Assistance à l'apprentissage de l'algorithmique : Méthode et outil pour l'évaluation et la rétroaction

Souleiman Ali Houssein

Communauté d'Universités et d'Établissements Lille Nord de France

Université de Lille

Université de Lille

École Doctorale SPI

Laboratoire CRIStAL

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE LILLE

Spécialité : Informatique et applications

Assistance à l'apprentissage de l'algorithmique : Méthode et outil pour l'évaluation et la rétroaction

Présentée et soutenue publiquement par

Souleiman Ali Houssein

Le 12 septembre 2019

JURY

M. Laurent GRISONI	Professeur des Universités, Université de Lille	Président du jury
Mme. Monique GRANDBASTIEN	Professeure émérite, Université de Lorraine	Rapporteur
M. Jean-Charles MARTY	Maître de Conférences, HDR Université Savoie-Mont Blanc	Rapporteur
M. Yvan PETER	Maître de Conférences, HDR Université de Lille	Directeur de Thèse

À mes parents, ma famille et à tout lecteur.

Résumé

La maîtrise des concepts fondamentaux de la programmation et la capacité à réaliser des programmes simples sont les objectifs essentiels de l'enseignement dans les cours d'introduction à l'informatique. L'enseignement et l'apprentissage de la programmation sont considérés comme complexes, ce qui explique le taux d'abandon important dans ces filières, largement documenté dans la littérature. De nombreux travaux proposent des environnements d'apprentissage de la programmation assistant les apprenants dans la maîtrise de la syntaxe et de la sémantique des langages de programmation. Toutefois, parmi les causes d'échecs, la littérature identifie un manque de capacité à décomposer et formaliser un problème sous forme d'algorithme.

Dans le cadre de cette thèse, nous nous intéressons à la phase d'analyse et de mise en solution algorithmique. Cette phase a pour rôle de faire acquérir à l'apprenant une démarche de résolution de problème et de formalisation de la solution. Dans cette phase l'apprenant décrit ou structure sa solution algorithmique à l'aide d'une notation formelle (pseudo-code) indépendant de tout langage de programmation.

L'étude de la littérature indique que l'amélioration de la capacité de résolution de problème passe par la pratique. Dans le cadre de la programmation, pour les novices, il est nécessaire de faire de nombreux exercices (résolution de problème) avec des niveaux de difficulté croissants. Il est par ailleurs nécessaire de fournir une rétroaction en rapport avec les erreurs commises par les apprenants.

Dans cet objectif nous proposons *AlgoInit*, un environnement Web pour l'apprentissage de l'algorithmique. En nous basant sur l'étude de la littérature, nous avons défini :

- Une modélisation basée sur la taxonomie de Bloom pour définir le niveau cognitif des exercices proposés ;
- Une approche d'évaluation basée sur la comparaison de la solution apprenant avec une solution modèle. Pour comparer les solutions (apprenant et modèles) décrites en pseudo-code, nous passons par une étape de transformation des solutions en arbres étiquetés construits à partir d'une base de règles ;
- Des règles pour fournir rétroactions et exercices progressifs en fonction du résultat de l'évaluation de la solution de l'apprenant.

Afin d'évaluer la potentialité de notre prototype, nous avons mené deux expérimentations à l'université de Djibouti. La première expérimentation a été consacrée à l'évaluation de la capacité de notre prototype à reconnaître les différentes solutions algorithmiques. Quant à la deuxième, elle a été consacrée à l'évaluation de l'intérêt pédagogique d'*AlgoInit.* Ces expérimentations ont montré des résultats probants sur la capacité d'*AlgoInit* à classer les solutions (correctes et incorrectes) et à fournir des rétroactions utiles. Les résultats indiquent également que notre système à une influence significative sur la capacité de résolution de problèmes des étudiants.

Mots-clés : ingénierie des EIAH, assistance, apprentissage de l'algorithmique, évaluation d'algorithme, rétroaction, génération d'exercices, évaluation formative

Abstract

Mastering the basic concepts of programming and the ability to carry out simple programs are the essential objectives of introductory computer science courses. The teaching and learning of programming is considered complex, which explains the high drop-out rate of these programs, widely documented in the literature. Many works provide learning environments for programming that assist learners in mastering the syntax and semantics of programming languages. However, among the causes of failure, the literature identifies a lack of ability to decompose and formalize a problem in the form of an algorithm.

In the context of this thesis, we are interested in the analysis and algorithmic solution phase. The purpose of this phase is to teach the learner a problem solving and formalization process. In this phase the learner describes or structures his algorithmic solution using a formal notation (pseudo-code) independent of any programming language.

The literature review indicates that improving problem solving skills requires practice. As part of programming for novices, it is necessary to do many exercises (problem solving) with increasing difficulty levels. It is also necessary to provide feedback related to the mistakes made by the learners.

For this purpose we propose *AlgoInit*, a Web environment for algorithmic learning. Based on the study of the literature, we defined:

- A modeling based on Bloom's taxonomy to define the cognitive level of the proposed exercises.
- An evaluation approach based on the comparison of a learning solution to a model solution. To compare the solutions (learner and models) described in pseudo-code, we go through a step of transforming solutions into labeled trees built from a rule base.
- Rules for providing feedback and progressive exercises based on the outcome of the learner's solution assessment.

To evaluate the potential of our prototype, we have conducted two experiments at the University of Djibouti. The first experiment was devoted to evaluating the ability of our prototype to recognize the different algorithmic solutions. The second one was devoted to the evaluation of the educational interest of *AlgoInit*. These experiments have shown convincing results on the ability of *AlgoInit* to classify solutions (correct and incorrect) and to provide useful feedback. The results also indicate that our system has a significant influence on students' problem solving ability.

Keywords: TEL, novice programming, CS1, automatic assessment, algorithm evaluation, formative feedback

Publications

- ***Article d'atelier avec comité de lecture :***

Ali Houssein, S., & Peter, Y. (2016). Etat de l'art des outils de soutien à l'enseignement / apprentissage de la programmation. In *Atelier Apprentissage Instrumenté de l'informatique (ORPHEE).*

Ali Houssein, S., & Peter, Y. (2017b). Outils d'assistance et les difficultés de l'enseignement / apprentissage de la programmation quelle aide ? In *Atelier apprentissage de la pensée informatique de la maternelle à l'université : Recherches, Pratique et Méthode, EIAH 2017.*

- ***Article de Conférence avec actes et comité de lecture :***

Ali Houssein, S., & Peter, Y. (2017 a). Orchestration and Adaptation of Learning Scenarios—Application to the Case of Programming Learning/Teaching. In *IEEE/ACS 14th international Conference on Computer Systems and Applications (AICSSA)* (pp. 7–11).

Ali Houssein, S., & Peter, Y. (2018). Evaluation of algorithms to support novice programmer. In *Proceedings of the 10th International Conference on Education Technology and Computers. ACM* (pp. 383–387).

Remerciements

Mes remerciements s'adressent à toutes les personnes qui m'ont soutenu et accompagné durant ces quatre dernières années.

Je souhaite tout d'abord remercier les membres du jury : Monique GRANDBASTIEN et Jean-Charles MARTY de m'avoir fait l'honneur d'accepter d'être rapporteur de cette thèse, ainsi que Laurent GRISONI d'avoir accepté d'être examinateur et président de jury.

Merci ensuite à Yvan PETER, pour avoir accepté d'être mon directeur de thèse et m'avoir permis d'intégrer l'équipe NOCE… merci pour m'avoir encadré scientifiquement pendant ma thèse… merci pour sa disponibilité indéfectible, ses conseils, ses remarques judicieuses qui ont enrichi mon travail et surtout d'avoir cru en moi et de m'avoir fait confiance.

Merci aussi aux membres de l'équipe NOCE et à son responsable Luigi LANCERI, pour son soutien, ses conseils, ses discussions et surtout pour la relecture de ma thèse.

Je remercie aussi l'université de Djibouti pour m'avoir accordé une bourse et surtout pour les congés doctoraux.

Enfin, je souhaite associer à ces remerciements mes parents, ma famille, mes frères et sœurs pour leur soutien, leur patience et leur investissement dans ce projet… merci aussi à mes amis pour leur encouragement et leurs conseils.

TABLE DES MATIERES

TABLE DES FIGURES

LISTE DES TABLEAUX

Introduction

Contexte

L'informatique est de nos jours un domaine attractif en terme d'emploi. C'est une discipline qui intervient dans de nombreux secteurs d'activité, tels que les banques, la santé, l'éducation, la défense, etc. Du fait de cette attractivité, l'enseignement de l'informatique est aujourd'hui un domaine incontournable, quel que soit le niveau d'enseignement (primaire, secondaire et université). Ainsi (Nijimbere, 2015) souligne que « *l'informatique, n'est pas seulement un élément de culture générale citoyenne, mais elle constitue aussi l'un des plus grands débouché d'emploi dans le monde […], et son enseignement semble donc une nécessité, pour ne pas vivre dans une nouvelle forme d'illettrisme* ». Dans la même optique Barack Obama, l'ancien président des Etats Unis, dans son discours en janvier 2016, lors du lancement de l'initiative « *Computer Science for All* »[1], a déclaré que dans le monde, de nos jours « *l'informatique n'est plus une compétence facultative, c'est une compétence de base, au même titre que les compétences de base (lecture, écriture et arithmétique) enseignées à l'école*». Cette initiative avait pour objectif d'enseigner aux jeunes américains les compétences de base de l'informatique, et plus précisément la programmation, à tous les niveaux scolaires. (Nijimbere, 2015) souligne que dans cet enseignement, l'apprentissage de l'algorithmique et de la programmation doit être une priorité du fait de son apport au développement intellectuel des apprenants.

Les composantes essentielles de l'enseignement de l'informatique en première année d'université, sont la maîtrise des fondamentaux de la programmation et la capacité de réaliser des programmes simples (Muratet, 2010). L'enseignement et l'apprentissage de ces composantes sont considérés comme des tâches complexes. Ceci est largement documenté dans la littérature. Cela se traduit par des abandons massifs des étudiants de cette discipline dès la première année (Winslow, 1996), (Jenkins, 2002), (Robins, Rountree, & Rountree, 2003), (Lahtinen, Ala-Mutka, & Järvinen, 2005), (Gomes & Mendes, 2007), (Tan, Ting, & Ling, 2009), (Luxton-Reilly et al., 2018). Selon le rapport ACM/AIS/IEEE, « *Les universités affirment que régulièrement 50 % ou plus de leurs étudiants ayant initialement choisi des études en informatique décident rapidement d'abandonner* » (ACM, AIS, & IEEE, 2005).

[1] *https: //obamawhitehouse.archives.gov/blog/2016/01/30/computer-science-all*

Pour sa part, Winslow, souligne qu'il faut environ 10 ans pour transformer un novice en expert de la programmation (Winslow, 1996). Dans la même optique (Mead et al., 2006) soulignent que « *au cours des 25 dernières années, des études nationales et internationales ont fourni des indicateurs empiriques qui montrent que l'apprentissage de la programmation est effectivement difficile pour la plupart des étudiants.* ».

Je suis enseignant à l'université de Djibouti et notre université est également touchée par ce phénomène. L'année dernière en première année, seulement 50 % des inscrits sont passés en deuxième année. L'université de Djibouti a également fermé temporairement la filière de DUT en informatique l'année dernière pour réfléchir à des nouvelles stratégies pour accompagner les apprenants dans leur réussite et ainsi former des informaticiens qualifiés.

Dans ce contexte, l'enseignement et l'apprentissage de la programmation ont fait l'objet des nombreux travaux. Selon (Pears et al., 2007) ces travaux se répartissent selon quatre grands axes de recherche : (i) ***Le curriculum*****,** qui s'intéresse au contenu du cours ; (ii) ***La pédagogie,*** qui porte sur les méthodes d'enseignement et d'apprentissage; (iii) ***Les langages de programmation*** les plus adaptés à l'enseignement des concepts de l'informatique ; (iv) ***Les environnements pour soutenir l'enseignement / l'apprentissage,*** afin de mieux assister l'enseignement et l'apprentissage des cours d'introduction.

Les travaux décrits dans ce manuscrit, portent sur le quatrième axe. Ainsi, notre thèse s'inscrit dans le champ de l'ingénierie des EIAH (Environnements Informatiques pour l'apprentissage Humain), qui, selon (Tchounikine, 2009), nous renvoie aux travaux, dont l'objectif est d'élaborer des concepts, des méthodes et techniques pour mettre en place des outils qui facilitent ou renforcent l'apprentissage humain.

Nous allons maintenant présenter brièvement dans la section suivante la question et l'objectif de la recherche.

Problématique et objet de recherche

(Qian & James, 2017) identifient trois types de connaissances à acquérir dans l'apprentissage de la programmation, il s'agit de : (i) ***connaissances syntaxiques***, c'est la connaissance des caractéristiques d'un langage de programmation, ses descriptions syntaxiques et ses règles. Par exemple, en Java, un point virgule est nécessaire pour terminer une instruction. Les erreurs souvent commises ou les difficultés des apprenants dans ce niveau de connaissance sont largement documentées dans (Altadmri & Neil, 2015). Selon (Qian &

James, 2017), les erreurs dans ce niveau sont facilement détectables et faciles à corriger ; (ii) ***connaissances conceptuelles***, qui font référence à la connaissance des concepts de programmation et de leur fonctionnement. Par exemple, connaître le fonctionnement de la boucle `pour`, ou comment un programme s'exécute ; les difficultés dans ce niveau sont liées à la compréhension du fonctionnement de ces concepts abstraits ; pour renforcer cette connaissance, des environnements mettent en œuvre des outils de visualisation ; (iii) ***connaissances stratégiques***, qui impliquent l'utilisation des connaissances syntaxiques et conceptuelles pour résoudre des problèmes. Les difficultés dans ce niveau sont liées à la capacité de choisir les concepts appropriés pour résoudre un problème. Pour renforcer cette capacité, les environnements mettent en œuvre des analyses sur les solutions des apprenants afin de leurs fournir des rétroactions. La plupart de ces environnements sont orientés vers l'analyse des erreurs syntaxiques et sémantiques (Rongas, Kaarna, & Kalviainen, 2004), (Pears et al., 2007).

Dans le cadre de cette thèse, nous nous intéressons à renforcer les connaissances stratégiques des apprenants. Il s'agit de renforcer leur capacité à reconnaître et à bien utiliser les concepts de programmation appropriés pour résoudre des problèmes. Pour cela nous nous intéressons à la phase d'analyse et sa mise en solution algorithmique. Cette phase a pour rôle de faire réfléchir l'apprenant à comment résoudre le problème. Dans cette phase l'apprenant décrit ou structure sa solution algorithmique à l'aide d'une notation formelle (pseudo code) indépendante de tout langage de programmation. Dans cette phase les apprenants se concentrent plus sur la résolution de problèmes et se soucient moins des détails syntaxiques des langages. Selon le rapport « Computing Curricula 2001 (CC 2001) », l'avantage de cette approche est que les étudiants apprennent à analyser et à réfléchir à comment construire la solution du problème avant de transformer cette solution dans un langage de programmation. De ce fait, lors de la transformation de la solution en un programme, les étudiants pourront plus facilement déboguer leurs erreurs et passer d'un langage à un autre.

Notre université de Djibouti, met en œuvre cette approche d'enseignement dans les cours d'introduction à la programmation. Mais, à cause du nombre important d'étudiants et des niveaux hétérogènes, les enseignants ont des difficultés à vérifier l'exactitude des solutions des apprenants et à les assister, ce qui est un problème récurrent (Gomes & Mendes, 2007), (Luxton-Reilly et al., 2018).

D'autre part, pour améliorer, la capacité de résolution de problème ou consolider les connaissances stratégiques des apprenants, ceux-ci doivent résoudre de nombreux problèmes

avec des niveaux de difficulté croissants (Beaubouef, Lucas, & Howatt, 2001), (Robins et al., 2003). Cette idée est appuyée par (Inventado, 2019) qui souligne que la pratique est une stratégie efficace pour l'enseignement de la programmation et que celle-ci favorise la maîtrise des connaissances stratégiques. Elle est efficace lorsque l'apprenant est confronté à de nombreux exercices (problèmes) appropriés et soutenu par des rétroactions.

Ainsi nos travaux se focalisent, dans un premier temps sur les moyens à mettre en œuvre pour détecter les erreurs dans les solutions algorithmiques des apprenants afin de les assister. Et, dans un deuxième temps, il s'agit de fournir assez d'exercices, afin de les confronter à différentes formes d'exercices pour renforcer leur capacité de résolution de problème. Ainsi, nous pouvons formuler, nos questions de recherche comme suit :

- **Comment évaluer la solution algorithmique de l'apprenant ?** Dans cette question, nous cherchons à définir si la solution de l'apprenant, est correcte ou incorrecte. Dans ce dernier cas, il s'agit de détecter et énumérer les erreurs conceptuelles. Nous cherchons à leur donner une signification, pour assister l'apprenant dans ces erreurs.
- **Comment accompagner l'apprenant pour consolider ses connaissances stratégiques ?** Dans cette question, nous cherchons à consolider les connaissances stratégiques des apprenants, en particulier renforcer la capacité d'utiliser les concepts appropriés pour résoudre des problèmes. Dans le cas où l'apprenant fournit une solution correcte il s'agit de fournir un autre exercice qui permet la progression de l'apprenant et dans le cas d'une solution incorrecte il s'agit d'assister l'apprenant afin de terminer l'exercice. De ce fait cette question peut se décomposer en deux points : d'abord *comment organiser les différents exercices en niveaux de difficulté croissants permettant une progression de l'apprenant ?* Et *comment assister ou accompagner l'apprenant qui a une difficulté pour résoudre un exercice donné ?*

Ces questions seront traitées dans l'objectif de proposer un environnement qui fournit des exercices d'apprentissage algorithmique dans le but d'améliorer la capacité de résolution de problèmes des apprenants dans ce domaine complexe.

Ces questions de recherche sont illustrées sur la figure 1.

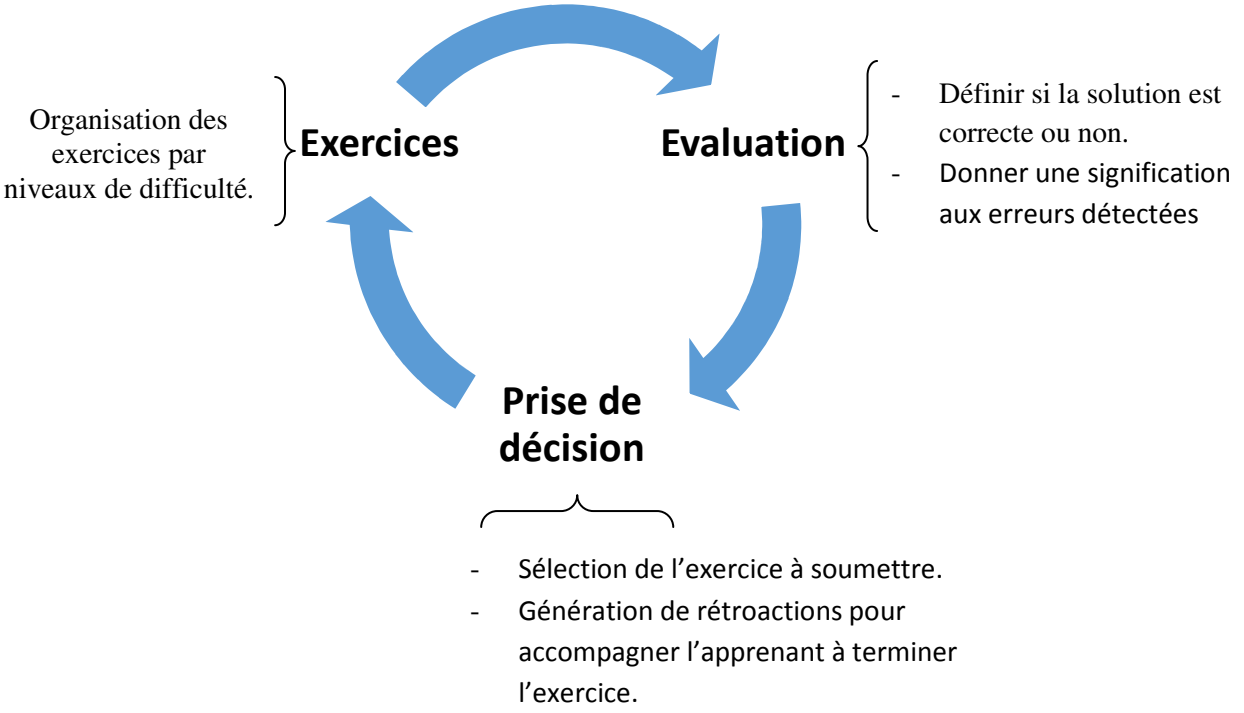


Figure 1 : interaction entre Exercices - Evaluation - Assistance

Méthodologie

Dans l'objectif de répondre à ces questions, nous avons adopté la méthodologie de recherche suivante :

1- Analyser la difficulté d'apprentissage et d'enseignement du cours d'introduction de la programmation, dans l'objectif d'identifier les verrous à lever.

2- Analyser les différentes approches d'évaluation, mise en œuvre dans les environnements d'apprentissage, et en particulier dans les environnements d'apprentissage de la programmation, pour détecter les erreurs des apprenants. Cette étude nous a permis d'identifier les solutions existantes et de proposer une approche d'évaluation afin de combler les lacunes identifiées.

3- Pour accompagner l'apprenant dans le cas d'une solution correcte et incorrecte et répondre ainsi aux deux dernières questions nous effectuons : d'une part une analyse des différentes formes d'assistance proposées et les techniques mises en œuvre dans les environnements d'apprentissage de cours d'introduction à la programmation. Cette étude nous a permis de proposer un mécanisme de génération de rétroactions. D'autre part étudier différents éléments de base qui

vont nous servir pour la proposition d'un modèle d'organisation des exercices de programmation du cours d'introduction.

4- Concevoir et implémenter un EIAH pour mettre en œuvre nos propositions afin d'améliorer la connaissance stratégique des apprenants dans le cours d'introduction à la programmation.
5- Évaluer l'environnement à travers des expérimentations réalisées dans un contexte d'apprentissage réel à l'université de Djibouti.

Contributions

Les contributions des travaux présentés dans cette thèse sont les suivantes :

1- Un modèle d'organisation des exercices (problèmes à résoudre). Cette organisation a été pensée dans l'objectif d'amener les apprenants à travailler sur des familles d'exercices, avec des niveaux de difficulté croissants pour améliorer leur capacité de résolution de problèmes.

2- Une approche d'évaluation de solution algorithmique. Cette approche permet de décider si la solution de l'apprenant est correcte ou non et d'identifier les erreurs conceptuelles, le cas échéant.

3- Un mécanisme, qui permet de fournir l'exercice suivant, mais également des rétroactions aux apprenants qui auront des difficultés pour terminer un exercice.

4- *AlgoInit*, un EIAH implémenté aux cours de nos travaux. Celui-ci permet d'évaluer, de proposer des rétroactions et des exercices de niveaux de difficulté différents.

5- L'expérimentation d'*AlgoInit* en milieu écologique et l'analyse des apports de ce dernier pour l'apprentissage de la mise en solution algorithmique.

Organisation du manuscrit

Ce manuscrit est organisé en deux grandes parties.

Partie 1 : État de l'art

Cette partie présente un état de l'art sur l'enseignement et l'apprentissage de la programmation en cours d'introduction et les environnements informatiques existants. Elle est organisée en trois chapitres. Dans le premier chapitre *(chapitre 1)*, nous présentons d'abord une revue de littérature sur les difficultés d'enseignement et d'apprentissage de la programmation dans les cours d'introduction. Ensuite, nous présentons les différentes approches d'enseignement mises en œuvre dans ce cours. L'étude menée dans ce chapitre nous a permis de nous positionner. Le second chapitre *(chapitre 2)* est dédié à l'évaluation. Nous présentons d'abord les différents types d'évaluation avant de passer en revue les différentes techniques d'évaluation dans les EIAH. Enfin nous abordons les différentes techniques ou approches d'évaluation mises en œuvre dans les environnements informatiques pour l'enseignement et l'apprentissage de la programmation. L'étude de ce chapitre nous a permis de souligner certaines lacunes à combler. Le troisième chapitre *(chapitre 3)* est consacré aux moyens de faire progresser l'apprenant dans sa capacité de résolution de problèmes. Il est organisé en deux sections. Dans une première section, nous présentons d'abord une revue de littérature sur les différentes formes et techniques de rétroaction dans le contexte de l'enseignement et l'apprentissage de la programmation. Dans la deuxième section de ce chapitre, nous présentons les éléments que nous avons utilisés pour proposer un modèle d'organisation des exercices. L'étude de la première section de ce chapitre nous a permis de souligner des lacunes à combler.

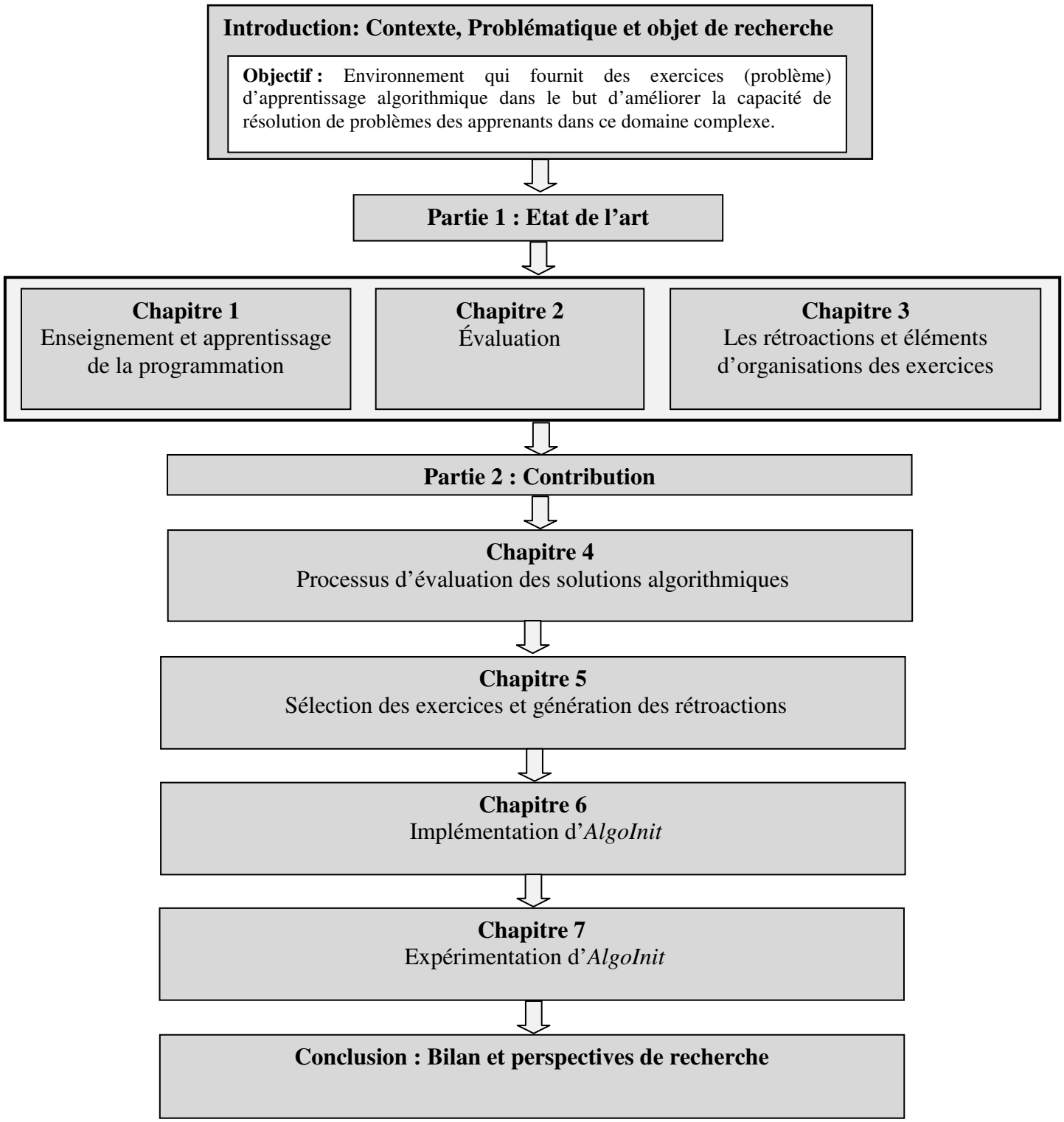
Partie 2 : Contributions

Nous présentons en quatre chapitres, l'ensemble de nos contributions. Dans le premier chapitre *(chapitre 4)* nous présentons notre approche d'évaluation. Cette approche permet de détecter et d'énumérer les erreurs commises par l'apprenant en donnant une signification à chacune d'elles. Cela permet de rendre leurs erreurs interprétables. Le second chapitre *(chapitre 5)*, quant à lui, se concentre sur l'assistance (génération des rétroactions et sélection de l'exercice suivant) fournie à l'apprenant et se base sur le résultat de l'évaluation présentée

dans le chapitre précédent. Le troisième chapitre *(chapitre 6)*, présente la réification de nos propositions dans un environnement Web appelé *AlgoInit*. Le dernier chapitre *(Chapitre 7)*, est consacré à la validation de nos propositions. Il aborde : (i) la précision d'*AlgoInit* pour identifier les erreurs dans les algorithmes ; (ii) l'analyse des effets des rétroactions et l'impact de notre environnement sur la capacité de résolution de problèmes et (iii) l'utilisabilité d'*AlgoInit*.

Nous terminons ce manuscrit par un chapitre, qui présente un bilan de nos travaux et nos perspectives de recherche.

PLAN DE THESE SCHEMATISE



Partie 1 : Etat de l’art

Chapitre 1 : Enseignement et Apprentissage de la programmation

Contenu

1.1 Introduction

Dans ce chapitre, nous présentons d'abord une revue de la littérature sur les difficultés liées au cours d'introduction à la programmation. Ensuite nous abordons les différentes approches d'enseignement pour ce cours, recensées dans le rapport « Computing Curricula 2001 (CC2001) » élaboré par l'ACM et l'IEEE (Engel & Roberts, 2001). Nous terminons par une synthèse des points abordés.

1.2 Difficultés liées à l'enseignement et l'apprentissage de la programmation

L'enseignement et l'apprentissage de la programmation sont complexes, en particulier dans les cours d'initiation. Cette difficulté est confirmée par le taux d'abandon des étudiants des cours d'introduction de premier cycle universitaire qui varie de 25% à 80% de par le monde (Kaasbøll, 2002). Cette difficulté est également soulignée par (Winslow, 1996), qui précise qu'il faut environ 10 ans pour transformer un novice en un expert de la programmation. Dans la même optique, (Mcgettrick et al., 2005) rapporté par (Djelil, 2016) souligne que « *l'enseignement des fondamentaux de la programmation au niveau universitaire est considéré comme l'un des défis majeurs de la didactique de l'informatique* ».

Cette difficulté est due au fait que la programmation nécessite plusieurs compétences (Renumol, Jayaprakash, & Janakiram, 2009) : (i) capacité de résolution de problèmes ; (ii) connaissance des outils de programmation ; (iii) maîtrise de la syntaxe et de la sémantique d'un langage de programmation.

Bien qu'il existe une diversité de travaux traitant de la difficulté de l'apprentissage de la programmation dans le cours d'introduction, ceux-ci convergent sur trois types de problématiques qui sont : (i) les problèmes liés à l'enseignement ; (ii) le manque de compétences des apprenants dans la méthode de résolution de problèmes ; (iii) le manque de compétences de codage (difficulté à traduire la solution dans un langage informatique, de trouver ou retracer les erreurs de leur programme…). Nous décrivons dans la suite ces trois types de problèmes.

1.2.1 Problème liés à l'enseignement

Le taux d'échec élevé dans les cours d'introduction de la programmation peut-être dû à un problème d'enseignement. Différentes études montrent qu'une des raisons de ce

problème vient du fait que les enseignants ne peuvent pas suffisamment assister ou aider les étudiants qui ont des difficultés à cause du nombre important d'étudiants dans la salle de classe (Jenkins, 2002), (Gomes & Mendes, 2007), (Renumol et al., 2009). Dans la même optique (Konecki, 2014) souligne que, face à des apprenants qui diffèrent de style et de vitesse d'apprentissage, les enseignants ont du mal à adapter leurs cours aux spécificités de chaque apprenant. En réalité, il est impossible que l'enseignant puisse donner une telle assistance ou personnalisation en raison des contraintes de temps et du nombre important d'étudiants avec des niveaux hétérogènes (certains étudiants ont une base de connaissances alors que d'autres sont novices). Une étude réalisée par (Bennedsen & Caspersen, 2007), souligne que le taux de réussite est important, lorsque le nombre d'étudiants n'est pas trop élevé (pas plus de 30 étudiants).

Les cours d'introduction à la programmation requièrent un apprentissage (Shaffer, Doube, & Tuovinen, 2003) : (i) ***déclaratif***, qui demande à connaître les différents concepts abstraits, c'est l'étape de l'acquisition de connaissance conceptuelle; (ii) ***procédural*** qui demande à utiliser les connaissances conceptuelles, pour résoudre des problèmes. Elle s'acquiert par la pratique. Dans cette étape la capacité de résolution de problèmes est mobilisée. Elle demande de la part des étudiants de choisir les concepts appropriés lors de la résolution d'un problème donné. Il demande également une pédagogie avec plus de pratique (Shaffer et al., 2003).

Un autre problème soulevé par les auteurs (Gomes & Mendes, 2007), (Jenkins, 2002) est que les enseignants se concentrent plutôt sur l'enseignement d'un langage de programmation et de ses détails syntaxiques (connaissance déclarative), au lieu de promouvoir ou d'insister sur la résolution de problèmes (connaissance procédurale). Ils soulignent également que le langage utilisé dans les cours est choisi en fonction de sa popularité dans l'industrie et non de son adéquation pédagogique.

1.2.2 Difficultés liées au manque de compétences en résolution de problèmes

La capacité de résolution de problèmes, est l'une des grandes compétences nécessaires du 21ème siècle[2]. Elle est considérée comme l'une des caractéristiques humaines la plus importante, qui aide l'individu à progresser. Elle demande à l'apprenant de résoudre des situations dans lesquelles aucune séquence d'étapes apparente pour atteindre l'objectif n'est

[2] http://www.oecd.org/pisa/keyfindings/PISA-2012-PS-results-fre-FRANCE.pdf

décrite (Greiff et al., 2017). Elle l'engage dans un processus cognitif, qui demande de comprendre d'abord la situation puis de chercher un moyen d'atteindre l'objectif (Jonassen, 2010). (Polya, 1957) a décomposé la stratégie de résolution de problèmes comme suit :

- Comprendre la nature du problème ;
- Élaborer un plan pour résoudre le problème ;
- Mettre en œuvre le plan ;
- Vérifier le résultat du plan.

a) La résolution de problème dans l'enseignement et l'apprentissage de la programmation

La stratégie de résolution de problèmes de (Polya, 1957) est généralement utilisée dans l'apprentissage de la programmation dans la mesure où l'exercice (problème) proposé est d'abord analysé (*comprendre le problème*), puis décrit sous forme d'algorithme (*élaborer un plan pour résoudre le problème*) et enfin la solution algorithmique est traduite en un programme valide (*mettre en œuvre et vérifier le résultat de son plan*). Ainsi, la capacité de résolution de problèmes est une étape importante dans l'apprentissage de la programmation. Knuth, rapporté par (Beaubouef & Mason, 2005) souligne que « *CS = problem solving* ». Selon (Qian & James, 2017), la capacité de résolution de problèmes, nommée *connaissance stratégique*, est la capacité de mobiliser les connaissances syntaxiques et conceptuelles de la programmation pour résoudre des problèmes. En d'autres mots, l'apprentissage de la programmation repose sur la résolution de problèmes. Selon (Keuning, Jeuring, & Heeren, 2018) le principal objectif de l'apprentissage de la programmation est la capacité à résoudre des problèmes. Dans la même optique (Eitelman, 2006) souligne que la résolution de problèmes est la fondation des compétences de la programmation. Plusieurs autres études ont confirmé que la capacité de résolution de problèmes est un prédictif significatif pour la performance de l'apprenant dans la programmation (Mayer, Dyck, & Vilberg, 1986), (Bergin & Reilly, 2006), (Lishinski, Yadav, Enbody, & Good, 2016).

b) Quelques difficultés liées à l'enseignement et à l'apprenant

Bien que dans la littérature, la résolution de problèmes soit décrite comme la base ou le principal objectif de l'apprentissage de la programmation, on constate que : (i) la plupart des enseignements actuels se concentrent plus sur l'enseignement d'un langage de programmation et de ses détails syntaxiques, au lieu de promouvoir ou d'insister sur la

résolution de problèmes. En d'autres mots, la plus grande partie de l'enseignement se concentre sur la quatrième phase de résolution de problèmes (*essayer et vérifier une solution* sans réfléchir au préalable à comment la mettre en œuvre) (Winslow, 1996), (Gomes & Mendes, 2007), (Jenkins, 2002); (ii) l'enseignant est confronté à un nombre important d'étudiants, ce qui rend difficile de réaliser ou d'effectuer plus d'exercices ; (iii) les apprenants ont des problèmes de compétences en résolution, ce qui se traduit selon (Gomes & Mendes, 2007) par :

- Une manque de capacité de compréhension du problème : les étudiants essaient de résoudre le problème sans le comprendre complètement : soit ils ont des difficultés d'interprétation, soit ils ont tendance à écrire les réponses sans réfléchir.
- Manque de persévérance : la résolution des problèmes exige des efforts et de la persévérance. Les étudiants, face à la difficulté du problème, préfèrent demander ou attendre la solution au lieu de continuer à essayer de résoudre le problème. Ceci est particulièrement important, étant donné que l'apprentissage est plus efficace lorsque les étudiants trouvent la solution, au lieu de lire la solution.
- Transfert de connaissances : beaucoup d'étudiants n'établissent pas d'analogie correcte avec les anciens problèmes rencontrés pour le transfert de connaissances sur le nouveau problème. Cette difficulté est détaillée dans la section ci-dessous.

c) Problème de transfert de connaissances

Un autre facteur clé de la capacité de résolution de problèmes est la construction de *schéma* (en anglais, *chunks*), qui est une abstraction d'une collection d'expériences que les experts utilisent pour analyser et résoudre des situations nouvelles. (Jonassen, 2010) souligne que la compétence de résolution de problèmes dépend d'un schéma bien construit, capable de reconnaître les différents types de problèmes afin d'activer ou d'appliquer une solution à un problème donné. Le schéma ainsi décrit, fait référence à une construction cognitive qui aide la personne à stocker, organiser et comprendre l'information (Mead et al., 2006). C'est d'ailleurs ce qui fait la différence entre le novice et l'expert. Par exemple (Sweller, 1988) et (Winslow, 1996) rapportent que les experts choisissent immédiatement la solution qui les conduit vers l'objectif, car ils reconnaissent chaque problème et chaque état du problème grâce à leurs expériences antérieures. Les novices, quand à eux, adoptent une stratégie de résolution de type essai-erreur. (Jonassen, 2010) a pour sa part déclaré que les novices ne possèdent pas un

schéma bien développé, ce qui les conduit à utiliser une stratégie de résolution faible telle que l'essai-erreur. (Gomes & Mendes, 2007) affirment que, c'est un problème de compétence en résolution de problèmes, qui est la cause la plus importante des difficultés de nombreux étudiants en cours d'introduction à la programmation.

