBoundsException.
    public RobotCommand getCommand(int idx)
            throws ArrayIndexOutOfBoundsException {
        return (RobotCommand) listOfCommands.elementAt(idx);
    }

    ...
```

Le bout de code ci-dessus montre quelques méthodes permettant d'ajouter une commande, d'en récupérer une et de connaître la taille de la liste de commandes. Tout d'abord, dans l'invariant de représentation, nous avons

indiqué que la liste de commandes ne pouvait pas contenir de RobotCommand « null ». C'est pourquoi lors de l'ajout d'une commande à l'aide de la méthode addCommand(...), nous devons vérifier si la commande à ajouter est différente de null. Cela va nous permettre de respecter les spécifications de la classe CommandList.

Nous avons également une méthode permettant de récupérer une commande. Le Vector que nous utilisons peut contenir n'importe quel type de données à l'aide de la classe mère commune à tous les types de données : Object. Cela signifie que lorsque l'on souhaite récupérer un élément de cette liste, il faut caster celui-ci en RobotCommand.

Voici maintenant un bout de code spécifique à la réimplémentation de la méthode clone() au sein de la classe CommandList :

```
...

//EFFECTS: crée une instance de CommandList
// à partir d'une autre instance de CommandList.
// Si cl == null, alors renvoie une
// NullPointerException.
private CommandList(CommandList cl)
        throws NullPointerException {
    if (cl == null)
        throw new NullPointerException();

    listOfCommands = new Vector();

    for (int i = 0; i < cl.size(); i++) {
        this.addCommand((RobotCommand) cl.getCommand(i).clone());
    }
}

...

public Object clone() {
    return new CommandList(this);
}

...
```

Comme pour RobotCommand, nous avons une méthode clone() qui ne fait rien d'autre que créer une nouvelle instance de CommandList à l'aide d'un constructeur privé auquel on fournit l'instance sur laquelle la méthode clone() est appelée. Dans ce constructeur privé, et ce pour éviter un problème de partage de référence sur des types mutables (Vector en étant un), nous avons décidé de copier un par un tous les éléments du Vector de l'instance passée en paramètre dans le Vector de l'instance en cours de création.

Bien que RobotCommand soit de facto un type immutable (aucune méthode ne permet de changer son état pour le moment), il n'a pas été spécifié comme

tel. Etant donné que celui-ci propose une méthode clone(), nous nous en servons lors de la copie des éléments du Vector de l'instance passée en paramètre dans le Vector de la nouvelle instance. Nous faisons également cela pour éviter un partage de référence qui pourrait s'avérer problématique dans le futur.

Pour terminer, voici le dernier bout de code d'intérêt concernant la classe CommandList :

```
...

//REQUIRES: l'état du CommandList ne peut pas être changé
// pendant l'utilisation de cet itérateur.
//EFFECTS: retourne un générateur permettant de générer toutes
// les valeurs contenues dans CommandList. Le générateur fournit
// des RobotCommand.
public Iterator robotCommands() {
    return new CommandListGen();
}

private class CommandListGen implements Iterator {
//OVERVIEW: classe "itérateur".
// Elle fournit les différentes commandes disponibles
// dans la liste de commandes dans l'ordre dans lequel
// elles ont été insérées.

    private int index;

    CommandListGen() {
        index = 0;
    }

    public boolean hasNext() {
        return index < listOfCommands.size();
    }

    public Object next() throws NoSuchElementException {
        if (!hasNext())
            throw new NoSuchElementException();
        return listOfCommands.elementAt(index++);
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}

...
```

Le concept OO couvert par ce bout de code est celui de l'abstraction par itération. Cette abstraction par itération permet de fournir à l'utilisateur un itérateur. Celui-ci va pouvoir l'utiliser pour parcourir un à un les éléments fournis par cet itérateur sans connaître l'ordre dans lequel ces éléments vont lui être fournis.

Pour créer un itérateur, nous avons dû créer une inner class implémentant l'interface Iterator. Cette inner class est un générateur et implémente donc les trois méthodes de cette interface. La méthode remove() étant cependant spécifiée comme étant optionnelle dans Iterator, nous avons choisi de ne pas lui fournir une véritable implémentation. C'est pourquoi, si elle est appelée, elle renvoie une UnsupportedOperationException.

Afin de permettre à l'utilisateur de la classe CommandList d'avoir accès à l'abstraction par itération, nous avons créé la méthode robotCommands() au sein de cette classe. Cette méthode retourne une instance de l'inner class CommandListGen (et donc un itérateur). Notez cependant la précondition de cette méthode qui demande que l'état de l'instance de CommandList sur laquelle elle a été appelée, ne soit pas modifié tant que l'utilisateur se sert de l'itérateur.

5.4.6 RobotBluetoothUnit & ComputerBluetoothUnit

Afin de permettre à un robot et un ordinateur de communiquer ensemble, nous avons créé les classes RobotBluetoothUnit et ComputerBluetoothUnit. Ces classes leur permettent de communiquer à l'aide de la technologie Bluetooth.

Voici un bout de code constituant une partie de l'implémentation de la classe RobotBluetoothUnit :