1.2.3 Manques de compétences de codage

L'apprentissage de la programmation passe par une étape, qui exige de l'apprenant qu'il écrive la solution d'un problème en un programme valide capable de s'exécuter sur un ordinateur. A cet effet, il doit utiliser un langage de programmation. Ce dernier est un ensemble de déclarations et d'énoncés déterministes, qu'il est possible, pour un être humain, de rédiger selon les règles d'une grammaire donnée et destinés à représenter les objets et les commandes pouvant entrer dans la constitution d'un programme. Le langage de programmation recouvre trois aspects fondamentaux (Granet, 2014) :

- *Lexical :* il s'agit des symboles ou caractères qui servent à la rédaction des programmes et les règles de formation des mots du langage.
- *Syntaxique :* est un ensemble de règle grammaticales qui organise les mots en phrases.
- *Sémantique :* permet de définir les règles qui donnent du sens aux phrases.

Les difficultés liées à cette étape ont été analysées par plusieurs auteurs (Lahtinen et al., 2005), (Milne & Rowe, 2002), (Tan et al., 2009), (Qian & James, 2017) et tous convergent sur les points suivants :

- Manque de maîtrise d'un environnement de développement ;
- Mauvaise connaissance des règles syntaxiques et sémantiques d'un langage de programmation.
- Faible capacité à décrypter les messages d'erreurs, ce qui limite la capacité des étudiants à corriger leurs programmes ;
- Manque de compréhension de la façon dont le programme s'exécute : les étudiants ont des difficultés sur les concepts liés à la mémoire (pointeur, liste chaînée…) (Milne & Rowe, 2002), (Lahtinen et al., 2005).

1.2.4 Synthèse

Nous avons vu dans ce chapitre que les difficultés liées à l'enseignement et l'apprentissage de la programmation, en cours d'introduction sont largement documentées dans la littérature. Ces différentes études convergent vers trois grands types de problèmes, résumé dans le *Tableau 1-1*. Nous observons également que le nombre important d'étudiants est une des causes principales des difficultés (difficulté d'assistance, manque d'exercices...). Autre point important, la plupart des enseignements sont focalisés sur l'enseignement d'un langage de programmation et ses détails syntaxiques plutôt que centrés sur la résolution de problèmes.

Dans le cadre de notre travail, nous nous intéressons à améliorer la capacité de résolution de problèmes des apprenants. A cet effet, dans notre proposition, nous tiendrons compte des critères suivants : (i) proposer de nombreux exercices pour pallier au manque de pratique ; (ii) proposer des exercices de même famille pour renforcer le schéma mental des novices et (iii) en cas de difficulté, proposer des rétroactions qui guident les apprenants dans la résolution de l'exercice.

Dans la section suivante, nous présentons les différentes approches d'enseignement, mis en œuvre dans les cours d'introduction afin de nous positionner sur l'approche que nous allons mettre en œuvre.

Problèmes liés à l'enseignement	**Capacité de résolution de problème**	**Problème de codage**
✓ Nombre important d'étudiants dans la salle de classe ; ✓ Manque d'assistance aux étudiants qui ont des difficultés ; ✓ Enseignements focalisés sur la connaissance syntaxique.	✓ Manque de pratique ; ✓ Capacité de compréhension du problème ; ✓ Schéma mental faible (difficulté de créer une analogie entre les différents problèmes) ; ✓ Manque de persévérance dans la résolution de problème.	✓ Utilisation d'un environnement de développement ; ✓ Difficulté à décrypter les messages d'erreurs. ✓ Difficulté à comprendre comment le programme s'exécute.

Tableau 1-1: Synthèse des différents types de problèmes liés à la difficulté de l'enseignement et l'apprentissage de la programmation en cours d'introduction.

1.3 Les différentes approches d'enseignement dans le cours d'introduction

Le rapport final du « Computing Curricula 2001 (CC2001) »(Engel & Roberts, 2001), présenté par le groupe IEEE et ACM, décrit six approches d'enseignement, mises en œuvre dans le cours d'introduction à la programmation. Nous présentons dans cette section ces six approches. L'objectif de cette présentation n'est pas de proposer une nouvelle approche mais de nous positionner.

1.3.1 Approche impérative (Imperative-first)

Cette approche est décrite comme la plus traditionnelle parmi les différentes approches présentées dans ce rapport. Dans cette approche, les apprenants abordent l'introduction de la programmation dans un style impératif. Pour effectuer des exercices et des exemples de programmation, les apprenants utilisent un langage Orienté-Objet. Même si un langage Orienté-Objet est utilisé, cette approche se concentre plus sur les aspects impératifs du langage comme les expressions, les structures des contrôles, les procédures et les fonctions, etc. Les concepts Orientés-Objet sont enseignés dans un cours suivant. L'inconvénient de cette approche est que les apprenants sont moins exposés aux concepts de la programmation Orientée-Objet et que celle-ci est enseignée tardivement.

1.3.2 Approche Orientée-Objet (Objects-first)

Dans cette approche, l'enseignement aborde d'abord les notions d'objet et d'héritage. Après avoir expérimenté ces notions dans un contexte de programmes interactifs simples, les enseignements introduisent les structures de contrôle classiques, mais dans un contexte lié à la conception Orientée-Objet. Les cours suivants couvrent les algorithmes, les structures de données fondamentales et les problèmes d'ingénierie logicielle plus en détails. Le véritable avantage de cette approche est sa popularité dans l'industrie mais également dans le milieu universitaire. L'inconvénient est que le langage utilisé dans les séances de travaux pratiques, tel que C++ ou Java, sont très complexes. Ainsi les enseignants passent par une étape d'introduction de ces langages.

1.3.3 Approche fonctionnelle (Functional-first)

Cette approche à été mise en place au MIT dans les années 80. Elle se caractérise par l'utilisation d'un langage de programmation fonctionnelle simple, tel que « *Scheme* », dès le premier cours. Elle présente les avantages suivants : (i) classe homogène, car les étudiants arrivant à l'université ont peu d'expérience de ce paradigme peu populaire en dehors de l'université ; (ii) les cours se concentrent sur les questions fondamentales, puisque l'apprentissage syntaxique dans ce langage est minimal ; (iii) des concepts importants sont enseignés, en particulier la récursivité, les structures de données et les fonctions en tant qu'objets de première classe. Les inconvénients de cette approche sont : (i) la réaction des étudiants face à l'apprentissage d'un langage non populaire dans le monde professionnel, surtout chez les étudiants novices, qui considèrent que le cours de programmation leur permet d'acquérir une compétence pratique ; (ii) les étudiants doivent penser de façon beaucoup plus abstraite qu'avec les langages de programmation traditionnels et cela trop tôt dans le cours. Même si cette façon de raisonner est utile et doit faire partie du programme d'apprentissage, l'enseigner trop tôt pourrait décourager les étudiants. La programmation Orientée-Objet et la conception sont abordées dans les cours suivants.

1.3.4 Approche Etendue (Breadth-first)

Cette approche soutient que les enseignements d'introduction à l'informatique, focalisés sur la programmation, donnent aux étudiants une vision limitée de la discipline. De ce fait cette approche, en plus des cours traditionnels (programmes, algorithmes et structures de données) aborde d'autres sujets tels que les mathématiques. L'avantage de cette approche est qu'elle donne aux étudiants une vision plus étendue de l'informatique afin de décider de poursuivre d'étudier l'informatique ou pas. Par contre, l'inconvénient est qu'elle surcharge le cours d'introduction.

1.3.5 Approche algorithmique (Algorithms-first)

Dans cette approche, les concepts de base de la programmation sont introduits. Les étudiants apprennent comment raisonner et expliquer les algorithmes qu'ils construisent pour résoudre des problèmes. Pour construire leurs solutions, les étudiants utilisent des notations formelles indépendantes de tout langage de programmation tel que le pseudo-code. Cette approche permet aux étudiants de se concentrer sur la résolution de problèmes sans se soucier

des détails syntaxiques des langages de programmation. Une fois que les étudiants ont une bonne compréhension des fondements algorithmiques, des structures des données et des structures de contrôles, ils peuvent commencer à utiliser un langage de programmation soit à mi-chemin du cours soit au début du cours. L'avantage de cette approche est que les étudiants apprennent à analyser et à réfléchir à la conception de la solution du problème avant de transformer cette solution dans un langage de programmation. Lors de transformation de la solution en un programme, les étudiants pourront plus facilement déboguer leurs erreurs et passer d'un langage de programmation à un autre. Par contre les reproches fait à cette approche sont : (i) le fait que les étudiants ne testent pas leurs solutions sur l'ordinateur est un frein pour leur motivation ; (ii) le processus de compilation et d'exécution du programme, qui est une compétence que les étudiants doivent maîtriser très tôt dans le cours, est vu tardivement; (iii) l'évaluation de l'exactitude de la solution décrite en pseudo-code, pour chaque étudiant demande beaucoup de temps aux enseignants.

Le rapport souligne que les compétences sur le raisonnement algorithmique et la transformation de solution en un programme valide sont essentielles. Ainsi il est utile de combiner l'approche de conception des algorithmes et des séances de travaux pratiques. Enfin, dans les cours suivant les étudiants apprennent la programmation Orientée-Objet.

1.3.6 Approche matérielle (Hardware-first)

Dans cette approche, les étudiants apprennent la base de l'informatique en commençant par l'ordinateur et évoluent vers des concepts plus abstraits. Elle commence par l'enseignement des circuits qui comprend le système de registres et les unités arithmétiques simples. Elle introduit également la machine de von Neumann. Une fois cette base établie, les étudiants apprennent la programmation Orientée-Objet dans un langage de haut niveau. Le rapport souligne que cette approche est intéressante pour un programme de formation génie informatique.

1.4 Synthèse

Dans ces différentes approches, les concepts de base telle que les structures conditionnelles, itératives, les fonctions et les structures de données sont abordées dans un premier temps avant d'aborder l'approche Orientée-Objet dans un deuxièmes temps (hors des cours d'initiation). Dans ces approches, l'utilisation d'un langage de programmation est soulignée sauf dans l'approche algorithmique. Ces approches induisent les étudiants et les

enseignants à se focaliser sur l'enseignement d'un langage et de ses détails syntaxiques. Par contre, l'approche algorithmique met en avant la connaissance stratégique (utilisation des connaissances conceptuelles) ou la capacité de résolution de problèmes, qui est un des éléments clés, souligné dans la section 1.1. Dans cette approche, les étudiants se concentrent plus sur la résolution de problèmes sans se soucier des détails syntaxiques et sémantiques des langages de programmation. Cette approche (algorithms-first) est également mise en œuvre à l'université de Djibouti, dans le cours d'introduction à la programmation.

Bien que la résolution de problème soit mise en avant dans l'approche algorithms-first, le rapport souligne, que les étudiants et les enseignants ont des difficultés à vérifier l'exactitude des solutions. C'est pour remédier à cela, que nous proposons, dans le cadre de cette thèse, une approche d'évaluation et un outil, afin de détecter les erreurs de conception dans les solutions des apprenants. L'objectif de cette évaluation automatisée est de proposer une assistance aux apprenants afin d'améliorer leur capacité de résolution de problèmes ou renforcer leurs connaissances stratégiques.

Dans le chapitre suivant, nous présentons, une étude bibliographique sur l'évaluation et les approches d'évaluation dans les environnements informatiques, avant de présenter les approches d'évaluation dans les environnements informatiques pour l'enseignement et apprentissage de la programmation.

Chapitre 2 : Évaluation

Contenu

2.1 Introduction

Dans le cadre de notre travail, nous cherchons à évaluer si la solution algorithmique des apprenants est correcte ou incorrecte. Dans ce dernier cas, il s'agit d'énumérer et donner une signification à cette erreur dans l'objectif d'accompagner l'apprenant pour résoudre l'exercice.

Dans ce chapitre, nous présentons une revue de littérature sur cette évaluation. Pour cela, nous présentons d'abord l'évaluation en général. Nous abordons ensuite les différentes techniques d'évaluation utilisées dans les EIAH. Pour terminer, nous présentons les différentes approches d'évaluation mises en œuvre dans les environnements d'apprentissage de la programmation.

2.2 Présentation de l'évaluation en général

L'évaluation est une activité importante dans tout processus d'apprentissage. Selon (Hadji, 1990), l'évaluation c'est « *mettre en relation des éléments issus d'un observable appelé référé et un référent pour produire de l'information éclairante sur le référé afin de prendre des décisions* ». L'évaluation ainsi définie met en relation : (i) un *référant*, qui spécifie les critères d'évaluation du résultat de l'activité ainsi que les compétences correspondantes ; et un (ii) *référé*, il s'agit de l'observable, qui sera évalué en fonction des critères définis précédemment (Durand, 2006). Dans la même optique (De Ketele, 2010) souligne que « *tous les auteurs actuels s'accordent sur le fait que le processus évaluatif consiste à confronter un référent à un référé* » mais « *ils divergent sur la finalité, pour les uns, il s'agit de « porter un jugement sur » ; pour les autres, de « fonder une prise de décision* ». Lorsqu'il s'agit de fonder une prise de décision, il s'agit soit (De Ketele, 2010) :

- De préparer une nouvelle action (*fonction d'orientation*) ;
- D'améliorer une action en cours (*évaluation formative ou régulation*) ;
- De certifier le résultat d'une action (*évaluation certificative ou sommative*).

En d'autres mots l'évaluation permet de mesurer l'atteinte des objectifs pour chaque apprenant. Elle permet également d'assurer une communication entre l'enseignant et l'apprenant. Elle renseigne l'apprenant sur l'état de ses connaissances et compétences et permet à l'enseignant de réguler l'apprentissage (Labat, 2002).

Nous présentons dans ce qui suit les différents types d'évaluation (formative et sommative).

2.2.1 Les différents types d'évaluation

Nous avons vu précédemment que les principales fonctions de l'évaluation sont : l'orientation, formative et sommative. Nous décrivons, dans ce qui suit les deux derniers types d'évaluation (évaluation formative et sommative) qui sont largement mises en œuvre dans les environnements d'apprentissage.

2.2.1.1 Évaluation formative

L'évaluation formative est un processus, qui permet de prendre des décisions pour améliorer la séquence d'apprentissage en cours de réalisation. Ce point de vue est appuyé par (Scallon, 2000), qui souligne que l'évaluation formative est un « *processus d'évaluation continue ayant pour objectif d'assurer la progression des individus engagés dans une démarche d'apprentissage ou de formation, selon deux voies possibles : soit par des modifications de la situation ou du contexte pédagogique, soit en offrant à chaque individu l'aide dont il a besoin pour progresser* ». Dans la même optique (Harlen, 2007) qualifie ce type d'évaluation comme un processus de recueil d'un ensemble d'informations dans l'activité d'apprentissage en cours de réalisation afin de le confronter à un ensemble de critères, que l'auteur qualifie comme objectifs pédagogiques. Ces critères permettent de situer la progression de l'apprenant par rapport à ces objectifs (*figure 2-1*). Selon (Harlen, 2007), les éléments clés de ce type d'évaluation sont :

- Des rétroactions individualisées pour chaque apprenant sur la manière d'améliorer ou de progresser pour un apprentissage donné ;
- Les apprenants s'expriment et communiquent leurs connaissances et compétences à travers des questions ouvertes ;
- Les enseignants utilisent les résultats de l'évaluation afin d'adapter ou d'ajuster leurs enseignements ;
- Elle instaure une communication entre l'enseignant et les apprenants.

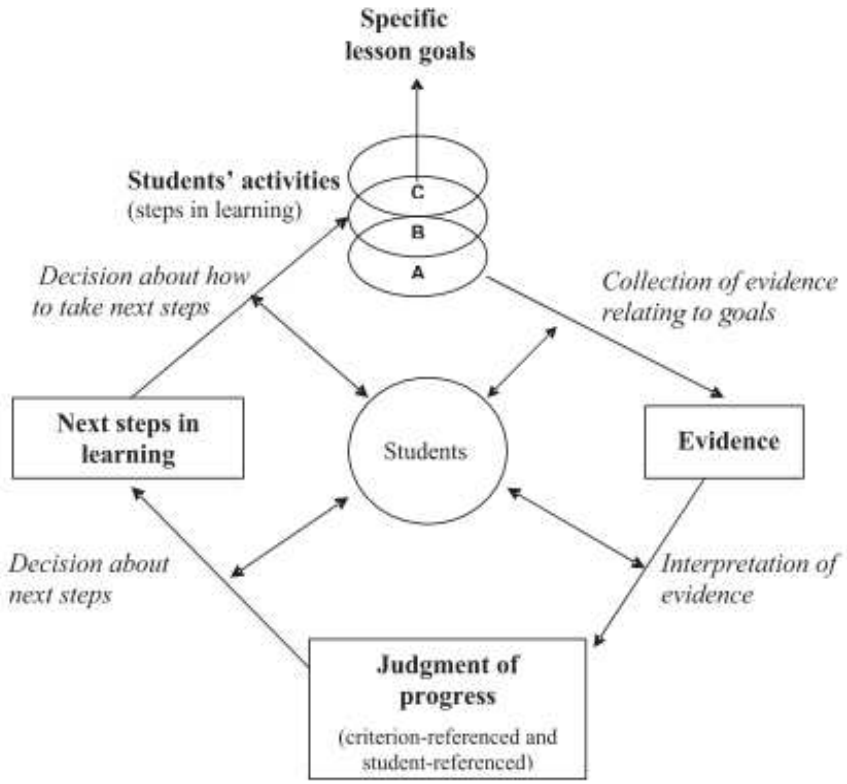


Figure 2-1: Evaluation des apprentissages - Evaluation Formative (Harlen, 2007)

2.2.1.2 Évaluation sommative

L'évaluation sommative a pour objectif principal de faire état de ce qui a été appris. Selon (Hadji, 1990) l'évaluation sommative est une « *évaluation par laquelle on fait un inventaire des compétences acquises, ou un bilan, après une séquence de formation d'une durée plus ou moins longue* ». En d'autres mots cette évaluation accorde une importance particulière à la performance et contrôle les connaissances de l'apprenant dans l'objectif de lui attribuer une note. Selon (Harlen, 2007), les différentes caractéristiques de cette évaluation sont :

- Elle est basée sur le jugement de l'enseignant ;
- Elle se déroule à un moment bien précis (après l'achèvement d'un cours, d'un ensemble d'objectifs…) pour donner un état sur la connaissance de l'apprenant ;
- Elle fournit des résultats exprimés en termes de note ;
- Elle nécessite une échelle de mesure ou des critères communs afin de juger tous les apprenants avec ces critères.

(Harlen, 2007) résume ce type d'évaluation sur la *figure 2-2*.

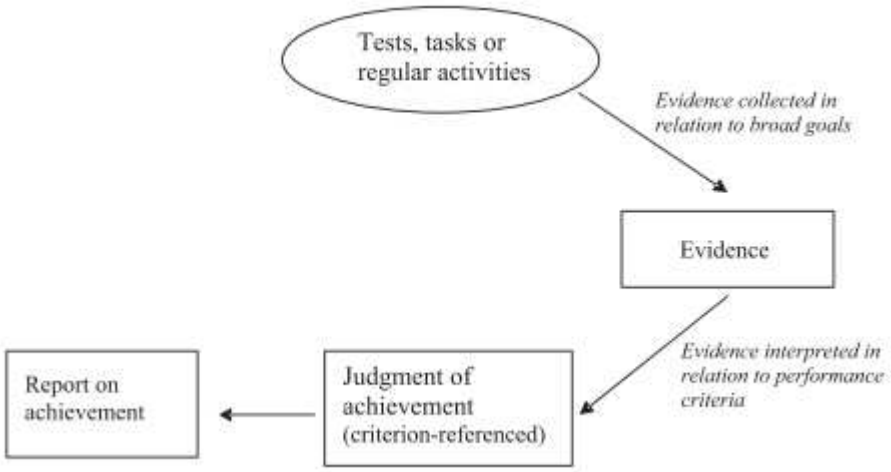


Figure 2-2: Evaluation d'apprentissage – Evaluation Sommative (Harlen, 2007)

2.2.1.3 Synthèse

Dans une évaluation, les auteurs s'accordent sur le fait qu'il s'agit de confronter la production de l'apprenant (*référé*) à des critères bien définis (*référent*), mais divergent sur la finalité de l'évaluation : (i) pour certains il s'agit de certifier ou d'établir un bilan communément appelé évaluation sommative. Celle-ci a généralement lieu après une séquence d'apprentissage ; (ii) pour d'autres elle sert de support à une prise de décision (régulation de l'apprentissage), communément appelé évaluation formative. Celle-ci a généralement lieu au cours de l'apprentissage.

Dans l'évaluation formative, l'assistance ou l'accompagnement de l'apprenant dans ses difficultés dans une activité en cours de réalisation est mise en avant. Dans le cadre de notre travail, nous nous situons donc dans ce type d'évaluation, ou nous cherchons à évaluer la solution algorithmique de l'apprenant afin de définir si sa solution est correcte ou incorrecte. Dans ce dernier cas il s'agit de détecter ses erreurs afin de l'assister par des rétroactions.

Nous présentons dans la section suivante les différentes techniques mises en œuvre dans les EIAH pour confronter le référé au référent pour détecter les erreurs de l'apprenant. Ensuite, nous présentons les différentes approches d'évaluation mises en œuvre dans les environnements d'apprentissage de la programmation.

2.3 Les techniques d'évaluation dans les EIAH

Le processus d'évaluation peut-être lourd à mettre en œuvre par l'enseignant pour un grand nombre d'apprenants. Les environnements informatiques permettent de faciliter le recueil des informations sur l'état des connaissances et compétences de l'apprenant. Selon (Labat, 2002), l'évaluation dans les EIAH sert à accompagner l'apprenant dans son apprentissage. Elle permet à l'enseignant et à l'apprenant lui-même de percevoir ses forces et ses difficultés. Pour (Juwah, 2003), l'évaluation doit :

- Être motivante pour l'apprenant ;
- Encourager une activité d'apprentissage soutenue ;
- Contribuer à la progression de l'apprenant ;
- Être faible en coût humain et facilement maintenable.

Nous présentons dans cette section, quelques techniques d'évaluation, généralement mises en œuvre dans les EIAH pour diagnostiquer les erreurs de l'apprenant.

2.3.1 Évaluation basée sur les cartes conceptuelles

Les cartes conceptuelles, initialement proposées par (Novak & Gowin, 1984), sont un moyen de représenter sous forme des graphes les concepts d'un domaine de connaissance et les liens existants entre ces concepts (Delorme, Delestre, & Pécuchet, 2004). Selon (Tribollet, Françoise, & Laurence, 2000) les cartes conceptuelles sont utilisées dans différents cas :

- La carte peut être construite par l'enseignant et lui servir ensuite à organiser son cours ;
- Elle peut être construite par l'enseignant et présentée aux apprenants comme une représentation non linéaire du domaine ;
- Elle peut être construite par un groupe d'apprenants, les divergences de point de vue entre participants pouvant générer des conflits sociocognitifs ;
- Elle peut être construite par un apprenant seul, cette carte pouvant alors être analysée par l'enseignant qui en déduira les conceptions et incompréhensions de l'apprenant.

Ce dernier cas correspond à la fonction d'évaluation, i.e., comparer la carte construite par l'apprenant à une carte de référence (carte construite par l'expert). L'outil DIOGen est un

exemple de cette technique d'évaluation. (Delorme et al., 2004) utilisent ce prototype pour évaluer l'acquisition de connaissances concernant l'utilisation de Java RMI. Pour cela, la carte est représentée sous forme d'une liste de triplets (sujet, lien, objet) (*figure 2-3*). Pour fournir une évaluation, le système compare la carte conceptuelle de l'apprenant à une carte de référence. Après analyse, trois résultats sont possibles :

- Cas 1 : un triplet se trouve sur les deux cartes.
- Cas 2 : un triplet est présent sur la carte de référence mais absent sur celle de l'apprenant : dans ce cas, soit l'apprenant n'a pas jugé nécessaire de représenter ce lien, soit il ne maîtrise pas le concept correspondant ;
- Cas 3 : un triplet de la carte de l'apprenant est absent sur la carte de référence : dans ce cas, soit l'apprenant a mal compris quelque chose, soit le lien proposé n'est pas faux, mais l'expert (enseignant) n'a pas jugé ce triplet nécessaire.

Chaque différence entre les deux cartes peut donc être interprétée comme une erreur de l'apprenant. Dans les cas 2 et 3, un questionnaire supplémentaire est utilisé pour lever les ambiguïtés.

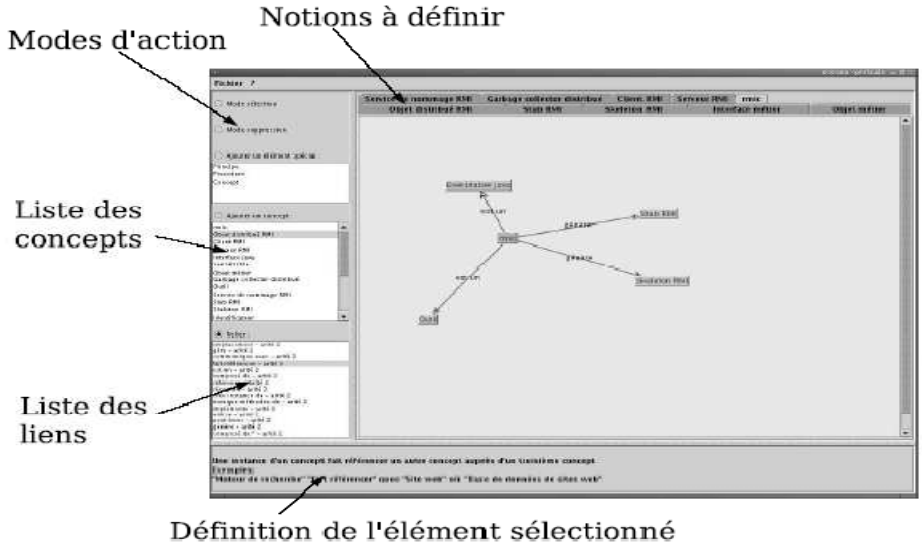


Figure 2-3: Réalisation d'une carte par l'apprenant dans DIOGen (Delorme et al., 2004)

Un autre exemple est l'outil COMPASS (COncept MaP ASSessment tool), qui met en œuvre une évaluation formative (Gouli, Gogoulou, Papanikolaou, & Grigoriadou, 2004). Il

comprend 5 modules (contrôle de l'interaction, diagnostic, génération de rétroaction, module de présentation et stockage des données). Le module stockage de données est composé d'un modèle de connaissance et un modèle apprenant. Dans le cadre de ce chapitre, nous nous intéressons au module de diagnostic. Ce module permet d'effectuer (i) un diagnostic sur les erreurs de l'apprenant. Pour ce faire l'outil effectue une comparaison entre la carte conceptuelle de celui-ci et celle d'un expert afin de détecter les concepts ou les liens manquants ; (ii) un diagnostic qualitatif (concepts non maîtrisés, idées fausses) sur la connaissance de l'apprenant. Celui-ci est basé sur la catégorisation des erreurs détectées selon des critères établis par l'enseignant ; (iii) une estimation quantitative du niveau des connaissances de l'apprenant sur les concepts liés à l'activité d'évaluation. Pour effectuer cette évaluation, COMPASS, propose à l'apprenant, soit de compléter une carte conceptuelle existante (concept ou lien manquant), soit de construire une nouvelle carte conceptuelle (*figure 2-4*).

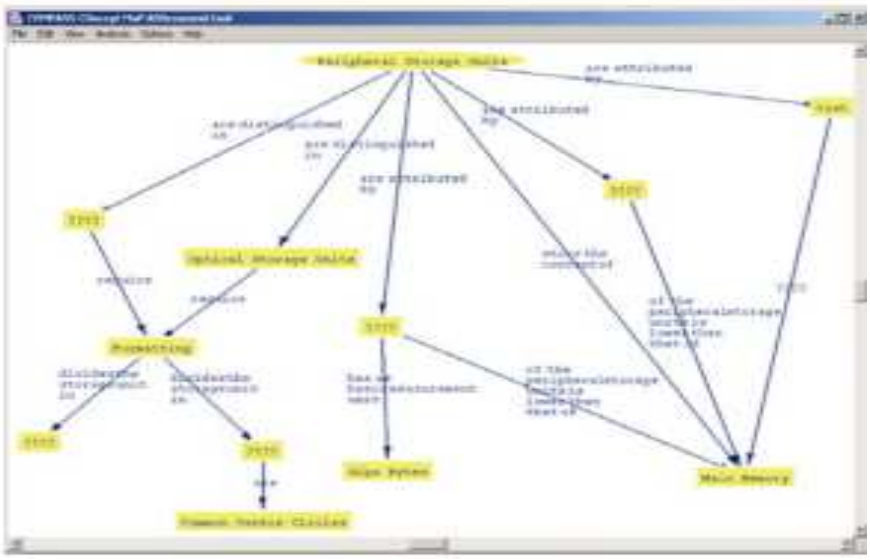


Figure 2-4: Réalisation / Complétion d'un Carte Conceptuelle dans COMPASS (Gouli et al., 2004)

2.3.2 Évaluation basée sur un plan de solution

Selon (Py, 1998), le plan de solution est «*comme une séquence ordonnée d'actions qui vise à satisfaire un but donné*». Dans cette technique d'évaluation, les solutions d'un problème donné sont modélisées sous forme de plan. Le système cherche à découvrir le plan mis en œuvre par l'apprenant suite à l'observation de ses actions. Cette technique est utilisée dans les domaines d'apprentissage qui demandent à ce que l'apprenant effectue un choix de

solution adaptée à la résolution d'un problème. Chaque problème peut avoir plusieurs plans de solution (Mufti-Alchawafa, 2008). De ce fait, la plupart des systèmes utilisant cette technique d'évaluation, utilisent comme base, une bibliothèque de plans, qui décrit l'ensemble des plans à reconnaître (Geib & Goldman, 2009).

Un exemple de cette technique d'évaluation est le système *VirtualLabs*, qui est un système de simulation d'un laboratoire pour l'enseignement de la chimie au niveau secondaire (Amir & Ya'akov, 2013). Il permet aux étudiants de réaliser une expérience sur des concepts de chimie. Pour ce faire l'apprenant réalise son expérience en spécifiant les actions à mener. Le système possède, pour un problème donné, plusieurs plans de solutions qui représentent les différentes combinaisons d'actions possibles et les différentes contraintes qu'un apprenant doit respecter pour résoudre un problème de dilution. L'outil, grâce à l'observation des actions de l'apprenant (*figure 2-5*), reconnaît le plan suivi et vérifie si la solution proposée correspond à un plan connu.

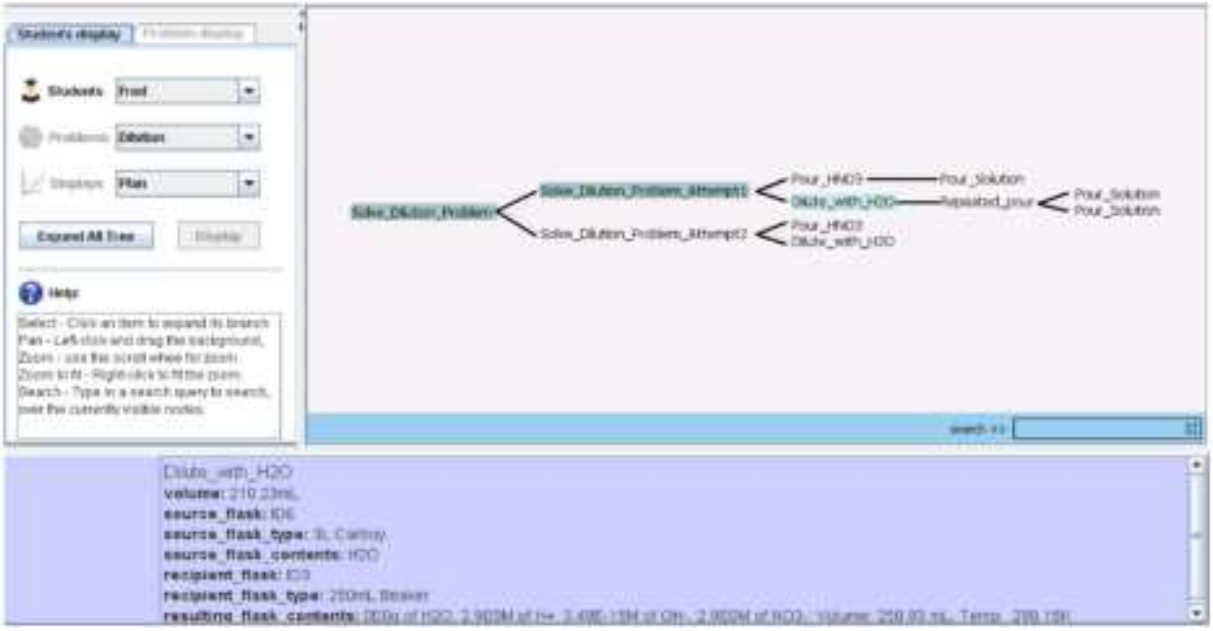
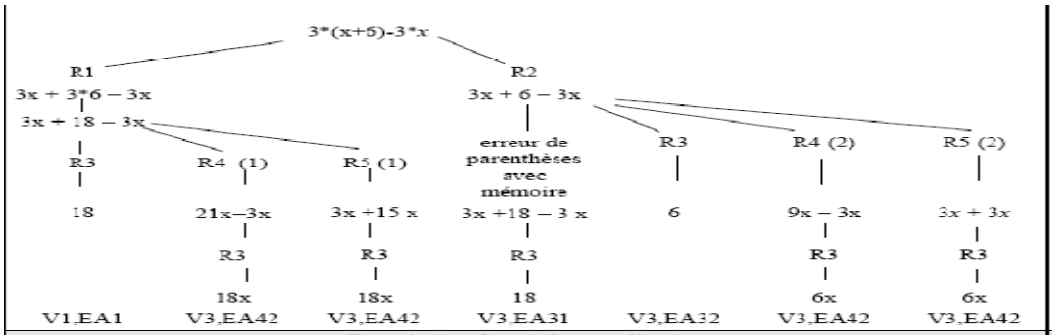


Figure 2-5: Plan de solution réalisé par l'apprenant dans l'outil VirtualLabs (Amir & Ya'akov, 2013)

Un autre exemple est l'outil *Pépinière*, un système de diagnostic automatique des réponses des élèves sur leur raisonnement algébrique (Prévit, 2008). Pour ce faire, ce système génère, dans un premier temps, automatiquement une grille de codage qui est sa bibliothèque de plans de solutions correctes et incorrectes. Celle-ci est représentée sous forme d'un arbre de solutions, où chaque branche de l'arbre est associée à un code. Chacun des codes est relié à une rétroaction donnée (*figure 2-6*). L'outil compare la solution de l'apprenant aux plans de

solutions, afin d'associer un code à la solution de l'apprenant. Selon le résultat du diagnostic, l'apprenant reçoit la rétroaction reliée au code.



Description des règles appliquées				
Règles correctes	R1 : A(B+C) → AB+AC		R3 : AC+BC→ (A+B)C	
Règles erronées	R2 : A(B+C) → AB+C	appliquée à :	3*(x+6)	
	R4 : AB+C→ B*(A+C)	appliquée à :	(1) 3x + 18	(2) 3x + 6
	R5 : A+BC→ (A+B)*C	appliquée à :	(1) 18 − 3x	(2) 6 − 3x
Description des codes				
Validité	V1 : traitement correct		V3 : traitement incorrect	
Utilisation des règles d'Écriture et de réécriture Algébrique	EA1 : Utilisation correcte des règles de transformation			
	EA31 : Utilisation inadaptée des parenthèses qui conduit à un résultat correct Remarque : certains élèves n'indiquent pas de parenthèses mais en conservent le sens. Cette erreur est codée EA31 (erreur de parenthèse avec mémoire) alors qu'une erreur de parenthèse sans mémoire de l'énoncé est codée EA32.			
	EA32 : Utilisation inadaptée des parenthèses qui conduit à un résultat incorrect			
	EA42 : Les règles de transformation utilisées assemblent les termes			

Figure 2-6: Bibliothèque de plan pour un problème donné (outil Pepinière) (Prévit, 2008)

2.3.3 QCM (Questionnaires à Choix Multiples)

Selon (Delorme et al., 2004) et (Chatzopoulou & Economides, 2010), cette technique d'évaluation est largement utilisée dans les EIAH, du fait de sa rapidité de mise en œuvre pour la vérification des réponses. Elle permet d'évaluer ou de mesurer la connaissance de l'apprenant, sur un domaine donné. Pour ce faire, les apprenants doivent répondre aux questions en choisissant une ou plusieurs réponse(s) à partir d'une liste donnée ou compléter un texte à trou. La technique d'évaluation consiste à comparer la/les réponse(s) choisie(s) par l'apprenant aux réponses attendues. (Nicol, 2007) souligne que « *les questionnaires à choix multiple exigent la sélection d'une réponse correcte à partir d'un ensemble de réponses alternatives, c'est-à-dire la reconnaissance de la réponse plutôt que la construction de la réponse* ». C'est pour cette raison que de nombreux chercheurs soutiennent que le QCM ne teste pas les processus cognitif de haut niveau (Nicol, 2007). Dans la même optique, (Durand, 2006), souligne que l'évaluation par le QCM, est généralement utilisée pour une évaluation sommative, afin de donner un jugement final sur la performance de l'apprenant. Un exemple connu est le test de TOEFL, largement utilisé pour évaluer le niveau d'anglais.

Le QCM est également utilisé pour une évaluation formative. Dans ce cas l'objectif est de détecter les difficultés de l'apprenant afin d'y remédier. Dans ce cas les outils sélectionnent les questions en fonction des réponses de l'apprenant. Par exemple une question facile en cas de mauvaise réponse, ou une question plus difficile en cas de réponse correcte. Cette technique est mise en œuvre dans CAT (Computer Adaptive Test), illustré *figure 2-7*. Pour minimiser les réponses aléatoires et encourager ainsi la réflexion et la confiance en soi, certains auteurs proposent une extension des QCM classiques, en ajoutant aux réponses une mesure du degré de confiance des apprenants sur une échelle allant de 1 (faible confiance) à 3 (haute confiance) (Gardner-Medwin, 2006). L'objectif est de renforcer, la validité et la fiabilité de l'évaluation.