```
public class RobotBluetoothUnit {
//OVERVIEW: cette classe est utilisable par un robot.
// Elle permet à un robot d'attendre une connexion
// Bluetooth venant d'un autre appareil (ordinateur ou
// robot).
//
// IR(c): c.connection != null
// A: Etat_Connexion
// FA(c): e1 où
//   e1 = {connecté, pasConnecté}

    private BTConnection connection;
    private DataInputStream dis;
    private DataOutputStream dos;
    private boolean connected;

    //EFFECTS: crée une instance de RobotBluetoothUnit.
    // this_post = <e1> où e1 = pasConnecté.
    public RobotBluetoothUnit() {
        connection = null;
        dis = null;
        dos = null;
        connected = false;
```

```
    }

    //EFFECTS: attend une connexion provenant d'un autre
    // appareil lorsqu'un appareil s'y connecte,
    // this_post = <e1> où e1 = connecté.
    // Si l'état du RobotBluetoothUnit est déjà "connecté",
    // renvoie une AlreadyConnectedException. Si la
    // tentative de connexion ne se passe pas correctement,
    // renvoie une ConnectionFailedException.
    public void waitForConnection()
            throws AlreadyConnectedException,
                ConnectionFailedException {
        if (!connected) {
            connection = Bluetooth.waitForConnection();
            if (connection != null) {
                dis = connection.openDataInputStream();
                dos = connection.openDataOutputStream();
                connected = true;
            }
            else throw new ConnectionFailedException();
        }
        else throw new AlreadyConnectedException();
    }

    ...
```

Cette classe permet à un robot d'attendre une connexion entrante à l'aide de la méthode waitForConnection(). Lorsque la connexion est établie avec, pour ce projet, un ordinateur, il est possible d'envoyer et de recevoir des messages.

Voici un bout de code de la classe ComputerBluetoothUnit :

```
public class ComputerBluetoothUnit {
//OVERVIEW: cette classe est utilisable par un ordinateur.
// Elle permet à un ordinateur de se connecter à un robot
// à l'aide de la technologie Bluetooth.
//
// IR(c): c.connector != null && c.dis != null &&
//  c.dos != null
// A: Etat_Connexion
// FA(c): e1 où
//   e1 = {connecté, pasConnecté}

    private boolean connected;
    private DataInputStream dis;
    private DataOutputStream dos;
    private NXTConnector connector;

    //EFFECTS: crée une instance de ComputerBluetoothUnit.
    // this_post = <e1> où e1 =  pasConnecté.
    public ComputerBluetoothUnit() {
        connected = false;
        dis = null;
        dos = null;
        connector = new NXTConnector();
    }
```

```
//EFFECTS: fournit la liste des robots dont le nom est
// le même que celui fournit en paramètre.
// Renvoie une NXTCommException en cas de problème au
// niveau de la tentative de recherche.
public static NXTInfo[] lookForRobots(string name)
        throws NXTCommException {
    NXTComm nxtComm = NXTCommFactory.createNXTComm(
        NXTCommFactory.BLUETOOTH);
    return nxtComm.search(name, NXTCommFactory.BLUETOOTH);
}

//EFFECTS: établit la connexion avec le robot
// dont les informations sont fournies en paramètre.
// Renvoie une AlreadyConnectedException si
// l'état de la connexion était déjà "connecté". Si
// la tentative de connexion rate, renvoie une
// ConnectionFailedException.
public void connect2Robot(NXTInfo robotInfo)
        throws AlreadyConnectedException,
            ConnectionFailedException {
    if (!connected) {
        connected = connector.connectTo(robotInfo, 0);
        if (connected) {
            dis = connector.getDataIn();
            dos = connector.getDataOut();
        }
        else throw new ConnectionFailedException();
    }
    else throw new AlreadyConnectedException();
}

...
```

Cette classe permet à un ordinateur de chercher un ou plusieurs robots atteignables à l'aide de la technologie Bluetooth et ayant un certain nom. Il est ensuite possible de se connecter à l'un de ces robots et de lui envoyer et de recevoir des messages.

5.4.7 AlphaRexProgram & RobotController

La classe RobotController se sert de la classe ComputerBluetoothUnit pour permettre à un utilisateur de se connecter à un robot. Après s'être connectée, elle récupère la liste des commandes disponibles auprès du robot auquel l'utilisateur a décidé de se connecter. Ensuite, ce même utilisateur peut se servir de cette classe pour visualiser ces commandes et en exécuter certaines.

La classe AlphaRexProgram se sert de la classe RobotBluetoothUnit pour attendre une connexion entrante de la part d'un ordinateur. Ensuite elle envoie à celui-ci la liste des commandes disponibles fournies par AlphaRexControlUnit et exécute les commandes que l'utilisateur souhaite exécuter.

Chapitre 6

Conclusion

6.1 Résumé des contributions

L'objectif de ce mémoire était de créer un support pédagogique à la programmation orientée-objet en utilisant le firmware leJOS et un robot LEGO Mindstorms NXT 2.0. Ce mémoire avait également pour but que ce support pédagogique soit ludique et attractif pour les étudiants. En effet, comme mentionné dans l'état de l'art, il est plus facile d'apprendre en s'amusant car cela suscite de l'intérêt chez les étudiants et les motive davantage à se consacrer pleinement à ce qu'ils entreprennent (Atmatzidou *et al.*, 2008, page 22).

Pour remplir ces objectifs, nous avons créé une dizaine de scénarios que nous avons regroupés dans deux projets distincts. Le premier est un projet regroupant l'ensemble des cas simples. Ces scénarios ont pour vocation de se focaliser sur un petit nombre de concepts OO à la fois.

Le deuxième projet, en revanche, ne contient qu'un seul cas. Celui-ci est considéré comme étant plus complexe car il s'agit en réalité d'un scénario de plus grande envergure, au cours duquel l'étudiant est amené à utiliser un grand nombre de concepts OO vus au cours des scénarios du premier projet et au cours de CPOO.

Afin de concevoir ces différents scénarios, nous avons tout d'abord pris le temps de tester les différentes possibilités offertes par le firmware leJOS (version 0.85). Nous avons donc effectué un certain nombre de tests avec les différents capteurs qui étaient en notre possession, nous avons construit différents modèles de robot, etc. Nous avons pris le temps de déterminer comment nous pourrions faire utiliser certains concepts OO par les étudiants à l'aide d'exercices et de ce firmware.

Dans le premier projet, nous avons précisé l'objectif (pour le corps professoral) et l'énoncé de chaque scénario. Nous y avons également inclus la liste des concepts OO abordés dans chacun de ces scénarios. Enfin, une liste de certaines erreurs que les étudiants pourraient être amenés à commettre est mise à leur disposition ainsi qu'une solution détaillée.

Dans certains cas, nous avons jugé intéressant de fournir quelques brèves explications sur les concepts OO à utiliser dans certains scénarios. C'est le cas notamment pour le principe de Liskov et la mutabilité.

Pour le deuxième projet, au chapitre 5, nous avons suivi les grandes lignes du premier. Il y dispose également d'un énoncé pour son scénario, d'une liste de certains points de matière abordés au cours de ce scénario, d'une liste d'erreurs jugées possibles ainsi qu'une solution fournie avec un maximum d'explications. Cette solution a été divisée en plusieurs sous-sections en raison de la taille beaucoup plus importante de ce scénario. La plupart de ces sous-sections concernent une ou plusieurs classes à concevoir dans le cadre de ce projet.

6.2 Analyse critique sur les résultats obtenus

Nous pensons qu'un robot LEGO Mindstorms NXT 2.0 programmé à l'aide du firmware leJOS peut véritablement être une valeur ajoutée dans le cadre d'exemples ou de travaux pratiques dans un cours de programmation orientée-objet. En effet, comme nous l'avons constaté dans le chapitre sur l'état de l'art, il est plus facile et plus efficace d'apprendre en s'amusant. Les personnes qui ont la possibilité d'apprendre en s'amusant font plus attention, parviennent souvent à mieux se concentrer et sont généralement plus enthousiastes dans leur étude.