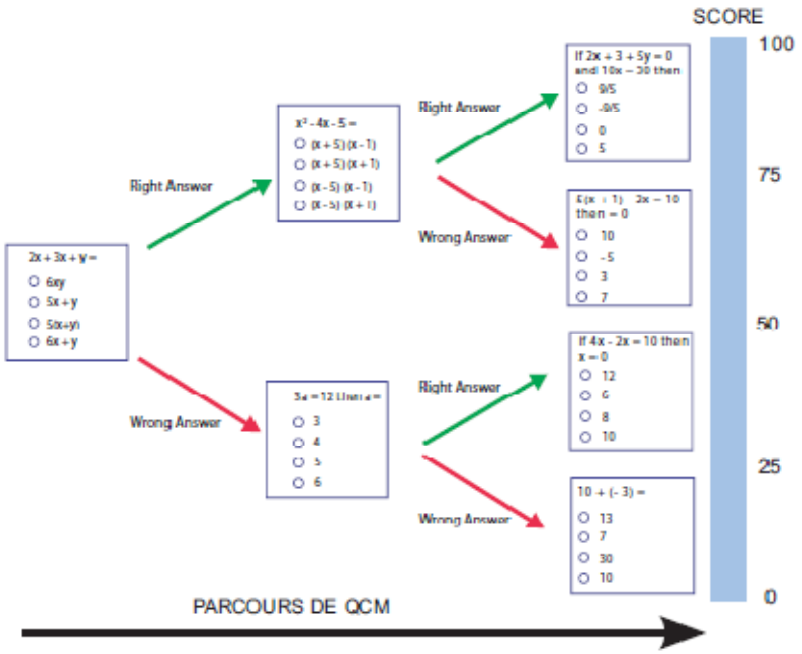


Figure 2-7: CAT : Principe (Durand, 2006)

2.3.4 Synthèse

Nous observons, sur les techniques d'évaluation présentées dans cette section, que le QCM et la carte conceptuelle sont bénéfiques dans le cadre d'une évaluation des connaissances déclaratives. L'approche basée sur un plan de solution, est utile pour évaluer une connaissance procédurale. Toutefois cette technique impose de prévoir, tous les plans de solutions possibles y compris les plans de solutions comportant des erreurs pour pouvoir établir un diagnostic.

Notre cadre d'étude est un environnement informatique pour l'apprentissage de l'introduction à la programmation. L'environnement sera basé sur une approche considérant que l'apprenant construit son savoir-faire en interagissant avec le système. Pour ce faire l'apprenant doit donner une solution algorithmique aux différents exercices proposés. Chaque exercice peut avoir plusieurs solutions. Dans ce cas, nous nous situons dans une évaluation à réponse ouverte. L'apprenant exprime sa créativité en mobilisant certaines connaissances apprises au préalable, ce qui rend l'analyse de sa solution complexe. Dans ce cadre là, la technique d'évaluation basée sur un plan de solution convient. Ainsi, pour diagnostiquer les erreurs des apprenants, nous confronterons la solution de l'apprenant à des solutions modèles (représentant les différentes solutions possibles). Contrairement à la technique d'évaluation basée sur un plan de solution nous chercherons à diagnostiquer les erreurs sans nous appuyer sur des solutions modèles comportant des erreurs.

Nous présentons, dans ce qui suit, les différentes techniques utilisées par les outils d'assistance de l'enseignement/apprentissage de la programmation, pour évaluer la production de l'apprenant. L'objectif de cette étude est de savoir comment on détecte les erreurs dans la solution apprenant.

2.4 L'évaluation dans les environnements d'enseignement/ apprentissage de la programmation

L'utilisation de l'évaluation dans les EIAH, permet de donner un résultat rapide après confrontation des données recueillies sur l'apprenant et du résultat attendu. Dans le cadre des outils assistant l'enseignement/apprentissage de la programmation, l'évaluation de la production (code) des apprenants repose sur la collecte ou l'extraction dans les programmes de ces apprenants de certaines informations (référé). Celles-ci seront comparées par la suite à certaines exigences ou à des solutions modèles (référant). Pour ce faire, les outils utilisent une stratégie basée sur l'analyse de programme de l'apprenant, en mettant en œuvre deux approches (Ala-Mutka, 2005), (Ihantola, Ahoniemi, Karavirta, & Seppälä, 2010), (Fonte, Da Cruz, Gançarski, & Henriques, 2013), (Arifi, Abdellah, Zahi, & Benabbou, 2015) :

- *Une approche dynamique* : cette approche est basée sur l'exécution du programme. Elle permet, à partir de cette exécution, de vérifier l'exactitude du programme d'un apprenant. Pour ce faire, elle effectue une comparaison, sur un ensemble des données de test, des résultats de sortie ou de la valeur de retour du programme de l'apprenant par rapport à des résultats attendus.

- *Une approche statique* : cette approche est basée sur l'analyse du programme sans l'exécuter. Elle permet de révéler la qualité du code, en analysant le code de l'apprenant, pour vérifier s'il se conforme à certaines règles. Pour ce faire, cette approche utilise plusieurs techniques telles que l'utilisation des métriques logicielles pour mesurer la complexité du programme (i.e., nombre de lignes de code, nombre de variables…), des techniques pour évaluer le style du programme, détecter les erreurs syntaxiques ou sémantiques, etc.

Nous présentons dans ce qui suit, en détail ces deux approches.

2.4.1 Approche dynamique

Cette approche repose sur l'exécution du programme de l'apprenant afin de vérifier la fonctionnalité (l'exactitude) ou la complexité. La vérification de la fonctionnalité ou l'exactitude du programme est la plus utilisée (Romli, Sulaiman, & Zamli, 2013). Pour vérifier l'exactitude du programme de l'apprenant, on teste celui-ci, à travers un ensemble de données de test. Par la suite, le résultat du programme de l'apprenant est comparé soit à un résultat attendu donné au préalable, soit au résultat obtenu après exécution d'une solution modèle avec les mêmes données de test.

Dans cette approche, les données de test sont une préoccupation fondamentale, car un programme comporte un grand nombre de chemins d'exécution qu'il faut tester. Dans la plupart des cas, les données de test sont fournies par l'enseignant au préalable, mais il existe également des outils qui utilisent des données de test générées automatiquement. Cette approche est mise en œuvre de différentes manières :

- L'enseignant préétablit les données de test. Le système compare les résultats de la solution modèle et ceux de l'apprenant.
- L'enseignant préétablit les données de test et le résultat attendu. Le système compare le résultat de la solution apprenant au résultat attendu.
- Les étudiants conçoivent eux-mêmes leurs jeux de test, afin de démontrer l'exactitude de leur code. Ainsi, ils soumettent leurs codes et leurs jeux de test. Le système d'évaluation vérifie avec ces jeux de test, s'ils couvrent tous les chemins d'exécution dans la solution modèle avant de lancer le test sur la solution apprenant.

Le système ASSYST est un exemple de ce type d'évaluation (Jackson & Usher, 1997). Ce système évalue la fonctionnalité, l'efficacité, le style de programmation ainsi que la complexité du programme soumis par l'apprenant. Pour vérifier la fonctionnalité, l'enseignant définit au préalable une spécification de ce que le programme correct doit produire. Le système exécute le programme de l'apprenant avec des données fournies par l'apprenant lui-même et par l'enseignant. Il analyse par la suite les résultats, en vérifiant s'ils sont conformes à la spécification.

Un autre exemple est l'outil TRY, qui est un utilitaire qui peut être intégré à n'importe quel système qui veut faire des tests automatiques sur la production de l'apprenant (Reek, 1989). TRY supporte plusieurs langages de programmation. Pour ce faire, cet outil permet à l'enseignant de concevoir les données de test qui seront stockées dans un fichier et non visibles par l'étudiant. TRY part du constat que si l'étudiant, prend connaissance des données de test, il va adapter son programme de tel sorte qu'il marche avec ces données et ne va pas produire un programme plus général. Une fois que l'étudiant soumet son programme, l'outil l'exécute avec les données fournies par l'enseignant et compare le résultat avec un fichier de résultat prédéfini. Un fichier journal (*figure 2-8*) enregistre le résultat de chaque test, que l'enseignant peut consulter par la suite pour voir l'état d'avancement de chaque apprenant. Si le test a échoué, l'apprenant peut modifier son programme et réessayer le test.

Le système BOSS, est composé d'un ensemble de programmes (Luck & Joy, 1999). Pour évaluer la production de l'apprenant, l'enseignant préétablit, un ensemble de données de test avec le résultat attendu pour chaque exercice, ensuite le système compare le résultat donné par l'exécution du programme de l'apprenant aux résultats attendus avec les données de test fournies.

Ces différents systèmes ont pour point commun l'utilisation des données de test pour comparer les résultats attendus à ceux de l'apprenant. Cela permet de valider (correct ou non) la solution de l'apprenant mais ne permet pas de diagnostiquer leurs erreurs (Bouhineau, 2013).

```
qcc1234   !    88/06/02   15:08:38   Log file created for Quai Chang Cain
qcc1234   p1   88/06/02   15:11:49   Failed test 1.
qcc1234   p2   88/06/04   12:02:02   Failed build.
qcc1234   p1   88/06/04   19:14:24   Failed test 4.
qcc1234   p1   88/06/07   10:21:10   Completed.
qcc1234   p2   88/06/07   17:22:32   Failed test 4.
qcc1234   p2   88/06/07   18:03:47   Failed test 7: Floating point exception
qcc1234   p2   88/06/07   20:42:41   Failed test 7: back pointer of root node not updated.
```

Figure 2-8: Exemple de fichier journal (Reek, 1989)

2.4.2 Approche statique

Cette approche est basée sur l'analyse du programme sans l'exécuter. Elle permet de vérifier/estimer la qualité du code en recueillant certaines informations sur le code de l'apprenant, pour vérifier s'il est conforme à certaines exigences (Ala-Mutka, 2005). Dans cette approche, différentes analyses du programme sont faites, telles que (Ala-Mutka, 2005), (Rahman & Nordin, 2007) :

- *Analyse du style du programme* : dans cette analyse, l'objectif est de vérifier la lisibilité du programme de l'apprenant afin de mesurer la qualité du code. Les critères de lisibilité portent sur la structure, la présence de commentaires, le fait d'avoir des noms de variables significatifs. Dans cette catégorie, on trouve Style++ qui permet de vérifier le style d'un code écrit en langage C (Ala-Mutka, Uimonen, & Järvinen, 2004). Un autre exemple est l'outil PASS [Wang Fu Lee], qui lui aussi vérifie certains facteurs de qualité pour les langages Ada, C et Java (F. L. Wang & Wong, 2008).
- *Utilisation de métriques logicielles* : Dans cette analyse, on cherche à évaluer la qualité des solutions fournies par l'apprenant. Pour ce faire, cette analyse se base sur des métriques de complexité du logiciel telle que la complexité cyclomatique et sur des guides de bonnes pratiques de programmation. Dans cette catégorie, on trouve Assyst (Jackson & Usher, 1997), BOSS (Luck & Joy, 1999), Style++ (Ala-Mutka et al., 2004) et ELP (Truong, Roe, & Bancroft, 2004) qui permettent de mesurer la qualité du code des apprenants.
- *Analyse des erreurs syntaxiques et sémantiques* : Cette analyse est la plus répandue. Elle permet de rechercher dans la solution de l'apprenant les erreurs syntaxiques et sémantiques (i.e., division par 0, boucle non fermée, etc.) pour un langage de programmation donné. L'objectif de cette analyse est d'accompagner les apprenants, en leur fournissant des rétroactions plus significatives par rapport aux commentaires du compilateur. Des exemples de tels outils sont Gauntled (Flowers, Carver, & Jackson, 2004) et CAP (Schorsch, 1995).
- *Analyse de similarité structurelle* : dans cette analyse, l'objectif est de comparer et de déterminer la similarité entre la structure du programme de l'apprenant et celle d'un expert (solution modèle) afin de détecter les erreurs des apprenants.

Dans le cadre de notre travail, comme souligné dans la section précédente (section 2.3.4), nous cherchons à diagnostiquer les erreurs des apprenants (évaluer si l'apprenant a utilisé les concepts de programmation appropriés et s'il les a utilisés au bon endroit lors de la conception de sa solution). Ce diagnostic se base sur la comparaison de la solution apprenant à une solution modèle. Notre travail se place donc dans le cadre de ce dernier type d'analyse (*analyse de similarité structurelle*). Nous abordons maintenant, comment cette analyse permet de détecter les erreurs des apprenants.

L'analyse de similarité structurelle est mise en œuvre soit en comparant une à une les instructions des deux solutions (apprenant et modèles), soit en passant par une étape de standardisation des solutions avant la comparaison (apprenant et modèle). Nous présentons dans ce qui suit ces deux modes de comparaison.

a) Comparaison entre les instructions d'une solution apprenant et des solutions modèles.

Cette technique repose sur la comparaison une à une des instructions. Dans le cas où plusieurs ordres sont possibles pour certaines instructions, l'enseignant doit écrire toutes les alternatives. De plus, l'enseignant doit également prévoir les modèles de solutions erronées pour identifier les erreurs de l'apprenant.

Un exemple est l'outil Algo+ (Bey & Bensebaa, 2011). Cet outil permet d'évaluer la solution algorithmique, décrite en pseudo-code. Pour ce faire, l'expert décrit, l'ensemble des solutions (correctes et incorrectes) possibles pour un exercice donné. Chacune des solutions comprend un ensemble d'opérations (*figure 2-9*) et est reliée à une note. Pour évaluer la solution de l'apprenant, l'outil la compare à différentes solutions modèles. Par cette comparaison, l'outil calcule une mesure de similarité qui permettra de choisir la note à attribuer à la solution apprenant.

Un autre exemple est l'outil WAGS, une application Web, pour évaluer automatiquement le code des apprenants écrit en langage Java, C ou Visual Basic (Zamin et al., 2006). Pour ce faire il compare la solution de l'apprenant aux différentes solutions données par l'enseignant (*figure 2-10*).

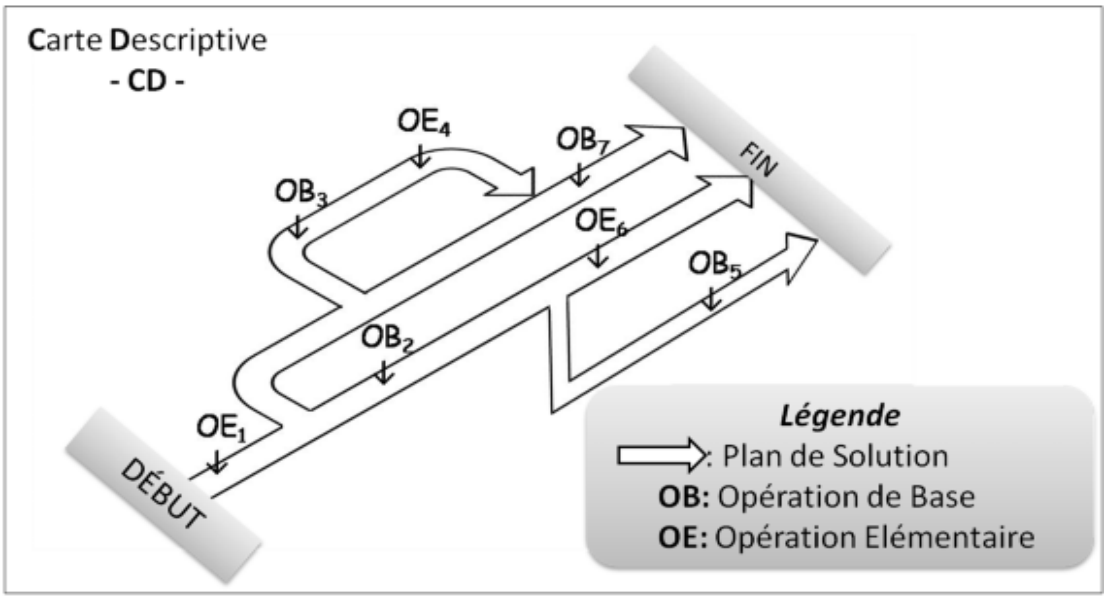


Figure 2-9: Solutions modèles en plan de solution (Bey & Bensebaa, 2011)

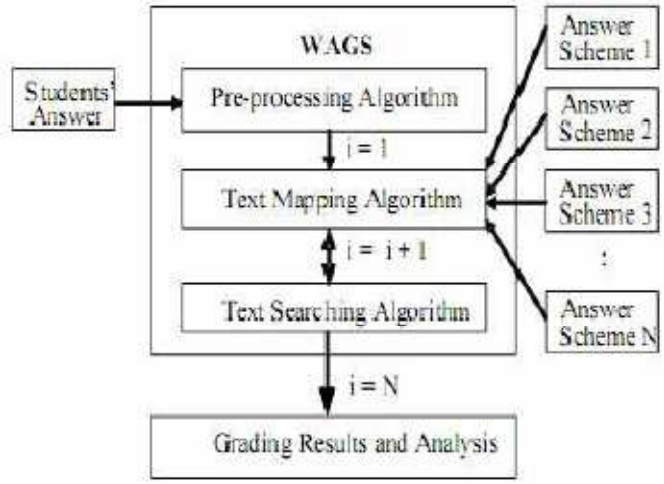


Figure 2-10: Processus d'évaluation dans WAGS (Zamin et al., 2006)

b) Comparaison entre la solution apprenant et des solutions modèles en passant par une étape de standardisation

Cette technique consiste à comparer et à analyser la similarité structurelle entre la solution apprenant et une solution modèle. Pour ce faire, certains outils mettant en œuvre cette technique, passent par une étape de transformation et de standardisation des solutions (solution apprenant et modèle) avant d'effectuer la comparaison.

Un exemple est l'outil ELP, un environnement Web, utilisé pour l'initiation à la programmation en Java (Truong et al., 2004). Pour cela, l'outil, fournit aux apprenants une

solution partielle pour un problème donné, que l'apprenant doit compléter (*figure 2-11*). Cela oblige l'apprenant à comprendre d'abord le code, avant de le compléter. Pour l'analyse, l'outil passe par un processus de transformation et de standardisation (*figure 2-12*). Cette transformation permet d'obtenir un AST (Abstract Syntax Tree) au format XML (*figure 2-13*). L'outil utilise un parseur Java pour construire cet AST. Après cette transformation, une analyse de la similarité structurelle est appliquée entre cette solution standardisée et une solution modèle, elle-même standardisée.

Un autre exemple est l'outil développé par (T. Wang, Su, Wang, & Ma, 2007). Celui-ci réalise une analyse de similarité structurelle entre la solution de l'apprenant et des solutions modèles, afin d'assister les apprenants dans l'apprentissage de la programmation en langage C. Pour cela, leur outil passe également par un processus de transformation et standardisation qui produit un graphe. Le graphe ainsi obtenu est un AST enrichi par des arcs. Ces arcs permettent de définir les dépendances entre les variables, les appels de fonctions, les données, les structures de contrôle, etc. (*figure 2-14*). Cet outil est plus orienté vers une évaluation sommative, en affectant une note à la solution apprenant. Pour cela, il calcule la similarité entre les graphes, en comparant les graphes de dépendances de la solution de l'apprenant et les graphes de dépendances des différentes solutions modèles.

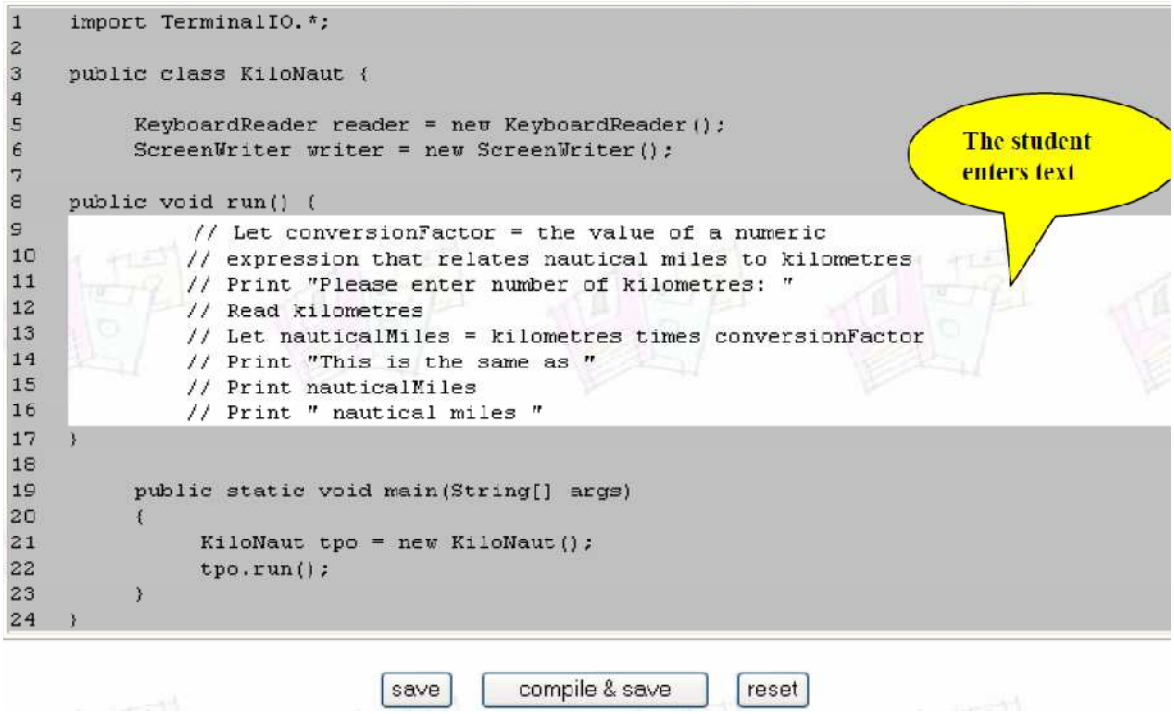


Figure 2-11: Exemple d'un exercice dans ELP (Truong et al., 2004)

Un autre exemple d'outil, mettant en œuvre une analyse de similarité structurelle, est celui développé par (Rahman, Nordin, & Che, 2008). Cet outil s'intéresse au langage de programmation C. Il passe également par un processus de transformation et standardisation, qui transforme la solution (code) de l'apprenant ainsi que les solutions modèles en pseudo-code, avant d'effectuer une comparaison. Cet outil est orienté vers une évaluation sommative, qui consiste à affecter une note à la solution apprenant. Pour cela, il recherche parmi les solutions modèles la solution la plus proche, en calculant le pourcentage de similarité.

Ces différents outils mettant en œuvre une analyse de similarité structurelle ont pour point commun la comparaison de solution apprenant à des solutions modèles. L'objectif de cette comparaison est de retrouver la solution modèle similaire à la solution apprenant afin de lui affecter la note rattachée à cette solution modèle. Ces outils ne proposent pas un moyen de diagnostiquer les erreurs de l'apprenant sauf pour l'outil Algo+ qui, pour détecter les erreurs des apprenants, s'appuie sur des solutions modèles avec erreurs.

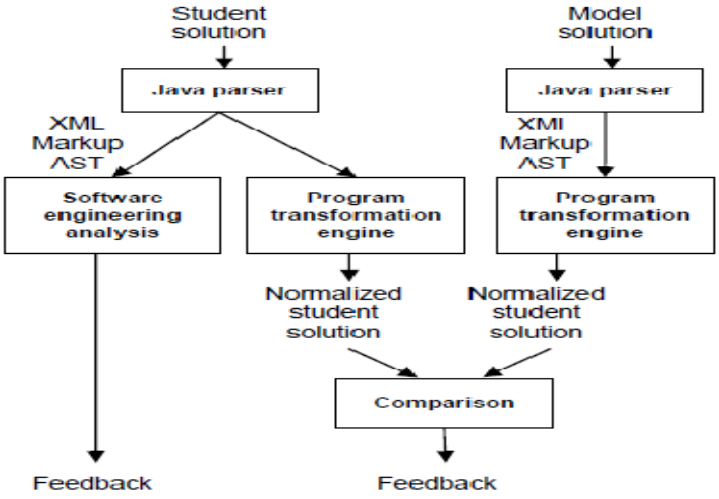


Figure 2-12 : Analyse statique dans ELP (Truong et al., 2004)

```
guess = reader.readInt("Guess a number " +
                "between 1 and 100 ");
while(guess != secret){
   if(guess < secret){
      writer.println("Your guess is low");
   }
   else {
      writer.println("Your guess is high");
   }
   guess = reader.readInt("Guess a " +
        "number between 1 and 100 " );
}
```

```
<gap>
<statements>
   <assignment>1</assignment>
   <methodCall>1</methodCall>
   <loop>
      <condition>
         <trueBranch>
            <methodCall>1</methodCall>
         </trueBranch>
         <falseBranch>
            <methodCall>1</methodCall>
         </falseBranch>
      </condition>
      <assignment>1</assignment>
      <methodCall>1</methodCall>
   </loop>
</statements>
</gap>
```

Figure 2-13: exercice complété et sa transformation en XML (Truong et al., 2004)

```
main()
{
      int a[10], i, max, j;
      for (i=0; i<10; i++)
            scanf("%d",&a[i]);
      max = a[0];
      j = 1;
      while (j < 10)
      {
            max = (a[j] > max) ? a[j] : max;
            j = j + 1;
      }
      printf("max=%d",max);
}
```

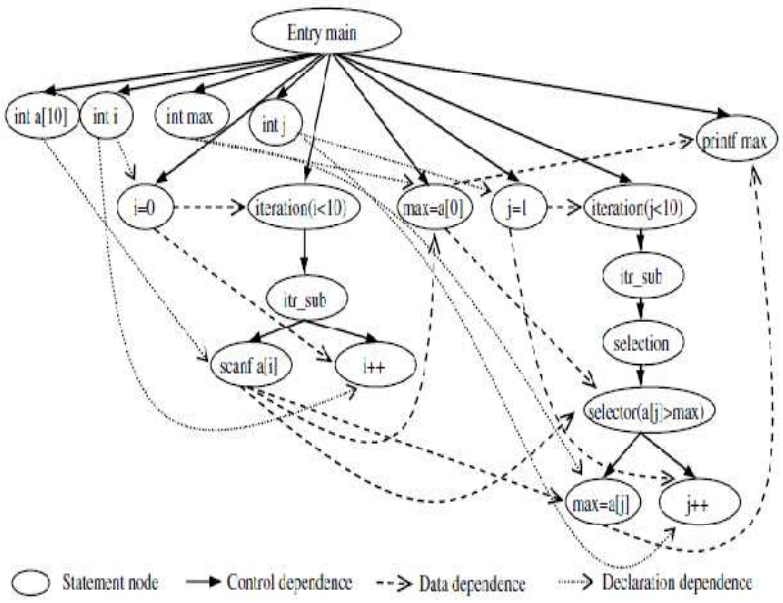


Figure 2-14 : Programme et son graphe de dépendance (T. Wang et al., 2007)

2.5 Synthèse

Nous avons souligné, dans le chapitre précédent (section 1.4), que notre cadre d'étude est un environnement qui permet d'évaluer les solutions algorithmiques des apprenants. L'objectif de cette évaluation est de détecter leurs erreurs afin d'assister les apprenants de manière pertinente. De ce fait nous nous situons dans le cadre des environnements mettant en œuvre une évaluation de type formative.

Pour mettre en œuvre ce type d'évaluation, nous devons diagnostiquer les erreurs des apprenants dans leurs solutions. Suite à l'étude des techniques d'évaluation dans les EIAH et dans les environnements d'apprentissage de la programmation, nous avons retenu une évaluation basée sur la comparaison des solutions apprenant à des solutions modèles. Dans cette étude nous avons relevé le manque de diagnostic d'erreurs dans la solution apprenant et, s'il est mis en œuvre, il s'appuie sur des solutions modèles comportant les différentes erreurs que l'apprenant pourrait commettre.

L'approche d'évaluation que nous mettrons en œuvre, essaiera de combler cette lacune (diagnostiquer les erreurs des apprenants sans s'appuyer sur un modèle de solution avec erreurs).

Après évaluation, si la solution de l'apprenant est correcte, il s'agira de lui donner un exercice suivant afin d'assurer sa progression. Pour une solution incorrecte, il s'agira d'accompagner l'apprenant afin de corriger ses erreurs en nous basant sur la production de rétroactions pertinentes.

Dans le chapitre suivant, nous présentons l'aspect technique de la production des contenus des rétroactions dans les environnements d'apprentissage de la programmation. Nous aborderons également les éléments nécessaires pour définir une organisation des exercices qui nous permettra de fournir à l'apprenant un exercice correspondant à son niveau de compétence.

Chapitre 3 : Les rétroactions et éléments d'organisation des exercices

Contenu

3.1 Introduction

Ce chapitre explore les moyens mis en œuvre pour accompagner l'apprenant novice afin de renforcer ou consolider ses connaissances stratégiques de programmation. Une première section est dédiée aux rétroactions destinées à accompagner l'apprenant pour terminer un exercice en cas de difficulté. Pour cela, nous présenterons dans cette section un panorama de différentes formes de rétroactions fournies aux apprenants et les techniques mises en œuvre dans les environnements d'apprentissage de la programmation pour générer ces rétroactions. Dans une seconde et dernière section, il s'agit d'assurer la progression de l'apprenant en lui proposant des exercices avec des niveaux de difficulté différents. Ainsi, nous présenterons dans cette section, les éléments pertinents pour définir l'organisation des exercices.

3.2 Les rétroactions

Selon (Boud & Molloy, 2013), la rétroaction est définie comme un processus de génération d'informations (commentaires) sur le travail d'un apprenant. Ces informations ont pour objectif d'améliorer le travail de l'apprenant. Elles l'aident à atteindre l'objectif fixé pour la tâche. Dans la même optique, (Ott, Robins, & Shephard, 2016) soulignent que la notion de rétroaction est perçue comme une information fournie à l'apprenant afin de réduire l'écart entre sa performance actuelle et la performance souhaitée.

Dans la littérature, la rétroaction est une information susceptible de corriger une imperfection. Elle est ainsi considérée comme une contribution importante à l'amélioration de l'apprentissage (Hattie & Timperly, 2007), (Carless, 2006).

Selon (Bosc-Miné, 2014), Les rétroactions fournies aux apprenants peuvent être de deux types :

- *Vérification,* qui est une information concernant « *la justesse ou l'inexactitude de la réponse* ». Ce type de rétroaction, fournit des informations sur la *performance* d'une réponse donnée par l'apprenant. Elle indique sous forme numérique ou graphique, le nombre ou la proportion de réussites de l'apprenant. Pour cela, elle donne une *appréciation sur la réponse fournie*. Celle-ci peut être une appréciation ordonnée de « très insuffisant » à « excellent » ou sous forme binaire (oui/non, correct/incorrect…). Dans le cas où l'apprenant ne donne pas la réponse attendue, la rétroaction fournit une

information indiquant la réponse correcte ou la meilleure solution possible pour un problème donné.

- *Élaboration*, qui est une information consistant à fournir « *des indices permettant de guider l'individu vers la réponse correcte ou vers l'amélioration des connaissances* ». Ce type de rétroaction a pour rôle d'assister l'apprenant dans la réalisation d'une tâche. A cet effet, la rétroaction fournit des informations, qui peuvent être interprétées comme une aide tutorielle. Cette dernière, selon (Narciss, 2008) permet de guider la réalisation d'une activité, en fournissant des indices, en suggérant une piste pour corriger les erreurs identifiées dans la réalisation de l'activité. La rétroaction d'aide tutorielle, peut également être, selon (Bosc-Miné, 2014) un guidage de l'action. Cette dernière se présente, lorsque l'apprenant n'a pas bien compris la consigne de l'activité. Pour cela, il s'agit de lui présenter un axe de compréhension ou une nouvelle formulation de la consigne.

Dans la rétroaction de type « *élaboration* », l'accompagnement de l'apprenant dans ses difficultés est mis en avant. Dans le cadre de nos travaux, nous nous situons donc dans ce type de rétroaction ou nous cherchons à fournir à l'apprenant des rétroactions qui le guident afin de terminer l'exercice. Dans ce qui suit, nous présenterons une étude de littérature en deux points : d'une part nous étudions les différentes formes de rétroactions de type élaboration, mises en œuvre dans le contexte de l'apprentissage de la programmation. D'autre part, nous présentons certains outils afin d'étudier les techniques mises en œuvre pour générer le contenu de ces rétroactions.

3.2.1 Formes de rétroactions générées dans les environnements d'apprentissage de la programmation

(Le, 2016) classe les différentes rétroactions générées par les environnements d'apprentissage de la programmation comme suit :

- Des rétroactions sur les erreurs syntaxiques (*Syntax Feedback)* : dans ces rétroactions, la plupart des environnements fournissent un message généré par le compilateur en ajoutant parfois quelques explications. Certains outils, fournissent des explications détaillées sur les erreurs syntaxiques commises par l'apprenant.
- Des rétroactions sémantiques (*Semantic Feedback)* : les informations fournies à l'apprenant dans ce type de rétroaction correspondent à des erreurs détectées dans la

solution de l'apprenant. Après analyse, les informations fournies à l'apprenant peuvent être : des indices qui aident à la résolution, des explications sur les erreurs, un exemple de la solution attendue.

- Des rétroactions sur le style de programmation (*Layout Feedback)* : ce type de rétroaction aide les apprenants sur deux points : (i) aider les apprenants à écrire un code conforme à une convention de codage spécifique et (ii) aider les apprenants à avoir des codes propres afin qu'ils puissent retrouver leurs erreurs. Les informations fournies aux apprenants dans ce type de rétroaction résultent d'une analyse faite sur le style de programmation dans les solutions apprenants.
- Des rétroactions sur la qualité (*Quality Feedback)* : ce type de rétroaction aide les apprenants à écrire un code qui respecte certaines règles de qualité. Les informations fournies dans ce type de rétroaction résultent d'une analyse faite avec des métriques du génie logiciel.

Dans le cadre de nos travaux, les apprenants seront amenés à résoudre des exercices de programmation. Dans ces exercices l'apprenant, doit être en mesure d'utiliser les concepts de programmations appropriés. Nous cherchons à détecter ses erreurs afin de l'aider à les corriger. Nous nous plaçons donc dans le cadre des rétroactions de type sémantique.

Ce type de rétroaction peut être affiné selon la classification de (Narciss, 2008) :

- Rétroaction sur les exigences de l'exercice (*Knowledge about task constraints*) : elle permet d'indiquer à l'apprenant les règles, les contraintes et/ou les exigences de l'exercice ;
- Rétroaction sur les connaissances conceptuelles (*Knowledge about concepts*) : elle permet de donner des conseils sur l'utilisation de certains concepts qui ne sont pas encore maîtrisés ;
- Rétroaction sur les erreurs (*Knowledge about mistakes*) : elle fournit des informations sur les erreurs commises par l'apprenant. Ces informations peuvent être une indication de la ligne d'erreur, une explication sur le type d'erreur ou les sources de l'erreur ;
- Rétroaction sur la façon de réaliser un exercice (*Knowledge about how to process the task*) : elle fournit des informations sur la façon de résoudre un exercice (conseil de résolution, stratégie à mettre en œuvre…) ;

- Rétroaction sur la métacognition (*knowledge about meta-cognition)* : elle fournit des informations susceptibles de favoriser les connaissances métacognitives et la stratégie nécessaire pour auto-réguler le processus d'apprentissage.

Nous décrivons dans ce qui suit certains outils mettant en œuvre ce type de rétroaction afin d'étudier les techniques mises en œuvre pour construire le contenu de ces rétroactions.

3.2.2 Techniques de génération des rétroactions dans les environnements d'apprentissage de la programmation

Dans cette section nous proposons l'étude de certains environnements. L'objectif de cette étude est de savoir comment le contenu des rétroactions fournies à l'apprenant est construit par les environnements d'apprentissage de la programmation. Nous avons retenu, dans notre étude, les environnements répondant aux critères suivants:

1. Leur proximité à notre contexte de recherche (cours d'introduction à la programmation, algorithmique).
2. Mise en œuvre d'une analyse basée sur la comparaison de la solution apprenant à des solutions modèles.
3. Génération de rétroactions de type sémantiques (« *Semantic Feedback* ») plus particulièrement destinées à la correction des erreurs des apprenants.

3.2.2.1 Ask-Elle

Ask-Elle est un système de tutorat en ligne (*figure 3-1*) (Gerdes, Heeren, & Jeuring, 2017). Il s'adresse à des étudiants en cours d'introduction à la programmation, pour les assister dans les étapes de développement du langage de programmation fonctionnelle Haskell. Cet environnement, permet de générer automatiquement des rétroactions. Ces dernières peuvent être des indices pour résoudre l'exercice ou une information sur les erreurs de l'apprenant. Pour ce faire, il se base sur des configurations préétablies par l'enseignant. Ce dernier, lorsqu'il ajoute un exercice, spécifie un ou plusieurs modèle(s) de solution(s) et certaines propriétés que chaque solution proposée doit satisfaire. D'autre part, les contenus des rétroactions à générer sont paramétrés par l'enseignant via des annotations faites sur la solution modèle (*figure 3-2*) et/ou sur les propriétés (*figure 3-3*).