Les travaux concernant la visualisation expliquent qu'elle peut permettre aux étudiants de mieux comprendre des concepts abstraits à l'aide de représentations graphiques. Cela a été utilisé dans différents outils à caractère pédagogique. Dans le support créé pour ce mémoire, nous pouvons voir le robot comme étant cette représentation qui va permettre de rendre certains concepts plus concrets pour les étudiants. Etant donné que notre support pédagogique se base également sur le constructivisme, il faut aussi s'assurer que, comme préconisé par cette théorie, les concepts les plus basiques soient bien compris par les étudiants avant qu'ils ne passent à des concepts plus complexes.

Nous pensons que la méthode consistant à fournir un énoncé ainsi qu'un maximum d'informations permettant aux étudiants de comprendre leurs erreurs et comment résoudre le scénario, permet de faciliter cela. En effet, en cas de difficulté pour parvenir à trouver une solution, les étudiants ont accès à une liste des principaux concepts concernés par le scénario qui pose problème, en plus de la solution détaillée. Cette liste les renvoie vers divers chapitres du cours de CPOO qui nous a servi de base dans l'élaboration de notre support pédagogique.

Pour toutes ces raisons, nous pensons que le support pédagogique qui a été créé dans le cadre de ce mémoire peut intéresser les étudiants et les aider à mieux appréhender la matière d'un cours de programmation OO.

Cependant, l'utilisation du firmware leJOS dans un cours de programmation peut également s'avérer problématique. Utiliser un tel firmware demande notamment aux étudiants qu'ils prennent du temps pour le tester et comprendre son fonctionnement. Ce firmware n'est pas non plus sans défaut. Lorsque nous avons tenté de faire communiquer un robot et un ordinateur ensemble à l'aide de chaîne de caractères, nous nous sommes rendus compte que ça ne marchait pas à moins de rajouter un caractère spécifique à la fin des chaînes de caractères envoyées (\n). Nous avons trouvé, dans la documentation de leJOS, des méthodes qui pourraient être plus indiquées pour l'envoi de chaînes de caractères. Cependant, certaines d'entre elles n'ont pas encore été implémentées dans la version que nous avons utilisée.

Nous avons aussi rencontré un problème au niveau du capteur de couleurs. Dans le firmware leJOS, il est possible d'utiliser ce capteur notamment pour allumer la lampe de celui-ci dans une certaine couleur et pour détecter la couleur de l'objet se trouvant en face de lui. Nous avons cependant constaté que, dans la version de leJOS que nous avons utilisée, la classe proposée n'était pas totalement fonctionnelle. En effet, nous pouvions par exemple faire appel à la méthode permettant de détecter la couleur mais cette méthode ne renvoyait aucun résultat valable. Pour régler en partie ce problème, nous avons dû mettre à jour un fichier classes.jar que les développeurs de leJOS ont mis à disposition un peu après la sortie de la version que nous avons utilisée dans le cadre de ce mémoire. Cependant la méthode readColor() ne permettait toujours pas de fournir le nom de la couleur détectée par le capteur, nous avons donc dû procéder en utilisant une autre méthode qui est devenue fonctionnelle depuis la mise à jour de ce fichier classes.jar.

Les étudiants peuvent également être confrontés à d'autres problèmes inhérents à l'utilisation de tels outils. Nous avons, par exemple, pu constater que

certains capteurs ne fonctionnent pas sous le firmware leJOS lorsqu'ils sont connectés à certains ports du robot LEGO Mindstorms NXT 2.0. L'utilisation de ces outils peut aussi être problématique car, comme expliqué plus haut, ils nécessitent que les étudiants effectuent davantage de recherche pour comprendre comment manipuler et programmer le robot à l'aide du firmware leJOS. Cela peut cependant également constituer un point positif car les étudiants sont confrontés à des types de problèmes pouvant survenir au cours de leur future carrière et il est donc important qu'ils soient en mesure de rechercher davantage d'informations par leurs propres moyens pour les résoudre ou les contourner.

Il y a aussi quelques différences notables entre le Java standard et le Java proposé via le firmware leJOS. Par exemple, la classe Vector n'est pas tout à fait la même. Un certain nombre de méthodes proposées par ces deux versions disposent d'un nom différent. Nous avons également pu constater cette problématique avec d'autres classes. Cela pourrait poser quelques problèmes aux étudiants dans un premier temps mais il s'agit aussi d'une problématique qui pourrait survenir dans le cadre de leur future carrière en tant que développeur.