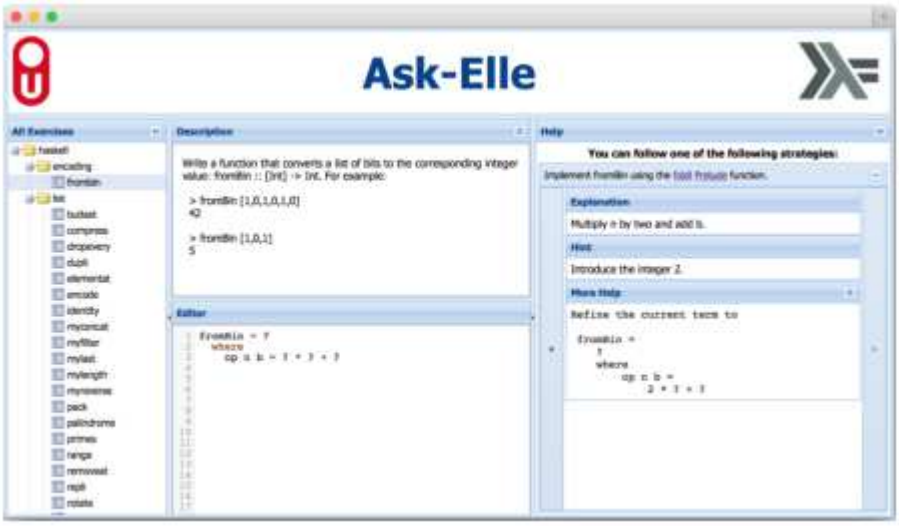


Figure 3-1: Interface apprenant Ask-Elle (Gerdes, Heeren, & Jeuring, 2017)

```
propModel f bs = feedback msg (output == model)
  where
    output = f bs
    model  = foldl (λn b → 2 * n + b) 0 bs
    msg    = "Your implementation is incorrect for the " ++
             "following input: " ++ show bs ++ "\nWe expected " ++
             show model ++ ", but we got " ++ show output
```

Figure 3-2: Annotation faite sur les propriétés

```
fromBin =
  {-# FB Define the fromBin function using foldl. The op... #-}
  foldl op 0
    where
      op n b = {-# FB Multiply n by two and add b. #-}2 * n + b
```

Figure 3-3: Rétroaction annotée dans le modèle solution

3.2.2.2 PROPL

PROPL est un système de tutorat, qui permet d'assister les apprenants dans la construction de la solution en pseudo-code d'un problème donné (Lane & VanLehn, 2005). Pour ce faire, PROBL instaure un tutorat basé sur le dialogue. Ce dialogue se fait en quatre étapes : (i) poser des questions qui permettent d'identifier si l'apprenant arrive à reconnaître

l'objectif du problème donné ; (ii) instaurer une dialogue qui permet de vérifier, comment l'apprenant décrit le plan pour arriver à l'objectif ; (iii) suggérer à l'apprenant une solution en pseudo-code (*figure 3-4*) et (iv) l'apprenant place des étapes dans le pseudo-code proposé. C'est à travers ce dialogue que le système assiste l'apprenant. À cet effet, il dispose d'une base de connaissance (*Knowledge Construction Dialogues*) qui a une structure hiérarchique, où chaque question est associée à un ensemble des réponses attendues.

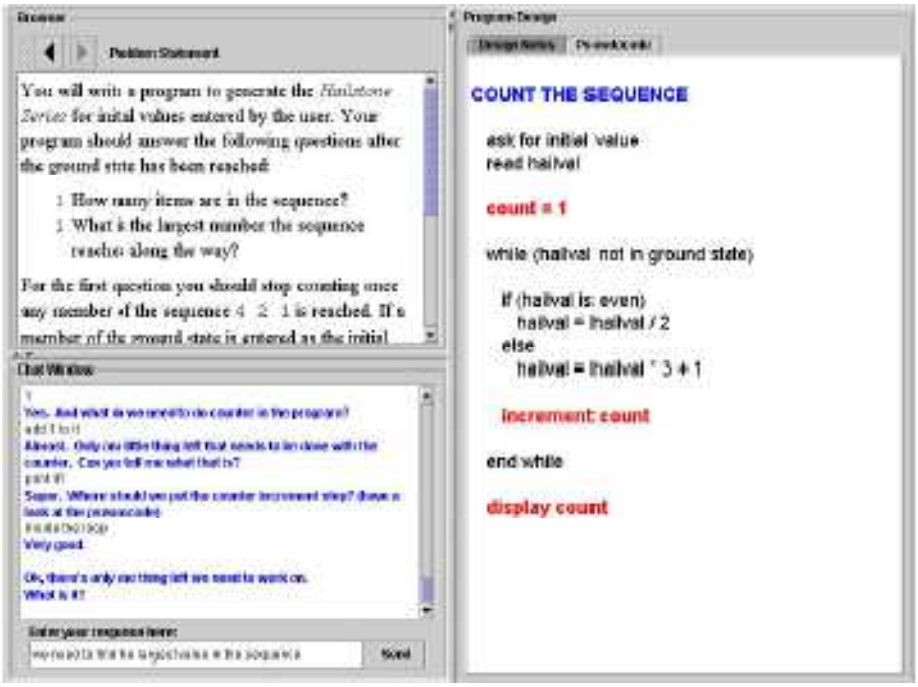


Figure 3-4: PROPL, solution en pseudo-code (Lane & VanLehn, 2005)

3.2.2.3 Algo+

Algo+, est un environnement qui permet d'analyser les solutions algorithmiques (solutions décrites en pseudo-code), soumises par les apprenants (Bey & Bensebaa, 2011). Après analyse l'environnement informe l'apprenant de la note obtenue et lui donne également une rétroaction. Le contenu de cette dernière, est paramétré par l'enseignant. Pour chaque exercice, l'enseignant propose un ensemble de solutions modèles correctes et incorrectes. Pour chaque solution modèle, il associe la rétroaction à générer. Cela implique que l'enseignant doit prévoir l'ensemble des solutions incorrectes pour un exercice donné.

3.2.2.4 ELP

ELP, est un environnement Web, destiné aux apprenants novices en programmation Java (Truong et al., 2004). Dans cet environnement les apprenants doivent compléter un code donné (figure 3-5). L'environnement analyse ensuite le bout de code ajouté par comparaison de la solution complétée avec une solution modèle et génère une rétroaction. Le contenu de cette dernière comprend : (i) un résumé montrant la différence structurelle entre les deux solutions. C'est à l'apprenant de comprendre cette différence (figure 3-6) et (ii) l'affichage de la solution correcte pour que l'apprenant puisse corriger son code.

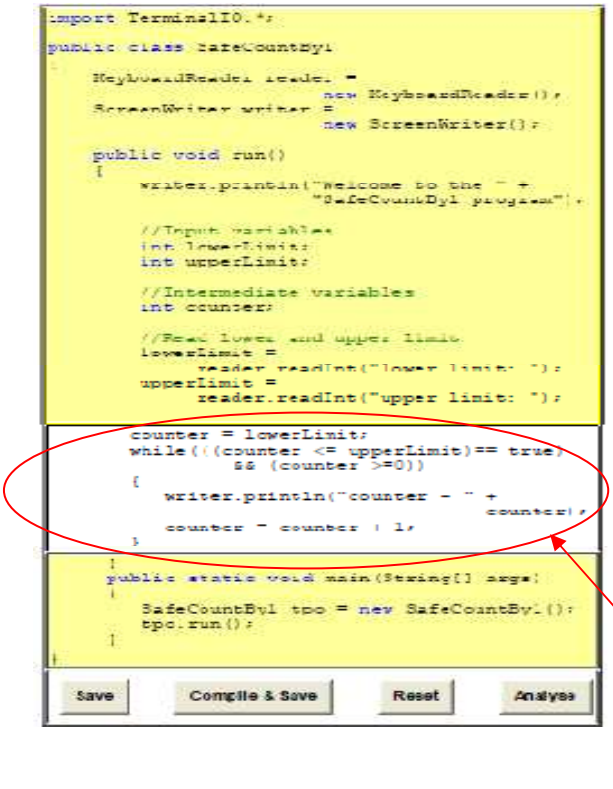


Figure 3-5 : Un exercice avec un code à compléter.

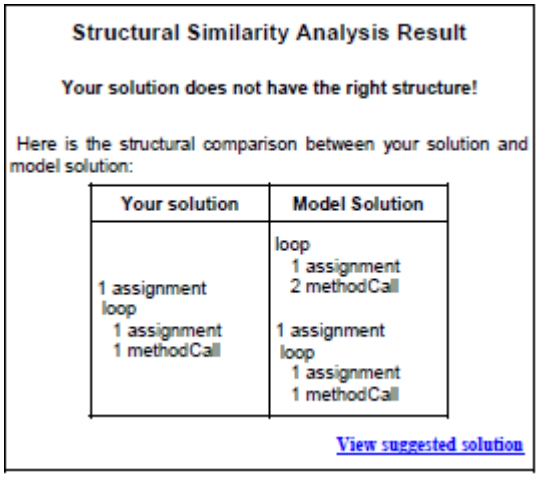


Figure 3-6: Différence structurelle entre la solution apprenant et modèle de solution.

3.2.3 Synthèse et positionnement

Dans cette section nous avons souligné que nos travaux se placent dans le cadre d'une rétroaction de type « élaboration ». Il s'agit de fournir à l'apprenant des indices et des explications sur ses erreurs.

Nous observons que les contenus des rétroactions, pour la plupart des environnements présentés, sont directement rattachés à une solution modèle et paramétrés par l'enseignant. Dans le cadre de notre travail, Nous avons souligné dans la section 2.5, que nous souhaitons

éviter d'avoir à énumérer les solutions modèles avec erreur pour diagnostiquer les erreurs des apprenants. Pour cela, nous souhaitons mettre en œuvre une génération automatique des contenus de rétroactions en fonction des erreurs diagnostiquées.

3.3 Eléments pertinents pour l'organisation des exercices

Pour améliorer la capacité de résolution de problèmes des apprenants novices en programmation, qui se traduit par le renforcement de leurs connaissances stratégiques, la littérature recommande la réalisation de nombreux exercices avec des niveaux de difficultés croissants. Les environnements d'apprentissage utilisent différentes techniques afin de fournir des exercices. Celles-ci sont classifiées, selon (Lefevre, 2009) comme suit :

- Génération automatique d'exercices : dans ce cas l'environnement génère automatiquement les énoncés d'exercices ainsi que leurs solutions, l'enseignant n'intervient pas.
- Génération semi-automatique : dans ce cas, l'environnement construit l'énoncé d'exercice en fonction de contraintes données par l'enseignant.
- Production manuelle : dans ce cas, l'environnement dispose d'une bibliothèque d'exercices et leurs solutions, construite et mise à jour par l'enseignant.

Nous cherchons à renforcer la capacité des apprenants à choisir les concepts de programmation appropriés pour résoudre un exercice. Pour cela l'apprenant doit être confronté à plusieurs exercices, mettant en œuvre un ou plusieurs concept(s) de programmation avec des degrés de difficulté différents. Par ailleurs, nous avons souligné dans la section 2.3.4 que notre approche d'évaluation, pour diagnostiquer les erreurs des apprenants, sera basée sur la comparaison des solutions apprenant à des solutions modèles. La génération des exercices sur différents niveaux et concepts et la génération de leurs solutions modèles est de notre point de vue impossible. Nous avons donc choisi la production manuelle.

Les exercices soumis aux apprenants seront issus d'une collection d'exercices organisée par niveaux et concepts, conçue par les enseignants. Pour proposer un modèle d'organisation des exercices nous nous basons sur les éléments suivants :

- Deux modélisations, trouvées dans la littérature, proposant une forme d'organisation des concepts de programmation pour le cours d'introduction.

- Les taxonomies d'objectifs d'apprentissage existantes afin de classer les exercices selon différents niveaux.
- La pédagogie de la maîtrise des apprentissages, « *Mastery Learning* », qui stipule que les concepts mis en œuvre dans l'exercice doivent être maîtrisés avant de passer à un autre exercice proposant d'autres concepts plus complexes.

Cette section présente ces éléments qui constituent la base de notre proposition de modèle d'organisation des exercices qui sera présenté dans le chapitre 6.

3.3.1 Modèle d'organisation des exercices

Nous avons trouvé deux modèles d'organisation des exercices, pour le cours d'introduction à la programmation qui sont présentés dans la suite.

3.3.1.1 Modèle de Chookaew et al.

Pour améliorer la maîtrise, par les apprenants, des concepts fondamentaux dans les cours d'introduction à la programmation, (Chookaew, Panjaburee, Wanichsan, & Laosinchai, 2014) proposent un environnement en ligne. Cet environnement réalise une personnalisation de l'apprentissage reposant sur :

- Des tests basés sur un modèle de « *relation d'effet des concepts* » (*concept-effect relationships*). ce modèle, présenté sur la *figure 3-7*, définit les relations entre les concepts qui doivent être appris dans un ordre donné. En d'autres mots l'apprentissage d'un concept donné peut nécessiter la connaissance préalable d'un ou plusieurs autres concepts. Ainsi lors de l'évaluation de l'exercice, l'environnement calcule un taux de réussite pour un concept donné.
- Le passage d'un concept à un autre est basé sur le taux de réussite du test précédent. Si le taux est compris entre 0 et 0,51 alors la maîtrise du concept mis en œuvre dans l'exercice est considéré comme faible, pour un taux compris entre 0,51 et 0,75 la réussite est moyenne, au delà l'apprenant a réussi et peut passer aux concepts suivants.

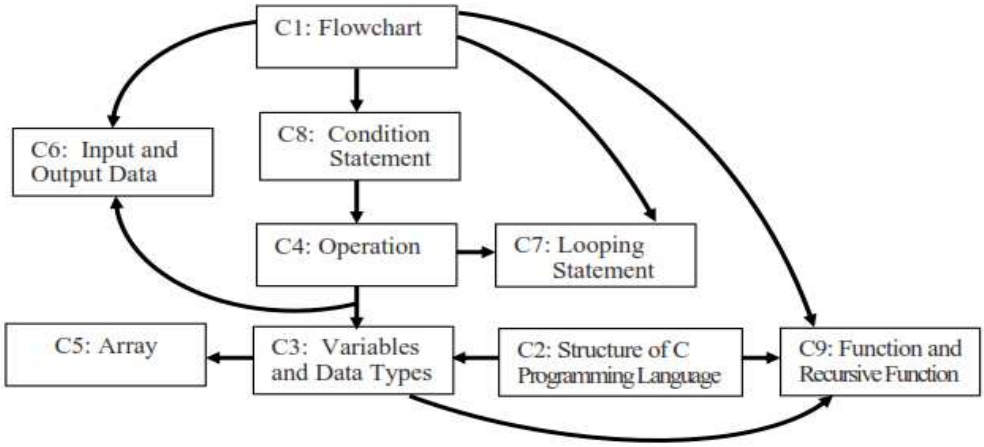


Figure 3-7 : Relation Concept-effet sur les concepts basiques de l'apprentissage de la programmation (Chookaew, Panjaburee, Wanichsan, & Laosinchai, 2014)

3.3.1.2 Modèle de Santos et al.

(Santos, Gomes, & Mendes, 2013) proposent un modèle de classification des exercices, qui permet de proposer aux apprenants, une séquence d'exercices selon leur niveau (*figure 3-8*). A cet effet, leur modèle comprend trois dimensions : (i) *sujet,* dans cette dimension, l'exercice est classé selon les concepts de programmation nécessaires pour résoudre l'exercice (structure de contrôle, variable, tableau…); (ii) *complexité,* dans cette dimension, l'exercice est classé selon son niveau de complexité, celui-ci est paramétré par l'enseignant. (iii) *niveau,* dans cette dimension l'exercice est classé par niveaux, selon deux paramètres qui sont la taxonomie de Bloom d'une part et un niveau (débutant, intermédiaire ou avancé) requis d'autre part.

Les auteurs utilisent un graphe avec une mesure de poids pour relier les exercices entre eux. Cette mesure est utilisée pour générer l'exercice suivant. La mesure du poids est une valeur entière positive (pour définir que l'exercice est plus difficile) ou négative (pour définir que l'exercice est plus facile). Par exemple dans le graphe (*figure 3-9*) l'exercice A est plus facile que l'exercice B et l'exercice C est plus difficile que l'exercice B.

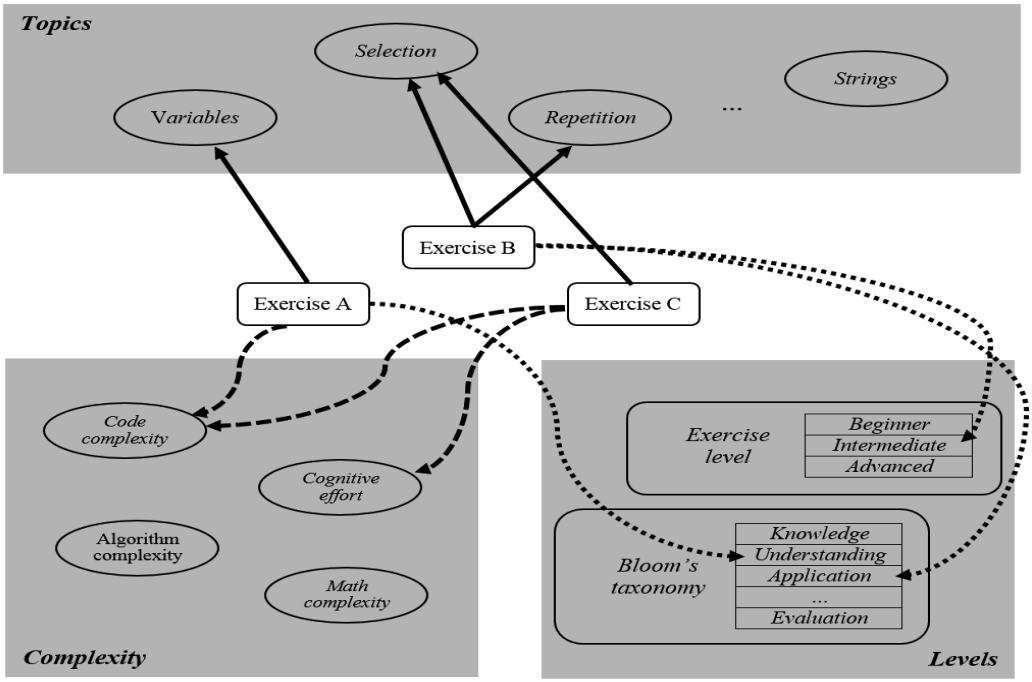


Figure 3-8 : Modèle d'activité (exercices) de (Santos, Gomes, & Mendes, 2013)

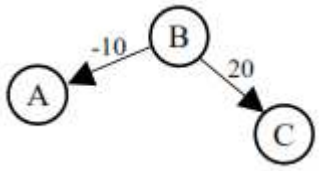


Figure 3-9 : Graphe d'exercice (Santos, Gomes, & Mendes, 2013)

3.3.2 Taxonomies des objectifs d'apprentissage

Les taxonomies d'objectifs d'apprentissage sont utilisées, selon (Le & Pinkwart, 2014) pour : (i) concevoir des cours à différents niveaux de granularité ; (ii) concevoir des moyens d'évaluation ; (iii) analyser les réponses des étudiants pour un exercice donné afin de mesurer la progression de l'apprenant. En d'autres mots les taxonomies sont un moyen de classement des objectifs d'apprentissage, d'une manière ordonnée. Elles sont aussi utilisées, pour structurer et classer, les exercices dans les environnements d'apprentissage. En ce qui concerne l'apprentissage de la programmation, plusieurs études confirment la pertinence de ces taxonomies (Selby, 2015).

Les taxonomies répartissent les objectifs d'apprentissage en trois catégories : cognitive, affective et psychomotrice. Dans ce document, nous nous intéressons au domaine cognitif. Nous présentons dans ce qui suit quelques taxonomies d'objectifs d'apprentissage souvent utilisées dans le domaine de l'enseignement et l'apprentissage de la programmation : à savoir la taxonomie de Bloom, la taxonomie de SOLO et la taxonomie de Bloom révisée. L'objectif de cette description n'est pas de proposer une nouvelle taxonomie mais de nous positionner.

3.3.2.1 Taxonomie de Bloom

La taxonomie de Bloom, créée par Benjamin Bloom, est l'une des premières taxonomies des objectifs d'apprentissage à être utilisée et reconnue en éducation (Raîche, 2004). Elle est, selon (Raîche, 2004) « *celle qui a été, et est encore, la plus utilisée pour modéliser les objectifs d'apprentissage en éducation* ». Dans la même optique (Shuhidan, Hamilton, & D'Souza, 2009) soulignent que les enseignants utilisent, souvent cette taxonomie pour composer des activités d'apprentissage sophistiquées avec des niveaux de difficulté croissants.

Cette taxonomie, dans le domaine cognitif, permet de classer les niveaux des compétences requises dans les exercices (Bloom, 1956). Elle permet ainsi de mesurer la progression de l'apprenant en effectuant une mesure dans un ordre de difficulté croissante. Cette taxonomie comprend six niveaux, du plus bas niveau au plus élevé (*figure 3-10).* L'apprenant avance dans la taxonomie, au fur et à mesure que ses connaissances progressent. Ces niveaux sont répartis comme suit :

- ***Connaissance :*** dans ce niveau, les activités affectées aux apprenants, vérifient juste s'ils/elles se rappellent certains éléments du cours. Selon (Kelly & Buckley, 2006), en apprentissage de la programmation, ce niveau permet de vérifier, si l'étudiant arrive à utiliser certains concepts. ceci peut se présenter à travers des activités très simples ou par des QCM (Lister & Leaney, 2003). On peut par exemple « demander aux étudiants de citer les différents types de boucles ».
- ***Compréhension :*** dans ce niveau, les activités affectées aux apprenants, vérifient si les apprenants ont bien compris certaines notions du cours. En apprentissage de la programmation, ce niveau propose des activités, pour vérifier, si l'étudiant a bien compris la spécificité de chaque concept. Par exemple avec une situation qui vérifie si les apprenants arrivent à différentier la boucle `Pour` et la boucle `Tant Que`. Pour

(Lister & Leaney, 2003), dans l'apprentissage de la programmation en Java, ce niveau permet de vérifier, si l'étudiant arrive à traduire un code écrit en pseudo-code en Java.

- ***Application :*** Dans ce niveau, les activités affectées aux apprenants, vérifient s'ils arrivent à mobiliser leurs connaissances sur des activités familières. Selon (Kelly & Buckley, 2006), (Lister & Leaney, 2003), il s'agit d'évaluer, la capacité de l'apprenant à appliquer des solutions, déjà vues, dans une situation similaire.
- ***Analyse :*** Dans ce niveau, on donne aux apprenants, des activités qui nécessitent une réflexion avancée. selon (Kelly & Buckley, 2006), ce niveau vérifie si l'apprenant arrive à décomposer un problème en sous-problèmes.
- ***Synthèse :*** Dans ce niveau, on donne aux apprenants, des activités qui nécessitent une certaine créativité. Selon (Kelly & Buckley, 2006), ce niveau souligne la créativité et l'originalité des pensées. Par exemple, dans une utilisation en programmation Java, (Lister & Leaney, 2003) rapportent qu'une tâche appropriée pour ce niveau serait de demander à décomposer une activité (problème) en classes, et de déterminer les méthodes appropriés pour chaque classe.
- ***Évaluation :*** Dans ce niveau, les apprenants s'entraînent à faire des hypothèses, à tester la qualité d'un produit ou choisir une solution parmi plusieurs à partir de certains critères. Par exemple, d'après (Kelly & Buckley, 2006), on propose des activités, ou l'apprenant, doit choisir une méthode de résolution par rapport à une autre.

Dans le cadre d'un cours d'introduction à la programmation, il est souligné que, compte tenu de la portée du cours, les niveaux ***analyse***, ***synthèse*** et ***évaluation***, sont considérés difficiles à mettre en œuvre (Fuller et al., 2007).

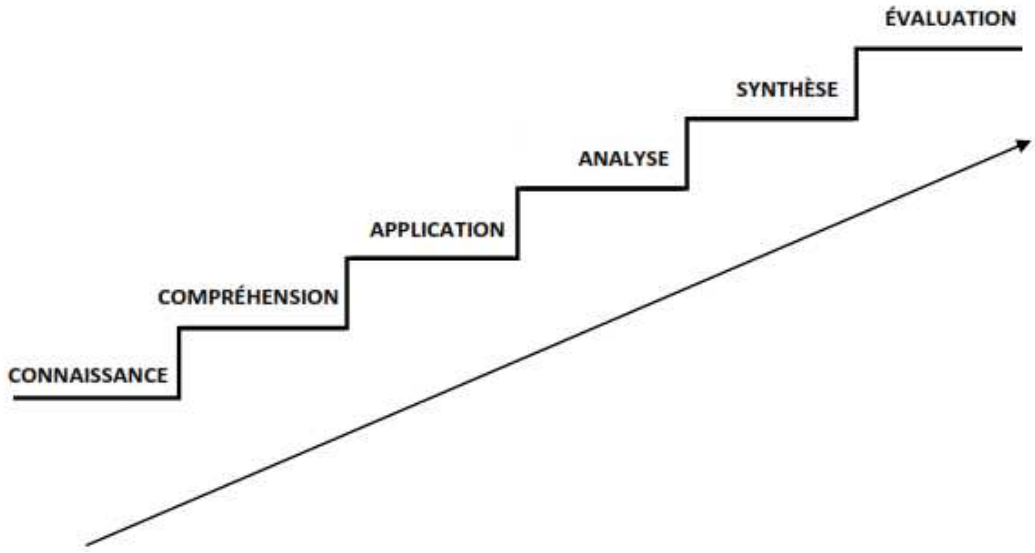


Figure 3-10 : Taxonomie de Bloom, domaine cognitif

3.3.2.2 Taxonomie de SOLO

La taxonomie de SOLO (*Structure of the Observed Learning Outcome*), contrairement à la taxonomie de Bloom, n'est pas utilisée pour classer les objectifs d'apprentissage. Elle permet plutôt de mesurer le niveau de compréhension de l'apprenant, sur ce qui est évalué (Fuller et al., 2007). Pour sa part (Chan, Tsui, Mandy, & Hong, 2002) soulignent que, dans la littérature, la taxonomie de SOLO est considérée comme un modèle hiérarchique qui convient pour mesurer le résultat de l'apprentissage suivant différents niveaux cognitifs. Cette taxonomie décompose le niveau de compréhension d'un apprenant pour un sujet donné en cinq niveaux (Whalley, Clear, Robbins, & Thompson, 2011). Les cinq niveaux de cette taxonomie sont décrits comme suit :

- ***Niveau Pré-Structurel (Pre-Structural Level) :*** Dans ce niveau l'apprenant n'a aucune compréhension. Il a des connaissances mal organisées, non structurées ou sans relations appropriées aux questions ou activité demandées.
- ***Niveau Uni-Structurel (Uni-Structural Level) :*** Dans ce niveau, l'apprenant ne peut traiter qu'un seul aspect. Il peut établir des liens et doit être capable de réciter (se souvenir des choses), traiter des instructions ou algorithmes simples, d'identifier, de nommer, de compter, etc.
- ***Niveau Multi-Structurel (Multi-Structural Level) :*** Dans ce niveau, l'apprenant peut traiter plusieurs aspects mais d'une manière indépendante et sans faire le lien. Il doit

être capable d'énumérer, de classer, de combiner, d'appliquer des méthodes, de structurer, d'exécuter une procédure, etc.

- ***Niveau Relationnel (Relational Level) :*** Dans ce niveau l'apprenant doit comprendre la relation entre plusieurs aspects et la façon de les assembler pour former une solution. L'apprenant peut ainsi avoir la compétence de comparer, relier, analyser, etc.
- ***Niveau Abstrait Étendu (Extended Abstract Level) :*** Dans ce niveau, qui est le plus élevé, l'apprenant peut généraliser la structure au-delà ce qui est donné pour ainsi produire des nouvelles connaissances. Mobiliser des idées pour résoudre des nouveaux cas. Il peut avoir la compétence de généraliser, émettre des hypothèses, critiquer, etc.

Contrairement à la taxonomie de Bloom qui mesure la progression de l'apprenant dans un ordre de difficulté croissant, pour la taxonomie de SOLO, ceci n'est pas une exigence. Ainsi, dans une activité (problème) donnée on peut évaluer le niveau de compréhension de l'apprenant dans différent niveaux de la taxonomie. Il est aussi possible de construire une situation dont le niveau supérieur est moins difficile que le niveau inférieur.

3.3.2.3 Taxonomie de Bloom révisée

La taxonomie de Bloom révisée, transforme le nom des niveaux de la taxonomie de Bloom en verbes afin de mieux décrire les objectifs d'apprentissage. Elle se présente également en deux dimensions (*figure 3-11*), contrairement à la taxonomie de Bloom qui ne présentait qu'une seule dimension (Anderson et al., 2001). Ces deux dimensions sont :

- la dimension **cognitive** (représentée par la colonne de la table *figure 3-11*, est représentée par six catégories : **Mémoriser**, **Comprendre**, **Appliquer**, **Analyser**, **Évaluer** et **Créer**. Chacune de ces catégories est associée à deux ou plusieurs processus cognitifs. Les niveaux de cette taxonomie, comme la taxonomie de Bloom originale, sont dans un ordre de difficulté croissant, i.e. le niveau **Comprendre** est considéré plus complexe, sur le plan cognitif que le niveau **Mémoriser**, **Analyser** est plus complexe que le niveau **Appliquer**, ainsi de suite.
- la dimension **connaissance** (représentée par la ligne de la table *figure 3-11*) permet de préciser la connaissance mise en œuvre pour un niveau de processus cognitif donné. Elle se présente en quatre catégories : **Factuel**, **Conceptuel**, **Procédural** et **Métacognitif**.

Dimension Connaissance	La Dimension Cognitive					
	1. Mémoriser	2. Comprendre	3. Appliquer	4. Analyser	5. Évaluer	6. Créer
A. Factuel						
B. Conceptuel						
C. Procédural						
D. Métacognitif						

Figure 3-11 : Table de taxonomie de Bloom Révisée en deux dimensions (Anderson et al., 2001)

3.3.3 La pédagogie de la maîtrise des apprentissages «Mastery Learning »

Cette approche pédagogique est centrée sur l'apprenant. Ce dernier doit démontrer la maîtrise d'un sujet donné avant de passer à un sujet plus difficile. Ainsi les sujets à étudier sont organisées d'une manière ordonnée. Selon (Bloom, 1971) cité dans (Guskey, 2007), cette approche permet dans un premier temps, à l'enseignant, d'organiser les concepts ou les compétences à faire travailler à l'apprenant en unités d'apprentissage. Le passage entre les unités d'apprentissage repose sur une évaluation formative (*figure 3-12*). Cette dernière permettent soit de proposer des activités correctives, qui visent à combler les lacunes des apprenants, soit de proposer des activités de renforcement. Les unités d'apprentissage doivent être organisées du plus facile au plus difficile.

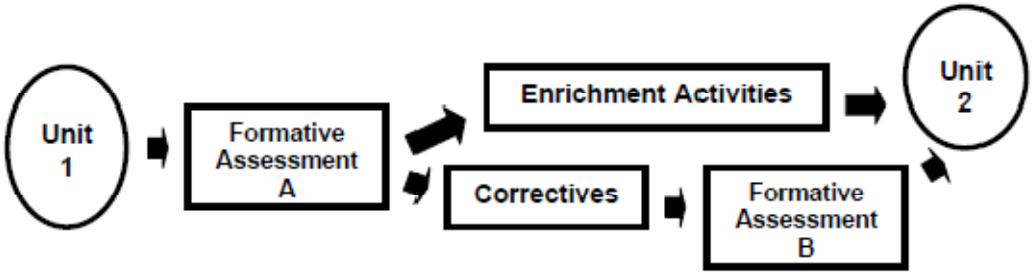


Figure 3-12 : Processus de « Mastery Learning » (Guskey, 2007)

3.4 Synthèse

Dans ce chapitre, l'objectif était d'étudier les moyens mis en œuvre pour renforcer les connaissances stratégiques de l'apprenant. Pour cela, nous avons présenté dans une première section, une étude sur la génération de rétroactions afin d'accompagner les apprenants qui ont une difficulté. Nous avons mis en évidence que les informations fournies à l'apprenant sont issues, pour la plupart, des solutions modèles paramétrées par l'enseignant. Ceci implique d'avoir des solutions modèles avec erreurs. L'approche défendue dans cette thèse est au contraire de construire les contenus des rétroactions selon le diagnostic des erreurs sans s'appuyer sur des solutions modèles avec erreurs.

Dans la deuxième section, nous avons étudié les différents éléments permettant d'accompagner l'apprenant dans sa progression, en lui proposant des exercices à des niveaux de difficultés croissants. Ces éléments vont servir de base pour notre proposition d'un modèle d'organisation des exercices. De cette étude, nous retenons que :

- Les concepts de programmation s'apprennent d'une manière ordonnée et que la maîtrise d'un ou plusieurs concepts peut faciliter la maîtrise d'autres concepts plus difficiles.
- Le taux de réussite d'un exercice nous informe sur le degré de maîtrise d'un ou plusieurs concepts.
- Le classement des exercices peut être réalisé selon un niveau cognitif. Pour cela nous retenons le classement selon la taxonomie de Bloom qui est plutôt destinée à classer, *a priori,* la difficulté des questions contrairement aux autres (Bloom révisé et SOLO) qui s'intéressent plutôt à classifier les réponses. Nous nous limitons, comme souligné par (Fuller et al., 2007) aux trois premiers niveaux de cette taxonomie.

La partie suivante de ce mémoire présente nos différentes propositions à savoir l'approche d'évaluation pour statuer sur la solution, le modèle d'organisation des exercices pour avoir une collection d'exercices organisée permettant de fournir à l'apprenant des exercices progressifs et le mécanisme de génération des rétroactions.

Partie 2 : Contributions

Cette partie décrit nos propositions concernant une approche d'évaluation, un modèle d'organisation des exercices, un mécanisme de génération des rétroactions et la réification de ces propositions dans un environnement Web appelé *AlgoInit*. L'objectif principal est de contribuer à l'accompagnement des apprenants dans l'apprentissage des cours d'introduction à la programmation. Cette partie détaille, dans un premier temps, notre **approche d'évaluation,** un processus qui est destiné à analyser la solution algorithmique proposée par l'apprenant pour résoudre un exercice. Cette solution sera décrite en pseudo-code. L'objectif de cette approche est de détecter les difficultés des apprenants dans l'utilisation des concepts de programmation (*chapitre 4*). Nous présentons ensuite notre **modèle d'organisation des exercices** et le **mécanisme de génération des rétroactions**, dont l'objectif est, en fonction des résultats de l'évaluation, soit de fournir un autre exercice, soit d'accompagner l'apprenant par des rétroactions pour terminer l'exercice *(chapitre 5)*. Nous présentons, dans le *chapitre 6*, la réification de ces propositions dans un environnement Web appelé *AlgoInit*. Enfin, cette partie s'achève sur la description de notre méthodologie d'expérimentation. Nous abordons ensuite, l'analyse des résultats. Ceux-ci portent sur le taux de reconnaissance des solutions algorithmiques des apprenants, l'apport de l'assistance proposée par *AlgoInit* ainsi que l'évaluation de l'utilisabilité de cet environnement *(chapitre 7)*.

Chapitre 4 : Processus d'évaluation des solutions algorithmiques

Contenu

4.1 Introduction

Pour détecter, les difficultés des apprenants à utiliser les concepts de programmation appropriés, il est nécessaire d'analyser leurs solutions. Ce chapitre propose une approche d'évaluation qui permet d'identifier les solutions correctes et, le cas échéant, d'identifier et d'énumérer les erreurs dans les solutions des apprenants.

Dans ce chapitre, nous présentons d'abord le principe général de notre approche d'évaluation. Nous donnons ensuite le détail des différentes étapes qui nous permettent de statuer sur la solution algorithmique d'un apprenant. Enfin, nous terminons par une conclusion concernant la proposition présentée dans ce chapitre.

4.2 Principe général de notre approche

Notre approche d'évaluation est inspirée de la méthode de décomposition de problèmes. Selon (HO, 2001) la première étape, pour résoudre un problème est de réaliser une analyse fonctionnelle qui consiste à décomposer un problème en sous-problèmes plus simples à résoudre. La résolution de ces sous-problèmes conduit à un résultat conforme au problème global initial.

Cette décomposition, nous donne une hiérarchie entre le problème initial et les différents sous-problèmes. On obtient ainsi un diagramme hiérarchique (arbre) sur lequel chaque nœud correspond à un sous-problème bien défini et facile à résoudre. Pour chacun de ces sous-problèmes (nœuds), la solution s'exprime par des actions (instructions) qui peuvent s'énoncer directement et sans ambiguïté. Ce diagramme hiérarchique ainsi formé est la solution du problème.

La solution algorithmique au problème donné est alors l'ensemble des solutions algorithmiques de chaque sous-problème dans l'ordre établi par l'arborescence (*figure 4-1*). Dans notre approche, chaque ligne de la solution algorithmique représente un sous-problème ou un nœud du diagramme hiérarchique. Dans la suite de ce document, nous utiliserons le terme arbre pour le diagramme hiérarchique.

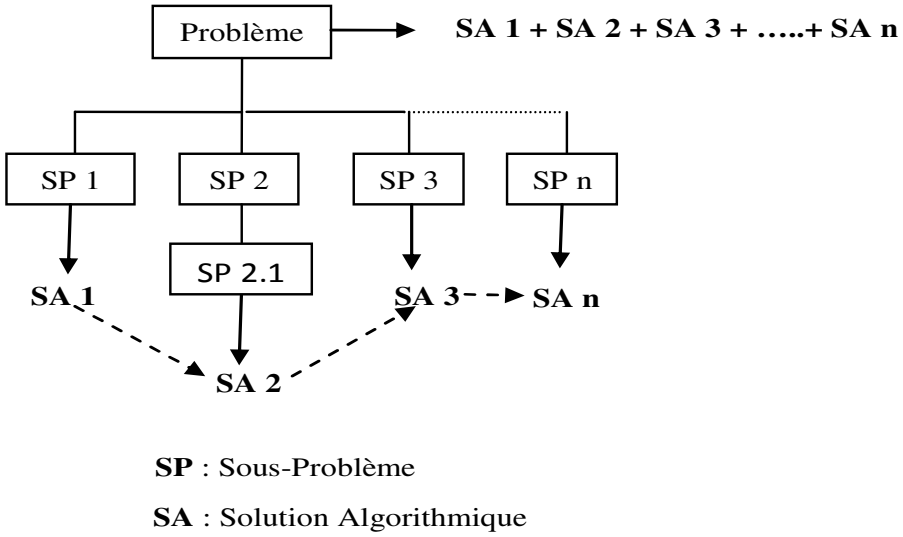


Figure 4-1: Décomposition d'un problème en sous-problèmes

4.2.1 Exemple de décomposition d'un problème

Prenons l'exemple suivant (Exemple 1) pour illustrer cette décomposition :

> ***Exemple 1 :*** *Écrire un algorithme, qui demande successivement des nombres, et qui ensuite, calcule et affiche la somme des nombres saisis. Il affiche également, combien il y a de nombres pairs parmi les nombres saisis. La saisie des nombres s'arrête, lorsque l'utilisateur saisit un zéro.*

Dans cet exemple, le problème peut être décomposé par l'arbre présenté *figure 4-2*. Par cette décomposition, nous obtenons des sous-problèmes (sur fond bleu), pour lesquels la solution (instructions sur fond jaune), peut s'écrire facilement et sans ambiguïté. Sur cet exemple, chaque ligne d'instruction est bien représentée par un nœud de l'arbre. Afin de diagnostiquer les erreurs des apprenants, nous associerons à chacun des nœuds une *étiquette*. Cette dernière correspondra à une catégorisation de la ligne d'instruction. Nous obtiendrons ainsi un *arbre étiqueté* dont tous les nœuds seront représentés par une étiquette correspondant au type d'action réalisé.

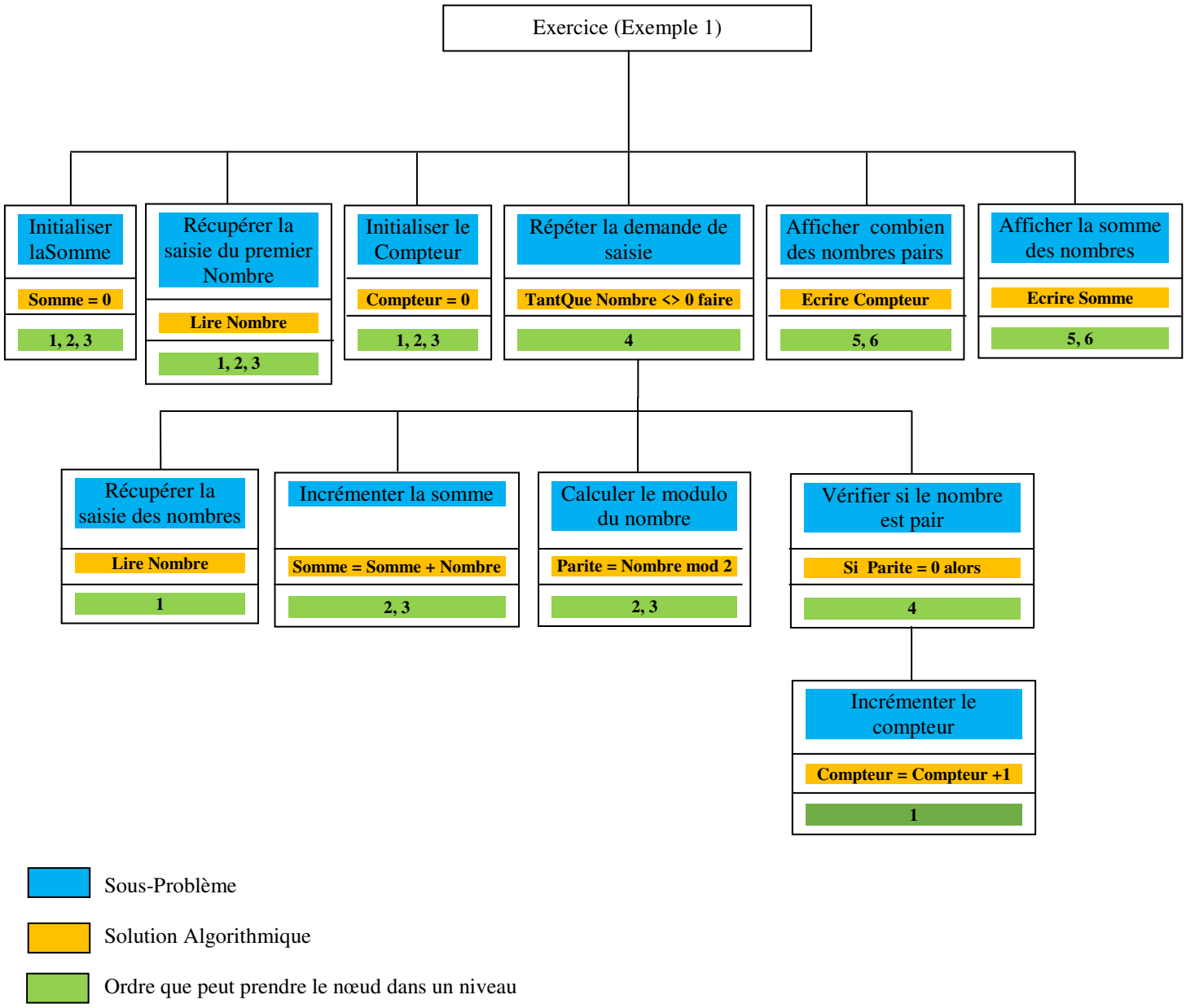


Figure 4-2: Diagramme hiérarchique après décomposition de l'exercice exemple 1

4.2.2 Les différentes étapes de notre processus d'évaluation

Nous rappelons qu'évaluer consiste à « *mettre en relation des éléments issus d'un observable appelé référé et un référent pour produire de l'information éclairante sur le référé afin de prendre de décisions* » (Hadji, 1990). Dans notre approche le ***référé*** est une solution modèle, préparé par l'expert (enseignant) et le ***référant*** c'est la solution de l'apprenant. Pour évaluer la solution de l'apprenant, nous procédons en deux étapes (*figure 4-3*) : (i) dans une

première étape, nous harmonisons les solutions algorithmiques (apprenant et modèle), en effectuant une transformation de ceux-ci en *arbres étiquetés* afin de permettre une comparaison. Le fait d'étiqueter, nous permet de donner une signification à chaque ligne de la solution algorithmique; (ii) dans une deuxième étape nous comparons les solutions ainsi standardisées afin de vérifier si les sous-problèmes attendus (représentés par une étiquette) ont été mis en œuvre et dans l'ordre.

Après cette description générale de notre approche d'évaluation, nous allons maintenant présenter comment nous construisons *l'arbre étiqueté* avant d'aborder l'étape d'évaluation proprement dite.

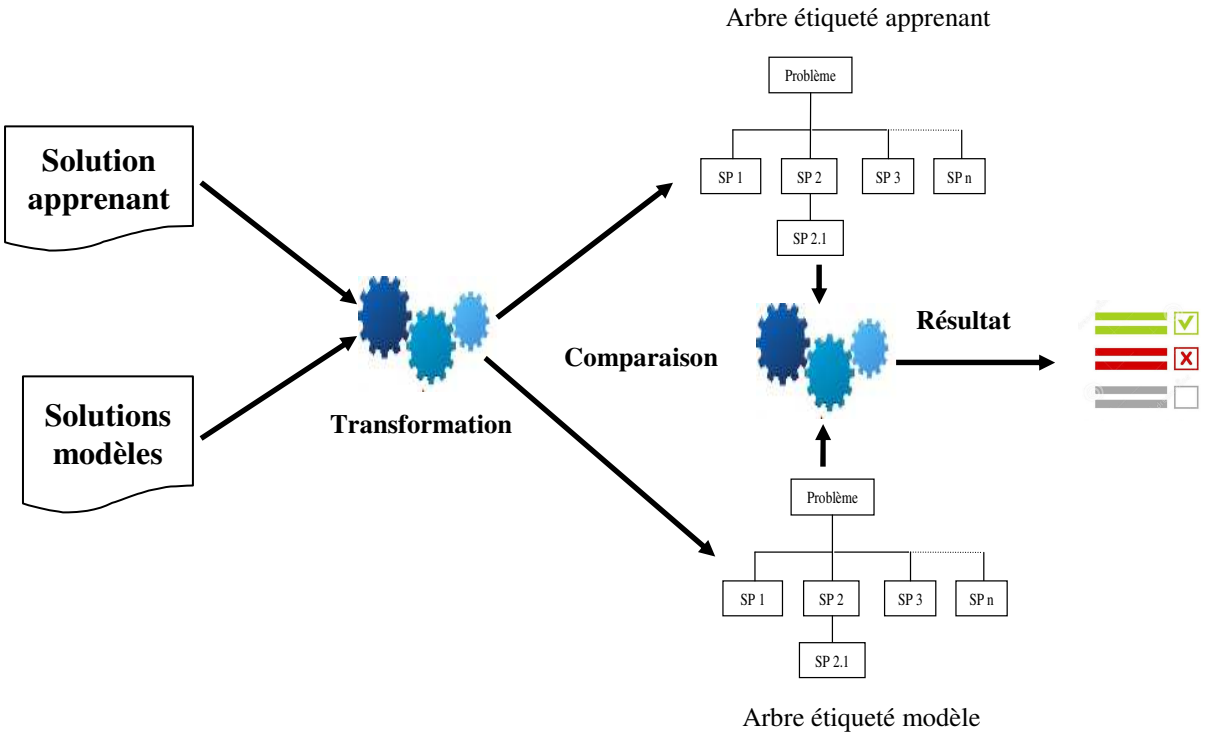


Figure 4-3: Processus d'évaluation

4.3 Transformation des solutions algorithmiques en arbres étiquetés

Notre évaluation est basée sur la comparaison de solutions modèles à chacune des solutions algorithmiques construites par les apprenants pour un exercice donné. Pour effectuer cette comparaison il faut que les solutions (modèle et apprenant) passent par une étape de standardisation. Cette étape correspond à la transformation des solutions algorithmiques en des arbres étiquetés.

Une solution algorithmique se présente comme une suite d'*instructions ordonnées*. Elle est rédigée en pseudo-code, langage naturel avec des notations formelles. Cette dernière est composée de deux parties : (i) une partie déclaration des variables et (ii) une partie solution algorithmique, composée des différentes instructions nécessaires. Cette dernière est délimitée par un ***Début*** et un ***Fin***. Chaque instruction joue un rôle dans la résolution du problème et l'ordre des instructions est important. Nous présentons maintenant les différentes catégories d'instructions. Ensuite, nous décrivons comment nous affectons les étiquettes et enfin, nous détaillons comment nous comparons les arbres étiquetés.

4.3.1 Catégories d'instructions

Une solution algorithmique est composée des différentes lignes d'instructions délimitées par un **Début** et une **Fin**. Chaque ligne d'instructions, peut être composée de :

- ***Opération Élémentaire (OE)*** : représente des instructions, qui permettent d'informer ou de récupérer des données, comme *Ecrire* (afficher une information), *Lire* (récupérer une valeur saisie), *Affecter* (affecter une valeur à une variable), etc.
- ***Structure Conditionnelle (SC)*** : représente des instructions qui permettent de vérifier une condition pour exécuter une action, comme *si, selon*…
- ***Structure Itérative (SI) :*** représente des instructions, qui permettent d'exécuter une action ou une suite d'actions autant de fois que nécessaire, comme *Pour, Tantque, Répéter*.
- ***Opération Arithmétique (OA)*** : ce sont des opérations de calcul, comme l'addition, la soustraction, la division, la multiplication, le modulo, etc.
- ***Opération de Comparaison (OC)*** : représente des opérations, qui permettent de comparer le contenu de deux variables ou le contenu d'une variable à une valeur (*supérieur (>), inférieur (<), égal (==), différent (<>)…).* Elles sont utilisées dans des instructions ou il y a une structure conditionnelle ou une structure itérative.
- ***Opérateur Logique (OL)*** : ces opérations (*et, ou)*, sont utilisées, dans les structures conditionnelles ainsi que dans les structures itératives afin de combiner différentes conditions.

Ces différentes catégories permettront de composer les différentes lignes d'instruction d'une solution algorithmique. Elles serviront à l'affectation des étiquettes de chacune des

lignes d'instructions pour construire l'arbre étiqueté. Nous présentons, dans ce qui suit, les traitements que nous effectuons sur ces solutions algorithmiques afin d'affecter une étiquette à chaque nœud de cet arbre.

4.3.2 Étiquetage des nœuds de l'arbre

Pour recomposer la solution algorithmique avec les opérations citées ci-dessus, en un arbre étiqueté ou chaque nœud représentera un sous-problème, nous nous basons sur les instructions que comprennent la solution et procédons comme suit :

- Chaque ligne de la solution correspond à un sous-problème. Ce dernier sera représenté par un nœud étiqueté dans l'arbre. L'ordre de chaque nœud dans l'arbre correspond à sa position dans la solution. En ce qui concerne la solution modèle fournie par l'enseignant, si plusieurs ordres sont possibles, il devra les définir (voir exemple *figure 4-2*, l'ordre est sur fond vert).
- Pour affecter l'étiquette à un nœud, nous nous basons sur les instructions que comprennent chaque ligne d'instruction (composée d'une opération ou d'un ensemble d'opérations) et une base de règles (Si *Condition* alors *Etiquette*). Par exemple, si une ligne d'action contient une structure conditionnelle alors l'étiquette retournée sera : *Vérifier.*

Une règle est composée de deux parties : (i) les conditions, qui doivent être vérifiées pour appliquer la règle ; (ii) l'étiquette, associée au nœud, si la condition est remplie. Cette base de règles a été construite en concertation avec quatre enseignants du cours d'introduction à la programmation à l'université de Djibouti. La base de règles n'est pas exhaustive et peut évoluer. Un extrait de cette base définie avec les enseignants est donné dans le *Tableau 4-1*.

L'étiquette à affecter, grâce à cette base de règles doit être significative car elle nous permettra d'associer une rétroaction fournie à l'apprenant suite à la détection d'une erreur. Cela sera détaillé dans le chapitre suivant.

Étiquette	Condition
Incrementation	Si la ligne est composée d'une variable, un *OE (affectation)*, la même *variable que celle identifiée premièrement*, un *OA(+)* et une valeur.
Decrementation	Si la ligne est composée d'une variable, un *OE (affectation)*, la même *variable que celle identifiée premièrement*, un *OA(-)* et une valeur.
Initialisation	*Si la ligne est composée d'une variable,* un *OE (affectation)* et une valeur.
Calcul	*variable* avec un *OE (affectation)* suivi d'autre *variable* ou valeur, ses variables et valeur sont séparés par un *OA*.
Verifie	Si la ligne contient une *SC*.
Repeter	Si la ligne contient une *SI.*
Donnee	Si la ligne contient un *OE (Lire)*
Afficher	Si la ligne contient un OE *(Ecrire)*

Tableau 4-1: Exemple de la base de règles

Pour construire l'arbre étiqueté, nous parcourons l'ensemble des lignes d'instructions décrites entre le *Début* et *Fin*. À la lecture de chaque ligne, nous appelons la base de règles qui nous retourne l'étiquette correspondant à cette ligne. Les étiquettes ainsi construites comprennent deux partie : (i) une étiquette construite grâce à la base de règles et (ii) le nom de la variable utilisée dans la construction de l'instruction.

Par exemple, pour la solution algorithmique de l'exemple 1 (*figure 4-2*), nous obtenons l'arbre étiqueté *figure 4-4*. Pour l'étiquette *verifeParite* : *Verifie* est construite par la présence de la *SC* dans l'instruction et *Parite* est le nom de variable utilisée par la *SC*.

Dans cet exemple, les sous-problèmes attendus sont :

- ***Au niveau 1*** : (i) initialiser la somme représentée par le nœud *InitialisationSomme* ; (ii) récupérer le nombre saisis représenté par le nœud *DonneeNombre* ; (iii) initialiser le compteur de nombres pairs représenté par le nœud *InitialisationCompteur ;* (iv) la répétition de la demande de saisie, représenté par le nœud *repeterNombre* ; (v) afficher combien il y a de nombres pairs représenté par le nœud *AfficherCompteur ;* (vi) afficher la somme des nombres représenté par le nœud *AfficherSomme.* Dans ce niveau, au total six sous-problèmes sont attendus. L'enseignant a indiqué différents ordres possibles pour les nœuds *InitialisationSomme, DonneeNombre, InitialisationCompteur* ainsi que pour les nœuds *AfficherCompteur* et *AfficherSomme*.

- ***Au niveau 2*** : (i) récupérer les nombres saisis représenté par le nœud *DonneeNombre* ; (ii) incrémenter la somme représenté par le nœud *IncrementationSomme* ; (iii) calculer le modulo du nombre saisi représenté par le nœud *CalculParite ;* (iv) vérifier si le nombre est pair représenté par le nœud *VerifieParite*. Dans ce niveau, au total quatre sous-problèmes sont attendus et l'enseignant a souligné que l'ordre des nœuds *CalculParite* et *IncrementationSomme* peuvent varier.
- ***Au niveau 3 :*** incrémenter le compteur des nombres pairs, représenté par le nœud *IncrementationCompteur*. Dans ce niveau un seul sous-problème est attendu.

L'arbre étiqueté, ainsi construit, nous détaillons dans la suite, comment nous comparons les arbres (modèle et apprenant) et comment cette comparaison nous permet de détecter les erreurs des apprenants en vue de leur fournir une assistante pertinente.

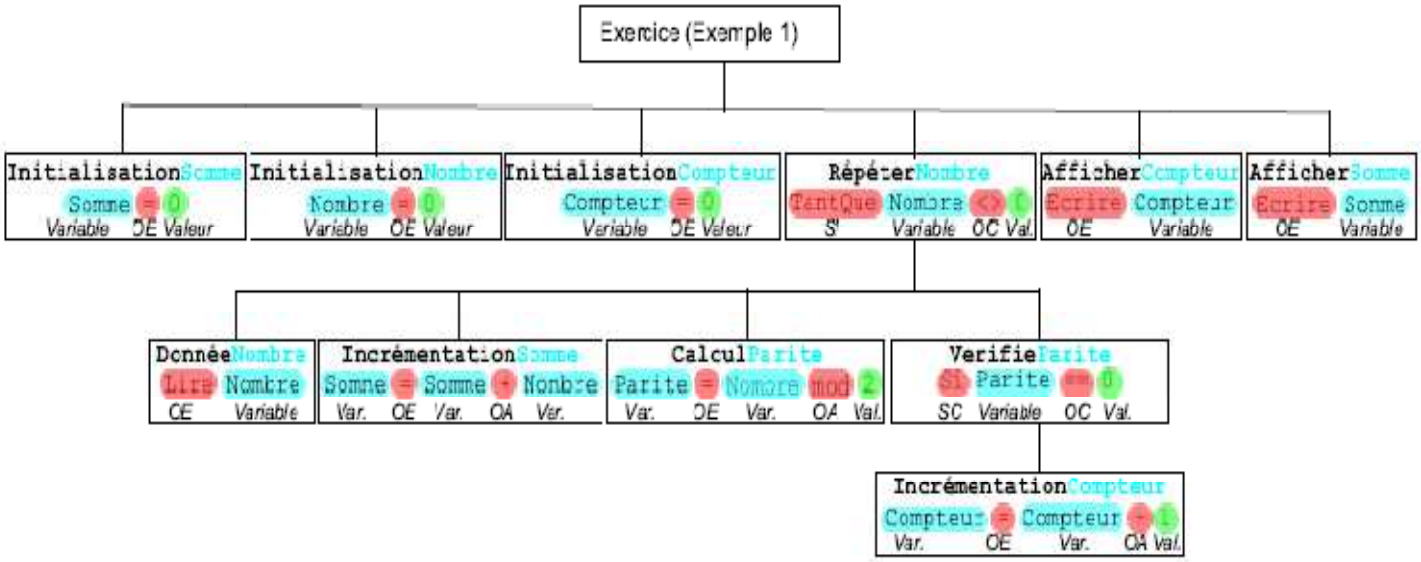


Figure 4-4: Arbre étiqueté de la solution modèle (Figure 4-2)

4.4 Comparaison de deux arbres étiquetés

Dans cette approche d'évaluation, l'objectif est de vérifier la cohérence des instructions écrites pour résoudre un problème donné. À cet effet, nous recherchons dans la solution apprenant :

- Si certaines lignes d'instructions attendues n'ont pas été mises en œuvre.

- Si certaines lignes d'instructions attendues ont été mises en œuvre, mais pas dans le bon niveau de l'arbre. Par exemple si un apprenant met une structure conditionnelle dans une boucle, alors que cette instruction est attendue à l'extérieur de la boucle.
- Si certaines lignes d'instructions attendues, dans un niveau, ont été mises en œuvre, mais pas dans le bon ordre. Par exemple, si dans un problème donné, il est attendu de mettre à jour une variable avant d'écrire une structure conditionnelle et que l'apprenant écrit la structure conditionnelle avant la mise à jour de la variable.
- Si certaines lignes d'instructions supplémentaires (non attendues), ont été mises en œuvre.

Pour détecter les anomalies citées ci-dessus, nous effectuons une comparaison entre la solution modèle et la solution apprenant avec l'algorithme suivant appliqué à chaque nœud de l'arbre en commençant au niveau 1 :

- ***Étape 1 :*** Nous récupérons, les nœuds (sous-problèmes) du niveau courant de chaque arbre (modèle et apprenant).
- ***Étape 2 :*** Nous comparons les nœuds ainsi récupérés. Pour chaque nœud (sous-problème) de l'arbre modèle, représentant les sous-problèmes attendus :
    - Si le nœud (sous-problème) attendu a été implémenté dans l'arbre représentant la solution algorithmique de l'apprenant, nous affectons la ***valeur 1*** à la variable ***resultatSA*** (représentant le résultat du sous-problème attendu). La ***valeur 0***, dans le cas contraire.
    - Si le nœud (sous-problème) attendu a été implémenté dans le bon ordre, nous affectons la ***valeur 1*** à la variable ***resultatOA*** (représentant le résultat de l'ordre attendu). La ***valeur 0***, dans le cas contraire.
    - Si des nœuds (sous-problèmes) non attendus sont présents dans l'arbre représentant la solution algorithmique de l'apprenant, alors nous affectons le nombre de ces nœuds supplémentaires à la variable ***Splus*** (représentant le nombre des sous-problèmes non attendus implémentés dans la solution de l'apprenant).

  Dans cette ***étape 2***, si les deux nœuds comparés possèdent des enfants (sous-nœuds), nous répétons l'***étape 1***.

Suite à cette comparaison, nous calculons deux indicateurs : (i) le premier va calculer le taux de réussite (*TR*). Cet indicateur, présenté en **(1)**, permet d'évaluer, à quel point l'apprenant a mis en œuvre les sous-problèmes attendus et dans l'ordre attendu. Nous associons un niveau de réussite à l'exercice (*Faible, Moyen ou Bon*) en fonction de cet indicateur. Cela nous sera utile dans la phase de génération de rétroaction, que l'on détaillera dans le chapitre suivant ; (ii) le deuxième indicateur quant à lui, indiquera le taux de sous-problèmes supplémentaires (*TSP*) trouvés dans la solution apprenant. Cet indicateur, présenté en **(2),** est utile, dans le cas où il est élevé. Cela peut révéler soit une grande difficulté de l'apprenant, soit une solution correcte qui n'avait pas été prévue par l'enseignant. Le système va alors soumettre la solution à l'enseignant pour vérification.

$$TR = \frac{\sum_{i=1}^{n} resultatSA * resultatOA}{NSA} \qquad \textbf{(1)}$$

$$TSP = \frac{Splus}{NSA} \qquad \textbf{(2)}$$

Ou :

resultatSA : Résultat (0 ou 1) du sous-problème attendu, pour le i^{ème} sous-problème attendu.

resultatOA : Résultat (0 ou 1) de l'ordre du sous-problème attendu, pour le i^{ème} sous-problème attendu.

Splus : Nombre de sous-problèmes supplémentaires.

NSA : Nombre de sous-problèmes attendus.

i = 1,2…..N : N est le numéro du sous-problème attendu.

L'algorithme suivant résume le processus de comparaison de deux arbres étiquetés décrits précédemment :

Algorithme de comparaison de deux arbres

Entrée : *arbre_apprenant ; arbre_modèle*

Début

Nœud = racine

Explorer ***(Nœud)***

Calcul du taux de réussite.

Calcul du taux de sous-problèmes supplémentaires.

Procédure Explorer (nœud)

Récupère la liste des éléments du nœud de l'arbre apprenant.

Récupère la liste des éléments du nœud de l'arbre modèle.

Comparer (liste_elements_arbre_apprenant, liste_elements_arbre_modèle, nœud)

Pour *chaque élément de l'arbre modèle* **faire**

Pour *chaque élément de l'arbre apprenant* **faire**

Si (élément_arbre_modèle=élément_arbre_apprenant et que les deux éléments ont des enfants**) Alors**

Nœud=element_arbre_modèle

Explorer (Nœud)

FinSi

FinPour

FinPour

Procédure Comparer (liste_elements_arbre_apprenant, liste_elements_arbre_modele, nœud)

resultatSA et resultatOA est par défaut 0 pour tous les élément de l'arbre modèle.
ordre = 1 // ordre de l'instruction courante dans la solution apprenant
Splus = 0

Pour *chaque élément de l'arbre apprenant* **faire**

trouve = 0

Pour *chaque élément de l'arbre modèle* **faire**

Si (élément_arbre_modèle = élément_arbre_apprenant**) Alors**

trouve=1

resultatSA pour l'élément de l'arbre modèle est mis à 1

Si (ordre attendu de l'élément arbre modèle = ordre de l'élément arbre apprenant**) Alors**

resultatOA pour l'élément de l'arbre modèle est mis à 1.

FinSI

FinSi

FinPour

Si (trouve = 0**) Alors**

Splus = Splus + 1

FinSi

o*rdre = ordre+1*

FinPour

4.5 Synthèse

Le travail exposé dans ce chapitre, concerne une technique d'évaluation des solutions algorithmiques. Dans cette technique nous avons proposé une évaluation basée sur la comparaison d'une solution modèle à une solution apprenant. Pour cela, nous proposons un traitement en deux étapes. Dans une première étape nous harmonisons les solutions algorithmiques (modèle et apprenant) en les transformant en des arbres étiquetés. Cette étiquette est affectée à chaque nœud grâce à une base de règles conçue avec des enseignants du cours d'introduction à la programmation de l'université de Djibouti. Dans la deuxième étape nous effectuons une comparaison des arbres ainsi obtenus. Cette comparaison, nous permet d'identifier et d'énumérer les erreurs conceptuelles de la solution apprenant.

La technique d'évaluation ainsi proposée permet de diagnostiquer les erreurs des apprenants sans s'appuyer sur une définition préalable des solutions modèles comportant des erreurs. Elle permet également de réduire les nombres de solutions modèles. Par ailleurs, grâce à l'étiquetage nous sommes en mesure de donner un sens aux erreurs commises par l'apprenant. Cela nous permettra de générer les rétroactions, ce que nous allons présenter dans le chapitre suivant.

Chapitre 5 : Sélection des exercices et génération des rétroactions

Contenu

5.1 Introduction

Dans le chapitre précédent, nous avons détaillé notre approche d'évaluation des solutions algorithmiques. Nous avons décrit comment cette approche nous permettait de qualifier la solution algorithmique de l'apprenant comme solution correcte ou incorrecte. Dans ce dernier cas, notre approche identifie et énumère les erreurs commises.

Ce chapitre présente les contributions relatives à l'assistance destinées à consolider les connaissances stratégiques des apprenants lors des cours d'introduction à la programmation. Il s'appuie sur l'état de l'art pour élaborer deux propositions : (i) un modèle d'organisation des exercices, associé à un mécanisme de sélection des exercices proposés à l'apprenant (ii) un mécanisme de rétroaction pour assister les apprenants qui ont des difficultés. Ces deux mécanismes s'appuient sur le fruit de l'évaluation de la solution apprenant, vue au chapitre précédent. Dans un premier temps, nous abordons le modèle général d'organisation des exercices à présenter aux apprenants, ensuite nous présentons le mécanisme de sélection des exercices à résoudre. Dans un deuxième temps, nous présenterons le mécanisme de génération des rétroactions qui permet d'accompagner l'apprenant qui rencontre des difficultés dans la résolution de l'exercice soumis.

5.2 Modèle d'organisation des exercices

Pour améliorer la capacité de résolution de problèmes des apprenants, la littérature recommande de les confronter à de nombreux problèmes avec des niveaux de difficultés croissants. Ce modèle vise à organiser les exercices à soumettre aux apprenants de manière à sélectionner les exercices selon ce principe.

Pour proposer ce modèle, nous nous appuyons sur deux éléments : d'une part, nous nous inspirons de l'approche de l'apprentissage par la maîtrise (« *Mastery learning* », présentée dans le chapitre 3). Cette approche stipule que ***l'apprenant doit démontrer qu'il a atteint un niveau de maîtrise approprié sur un sujet donné avant de passer à un sujet plus avancé*** (Luxton-Reilly et al., 2018). Les concepts ou compétences à faire travailler à l'apprenant sont organisés en ***unités d'apprentissage***. Le passage entre les unités repose sur une évaluation formative.

D'autre part, dans le cadre de l'apprentissage de la programmation, il est souligné que l'apprenant doit maîtriser les concepts de programmation de base avant d'aborder des concepts plus complexes (Luxton-Reilly et al., 2018). Ces chercheurs ont également souligné

qu'il est nécessaire de soumettre aux apprenants des exercices vérifiant la maîtrise de l'utilisation d'un concept avant de leur proposer des exercices plus complexes invoquant plusieurs concepts. Cela permet de renforcer leurs *connaissances stratégiques*.

Ainsi, en se basant sur ces deux éléments, notre modèle se présente comme suit :

- Pour respecter la contrainte d'organisation des activités par unité d'apprentissage, et faire progresser l'apprenant d'une unité à l'autre, nous utilisons la taxonomie de Bloom du domaine cognitif (voir chapitre 3). Dans cette taxonomie, les niveaux cognitifs sont organisés en six niveaux, de difficulté croissante. L'apprenant doit maîtriser les opérations d'un niveau avant de passer au niveau suivant.
- La contrainte suivante est que les concepts de programmation s'apprennent d'une manière progressive et qu'il faut maîtriser certains concepts avant d'aborder d'autres concepts plus difficiles. Pour cela, nous répartissons les concepts selon un niveau de difficulté. Pour chaque niveau cognitif, il y aura plusieurs niveaux de difficulté et dans chaque niveau de difficulté, nous abordons un ou plusieurs concepts de programmation.
- Dans chaque niveau de difficulté, plusieurs exercices sont proposés, cela a pour objectif de confronter les apprenants à des exercices de même famille. Il s'agit de les soumettre des exercices de même niveau de difficulté et mettant en œuvre les mêmes concepts de programmation afin de vérifier s'ils arrivent à faire l'analogie entre les exercices.
- Le passage d'un niveau cognitif à un autre est conditionné par la réalisation de l'ensemble des exercices de tous les niveaux de difficulté d'un niveau cognitif. Pour cela, chaque exercice sera évalué. Cette évaluation sera formative (chapitre 2). Dans le cadre de nos travaux, le but de l'évaluation est de fournir à l'apprenant qui a des difficultés des rétroactions pour améliorer sa solution.

Ainsi, nous proposons un modèle d'exercice à différent niveaux (*figure 5-1*). Cela nous permet d'avoir une collection d'exercices préparés par les enseignants selon ce modèle. Nous détaillons dans ce qui suit les différents niveaux de ce modèle ainsi que l'algorithme de sélection de l'exercice suivant.

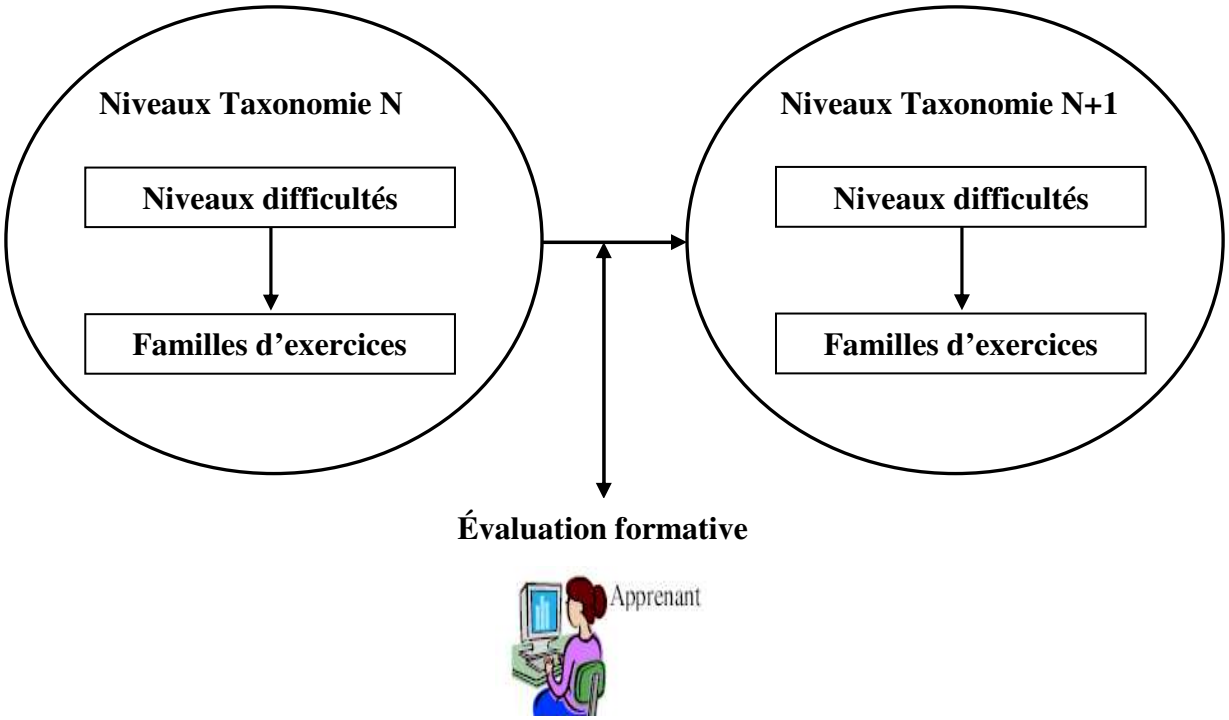


Figure 5-1: modèle d'organisation des exercices

5.2.1 Niveau de taxonomie de Bloom

Notre modèle d'organisation des exercices, nous permet d'avoir une collection d'exercices organisée mettant en œuvre les différents concepts de programmation à des degrés de difficulté différents. À cet effet, notre modèle répartit les exercices selon les niveaux cognitifs de la taxonomie de Bloom (présentée au chapitre 3). Dans le contexte de l'apprentissage de la programmation du cours d'introduction, nous allons nous limiter aux trois premiers niveaux (*figure 5-2*) : (i) ***connaissance*** : dans ce niveau, les exercices proposés ont pour objectif de faire découvrir et assimiler l'utilisation des différents concepts de la programmation. On aura, par exemple, des exercices qui font découvrir aux apprenants l'utilisation des concepts suivant : *afficher / récupérer des données, structure conditionnelle, les différentes structures répétitives…* ; (ii) ***compréhension*** : dans ce niveau, les exercices proposés permettent de vérifier, si l'apprenant a bien compris les différents concepts de programmation acquis dans le niveau connaissance. Il s'agit alors de fournir des exercices qui vérifient, si les apprenants ont bien compris ces différents concepts et arrivent à utiliser le concept approprié. On pourra, par exemple, donner des exercices qui vérifient si l'apprenant arrive à faire la différence entre l'utilisation des concepts des boucles `Pour` et `TantQue` ; (iii) ***Application*** : dans ce niveau, il s'agit de proposer des exercices invoquant plusieurs concepts à la fois. Cela permet de vérifier si les apprenants arrivent à mobiliser les différents

concepts nécessaires pour résoudre l'exercice. On aura, par exemple, un exercice qui vérifie, si l'apprenant arrive à utiliser ensemble les concepts *structure conditionnelle* et *structure répétitive* ou *structure conditionnelle*, *répétitive* et *tableau*.

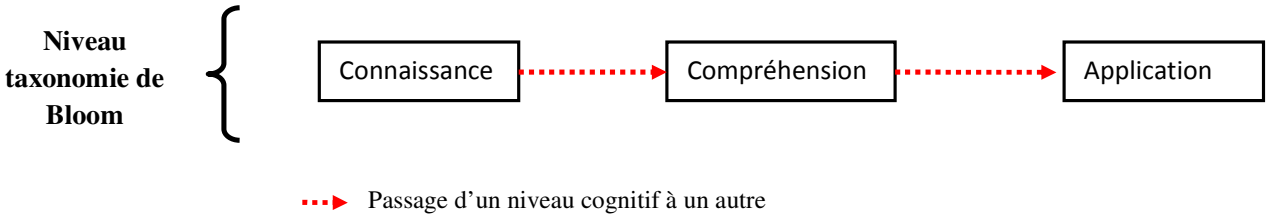


Figure 5-2: Affecter un exercice à un niveau de taxonomie de Bloom

5.2.2 Organisation en niveaux de difficulté

Dans la section précédente, nous avons réparti les exercices en trois niveaux cognitifs croissants et dans chaque niveau cognitif, les exercices proposés mettent en œuvre un ou plusieurs concepts. Ces concepts ont des complexités différentes et la maîtrise des uns facilite l'acquisition des autres. Nous allons donc répartir les exercices selon des niveaux de difficulté en fonction des concepts qu'ils invoquent, cela pour chaque niveau de la taxonomie de Bloom (*figure 5-3*). Par exemple, pour le niveau connaissance de la taxonomie de Bloom le premier niveau de difficulté correspond à des exercices qui vérifient l'utilisation de l'*affichage*, la *récupération de données dans des variables* et l'*utilisation des variables*. Dans un deuxième niveau on trouvera des exercices qui mettent en œuvre la *structure conditionnelle*, dans un troisième niveau des exercices qui mettent en œuvre des *structures conditionnelles imbriquées*…

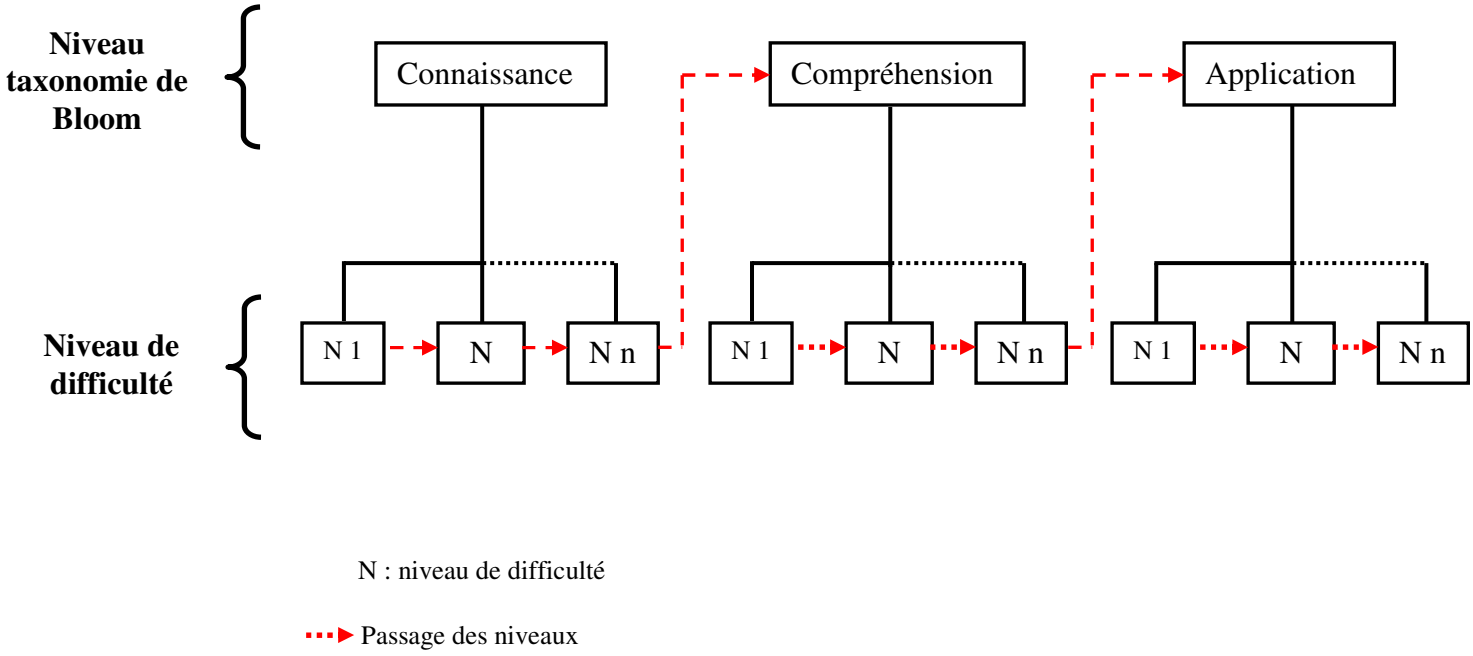


Figure 5-3: Les différents niveaux de difficultés selon les concepts abordés, dans chaque niveau de taxonomie

5.2.3 Organisation en familles d'exercices

Dans la section précédente on vient de présenter différents niveaux de difficulté pour chaque niveau de la taxonomie. Ces niveaux de difficulté sont classés, selon le(s) concept(s) qui seront manipulés dans l'exercice. Le rôle de ce niveau est de proposer différents exercices pour le même niveau de difficulté. En d'autres mots, il s'agit de spécifier un ensemble d'exercices pour le(s) concept(s) abordé(s) dans les différents niveaux de difficulté (*figure 5-4*). L'objectif est de renforcer la capacité des apprenants à faire les analogies entre les différents exercices. Aboutir à cette capacité d'analogie est une difficulté soulignée dans la capacité de résolution chez les apprenants (voir chapitre 1).