6.3 Perspectives futures

Si ce mémoire devait être repris par d'autres étudiants, nous pensons qu'il serait intéressant d'agrémenter notre travail d'une utilisation plus poussive du firmware leJOS. En effet, il existe énormément de classes au niveau de l'API de ce firmware pour le robot et pour l'ordinateur qui n'ont pas été utilisées alors qu'elles auraient pu être utiles. Par exemple, nous pensons à la classe Command qui permet de faire exécuter des actions à un robot LEGO Mindstorms à distance, par exemple à partir d'un ordinateur.

La version 0.9 de leJOS ayant été rendue disponible, il serait également intéressant de voir si certains bugs ou problèmes que nous avons pu rencontrer sont désormais réglés. Nous pensons, dans ce cas-ci par exemple, au problème que nous avons rencontré au niveau de la communication entre un ordinateur et un robot à l'aide de chaînes de caractères. La gestion de la lecture de couleurs à l'aide d'un capteur pourrait également avoir été réglée. Nous avons dû gérer celle-ci d'une autre manière car la méthode readColor() du capteur de couleurs ne fonctionnait pas.

Notre outil devrait être évalué par une véritable utilisation dans le cadre de travaux pratiques d'un cours de programmation orientée-objet. Ceci nous

permettrait d'obtenir un retour d'expérience pertinent et provenant de potentiels utilisateurs finaux.

Bibliographie

Atmatzidou, S., Markelis, I. et Demetriadis, S. (2008). The use of lego mindstorms in elementary and secondary education : game as a way of triggering learning. *In Workshop proceedings of SIMPAR 2008*, pages 22–30.

Ben-ari, M. (2008). Constructivism in computer science education 1.

Bruce, K. B. (2005). Controversy on how to teach cs 1 : a discussion on the sigcse-members mailing list. *SIGCSE Bull.*, 37:111–117.

Chalk, B. et Fraser, K. (2008). Higher education academy centre for ics.

Cooper, S., Dann, W. et Pausch, R. (2003). Teaching objects-first in introductory computer science. *SIGCSE Bull.*, 35:191–195.

Georgantaki, S. et Retalis, S. (2007). Using educational tools for teaching object oriented design and programming.

Henriksen, P. et Kölling, M. (2004). greenfoot : combining object visualisation with interaction. *In Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 73–82, New York, NY, USA. ACM.

Heymans, P. (2010). Cours de conception et programmation orientées-objet (cpoo). FUNDP.

Lawhead, P. B., Duncan, M. E., Bland, C. G., Goldweber, M., Schep, M., Barnes, D. J. et Hollingsworth, R. G. (2002). A road map for teaching introductory programming using lego© mindstorms robots. *In Working group reports from ITiCSE on Innovation and technology in computer science education*, ITiCSE-WGR '02, pages 191–201, New York, NY, USA. ACM.

LEGO (2011). Site officiel de lego (13 août 2011).

Liskov, B. et Guttag, J. (2001). *Program Development in Java - Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley.

Mödritscher, F. (2006). e-learning theories in practice : A comparison of three methods. *Journal of Universal Science and Technology of Learning*, 0(0):3–18.

Mergel, B. (1998). Instructional design and learning theory. Mémoire de D.E.A., University of Saskatchewan.

Naps, T. L., Rossling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S. et Velazquez-Iturbide, J. A. (2002). Exploring the role of visualization and engagement in computer science education. *SIGCSE Bull.*, 35:131–152.

Sanders, D. et Dorn, B. (2003). Jeroo : a tool for introducing object-oriented programming. *SIGCSE Bull.*, 35:201–204.

Satratzemi, M., Xinogalos, S. et Dagdilelis, V. (2003). An environment for teaching object-oriented programming : Objectkarel. *Advanced Learning Technologies, IEEE International Conference on*, 0:342.

Van Haaster, K. et Hagan, D. (2004). Teaching and learning with bluej : an evaluation of a pedagogical tool.

Xinogalos, S. (2010). An interactive learning environment for teaching the imperative and object-oriented programming techniques in various learning contexts. *In* Lytras, M. D., Ordonez De Pablos, P., Ziderman, A., Roulstone, A., Maurer, H. et Imber, J. B., éditeurs : *Knowledge Management, Information Systems, E-Learning, and Sustainability Research*, volume 111 de *Communications in Computer and Information Science*, pages 512–520. Springer Berlin Heidelberg.