Le modèle ainsi défini, permet à l'apprenant de résoudre des exercices correspondants à différents niveaux cognitifs. Nous présentons dans ce qui suit la sélection des exercices à soumettre à l'apprenant ainsi que les assistances fournies sous forme des rétroactions pour ceux qui n'arrivent pas à résoudre l'exercice.

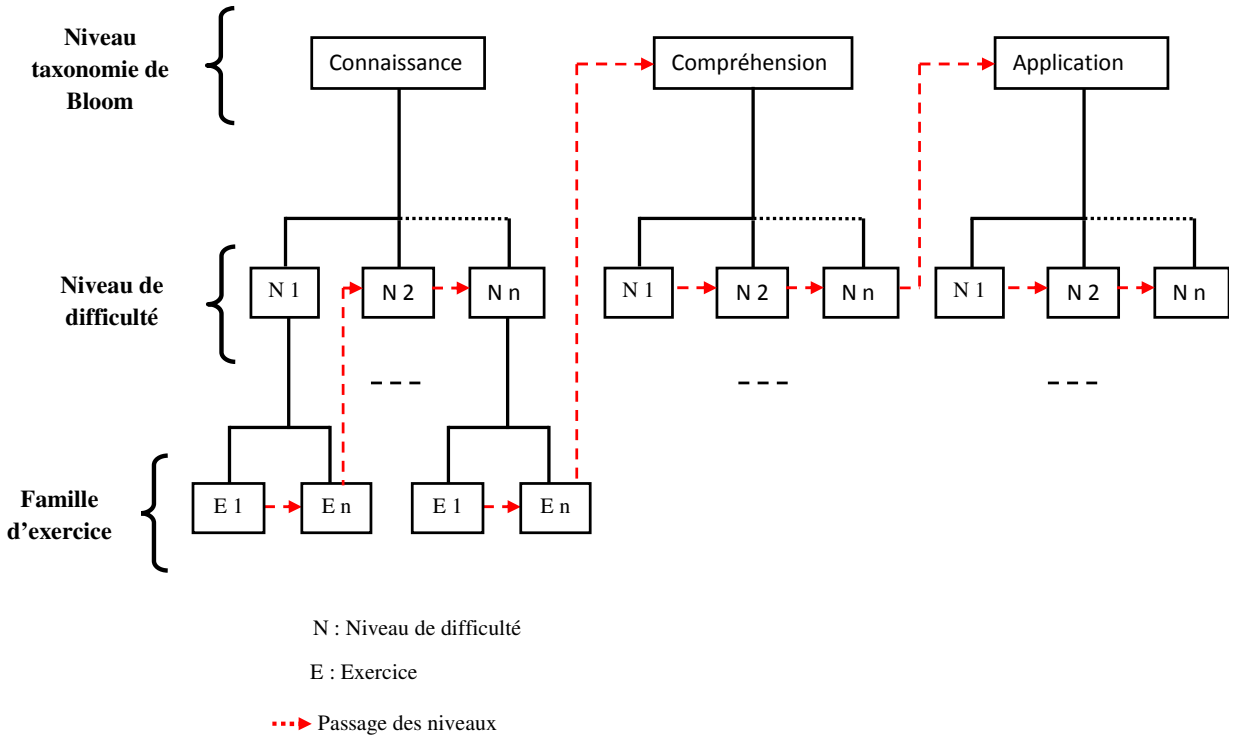


Figure 5-4: Paramétrage niveau famille de problèmes.

5.3 Sélection de l'exercice suivant

Nous avons défini, dans la section précédente, un modèle d'organisation des exercices à soumettre à l'apprenant. Pour sélectionner l'exercice suivant, sur la base de ce modèle, nous nous basons sur l'historique des exercices résolus par l'apprenant. Chaque exercice correspond à un niveau de difficulté et un niveau de taxonomie donné. Le résultat de l'évaluation nous renvoie un état de résolution (voir section 4.4) qui peut être :

- ***Faible*** : indique que l'apprenant a commis beaucoup d'erreurs, ce que nous interprétons comme le fait que l'apprenant n'a pas compris l'exercice. Dans ce cas l'apprenant obtient une rétroaction qui lui explique l'exercice avant de faire un nouvel essai ;
- ***Moyen*** : indique que l'apprenant a commis des erreurs minimes qu'il peut corriger. Dans ce cas l'apprenant obtient une rétroaction descriptive de ses erreurs avant de reprendre l'exercice.
- ***Bon*** : indique que l'apprenant a bien résolu l'exercice et qu'il peut passer à l'exercice suivant. Dans ce cas, la sélection de l'exercice suivant est donnée par l'algorithme ci-après.

L'algorithme présenté ci-après nous permet, sur la base de l'état du dernier exercice, soit d'afficher la rétroaction adaptée et inviter l'apprenant à refaire l'exercice, soit de lui soumettre l'exercice suivant.

Algorithme de sélection des exercices

Entrée : *État du dernier exercice traité*

Début

Si (*l'apprenant est nouveau*) **Alors**

Exercice_suivant est le premier exercice du premier niveau de difficulté du niveau de taxonomie Connaissance.

Sinon

Si (État est « Faible ») **Alors**

Afficher une rétroaction expliquant comment résoudre l'exercice

Exercice_suivant *est l'exercice en cours*

SinonSi (État est « Moyen ») **Alors**

Afficher une rétroaction expliquant comment résoudre l'exercice

Exercice_suivant *est l'exercice en cours*

Sinon

Exercice_suivant *est l'exercice suivant appartenant au même niveau de difficulté et au même niveau de taxonomie*

Si (aucun exercice n'est trouvé) **Alors**

Exercice_suivant *est le premier exercice du niveau de difficulté suivant dans le même niveau de taxonomie.*

Si (aucun exercice n'est trouvé) **Alors**

Exercice_suivant *est* le premier exercice du premier niveau de difficulté dans le niveau de taxonomie suivant.

Si (aucun exercice n'est trouvé) **Alors**

Informer à l'apprenant que tous les exercices sont résolus.

FinSi

FinSi

Finsi

FinSi

5.4 Génération des rétroactions

Les rétroactions ont pour objectif de permettre aux apprenants d'atteindre l'objectif fixé pour un exercice donné. Les rétroactions que nous mettons en œuvre sont de type *élaboration* (voir chapitre 3). Il s'agit de fournir des indices ou des explications destinés à accompagner l'apprenant dans la réalisation de l'exercice.

Dans le cadre de la résolution d'un exercice, un apprenant peut être confronté principalement à deux types de problèmes : (i) un manque de compréhension de l'exercice qui ne lui permet pas d'élaborer une solution cohérente ou (ii) une difficulté à construire une solution correcte par manque de maîtrise de concepts algorithmiques. Nous avons donc élaboré un mécanisme de rétroaction pour prendre en compte ces deux cas.

Nous détaillons dans ce qui suit, dans un premier temps, une présentation du principe de génération des rétroactions, ensuite nous abordons comment ces deux cas sont pris en charge.

5.4.1 Principe de génération des rétroactions

Dans le *chapitre 5*, nous avions détaillé notre approche d'évaluation. Cette approche permet d'identifier et d'énumérer les erreurs de l'apprenant. Ces erreurs sont interprétables grâce aux étiquetages. Nous avons également décrit dans ce chapitre, le calcul de taux de réussite. Ce dernier est un indicateur, qui nous informe sur le type de difficulté de l'apprenant :

- Un problème de compréhension de l'exercice se traduit par un faible taux de réussite (inférieur à 51%)[3]. Dans ce cas nous fournissons à l'apprenant une reformulation de l'exercice. Cette idée est appuyée par (Bosc-Miné, 2014), qui souligne que « *le fait de ne pas réussir une tâche peut être expliqué par une mauvaise représentation des consignes. Le guidage de l'action peut alors être axé sur la compréhension de celles-ci au moyen de rappel des règles ou de la formulation différente de celles-ci.* ». ***Le contenu de cette rétroaction est fourni par l'enseignant, lors du paramétrage des exercices.***
- Un problème de mise en œuvre de l'exercice est indiqué par un taux de réussite supérieur à 51%. Ce taux reflète un nombre d'erreurs limité pour lesquelles nous

[3] Ce seuil est utilisé dans le travail de (Chookaew et al., 2014) décrit dans la section 3.3.1.1

allons générer des rétroactions en fonction du type d'erreur et de l'étiquette correspondante.

Le principe de génération de nos deux types de rétroactions est illustré sur la *figure 5-5*. Nous avons indiqué ci-dessus que la rétroaction explicative (*cas 1 : figure 5-5*) est paramétrée par l'enseignant lors de la création de l'exercice.

Nous détaillons ci-dessous comment nous formulons le contenu de la deuxième forme de rétroaction (*cas 2 : figure 5-5*).

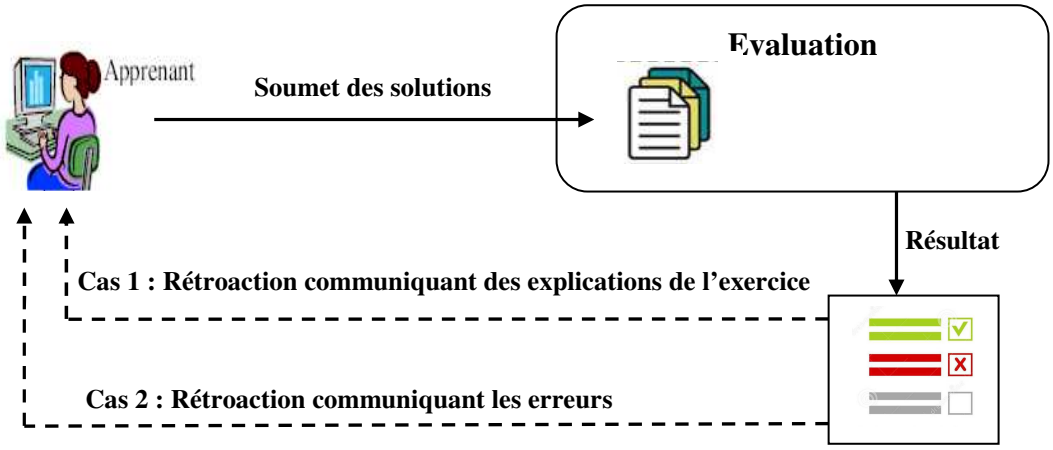


Figure 5-5: Processus de génération de rétroaction

5.4.2 Rétroactions communiquant les erreurs

La procédure de production du contenu de la rétroaction est basée sur deux éléments (*figure 5-6*) : (i) les étiquettes, dont la construction a été détaillée dans le chapitre précédent, et (ii) la nature de l'erreur qui peut être :

- Des instructions attendues non mises en œuvre ;
- Des instructions attendues non mises en œuvre au bon endroit ;
- L'utilisation d'un concept inapproprié.
- Des instructions supplémentaires.

La production du contenu de la rétroaction se construit alors comme suit, suivant le type d'erreur :

- **Instructions non mises en œuvre** : le contenu débute par, *tu as oublié de mettre en œuvre les instructions suivantes :* et pour énumérer ces instructions nous récupérons

dans le résultat de l'évaluation l'ensemble des instructions oubliées. Chacune des instructions est représentée par une étiquette qui les traduits. Ainsi pour interpréter les étiquettes, nous utilisons une base de règle qui est de la forme SI *condition* alors *phrase*. La *condition* vérifie le contenu de l'étiquette. Celui-ci permet de construire le contenu de la rétroaction. Par exemple si le contenu de l'étiquette est « *verifie* », nous informons l'apprenant qu'il a oublié de mettre en œuvre une *structure conditionnelle*.

- **Instruction mise en œuvre mais pas au bon endroit** : dans ce cas nous avons deux types d'erreurs possibles : soit l'apprenant a inversé l'ordre des certaines instructions, soit il a mis certaines instructions dans un mauvais niveau de l'arbre. Le contenu de la rétroaction débute :
    - Pour le premier cas, par la phrase : *Tu n'as pas mis dans le bon ordre les instructions suivantes :* et on énumère les instructions qu'il n'a pas mises dans le bon ordre selon le même principe que précédemment.
    - Pour le deuxième cas, par la phrase : *Tu n'as pas mis au bon endroit les instructions suivantes* : et on énumère les instructions qui n'ont pas été mises au bon endroit selon le même principe. Par exemple l'apprenant met une instruction d'incrémentation dans une structure conditionnelle au lieu de la mettre à l'extérieur de cette structure.
- **Utilisation d'un concept non approprié** *:* pour ce type d'erreur, le contenu de la rétroaction est : *tu as utilisé le concept x au lieu d'utiliser le concept y.*
- **Instruction supplémentaire** : le contenu débute par la phrase *les instructions suivantes sont inutiles :* et on énumère les instructions concernées.

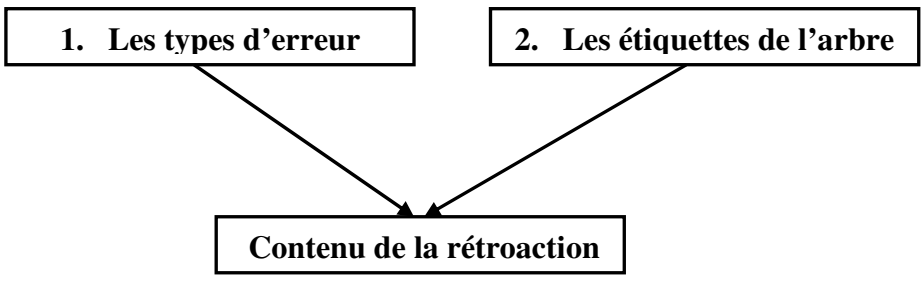


Figure 5-6: Procédure de la production du contenu de la rétroaction

5.4.3 Exemple de génération des rétroactions communiquant les erreurs

Pour illustrer, la génération de rétroactions communiquant les erreurs aux apprenants, nous reprenons l'exemple 1 du chapitre 4. Sur cet exercice, une solution possible de l'apprenant, après transformation peut être (*figure 5-7*).

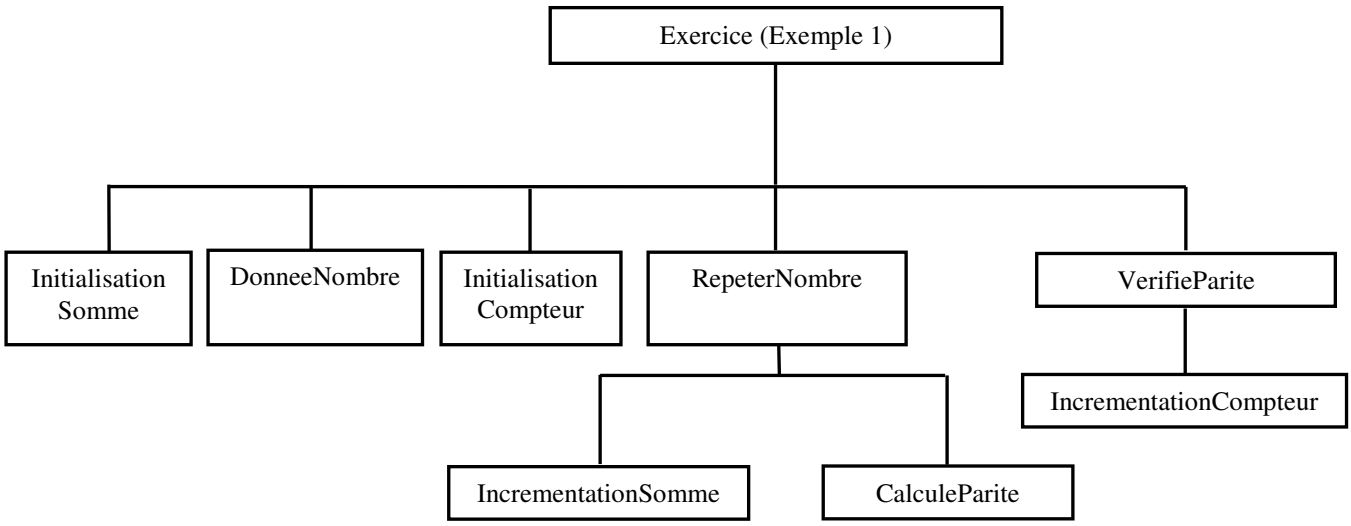


Figure 5-7: Arbre étiqueté d'une solution apprenant

En comparant cette solution apprenant à la solution modèle de la *figure 4-4*, nous observons que l'apprenant à oublié certaines instructions et qu'il n'a pas mis dans le bon niveau certaines instructions. Pour cela nous retournons à l'apprenant les rétroactions suivantes :

Tu as oublié de mettre en œuvre les instructions suivantes :

- Demander à l'utilisateur : Nombre, dans la boucle `TantQue` (généré grâce à l'étiquette *DonneeNombre*).
- Afficher : Somme (généré grâce à l'étiquette *AfficherSomme*)
- Afficher : Compteur (généré grâce à l'étiquette *AfficherCompteur*)

Tu n'as pas mis au bon endroit les instructions suivantes :

- La Structure Conditionnelle qui vérifie : Parite, (généré grâce à l'étiquette *VerifieParite*) tu devrais le mettre dans la boucle `TantQue` (généré grâce à

l'étiquette *RepeterNombre*, qui est le nœud dans lequel devrait se trouver l'instruction *VerifieParite*).

5.5 Synthèse

Notre objectif est de faire travailler les apprenants sur des exercices avec des niveaux de difficultés croissants afin de renforcer leurs connaissances stratégiques dans la conception des solutions algorithmiques. Pour cela nous avons proposé un modèle qui nous permet d'avoir une collection d'exercices organisée. En utilisant cette collection, nous pouvons soumettre des exercices à l'apprenant. Selon le résultat de l'évaluation, nous proposons à l'apprenant soit l'exercice suivant, soit une assistance par des rétroactions afin de le guider dans la résolution de l'exercice. La génération des contenus de ces rétroactions est basée sur les types d'erreurs et l'étiquetage fait lors de l'évaluation.

Dans le chapitre suivant, nous présentons la réification de nos propositions (approche d'évaluation, sélection des exercices et mécanisme de génération des rétroactions) dans un environnement Web appelé *AlgoInit.*

Chapitre 6 : Implémentation de l'environnement d'évaluation et d'assistance AlgoInit

Contenu

6.1 Introduction

Dans les deux chapitres précédents, nous avons présenté nos propositions concernant une approche d'évaluation, un mécanisme de génération des rétroactions pour assister les apprenants qui ont des difficultés et un modèle d'organisation des exercices. Ce chapitre présente la réification de nos propositions dans un environnement Web, que nous avons appelé *AlgoInit*. Il est élaboré pour soumettre des exercices aux apprenants, évaluer leurs réponses et assister, à travers la génération de rétroactions, les étudiants qui ont des difficultés à réaliser un exercice. L'objectif pédagogique de ce prototype est d'aider les apprenants à mieux utiliser les concepts de base de la programmation.

Ce chapitre a pour objectif de décrire *AlgoInit*. Il s'agit de présenter l'implémentation de différentes approches décrites dans les chapitres précédents : (i) l'approche d'évaluation, dans cette approche, nous présentons l'implémentation du parseur (transformation des solutions en des arbres étiquetés) et la comparaison des solutions ainsi harmonisées; (ii) nous présentons l'implémentation de la génération des rétroactions, l'organisation des différents exercices ainsi que la sélection de l'exercice à soumettre à l'apprenant.

Nous présentons dans ce chapitre, d'abord l'architecture générale, ensuite le développement d'*AlgoInit* et les outils utilisés pour le mettre en œuvre.

6.2 Présentation de l'architecture générale

AlgoInit est une application Web, organisée classiquement en trois niveaux (*figure 6-1*) :

- Le premier niveau, concerne l'interface utilisateur. Cette interface doit être accessible via différents navigateurs et se divise en 2 parties : (i) **une partie pour les apprenants**. Cette interface a pour rôle de permettre à l'apprenant de recevoir l'énoncé de l'exercice à résoudre, de rédiger la solution algorithmique, de la soumettre et de recevoir les différentes rétroactions dans le cas où la solution est incorrecte; (ii) **une partie pour les enseignants**. Cette interface, permet à l'enseignant de rédiger les différents exercices, les solutions modèles, la rétroaction (reformulation de l'énoncé) à afficher au cas où l'apprenant n'a pas compris l'exercice soumis. Il peut également paramétrer les différents niveaux (taxonomie et niveau de difficulté) de l'exercice.
- Le deuxième niveau, concerne les traitements. Il permet de traiter les requêtes envoyées par le client et se charge également de lui envoyer la réponse. Il est constitué

de deux modules : le module d'évaluation, qui est chargé de transformer les solutions en arbres étiquetés, de réaliser la comparaison des arbres afin de statuer sur la solution de l'apprenant ; le module de prise de décision, qui est chargé de sélectionner les exercices et de générer les rétroactions pour accompagner l'apprenant au cas où celui-ci n'a pas résolu l'exercice.

- Le troisième niveau, permet quant à lui, d'assurer la gestion des données (profil apprenant, les différents exercices, …).

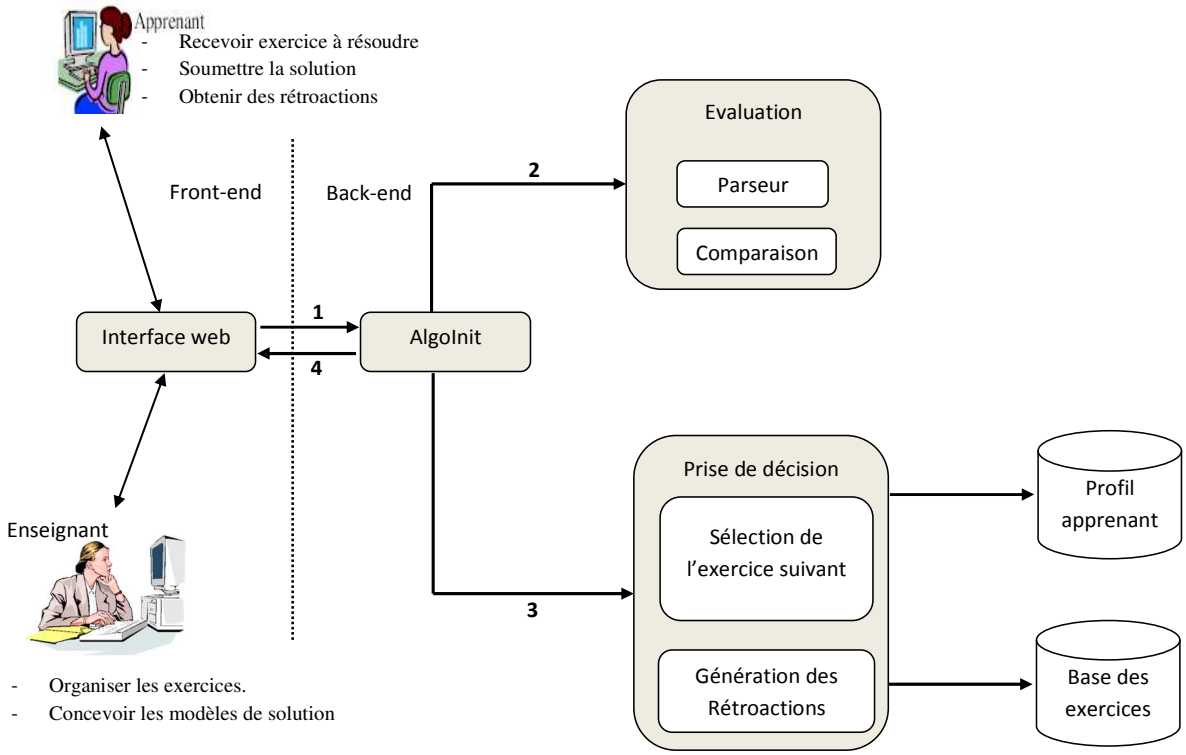


Figure 6-1: Architecture de l'environnement AlgoInit

6.3 Implémentation d'AlgoInit : les outils utilisés dans les différents niveaux

La *figure 6-2*, décrit l'architecture technique d'*AlgoInit*. Nous détaillons dans cette section la mise en œuvre d'*AlgoInit* dans les différents niveaux décrit ci-dessus.

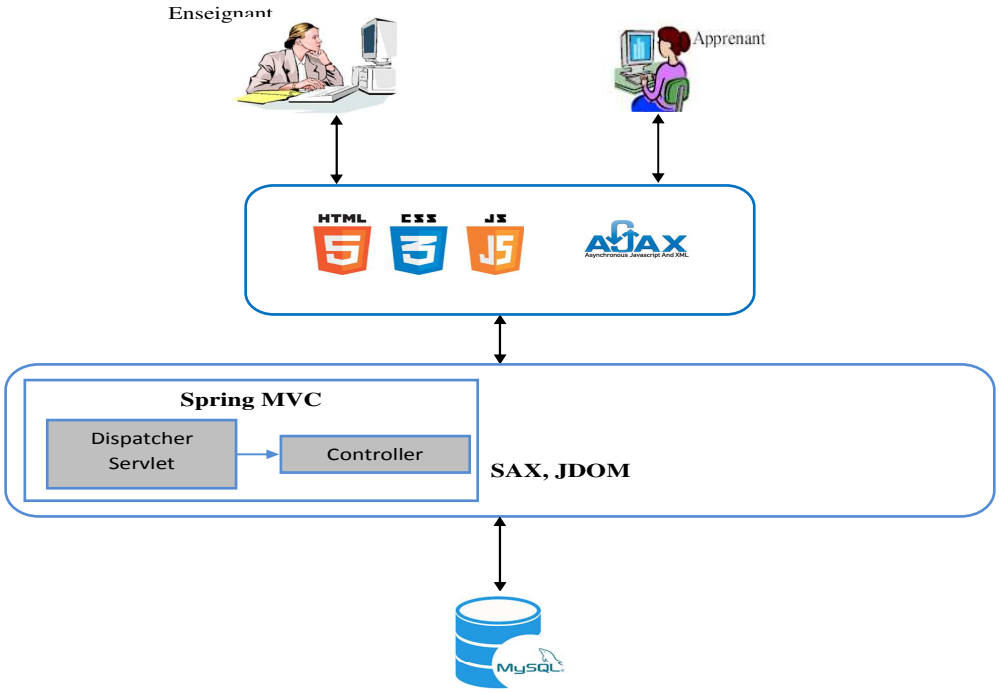


Figure 6-2: Architecture technique d'AlgoInit

6.3.1 Les interfaces utilisateurs de l'environnement AlgoInit

La conception de cette interface est basée sur HTML, CSS, JavaScript et AJAX. Suivant son rôle, l'utilisateur est dirigé soit : (i) vers l'**interface étudiant** *(figure 6-3)* où celui-ci rédige la solution de l'exercice donné et la soumet pour correction ; soit (ii) vers **l'interface enseignant** (*figure 6-4*). Sur cette interface, l'enseignant rédige l'énoncé de l'exercice, la rétroaction (information aidant à résoudre l'exercice) et les différents paramétrages de niveaux de l'exercice ainsi que les solutions modèles. Lorsque l'enseignant rédige la solution modèle, il a la possibilité d'ajouter les ordres que peut occuper une ligne d'instruction (*figure 6-5*).

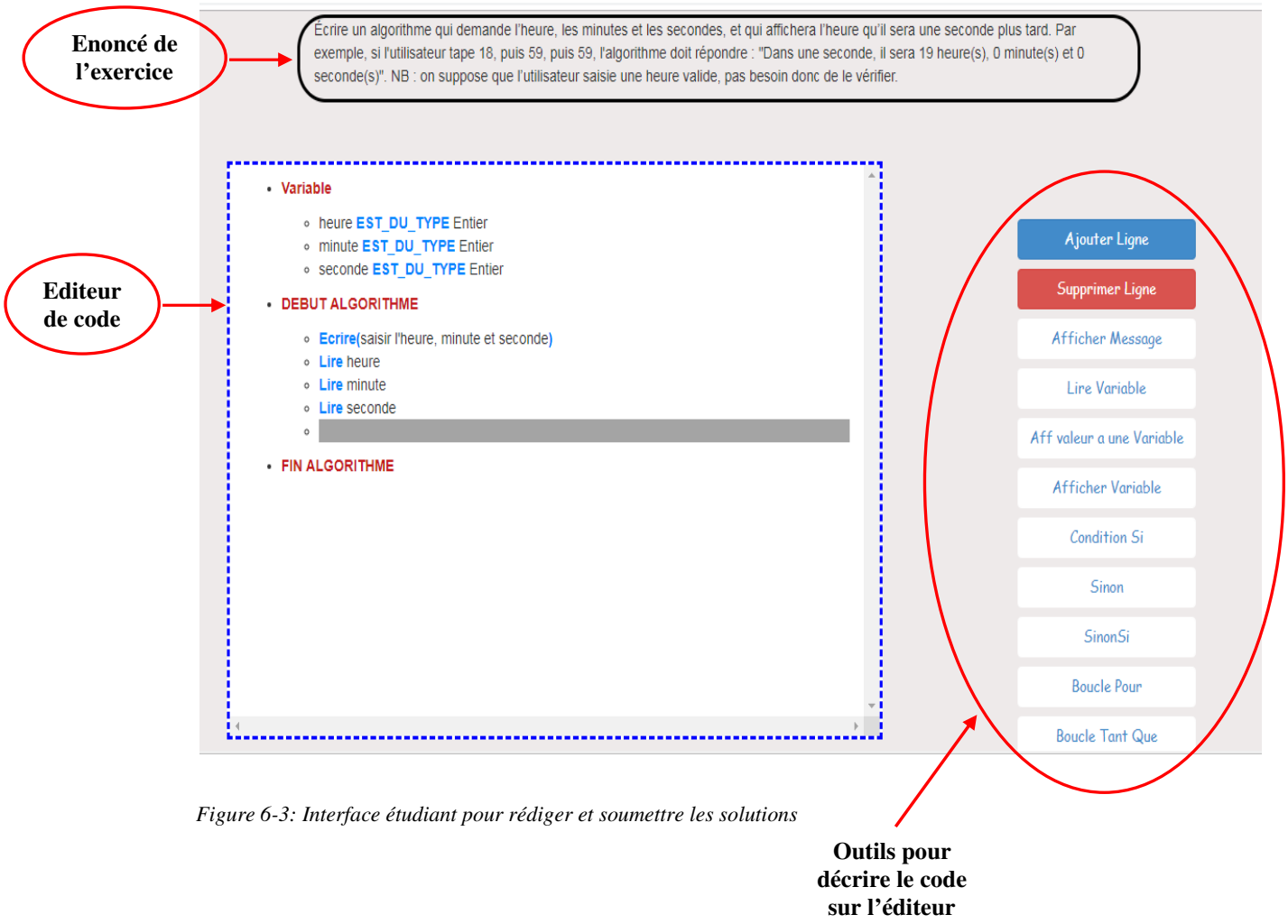


Figure 6-3: Interface étudiant pour rédiger et soumettre les solutions

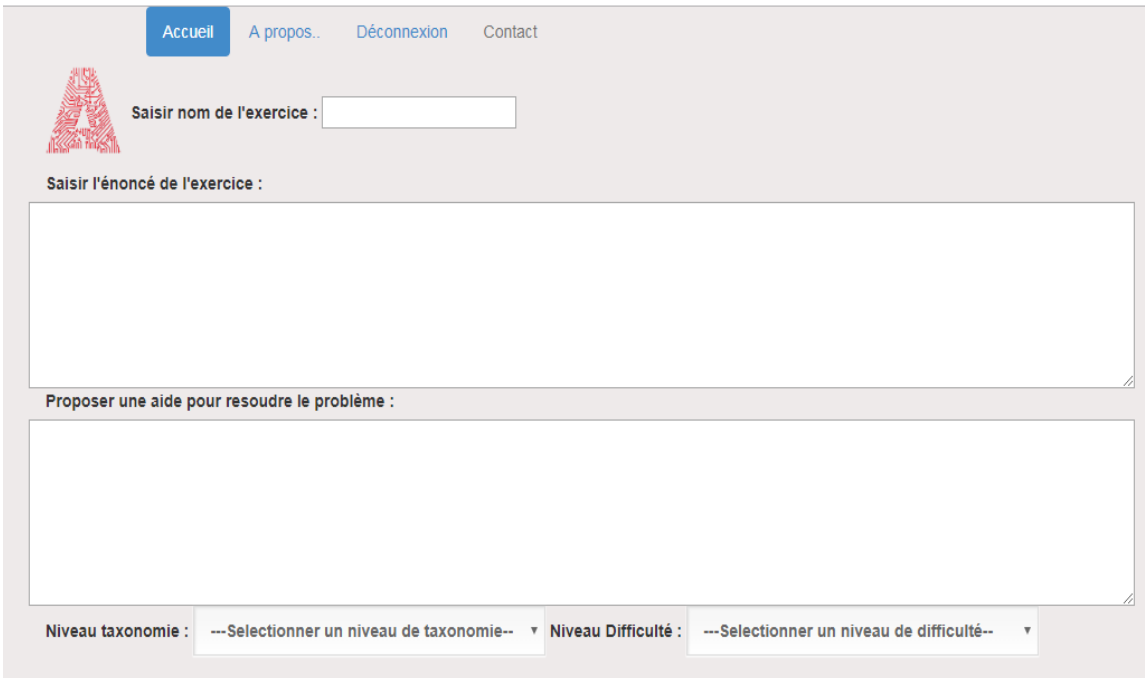


Figure 6-4 : Interface enseignant pour paramétrer les exercices

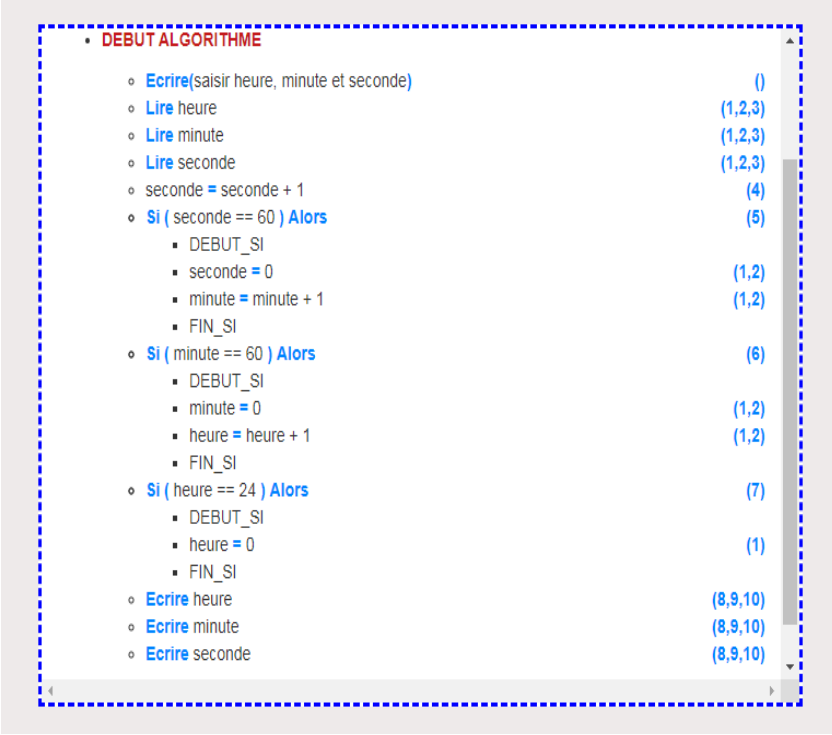


Figure 6-5: Solution modèle ajoutée par l'enseignant

6.3.2 Les différents traitements réalisés par AlgoInit

Nous utilisons Java et ses technologies pour mettre en œuvre le traitement des requêtes provenant de l'interface utilisateurs. Pour mettre en œuvre le prototype, nous avons utilisé le *framework Spring*. Ce *framework* libre simplifie la conception des applications Java EE. Il est modulaire et on choisit les modules appropriés en fonction des fonctionnalités nécessaires. Dans le cadre de la conception de notre prototype, qui concerne une application Web, nous utilisons le module *Spring MVC*.

Dans ce qui suit, nous présentons les différents traitements implémentés dans AlgoInit. Ceci se présente comme suit :

- L'**approche d'évaluation** qui est chargée d'analyser la solution algorithmique de l'apprenant. Dans cette approche nous allons présenter deux traitements : la transformation des solutions algorithmiques (apprenant et modèle) en arbres étiquetés et la comparaison entre ces derniers afin de statuer sur la solution de l'apprenant.
- La **prise de décision**, qui doit soit générer des rétroactions, soit fournir l'exercice suivant. Ce traitement se base sur le résultat fourni par l'évaluation. Dans cette étape nous présentons le traitement pour la génération des rétroactions communiquant à l'apprenant ses erreurs.

Les résultats de la comparaison effectuée par le *module d'évaluation*, permettent d'enrichir notre base de données. Ceci permet de sauvegarder l'historique des résultats (taux de réussite) pour chaque soumission effectuée et permet également de mettre à jour le profil de l'apprenant en sauvegardant l'état (faible, moyen, bon) pour chaque exercice d'un niveau de difficulté et d'un niveau cognitif donné.

Le *module prise de décision* se base sur le profil de l'apprenant, précédemment mis à jour par le module évaluation, pour la sélection des exercices ainsi que la génération des rétroactions à soumettre.

Toutes les données sont stockés dans une base de données de type MySQL.

La *figure 6-12* (*p.106*) montre le diagramme de classe UML de notre prototype. Nous décrivons dans la suite les deux modules (évaluation et prise de décision)

6.3.2.1 Implémentation du processus d'évaluation

Ce module repose sur un parseur qui permet de transformer les solutions algorithmiques en une représentation XML. Celle-ci sert de base pour la comparaison des solutions modèle et apprenant. Nous présentons dans la suite l'implémentation de ces deux traitements.

a) Transformation des solutions algorithmiques en arbres XML

Les solutions algorithmiques vont être représentées sous forme hiérarchique (*figure 6-5*). Pour transformer ces solutions en arbres étiquetés représentés sous forme XML, il nous faut un parseur de fichier XML pour lire ces hiérarchies et les transformer. Pour implémenter ce parseur, nous avons utilisé l'API SAX (Simple API for XML).

La représentation XML (*figure 6-6*) est construite au fur et à mesure que l'apprenant ou l'enseignant construit sa solution. Chaque ligne d'instruction est représentée par une balise.

L'API SAX est utilisée pour parser cette représentation. Elle permet un traitement événementiel des fichiers XML, c'est-à-dire qu'il va engendrer un événement pour les différents nœuds (i.e. balises) que nous utiliserons afin de générer l'arbre étiqueté.

Lors de la transformation de cette représentation XML en arbre XML étiqueté nous utilisons une base de règles (voir chapitre 4) qui nous permet de sélectionner l'étiquette pertinente. Par exemple pour la solution modèle de *figure 6-5*, après transformation nous

obtenons l'arbre étiqueté (arbre XML) *figure 6-8*. Sur cet arbre les balises représentent les étiquettes.

Notre base de règles, qui nous permet de traduire chaque ligne d'instruction en une étiquette, est stockée dans une table de notre base de données. Pour retourner l'étiquette correspondant à chacune des lignes d'instructions, nous utilisons une requête SQL. Ceci est illustré sur la *figure 6-7*.

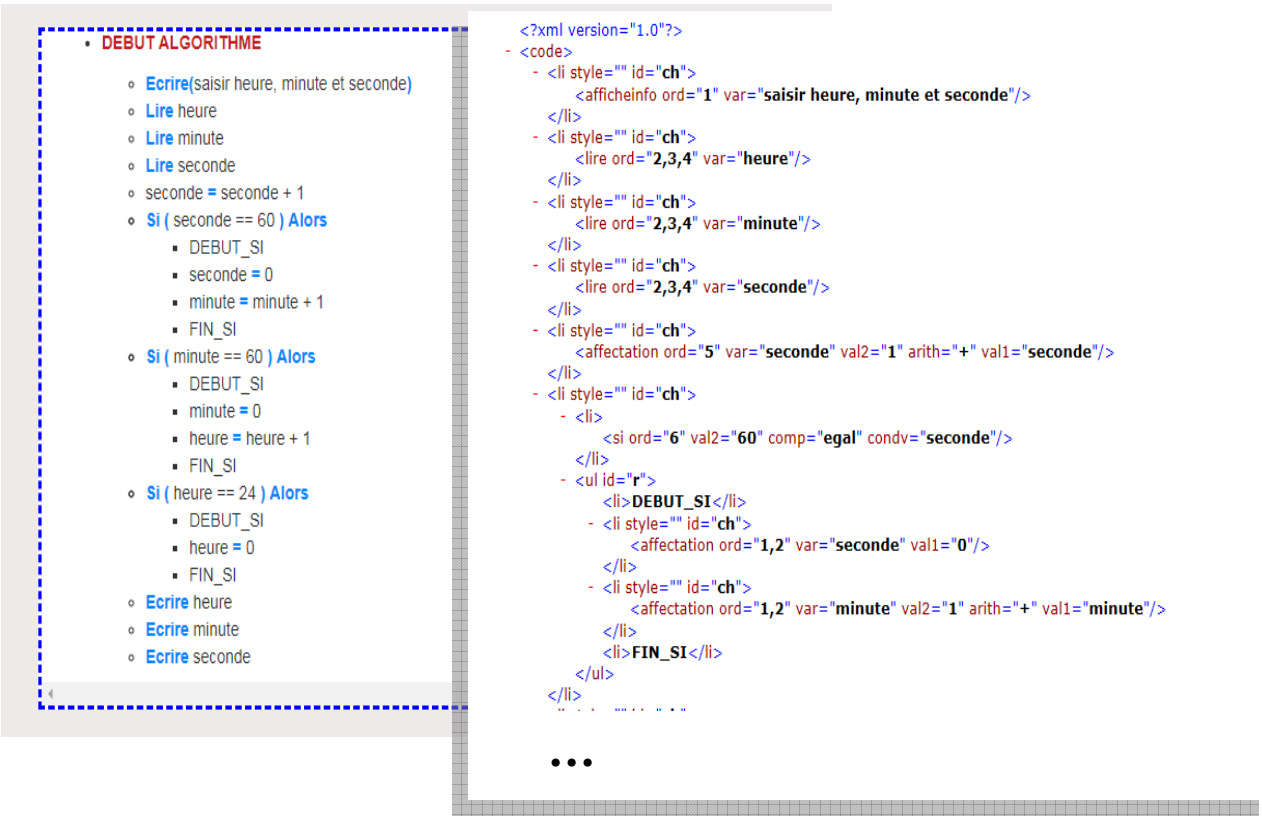


Figure 6-6: Format XML de la solution lors de sa soumission

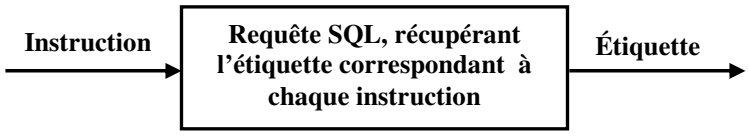


Figure 6-7: Règles de production des étiquettes représentant la balise de chaque ligne

```
<?xml version="1.0"?>
- <doc>
    <AfficheInfo ordre="" SA="Ecrire ( saisir heure, minute et seconde)"/>
    <Donneeheure ordre="1,2,3" SA="lire heure"/>
    <Donneeminute ordre="1,2,3" SA="lire minute"/>
    <Donneeseconde ordre="1,2,3" SA="lire seconde"/>
    <Incrementationseconde ordre="4" SA="seconde= seconde + 1"/>
  - <verifieseconde ordre="5" SA="Si seconde egal 60">
      <Initialisationseconde ordre="1,2" SA="seconde= 0 "/>
      <Incrementationminute ordre="1,2" SA="minute= minute + 1"/>
    </verifieseconde>
  - <verifieminute ordre="6" SA="Si minute egal 60">
      <Initialisationminute ordre="1,2" SA="minute= 0 "/>
      <Incrementationheure ordre="1,2" SA="heure= heure + 1"/>
    </verifieminute>
  - <verifieheure ordre="7" SA="Si heure egal 24">
      <Initialisationheure ordre="1" SA="heure= 0 "/>
    </verifieheure>
    <Afficherheure ordre="8,9,10" SA="Ecrire heure"/>
    <Afficherminute ordre="8,9,10" SA="Ecrire minute"/>
    <Afficherseconde ordre="8,9,10" SA="Ecrire seconde"/>
  </doc>
```

Figure 6-8: Arbre XML construit avec SAX

b) Comparaison des arbres étiquetés

L'étape précédente, nous permet de disposer de solutions algorithmiques standardisées au format XML. Dans une deuxième étape nous effectuons la comparaison des arbres XML (apprenants et modèles).

Pour naviguer dans les arbres XML et pouvoir comparer les nœuds, nous utilisons l'API DOM. Cette API fournit un moyen simple et efficace pour manipuler un arbre XML.

Après avoir parsé nos arbres XML (apprenant et modèle), nous obtenons des arbres DOM que nous pouvons manipuler pour effectuer la comparaison entre eux. Pour cela nous appliquons l'algorithme présenté dans le chapitre 4. Les résultats de cette évaluation sont enregistrés dans notre base de données comme décrit précédemment.

6.3.2.2 Génération des rétroactions et sélection des exercices

Dans cette section nous présentons l'implémentation de la génération des rétroactions et la sélection des exercices à soumettre dans *AlgoInit*.

a) Génération des rétroactions

Les rétroactions fournies à l'apprenant se présentent sous deux formes (voir chapitre 5): une rétroaction indiquant à l'apprenant comment résoudre l'exercice. Celle-ci est paramétrée par l'enseignant lors de la création de l'exercice ; le deuxième type de rétroaction, indique quant à lui, les différentes erreurs commises par l'apprenant. Nous présentons ici l'implémentation dans *AlgoInit* de cette deuxième forme de rétroaction.

Pour générer ces rétroactions nous nous appuyons sur les résultats (*resultatSA et resultatOA*) des différents sous-problèmes attendus stockés dans notre base de données. Ces résultats nous permettent de classer les erreurs selon les types d'erreurs décrites dans le chapitre 4. Pour énumérer les erreurs commises dans chacun des types d'erreurs, nous cherchons tous les sous-problèmes attendus pour lesquels la variable *resultatSA* ou *resultatOA* est à 0. Ensuite, nous traduisons ces sous-problèmes (étiquette) grâce à notre base de règles.

L'utilisation de cette base de règles est illustrée sur la *figure 6-9.*

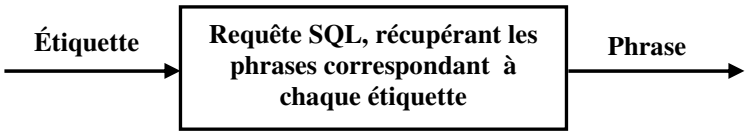


Figure 6-9: Règles de production les contenus de rétroaction

La solution algorithmique de la *figure 6-10* présente un exemple d'une solution apprenant. En se basant sur la solution modèle de la *figure 6-5*, nous observons que cet étudiant à oublié de mettre en œuvre l'incrémentation de la seconde, minute, heure et qu'il n'a pas mis au bon endroit les affichages. Pour cette solution les rétroactions affichées par *AlgoInit* est représenté par la *figure 6-11*.

b) Sélection des exercices

Pour sélectionner l'exercice suivant, nous implémentons l'algorithme présenté dans le chapitre 5. Celui-ci se base sur le profil apprenant et la collection d'exercices de notre base de données (*figure 6-12*).

Figure 6-10: Solution algorithmique d'un apprenant

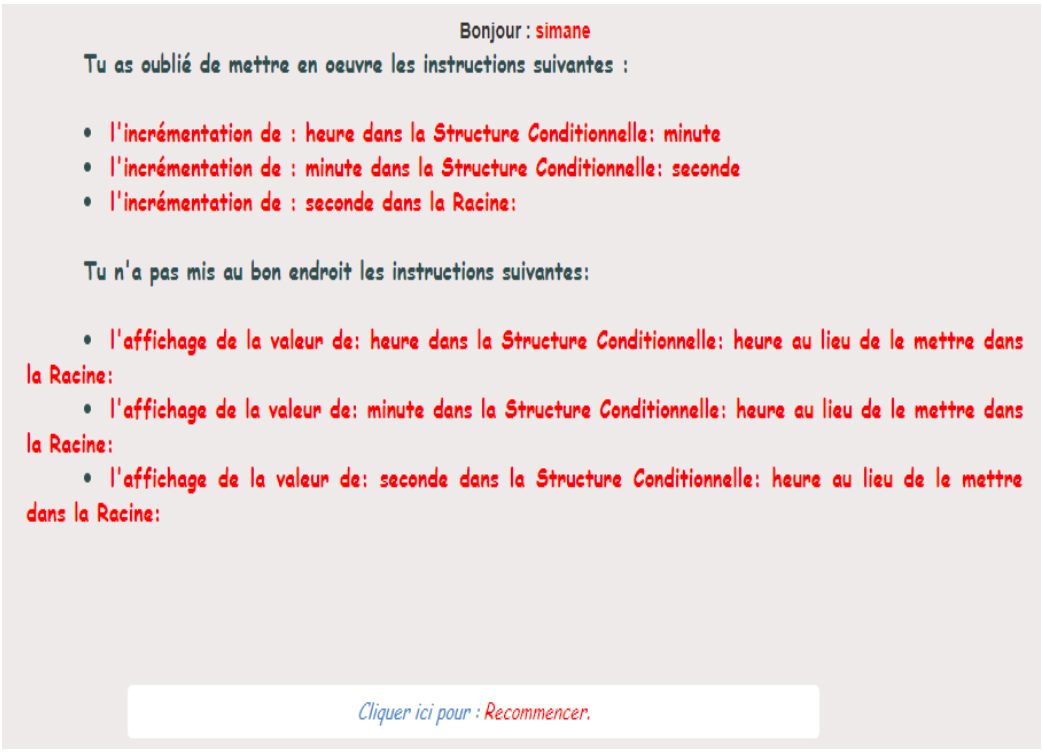


Figure 6-11: Rétroaction communiquant les erreurs de l'apprenant

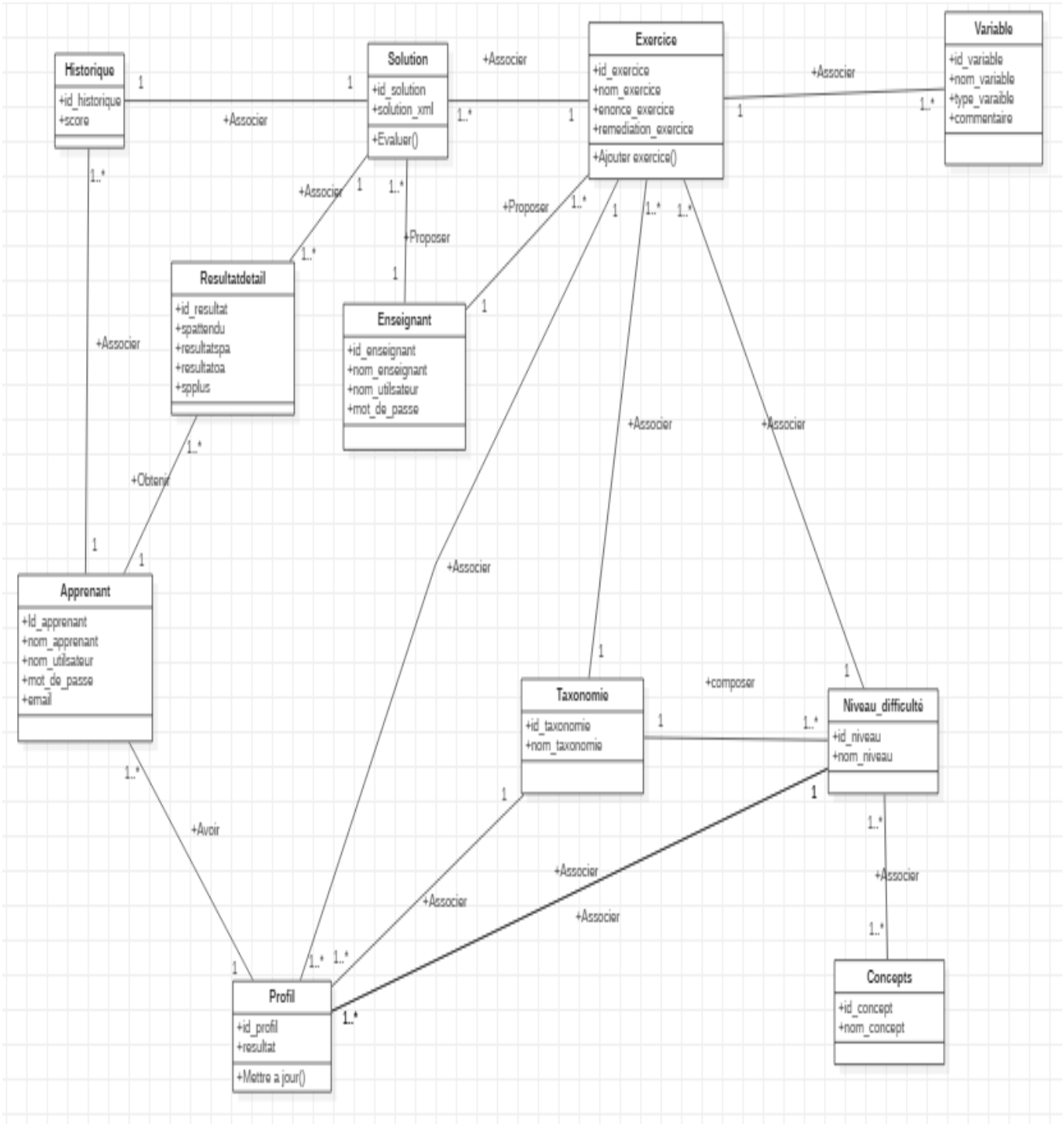


Figure 6-12: Diagramme de classe pour le prototype AlgoInit

6.4 Synthèse

Nous avons présenté dans ce chapitre le développement de l'environnement Web *AlgoInit*. Nous avons focalisé notre présentation sur l'implémentation du processus d'évaluation, la génération des rétroactions communiquant les erreurs à l'apprenant et la sélection des exercices à lui proposer. L'objectif pédagogique de cet environnement est de proposer une évaluation automatisée afin d'accompagner les apprenants qui ont des difficultés et proposer également de nombreux exercices pratiques pour renforcer leurs connaissances stratégiques (savoir utiliser les concepts appropriés dans différents exercices).

Dans le chapitre suivant, nous présenterons les expérimentations réalisées afin d'évaluer l'efficacité de notre système à évaluer les différentes solutions algorithmiques et son apport pédagogique.

Chapitre 7 : Expérimentations

Contenu

7.1 Introduction

L'objectif pédagogique d'*AlgoInit* est d'assister les étudiants novices afin de renforcer leurs connaissances stratégiques de manière à ce qu'ils sachent utiliser les concepts de base de la programmation de manière appropriée. Afin d'étudier la potentialité d'*AlgoInit,* nous avons mené deux études expérimentales.

Ce chapitre a pour objectif de décrire les deux expérimentations que nous avons réalisées avec le prototype *AlgoInit*. La première expérimentation a été consacrée à l'évaluation de la capacité de notre prototype à reconnaître les différentes solutions algorithmiques. Quant à la deuxième expérimentation, elle a été consacrée à l'évaluation de l'intérêt pédagogique d'*AlgoInit*. Nous avons cherché à évaluer cet intérêt à travers trois axes : l'impact des rétroactions fournies par *AlgoInit*, l'impact global sur les résultats des étudiants ainsi que l'utilisabilité du système.

7.2 Expérimentation #1 : Taux de reconnaissance des erreurs dans AlgoInit

7.2.1 Objectif de l'expérimentation

L'objectif d'*AlgoInit* est de fournir aux étudiants des exercices et des rétroactions pertinents en fonction de leurs erreurs. Aussi, une première étape consiste à vérifier le taux de reconnaissance des erreurs des apprenants sur leurs solutions algorithmiques. Pour cela, nous avons mené une première expérimentation destinée à évaluer la capacité de reconnaissance de notre approche d'évaluation. Dans un premier temps, nous allons préciser le cadre expérimental (contexte et exercices utilisés) avant de présenter les résultats de cette expérimentation.

7.2.2 Contexte de l'expérimentation

Cette expérimentation a été réalisée dans le contexte du cours de première année de DUT GEII (Génie Électrique et Informatique Industrielle) de l'IUTI de l'Université de Djibouti. Dans cette filière, le cours (Algorithmique, Programmation) d'introduction à la programmation a lieu au 1er semestre, dans le cadre de l'unité d'enseignement « *informatique des systèmes industriels* ». Dans ce cours, la base des concepts de la programmation est

enseignée : *types de donnée, variables, structure conditionnelle, structure répétitive, introduction aux tableaux*. Ce cours est organisé comme suit : (i) une séance de cours magistral, où l'enseignant présente aux étudiants les différents concepts ; (ii) une séance de TD. Durant cette séance l'enseignant propose aux étudiants des exercices à résoudre. Les étudiants décrivent leurs solutions sous forme algorithmique en pseudo-code ; (iii) et une séance de TP, durant laquelle les étudiants traduisent leur solutions algorithmiques en langage C.

La séance de TD permet de manipuler les concepts vus en cours et de les renforcer à travers un ensemble d'exercices se traduisant par l'expression d'algorithmes. C'est dans ce contexte que l'utilisation d'*AlgoInit* est le plus pertinent.

7.2.3 Population de test

Dans le cadre de cette expérimentation, nous avons sélectionné 30 étudiants selon leur niveau, afin d'avoir des étudiants de niveaux hétérogènes. Pour les sélectionner, nous nous sommes basés sur leurs notes de contrôle continu en « *Algorithmique, Programmation* » *Tableau 7-1*.

	Note<10	**10<Note<13**	**Note>13**
Nombre	10	10	10

Tableau 7-1: Répartition des niveaux de la population

Au moment de l'expérimentation, les étudiants ont étudié *les variables, l'affichage et la récupération des données, les structures conditionnelle et répétitive.*

7.2.4 Les différents exercices

Pour réaliser cette expérimentation, nous avons, dans un premier temps, demandé à des enseignants de préparer des exercices avec des niveaux de difficulté différents (selon le modèle présenté au chapitre 5). Nous avons ensuite, intégré ces exercices dans *AlgoInit*. Nous avons également fait une phase de présentation et d'utilisation d'*AlgoInit* avec les étudiants.

Dans le cadre de cette expérimentation, nous avons enlevé dans *AlgoInit* les conditions de passage d'un exercice à un autre, puisque l'objectif de cette première expérimentation est le taux de reconnaissance des erreurs sur différents exercices. De cette façon, les étudiants ont eu accès à la totalité des exercices disponibles dans *AlgoInit*.

Dans le cadre de l'expérimentation, nous avons sélectionné trois exercices, avec des niveaux de difficultés croissants. Ces niveaux de difficultés ont été choisis selon les trois niveaux de taxonomie de Bloom (voir chapitre 5). L'objectif de cette sélection est d'évaluer la capacité d'*AlgoInit* à reconnaître des solutions avec des niveaux de difficulté croissants. En effet, chaque niveau de difficulté diffère par : (i) le nombre d'instructions attendues ; (ii) la variabilité des solutions et (iii) les concepts utilisés.

Compte tenu de ces critères, nous avons sélectionné les exercices suivants :

> ***Exercice 1 :*** *Écrire un algorithme qui demande un nombre à l'utilisateur, puis qui calcule et affiche le carré de ce nombre.*

Cet exercice correspond au ***niveau connaissance*** de la taxonomie de Bloom. Il est attendu de l'apprenant qu'il arrive à utiliser les concepts *afficher*, *récupérer* (i.e., lire une entrée au clavier) et *faire le calcul*.

> ***Exercice 2 :*** *Écrire un algorithme qui demande l'heure, les minutes et les secondes, et qui affichera l'heure qu'il sera une seconde plus tard.*
>
> *Par exemple, si l'utilisateur tape 18, puis 59, puis 59, l'algorithme doit répondre : "Dans une seconde, il sera 19 heure(s), 0 minute(s) et 0 seconde(s)".*
>
> *NB : on suppose que l'utilisateur saisit une heure valide, pas besoin donc de le vérifier.*

Dans cet exercice, le ***niveau compréhension*** de la taxonomie de Bloom est mis en œuvre. Il évalue, si les étudiants ont bien compris l'utilisation du concept de *structure conditionnelle*.

> ***Exercice 3 :*** *Écrire un algorithme qui demande à l'utilisateur de saisir les notes de N étudiants d'une classe, calcule la moyenne de la classe et affiche la moyenne, la meilleure et la plus mauvaise note.*

Cet exercice, correspond au ***niveau application*** de la taxonomie de Bloom. Il demande à ce que l'utilisateur, à travers cet exercice mobilise les différents concepts étudiés : *structure répétitive* (utiliser la boucle appropriée, dans cet exercice l'utilisation de la boucle `Pour` est attendue), *structure conditionnelle*, *calcul*, *afficher* et *récupérer une donnée*.

7.2.5 Collecte et méthodologie d'analyse des données

Pour analyser les résultats de l'expérimentation, nous comparons le résultat (correct ou incorrect) de la correction effectuée par l'enseignant et celle effectuée par *AlgoInit* afin de vérifier si l'analyse d'*AlgoInit* concorde avec celle faite par l'enseignant.

Pour effectuer cette comparaison, nous utilisons la « *matrice de confusion* » également appelé « *matrice de classification* ». Cette dernière permet de mesurer la performance d'un modèle en vérifiant à quelle fréquence ses prédictions sont exactes par rapport à la réalité. Elle est généralement utilisée pour évaluer les systèmes de recommandation en mesurant certaines valeurs[4] comme la valeur de précision. Cette dernière correspond au calcul de la différence entre les valeurs prédites par le système de recommandation et les valeurs réelles fournie par l'utilisateur (Haydar, 2014), (Piton, 2011). Dans notre cas, nous l'utilisons pour mesurer si *AlgoInit* classe bien les solutions correctes *et* incorrectes. Dans le cas d'une solution incorrecte nous vérifions si les erreurs soulignées par *AlgoInit* correspondent réellement aux erreurs soulignées par l'enseignant.

Les lignes de cette matrice de confusion indiquent les valeurs prédites par le modèle tandis que les colonnes représentent les valeurs réelles. Les catégories utilisées dans l'analyse sont :

- ***Vrai Positif (VP):*** représente, le cas où la prédiction est positive, et où la valeur réelle est également positive. *Dans notre cas, cela correspond à une solution juste, reconnue comme correcte par AlgoInit.*
- ***Vrai Négatif (VN) :*** représente, le cas où la prédiction est négative, et où la valeur réelle est également négative. *La solution de l'apprenant est fausse et AlgoInit a bien détecté que celle-ci est incorrecte.*
- ***Faux Positif (FP) :*** représente, le cas où la prédiction est positive tandis que la valeur réelle est négative. *La solution de l'apprenant est fausse mais AlgoInit l'a classée comme correcte.*
- ***Faux Négatif (FN) :*** représente, le cas où la prédiction est négative tandis que la valeur réelle est positive. *La solution de l'apprenant est juste mais AlgoInit l'a classée comme incorrecte.*

[4] https://jcrisch.wordpress.com/2015/05/04/valider-un-modele-statistique-avec-la-cross-validation/

Dans le cas de notre expérimentation, la prédiction correspond au classement effectué par *AlgoInit* et la valeur réelle représente le classement des solutions algorithmique effectué par l'enseignant. Nous avons souligné plus haut que nous évaluons si *AlgoInit* classe bien les solutions correctes mais également s'il classe bien les solutions incorrectes. Ainsi, nous calculons les valeurs de précision d'*AlgoInit* à classer les solutions comme correctes (précision sur les vraies positifs) (1) et incorrectes (précision sur les vraies négatifs) (2), comme suit :

$$\Pr SC = \frac{VP}{VP + FP} \qquad \textbf{(1)}$$

$$\Pr SIC = \frac{VN}{VN + FN} \qquad \textbf{(2)}$$

Ou :

PrSC : correspond à la précision d'*AlgoInit* pour détecter les solutions correctes.

PrSIC : correspond la précision d'*AlgoInit* pour détecter les solutions incorrectes.

Dans la suite, le résultat de l'expérimentation pour chaque exercice est représenté par cette matrice.

7.2.6 Résultat de l'expérimentation

7.2.6.1 Résultat exercice 1

Le résultat de la matrice de confusion, pour l'exercice 1, est représenté par le *Tableau 7-2*.

Expert (Enseignant) / **AlgoInit**	**Positif**	**Négatif**
Positif	**30**	-
Négatif	-	-

Tableau 7-2: Matrice de confusion pour l'exercice 1

Nous observons que la totalité des solutions soumises par les apprenants ont été reconnues par *AlgoInit* comme des solutions correctes. Les 30 solutions fournies par les

apprenants ont été décrite de la même façon (*figure 7-1*). Pour cet exercice la précision de notre système pour détecter les solutions correctes est de :

$$\Pr SC = \frac{VP}{VP + FP} = \frac{30}{30 + 0} = 1 = 100\%$$

Les exercices suivants vont nous permettre de vérifier le comportement de notre système face à des solutions plus complexes et de plus grande variabilité.

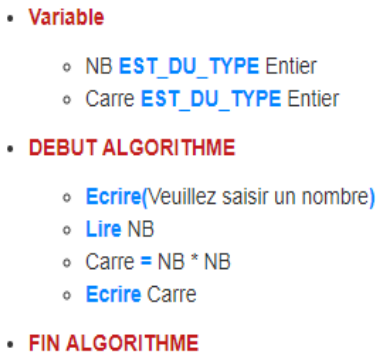


Figure 7-1: [Exercice 1] solution fournie par les apprenants

7.2.6.2 Résultat exercice 2

Dans cet exercice, comme souligné précédemment, plusieurs solutions sont possibles. Le résultat de la matrice de confusion, pour cet exercice, est donné par le *Tableau 7-3*. Nous observons qu'*AlgoInit* a reconnu 16 solutions comme correctes et 14 solutions comme incorrectes. Ainsi la précisons d'*AlgoInit* pour la reconnaissance des solutions correctes est : $\Pr SC = \frac{VP}{VP + FP} = \frac{16}{16 + 0} = 1$ c'est-à-dire 100%. Tandis que la précision pour la reconnaissance des solutions incorrectes est : $\Pr SIC = \frac{VN}{VN + FN} = \frac{13}{13 + 1} = 0.93$, c'est-à-dire 93%.

Expert (Enseignant) / AlgoInit	Positif	Négatif
Positif	**16**	-
Négatif	**1**	**13**

Tableau 7-3: Matrice de confusion pour l'exercice 2

Pour cet exercice, *AlgoInit* a bien classé la plupart des solutions (correctes et incorrectes) à une exception près. Nous allons maintenant étudier les différents résultats pour confirmer le comportement *d'AlgoInit.*

a) Exercice 2 : Vrais Positifs (VP)

Dans cet exercice nous observons que 16 solutions sont des vrais positifs (solution évaluée par *AlgoInit* et l'enseignant comme solution correcte). Nous avons trouvé deux solutions différentes rendues par les apprenants (*figure 7-2*). Nous avons trouvé dix solutions décrites comme la première solution (*figure 7-2* : gauche) et six solutions décrites comme la deuxième solution (*figure 7-2* : droite). Sur ce résultat, nous soulignons qu'*AlgoInit* arrive à détecter des solutions décrites différemment (ordre des instructions différent). Cette reconnaissance est faite grâce à une seule solution modèle contrairement aux systèmes étudiés dans l'état de l'art qui nécessitent un modèle pour chaque type de solution.

```
DEBUT ALGORITHME
    Ecrire(saisir heure, minutes et seconde)
    Lire heure
    Lire minute
    Lire seconde
    seconde = seconde + 1
    Si ( seconde == 60 ) Alors
        DEBUT_SI
        seconde = 0
        minute = minute + 1
        FIN_SI
    Si ( minute == 60 ) Alors
        DEBUT_SI
        minute = 0
        heure = heure + 1
        FIN_SI
    Si ( heure == 24 ) Alors
        DEBUT_SI
        heure = 0
        FIN_SI
    Ecrire heure
    Ecrire minute
    Ecrire seconde
```

```
DEBUT ALGORITHME
    Ecrire(saisir heure, minutes et seconde)
    Lire heure
    Lire minute
    Lire seconde
    seconde = seconde + 1
    Si ( seconde == 60 ) Alors
        DEBUT_SI
        minute = minute + 1
        seconde = 0
        FIN_SI
    Si ( minute == 60 ) Alors
        DEBUT_SI
        heure = heure + 1
        minute = 0
        FIN_SI
    Si ( heure == 24 ) Alors
        DEBUT_SI
        heure = 0
        FIN_SI
    Ecrire heure
    Ecrire minute
    Ecrire seconde
```

Figure 7-2: [Exercice 2] Deux solutions différentes, catégorisées Correcte par AlgoInit

b) Exercice 2 : Vrais Négatifs (VN)

Nous observons que 13 solutions sont classées vrais négatifs (solution évaluées par *AlgoInit* et l'enseignant comme solutions incorrectes). Parmi ces solutions, les erreurs détectées peuvent être catégorisées comme suit :

- ***Instructions non mise en œuvre par les étudiants :*** 8 solutions. La *figure 7-3* montre des exemples de cette catégorie. Dans le premier cas, *AlgoInit* a détecté que l'étudiant a oublié l'instruction qui vérifie l'heure (*figure 7-3 : gauche)*. Cette instruction permet de vérifier si l'heure est égale à 24, si c'est le cas la variable heure doit être affectée à 0. Dans le deuxième cas, *AlgoInit* a détecté que l'étudiant a oublié de mettre en œuvre les instructions qui vérifient les secondes, minutes et heures (*figure 7-3 : droite*).
- ***Instructions mal placées :*** 3 solutions. La *figure 7-4 (gauche)* montre un exemple de cette catégorie.
- ***Instruction non mise en œuvre par les étudiants et avec des instructions mal placées*** : 2 solutions. La *figure 7-4* (*droite*) montre un exemple de cette catégorie.

Figure 7-3: [Exercice 2] solution avec erreur (manque des instructions)

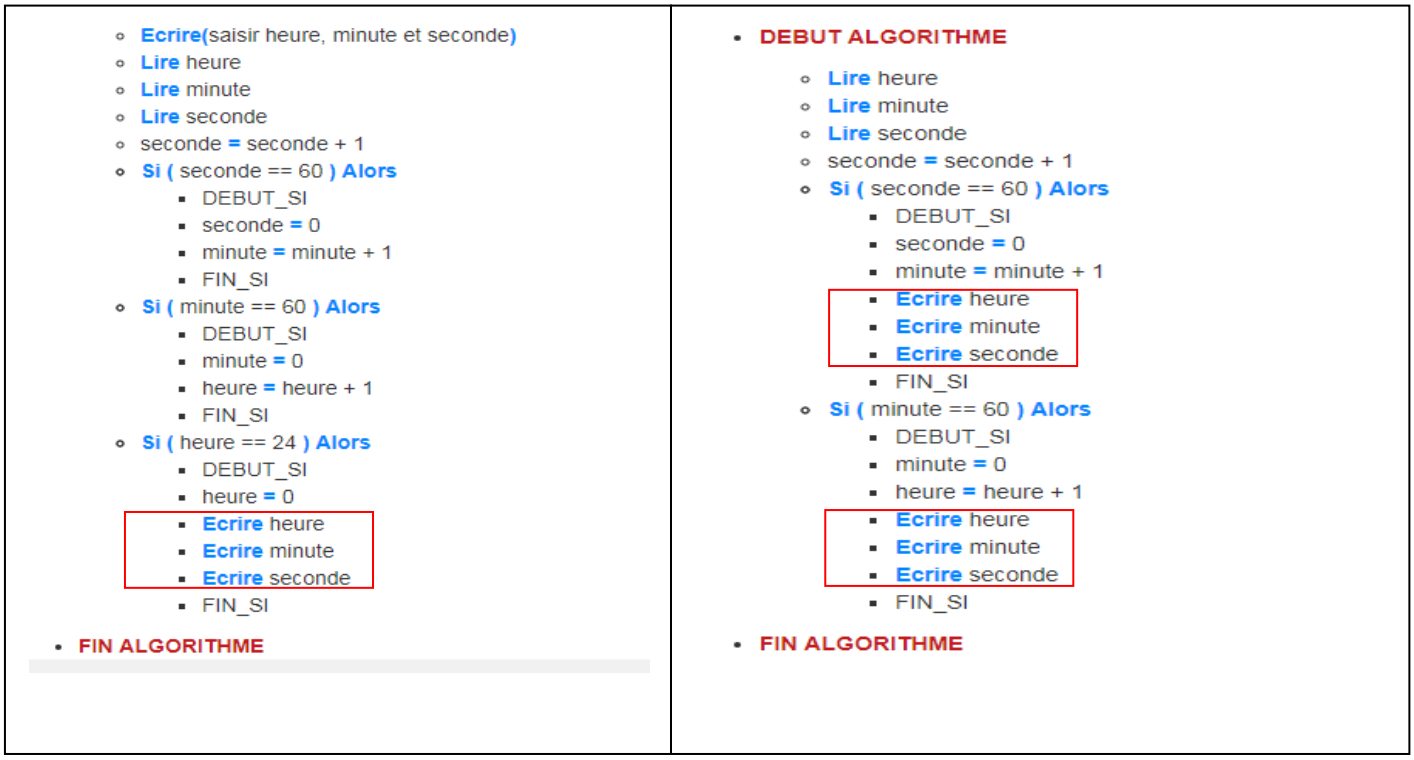


Figure 7-4: [Exercice 2] solution avec erreur (manque des instructions + instructions mal placées)

c) Exercice 2 : Faux Négatifs (FN)

Nous observons que dans cet exercice, une solution est classée faux négatif (solution évaluée par *AlgoInit* comme solution incorrecte et par l'enseignant comme solution correcte). Cette solution est présentée *figure 7-5*. *AlgoInit* a évalué que toutes les instructions attendues ont bien été mise en œuvre et dans l'ordre. Toutefois, *AlgoInit* souligne, qu'il y a des instructions supplémentaires. L'enseignant considère que ces instructions ne faussent pas la solution qui reste correcte. Si l'enseignant juge que cette solution est correcte, il pourra l'ajouter dans la base de solutions modèles pour cet exercice.

```
seconde = seconde + 1
Si ( seconde == 60 ) Alors
    DEBUT_SI
    seconde = 0
    minute = minute + 1
    Sinon
        seconde = seconde
    FIN_SI
Si ( minute == 60 ) Alors
    DEBUT_SI
    minute = 0
    heure = heure + 1
    Sinon
        minute = minute
    FIN_SI
Si ( heure == 24 ) Alors
    DEBUT_SI
    heure = 0
    Sinon
        heure = heure
    FIN_SI
Ecrire heure
Ecrire minute
Ecrire seconde
```

Figure 7-5: [Exercice 2] solutions avec erreur (instruction supplémentaire)

7.2.6.3 Résultat exercice 3

Cet exercice est le plus compliqué et celui qui présente la plus grande variabilité. Le résultat de la matrice de confusion est donné par le *Tableau7-4*. Nous observons qu'*AlgoInit* a reconnu 8 solutions comme correctes et 22 solutions comme incorrectes. Ainsi la précision d'*AlgoInit* pour la reconnaissance des solutions correctes est : $\Pr SC = \frac{VP}{VP + FP} = \frac{8}{8+0} = 1$ c'est-à-dire 100%. Tandis que la précision pour la reconnaissance des solutions incorrectes est : $\Pr SIC = \frac{VN}{VN + FN} = \frac{20}{20+2} = 0.91$, c'est-à-dire 91%.

Expert (Enseignant) / AlgoInit	Positif	Négatif
Positif	**8**	-
Négatif	**2**	**20**

Tableau7-4: Matrice de confusion pour l'exercice 3

Pour cet exercice, *AlgoInit* a encore une fois bien classé la plupart des solutions (correctes et incorrectes) à deux exceptions près. Nous allons maintenant étudier les différents résultats pour confirmer le comportement d'*AlgoInit*.

a) Exercice 3 : Vrais Positifs (VP)

Dans cet exercice nous observons que 8 solutions sont des vrais positifs (solution évaluée par *AlgoInit* et l'enseignant comme solution correcte). Parmi ces solutions, nous avons trouvé six écritures différentes. Cela confirme qu'*AlgoInit* arrive à détecter des solutions décrites différemment à partir d'une seule solution modèle.

b) Exercice 3 : Vrais Négatifs (VN)

Nous observons que 20 solutions sont classées en vrais négatifs (solution évaluée par *AlgoInit* et l'enseignant comme solutions incorrectes). Parmi ces solutions, les erreurs détectées peuvent être catégorisées comme suit :

- ***Instructions non mise en œuvre par les étudiants*** : 17 solutions.
- ***Instructions mal placées :*** 3 solutions.

c) Exercice 3 : Faux Négatifs (FN)

Nous observons, que dans cet exercice, deux solutions (*figure 7-6*) sont classées dans la catégorie **faux négatif** (solution évaluée par *AlgoInit* comme solution incorrecte et par l'enseignant comme solution correcte).

Pour la première solution, *AlgoInit* a détecté une instruction mal placée (*figure 7-6* : gauche). Après étude de la solution, l'enseignant considère que cette instruction ne fausse pas la solution. Pour remédier à cette erreur, l'enseignant ajoutera un nouveau modèle dans la base de solution modèle.

Pour la deuxième solution, *AlgoInit* a détecté deux types d'erreurs (*figure 7-6* : droite): (i) des instructions supplémentaires et (ii) l'utilisation d'un boucle non attendue. Dans le paramétrage de l'exercice, l'enseignant avait indiqué que l'utilisation d'une *boucle* `Pour` était attendue et l'apprenant a utilisé `TantQue`. Pour cette erreur *AlgoInit, informera l'apprenant qu'il doit utiliser la boucle appropriée (boucle* `Pour` *au lieu de* `Tantque`*)*. Cet exercice est donc classé dans faux négatif pour le type d'erreur instruction supplémentaire uniquement.

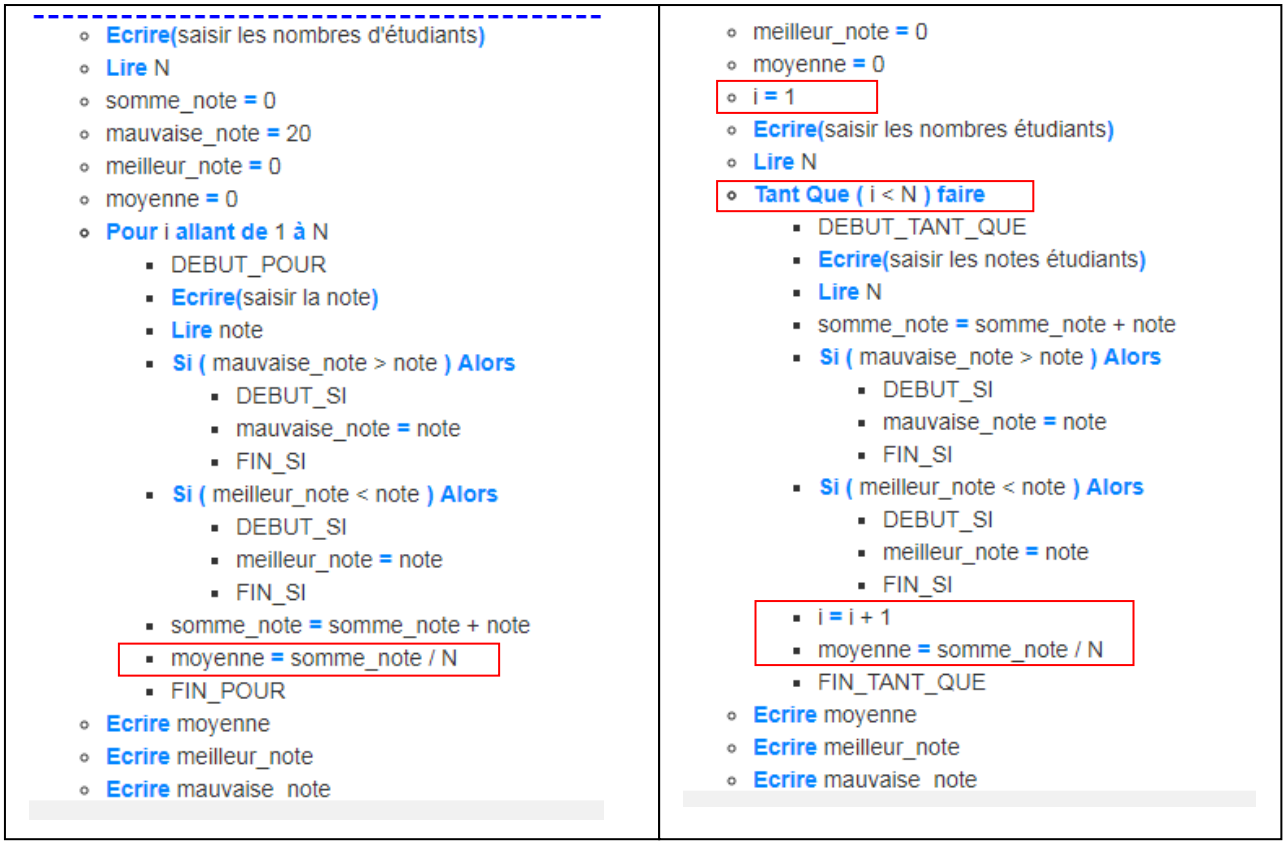


Figure 7-6: [Exercice 3] solution classifiée dans Faux Négatif

7.2.6.4 Résultat global

Nous représentons sur le *Tableau7-5*, le résultat de la matrice de confusion sur l'ensemble des solutions. Nous observons qu'*AlgoInit* a bien reconnu 54 solutions correctes et 36 solutions incorrectes. Ainsi la précisons d'*AlgoInit* pour la reconnaissance des solutions correctes est de 100%. Tandis que la précision d'*AlgoInit* pour classer les solutions fausses est de 92% ($\frac{33}{33+3} = 0.92$).

Expert (Enseignant) / AlgoInit	Positif	Négatif
Positif	**54**	-
Négatif	**3**	**33**

Tableau7-5: Matrice de confusion pour les trois exercices

7.2.7 Synthèse

Cette première expérimentation avait pour but de valider le mécanisme d'évaluation des solutions algorithmiques par *AlgoInit*. Cette étape était nécessaire pour s'assurer d'une bonne reconnaissance qui va servir de base à la rétroaction et à la sélection des exercices suivants.

Pour cette validation, nous avons choisi trois exercices de niveaux de difficulté croissants selon les trois premiers niveaux de la taxonomie de Bloom. Cela nous a permis de confronter *AlgoInit* à des niveaux de variabilité et de complexité croissants des solutions. Nous observons que la performance d'*AlgoInit* pour la reconnaissance des solutions correctes est de 100%. Nous soulignons, la reconnaissance correcte par *AlgoInit*, des solutions écrites avec des ordres d'instructions différents.

La reconnaissance par *AlgoInit* des erreurs dans les solutions apprenants est également très importante. Nous soulignons qu'*AlgoInit* a raté 3 solutions sur les 36 solutions classifiées comme fausses. Ce défaut peut être corrigé en enrichissant la base des solutions modèles. Toutefois, *AlgoInit* arrive à bien identifier les erreurs de conception dans les solutions algorithmiques des apprenants comme : (i) des instructions attendues, non mises en œuvre ; (ii) des instructions attendues mises en œuvre mais au mauvais endroit ; ou (iii) l'utilisation d'un concept inapproprié. Dans le cas où il y a des instructions supplémentaires, la solution peut quant même être correcte. Pour cela, l'enseignant vérifie la solution, et dans le cas où celle-ci est correcte, elle sera ajoutée dans la base des solutions modèles.

Dans la suite, nous présentons une deuxième expérimentation, qui elle, vise à évaluer l'apport de l'assistance fournie par *AlgoInit* suite à la détection des erreurs.

7.3 Expérimentation #2 : Analyse de l'impact et de l'utilisabilité d'AlgoInit

7.3.1 Objectif de l'expérimentation

Notre première expérimentation nous a permis de valider la capacité d'*AlgoInit* à reconnaître les erreurs dans les solutions apprenants. Cette détection d'erreurs a pour objectif de permettre à *AlgoInit* de leur offrir des rétroactions pertinentes et une série d'exercices de niveaux de difficulté cognitive croissants. L'objectif de cette deuxième évaluation est de mesurer l'impact d'*AlgoInit* sur l'apprentissage des apprenants. Dans cet objectif, nous allons traiter dans cette section les questions suivantes :

1. *Quel est l'impact des rétroactions fournies par AlgoInit sur le processus de résolution des exercices ?*
2. *Quel est l'impact de l'utilisation d'AlgoInit sur les capacités de résolution des problèmes des apprenants ?*
3. *Comment les apprenants évaluent l'utilisabilité d'AlgoInit ?*

Dans la suite, nous commencerons par la présentation du contexte d'expérimentation. Nous aborderons ensuite les trois questions posées ci-dessus.

7.3.2 Contexte de l'expérimentation

Cette expérimentation à été réalisée dans le même contexte que l'expérimentation précédente (cours de première année de DUT Génie Électrique et Informatique Industrielle de l'IUTI de l'université de Djibouti) et avec les mêmes étudiants. Pour réaliser cette expérimentation, nous avons intégré 12 exercices dans le système *AlgoInit*. La répartition des exercices dans les différents niveaux est donnée dans le *Tableau 7-6*.

Cette expérimentation s'est déroulée en 2 séances de 1h30 chacune. Pour cette expérimentation, nous avons activé dans le prototype le conditionnement du passage d'un exercice à un autre et la génération des rétroactions. Nous avons, également expliqué aux étudiants que l'environnement leur affecterait des exercices à résoudre et qu'en cas de non résolution, des rétroactions leurs seraient fournies.

Taxonomie	Niveau de difficulté	Concept mis en œuvre / objectif	Nombre d'exercices
Connaissance	1	Variables, affectation.	1
Connaissance	2	Structure conditionnelle.	2
Connaissance	3	Structures conditionnelles imbriqués.	2
Connaissance	4	Structure itérative.	2
Compréhension	1	Différencier l'utilisation des structures conditionnelles et structures conditionnelles imbriquées.	1
Compréhension	2	Utilisation de la structure répétitive appropriée.	1
Application	1	Utilisation de multiples concepts (structure conditionnelle + structure répétitive) pour résoudre un exercice.	3

Tableau 7-6: Répartition des différents exercices

7.3.3 Question 1 : quel est l'impact des rétroactions fournies par AlgoInit ?

Nous nous intéressons ici à vérifier l'effet des rétroactions fournies par *AlgoInit* aux apprenants dans le cas où ces derniers n'arrivent pas à résoudre un exercice. Nous présentons d'abord les traces collectées ainsi que notre méthodologie d'analyse des données. Nous rendons ensuite compte des résultats de l'évaluation.

7.3.3.1 Collecte et méthodologie d'analyses de données

Pour traiter cette première question, nous nous basons sur les données obtenues grâce aux traces des étudiants (solutions soumises) et les rétroactions obtenues par les apprenants durant la réalisation des différents exercices. Nous obtenons ainsi pour chaque essai non fructueux une rétroaction spécifique.

Pour vérifier l'impact de la rétroaction fournie par *AlgoInit,* nous étudions l'effet de cette rétroaction sur l'essai suivant.

Nous définissons ci-dessous une **rétroaction à effet positif** (RP) et une **rétroaction sans effet** (RSE) et nous illustrons ces effets sur la *figure 7-7*.

Définition 1 : *une rétroaction est qualifiée comme ayant un effet positif (RP) : (i) si la rétroaction générée a permis à l'apprenant de résoudre l'exercice à l'essai suivant (figure 7-7: cas 1) ou (ii) si la rétroaction fournie lors de l'essai suivant est différente de celle de l'essai courant (figure 7-7: cas 2).*

Définition 2 : *une rétroaction est qualifiée comme étant sans effet (RSE), si elle n'a engendré aucune amélioration sur la résolution de l'exercice à l'essai suivant (i.e., même rétroaction) (figure 7-7: cas 3).*

Cas 1 : E(i) exercice non résolu ⟶ R 1 ⟶ E(i+1) exercice résolu ⟶ RP

Cas 2 : E(i) exercice non résolu ⟶ R 1 ⟶ E(i+1) exercice non résolu ⟶ R 2 ≠ R1 ⟶ RP

Cas 3 : E(i) exercice non résolu ⟶ R 1 ⟶ E(i+1) exercice non résolu ⟶ R 2 = R1 ⟶ RSE

E(i) : essai à l'étape i. R : Rétroaction

Figure 7-7: Illustration des effets de rétroaction pour un exercice

Nous calculons par la suite la fréquence des rétroactions qui ont eu un effet positif (RP) et des rétroactions qui ont été sans effet (RSE) pour chaque étudiant durant leur résolution de l'ensemble des exercices.

Pour chaque exercice, l'étudiant reçoit des rétroactions à chaque essai. Pour cela, nous calculons le taux des rétroactions à effet positif (RP) et de rétroactions sans effet (RSE), pour chaque étudiant comme suit :

$$RP = \sum_{i=1}^{n} \frac{NRPi}{NRi} \qquad RSE = \sum_{i=1}^{n} \frac{NRSEi}{NRi}$$

Ou :

NRP : Nombre de rétroactions positive pour i^{ème} exercice.

NRSE : Nombre de rétroactions sans effet pour i^{ème} exercice.

NR : Nombre de rétroactions obtenues pour i^{ème} exercice.

i : 1, 2, 3…N exercice.

Nous avons également soumis aux étudiants un questionnaire à la fin de l'expérimentation lors de la deuxième séance. Ce questionnaire (*Tableau 7-8*) avait pour objectif de recueillir le ressenti des étudiants sur l'apport des rétroactions fournies par *AlgoInit*.

7.3.3.2 Résultats

Nous avons calculé les taux de rétroactions à effet positif (RP) et de rétroactions sans effet (RSE) pour l'ensemble des étudiants. Ce résultat est représenté au *Tableau 7-7*. Nous observons, que la fréquence des rétroactions ayant un effet positif est largement plus importante (84%) que les rétroactions sans effet (16%).

	RP	RSE
Fréquence	84 %	16%

Tableau 7-7: Résultat des fréquences des effets des rétroactions fournies par AlgoInit

Les réponses au questionnaire (*Tableau 7-8*) corroborent les résultats de l'analyse des traces. En effet, les étudiants ont jugé les rétroactions compréhensibles (96%) et 80% d'entre eux pensent que les rétroactions les ont aidés à corriger leurs erreurs.

Questions	**Oui**	**Non**
Les rétroactions données sont compréhensibles et claires.	29	1
Les rétroactions obtenues après chaque exercice non résolu, vous ont aidé à le résoudre.	24	6

Tableau 7-8: Questionnaire sur le ressenti des étudiants sur les rétroactions reçues

7.3.4 Question 2 : quel est l'impact de l'utilisation d'AlgoInit sur les capacités de résolution des problèmes des apprenants ?

Par cette question nous cherchons à évaluer l'impact de l'utilisation d'*AlgoInit* sur la capacité de résolution des problèmes des apprenants. Pour vérifier cela, nous présentons d'abord la collecte et la méthodologie d'analyse des données et par la suite le résultat de l'évaluation.

7.3.4.1 Collecte et méthodologie d'analyse des données

Pour répondre à cette question, nous avons comparé les notes obtenues, après un test, par les apprenants du groupe ayant utilisé *AlgoInit (Groupe 1)* avec celles d'un groupe contrôle qui n'a pas utilisé l'environnement (Groupe 0). Ce dernier groupe a été constitué selon les mêmes critères que pour le groupe 1, à savoir 10 étudiants faibles, 10 moyens et 10

forts. Le groupe de contrôle (Groupe 0) a fait deux séances TD (sur les mêmes exercices) de 1h30 chacune, avec un enseignant.

Les étudiants de deux groupes ont par la suite été soumis à un test comportant trois exercices. Nous avons indiqué aux étudiants que le test serait noté afin de nous assurer qu'il soit fait sérieusement. La correction a été faite par un seul enseignant et nous avons récupéré par la suite les notes de ces 60 étudiants.

Nous calculons ensuite la moyenne de chaque groupe (Groupe 1 et Groupe 0). L'objectif est de comparer ces moyennes afin de vérifier s'il y a une différence et déterminer si cette différence est significative.

Pour vérifier cela, nous utilisons un test statistique, le **test t de Student**. Selon (Ramousse, Le Berre, & Le Guelte, 1996), ce test statistique repose sur la comparaison des moyennes de deux échantillons indépendants avec des données quantitatives. Ce test donne une décision statistique par un rejet ou non de l'hypothèse nulle (hypothèse qui suppose qu'il n'y a pas une différence significative entre les deux moyennes comparées). Pour utiliser ce test, (Ramousse et al., 1996) recommandent de vérifier : (i) le test de normalité et (ii) l'homogénéité des variances dans les deux échantillons. Si ces deux conditions sont vérifiées, alors le **test t de student** peut être utilisé.

Pour interpréter le résultat il faudra formuler deux hypothèses : H0 (hypothèse nulle) et H1. Ainsi nous formulons les hypothèses suivantes :

- **H0** : il n'y a pas une différence significative entre la moyenne du Groupe 1 et du Groupe 0.
- **H1** : il y a une différence significative entre la moyenne du Groupe 1 et du Groupe 0.

Pour analyser le résultat, nous avons utilisé le logiciel R.

7.3.4.2 Résultat

Pour nous assurer que ce test est valable sur nos données, nous avons donc réalisé les vérifications suivantes :

- dans une première étape, nous avons vérifié le test de normalité en utilisant le logiciel R, ce résultat à été positif.

- Dans une deuxième étape, nous avons vérifié l'homogénéité des variances dans les deux échantillons, ce résultat s'est également avéré positif.

Suite aux résultats positifs de deux étapes ci-dessus, nous avons effectué le **test t student** sur le logiciel R, le résultat est représenté sur la *figure 7-8*.

```
        Welch Two Sample t-test

data:  notes by gr
t = -2.6671, df = 54.722, p-value = 0.01004
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -3.6780933 -0.5219067
sample estimates:
mean in group 0 mean in group 1
           9.95           12.05
```

Figure 7-8 : Résultat de t-test (test student) sur les notes de deux groupes

La moyenne du groupe 0 (groupe contrôle) est de 9,95. Celle du groupe 1 est de 12,05. Ce qui nous donne une différence non négligeable de 2,55 points. Le *test t* nous montre que l'hypothèse *H0* est rejetée et nous donne surtout comme résultat une valeur $p = 0,01$. Cette valeur est bien inférieure à la valeur généralement acceptée ($p < 0,05$). Cela nous permet d'affirmer que cette différence de moyenne entre les deux groupes est significative et nous permet de conclure que l'utilisation d'*AlgoInit* a très certainement eu un impact positif sur la moyenne obtenue par le groupe qui l'a utilisé.

7.3.5 Question 3 : comment les apprenants évaluent l'utilisabilité d'AlgoInit ?

Nous nous intéressons à travers cette question à l'utilisabilité d'*AlgoInit*. Pour vérifier cela, nous présentons d'abord la méthodologie pour évaluer l'utilisabilité et par la suite le résultat de l'évaluation.

7.3.5.1 Collecte et méthodologie d'analyse des données

Pour évaluer l'utilisabilité, nous avons utilisé le questionnaire SUS (*System Usability Scale*). C'est un outil standard, simple, rapide à remplir et facile à comprendre pour les utilisateurs (Bangor, Kortum, & Miller, 2009). Il est largement utilisé pour évaluer l'utilisabilité des systèmes. Le questionnaire comprend dix items, qui permettent de

déterminer le niveau de satisfaction des utilisateurs d'un système. Ce niveau de satisfaction peut être exprimé sur une échelle de Lickert à cinq niveaux allant de « Pas du tout d'accord » à « Tout à fait d'accord ».

Pour que le questionnaire SUS soit interprétable, (Tullis & Stetson, 2004) recommandent au moins 12 participants au sondage. Pour mesurer l'utilisabilité d'*AlgoInit*, nous avons 30 participants. Le questionnaire SUS, présenté dans le *Tableau 7-9*, a été distribué aux étudiants dès la fin de la deuxième séance de la deuxième expérimentation.

Pour calculer le score SUS, on procède comme suit (Bangor et al., 2009) : (i) pour les items impairs (1, 3, 5, 7 et 9) on soustrait un point au score donné par l'utilisateur. Par exemple si l'utilisateur coche 3 le score est de 3-1=2 ; (ii) pour les items pairs (2, 4, 6, 8 et 10) on calcule cinq moins le score donné par l'utilisateur. Par exemple si l'utilisateur coche 1 (pas du tout d'accord) le score est de 5-1=4 ; (iii) on additionne les scores ainsi calculés pour obtenir un total sur 40. Par la suite on multiple ce score par 2,5 pour obtenir un score sur 100. Une fois le score final obtenu pour chaque utilisateur, nous calculons la moyenne de tous ces scores. Cette moyenne représente le score SUS obtenu par notre système.

Pour interpréter le résultat (score SUS) ainsi obtenu, nous avons utilisé la plage d'acceptabilité d'un système proposé par (Bangor et al., 2009) présenté *figure 7-9*.

1. Je pense que j'aimerais utiliser ce système fréquemment
2. J'ai trouvé ce système inutilement complexe.
3. J'ai trouvé ce système facile à utiliser.
4. Je pense que j'aurais besoin d'un support technique pour être capable d'utiliser ce système
5. J'ai trouvé que les différentes fonctions de ce système étaient bien intégrées.
6. J'ai trouvé qu'il y avait trop d'incohérences dans ce système.
7. Je suppose que la plupart des gens apprendraient très rapidement à utiliser ce système
8. J'ai trouvé ce système très contraignant à utiliser.
9. Je me suis senti très confiant en utilisant ce système.
10. J'ai dû apprendre beaucoup de choses avant de pouvoir utiliser ce système

Tableau 7-9: Questionnaire SUS

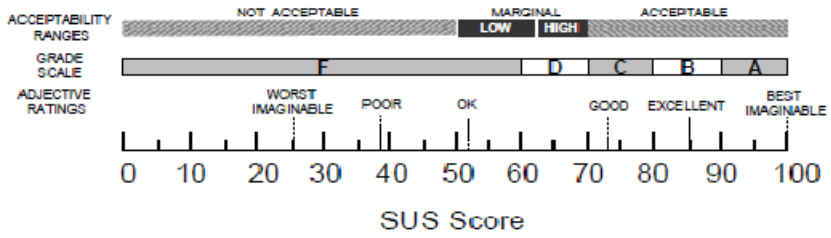


Figure 7-9: Plage d'acceptabilité d'un système par le score SUS

7.3.5.2 Résultat

Le score SUS obtenu pour *AlgoInit* est de 71,83. Selon la plage d'acceptabilité, *AlgoInit* peut être qualifié comme un « Bon » système d'après les étudiants interrogés.

7.3.6 Synthèse

Après une première expérimentation dédiée à l'efficacité de notre système, cette deuxième expérimentation était orientée sur son intérêt pédagogique. Nous avons cherché à évaluer cet intérêt selon trois axes : l'impact des rétroactions fournies par *AlgoInit*, l'impact global sur les résultats des étudiants ainsi que l'utilisabilité du système.

En ce qui concerne le premier axe, *l'impact des rétroactions,* l'analyse des traces ainsi que les retours des étudiants nous indiquent un effet positif encourageant. Sur cet axe, nous nous sommes limités à vérifier si les rétroactions générées suite à une détection d'erreurs, permettait à l'apprenant de corriger celles-ci. Nous pourrons dans une expérimentation de longue durée avec plus d'exercices évaluer l'effet de ces rétroactions entre les différents exercices d'une même famille et l'effet de ces rétroactions sur les différents niveaux de difficultés.

Nous avons comparé la moyenne des notes obtenues par le groupe ayant utilisé *AlgoInit* et celle d'un groupe contrôle assisté par un enseignant pour répondre à la deuxième question. L'analyse statistique de ces résultats a montré que l'utilisation d'*AlgoInit* a une influence significative sur les notes. Bien que le résultat obtenu pour cette question soit positif et encourageant, il nous semble raisonnable de mener des expérimentations sur des longues périodes afin de confirmer le résultat obtenu.

Enfin, nous avons utilisé le questionnaire *System Usability Scale* (SUS) pour évaluer l'utilisabilité d'*AlgoInit*, le résultat de ce test est plutôt correct avec un score supérieur à 71/100 indiquant une bonne utilisabilité.

Conclusion générale

Bilan

Dans le cadre de cette thèse, nous avons abordé la question de l'amélioration des connaissances stratégiques des apprenants dans les cours d'introduction à la programmation. L'objectif est de consolider leur capacité à utiliser les concepts appropriés lors de la résolution de problèmes de programmation. Pour arriver à cet objectif la littérature recommande que l'apprenant réalise de nombreux exercices avec des niveaux de difficulté croissants. La littérature a également souligné qu'à cause du nombre important d'apprenants et de l'hétérogénéité des niveaux, les enseignants ont des difficultés à vérifier l'exactitude de leurs solutions et à fournir l'assistance nécessaire.

Nous nous sommes donc attachés à identifier un processus d'évaluation qui permet de détecter rapidement les erreurs des apprenants, de les assister dans la correction de ces erreurs et qui serve de support pour proposer différents exercices de niveaux progressifs.

Notre problématique s'est donc axée sur trois points : *comment évaluer la solution algorithmique pour que la détection d'erreur soit significative ? Comment assister ou accompagner un apprenant qui a une difficulté pour résoudre un exercice ? Et enfin, comment organiser les exercices en niveaux de difficulté croissants permettant une progression de l'apprenant ?*

Pour répondre à la première question, ***nous avons proposé une approche d'évaluation***, qui permet de détecter les erreurs de conception des apprenants et de leur associer une signification. Ce processus est basé sur la comparaison des solutions algorithmiques des apprenants à des solutions modèle préparées par l'enseignant. Cette comparaison est réalisée en deux étapes. La première étape consiste à transformer les solutions (apprenant et modèle) en des arbres étiquetés. Chaque nœud de cet arbre est représenté par une étiquette qui traduit la signification d'une ligne d'instruction. Pour construire les étiquettes de chaque instruction, nous avons défini une base de règles en collaboration avec les enseignants de l'université de Djibouti. Une fois les solutions standardisées, la deuxième étape consiste à comparer les deux arbres étiquetés pour statuer sur la solution de l'apprenant (correcte ou incorrecte). Selon le score de l'apprenant un état (**faible, moyen, bon**) de résolution de l'exercice est affecté dans le profil de l'apprenant.

Notre contribution par cette approche d'évaluation, se présente comme un moyen de diagnostiquer les erreurs de conception des apprenants sans nous appuyer sur des solutions modèle avec erreur. Cette approche propose également un moyen de réduire le nombre de solutions modèles.

Pour répondre à la deuxième question, ***nous avons proposé un mécanisme de génération des rétroactions***. Celles-ci sont de deux types : (i) une rétroaction communiquant à l'apprenant une explication sur la façon de résoudre l'exercice. Elle est paramétrée par l'enseignant et est fournie à l'apprenant dans le cas où le nombre d'erreurs dans la solution dépasse un seuil significatif ; (ii) une rétroaction communiquant à l'apprenant ses erreurs. Le contenu de cette rétroaction est construit en se basant sur un type d'erreur et les étiquettes construites lors de l'évaluation. Pour énumérer les erreurs de l'apprenant selon des phrases compréhensibles, nous utilisons une base de règles qui transforme les étiquettes en des phrases (contenu de la rétroaction).

Pour répondre à la dernière question, ***nous avons proposé un modèle d'organisation des exercices.*** Nous avons organisé nos exercices selon les trois premiers niveaux cognitifs de la taxonomie de Bloom. Dans chaque niveau cognitif, nous répartissons nos exercices en niveaux de difficulté selon le concept qu'il met en œuvre. Pour chaque niveau de difficulté on peut proposer plusieurs exercices afin de favoriser l'acquisition des compétences liées à l'utilisation de ces concepts.

En nous basant sur ces modèles et algorithmes, ***nous avons développé un environnement Web nommé AlgoInit***. C'est dans cet environnement que nous avons réifié nos propositions. Ceci nous a également permit d'évaluer nos propositions à travers des expérimentations en milieu écologique.

Nous avons ainsi mené deux expérimentations : une première qui était destinée à évaluer l'efficacité de notre approche d'évaluation et une autre orientée sur l'intérêt pédagogique de notre système

Pour la première expérimentation, nous nous sommes inspirés des méthodologies d'évaluation des systèmes de recommandation. Dans ce cadre, nous avons utilisé une matrice de confusion pour étudier la précision d'*AlgoInit*, c'est-à-dire sa capacité à reconnaître des solutions correctes et incorrectes. Pour cela, nous avons comparé l'évaluation réalisée par notre environnement et celle d'un enseignant à travers 3 exercices. *Le résultat de cette évaluation nous a montré que notre environnement arrivait à reconnaître avec une précision de 100% les solutions correctes. En ce qui concerne les solutions incorrectes, le taux de*

reconnaissance est de 92%. L'analyse a montré que les cas où *AlgoInit* n'a pas reconnu une solution correcte correspondaient à des cas où il y a des instructions supplémentaires ou placées au mauvais endroit par rapport à la solution modèle.

Pour la deuxième expérimentation, nous avons cherché à évaluer l'intérêt pédagogique à travers trois axes : l'impact des rétroactions fournies par notre environnement, l'impact global sur les résultats des étudiants ainsi que l'utilisabilité du système.

En ce qui concerne, le premier axe, *l'impact des rétroactions*, nous avons cherché à évaluer l'apport des rétroactions générées par notre système sur la correction faite par les apprenants suite à des erreurs relevées dans leurs solutions. Pour évaluer ceci, nous nous sommes basé sur l'analyse des traces des solutions soumises par les apprenants suites aux rétroactions et sur des questionnaires soumis aux apprenants à la fin de l'expérimentation. Le résultat de cette évaluation à été généralement positif.

En ce qui concerne, le deuxième axe, *l'impact global sur les résultats des étudiants*, nous avons cherché à évaluer l'apport de l'utilisation de notre environnement sur les résultats des étudiants. Pour évaluer ceci, nous avons comparé les moyennes des notes obtenues par deux groupes : un groupe ayant utilisé notre environnement et un groupe témoin qui ne l'a pas utilisé. Le résultat de l'analyse statistique a indiqué que notre environnement a une influence significative sur les résultats des étudiants.

Pour le dernier axe concernant, *l'utilisabilité d'AlgoInit*, nous avons utilisé le classique questionnaire *System Usability Scale* (SUS). Le résultat est correct avec un score indiquant une bonne utilisabilité (71%). Nous avons toutefois des pistes d'amélioration immédiates suite aux discussions avec les apprenants (affichage des rétroactions, reprise de la solution erronée…).

Perspectives

Nos principales perspectives pour ce travail sont liées à un souhait d'amélioration de notre environnement *AlgoInit* et une volonté d'améliorer nos propositions (approche d'évaluation, assistance et organisation des exercices) ainsi qu'une perspective d'expérimentation. Nous présentons ici ces principaux souhaits d'amélioration.

Perspectives expérimentales

A moyen terme, afin de renforcer nos premiers résultats, nous envisageons de mener une expérimentation à plus large échelle : d'abord avec un nombre important d'étudiants (environ 200 étudiants) ; ensuite avec une grande quantité d'exercices et la possibilité d'utiliser l'environnement à distance (afin de permettre aux étudiants de continuer à l'utiliser en dehors de l'université). L'ajout d'exercices nous permettra de valider et d'enrichir si nécessaire notre base de règles.

Du point de vue expérimental, cela nous permettra :

- De mener une analyse plus fine de l'impact des rétroactions sur la capacité de résolution de problème. Dans cette analyse, nous cherchons à vérifier le nombre de rétroactions entre les différents niveaux et le nombre de rétroactions entre les différents exercices de même famille. Cela nous permettra de percevoir l'acquisition des concepts par les apprenants si le nombre de rétroactions liées à un concept donné tend à diminuer ;
- D'effectuer une évaluation pré/post des capacités de résolutions de problèmes des apprenants ;
- D'évaluer la capacité des apprenants à transposer les solutions algorithmiques dans divers langages de programmation. Dans cette évaluation nous mettrons en œuvre un moyen de récupérer les traces d'exécution des programmes des apprenants.

Perspectives de recherche

Notre modèle de sélection des exercices est relativement linéaire et constitue une progression à travers les différents niveaux de concepts et de difficulté. Il nous semble intéressant de réfléchir à une progression plus personnalisée incluant des exercices de remédiation. Cette réflexion implique un travail selon trois axes :

- Un premier axe porte sur la diversification des formes d'exercices. Ces exercices seront sélectionnés selon le niveau de l'apprenant et ses résultats. Cela nous amène à une réflexion sur une approche d'évaluation générique, qui doit être capable d'évaluer différentes formes d'exercices (QCM, résoudre un exercice, question à trou, lire un code…).

- Cela nous conduit également à repenser notre modèle d'organisation des exercices afin d'intégrer les remédiations, de telle sorte que chaque apprenant puisse suivre un chemin d'exercice adapté selon son niveau.
- La personnalisation des parcours met en exergue la question de la modélisation de l'apprenant. Pour cela le système doit tenir compte des besoins, des connaissances ainsi que l'historique de résolution des exercices des apprenants. Ainsi il existe plusieurs approches pour construire le modèle apprenant (Chrysafiadi & Virvou, 2013). dans cette perspective il s'agit de répondre à la question : Comment représenter l'apprenant afin de l'aider ou l'assister dans ses difficultés ? Dans cette question nous chercherons à mettre en œuvre un modèle de prise de décision qui se basera sur le modèle apprenant pour générer l'assistance adéquate ainsi que l'exercice suivant.

Une autre perspective à plus long terme, concerne un changement du contexte d'utilisation. Nous envisageons d'étendre l'utilisation de notre outil dans l'enseignement secondaire (lycée) à Djibouti où l'algorithmique est intégrée au programme de mathématiques de la seconde à la terminale. Cela soulève la question de l'utilisation d'un langage visuel type Scratch pour aborder les notions d'algorithmique et du passage graduel aux langages textuels : en pseudo-code puis dans un langage de programmation. Pour cette perspective nous pouvons nous inspirer des travaux de (Muratet, 2010).

Annexe

A. Base de règles

Les règles sont exprimées à partir des catégories d'instructions présentés page 72. La base de règles représentées ci-dessous est celle que nous avons utilisée lors de nos différentes expérimentations.

Étiquette	Condition
Incrementation	Si la ligne est composée d'une variable, un *OE (affectation)*, la même *variable que celle identifiée premièrement*, un *OA(+)* et une valeur ou variable.
Decrementation	Si la ligne est composée d'une variable, un *OE (affectation)*, la même *variable que celle identifiée premièrement*, un *OA(-)* et une valeur ou variable.
Initialisation	*Si la ligne est composée d'une variable,* un *OE (affectation),* et une valeur.
Calcul	*variable* avec un *OE (affectation)* suivi d'autre *variable* ou valeur, ses variables et valeur sont séparés par un *OA*.
Verifie	Si la ligne contient une *SC(Si).*
VerifieImbrique	*Si la ligne contient une SC(SinonSi).*
VerifieSelon	*Si la ligne contient une SC(Selon).*
Default	*Si la ligne contient une SC(Sinon).*
Repeter	Si la ligne contient une *SI.*
Donnee	Si la ligne contient un *OE (Lire)*
Afficher	Si la ligne contient un *OE(Ecrire)*

Toutefois cette base de règle n'est pas exhaustive et peut évoluer. Pour l'étendre, nous souhaitons collaborer avec des enseignants, de cette matière algorithmique et programmation, d'autres universités. Pour cela nous souhaitons proposer une page web où les enseignants peuvent compléter un formulaire selon le format suivant :

Etiquette	Condition	Explication

L'étiquette représente le mot à associer au nœud et la condition représente la règle associée pour générer cette étiquette. L'explication traduit en une phrase l'étiquette. Celle-ci est essentielle dans la génération des rétroactions.

Une fois que l'enseignant a complété une ou plusieurs lignes de la base de règles, il soumettra pour validation et les règles ainsi validées par la majorité des enseignants seront intégrées dans notre base de règles.

B. Exemple d'exercices modélisés

Enoncé :	Réaliser un algorithme permettant à un abonné EDF de calculer lui-même le montant de sa prochaine facture trimestrielle. Chaque abonné doit payer tous les trimestres un abonnement fixe de 200 DJF et 30 DJF par KW/H consommé. Pour calculer la facture il faut donc connaître les relevés de compteur de début et fin de période.
Niveau taxonomie :	Connaissance
Niveau difficulté :	1
Exercice N° :	1

Enoncé :	Ecrire un algorithme qui permet de déterminer si un entier lu est pair ou impair.
Niveau taxonomie :	Connaissance
Niveau difficulté :	2
Exercice N° :	1

Enoncé :	Ecrire un algorithme ou programme demandant la saisie d'une valeur entière et affiche ensuite si cette valeur est un multiple de 7.
Niveau taxonomie :	Connaissance
Niveau difficulté :	2
Exercice N° :	2

Enoncé :	Les étudiants ayant passé l'examen d'algorithmique en session de Juin ont été classés selon leurs notes en trois catégories : - pour une note strictement inférieure à 5, l'étudiant est éliminé. - pour une note supérieure ou égale à 5 et strictement inférieur à 10, l'étudiant passe la session de rattrapage. - pour une note supérieure ou égale à 10, l'étudiant valide le module. Ecrivez un algorithme qui demande à l'utilisateur d'entrer la note du module et affiche la situation de l'étudiant selon sa note.
Niveau taxonomie :	Connaissance
Niveau difficulté :	3
Exercice N° :	1

Enoncé :	Ecrire un algorithme qui affiche le mois en toute lettre selon son numéro en utilisant la structure conditionnelle `Selon`.
Niveau taxonomie :	Connaissance
Niveau difficulté :	3
Exercice N° :	2

Enoncé :	Ecrire un algorithme qui demande un nombre de départ, et qui calcule la somme des entiers jusqu'à ce nombre en utilisant le boucle Pour. Par exemple, si l'on entre 5, le programme doit calculer : 1 + 2 + 3 + 4 + 5 = 15
Niveau taxonomie :	Connaissance
Niveau difficulté :	4
Exercice N° :	1

Enoncé :	Écrire un algorithme qui fait la somme des données saisit par l'utilisateur, la saisie des nombres s'arrête lorsque l'utilisateur saisie un 0.
Niveau taxonomie :	Connaissance
Niveau difficulté :	4
Exercice N° :	2

Enoncé :	Ecrire un algorithme qui demande un nombre à l'utilisateur, et l'informe ensuite si ce nombre est positif, négatif ou nul.
Niveau taxonomie :	Compréhension
Niveau difficulté :	1
Exercice N° :	1

Enoncé :	On place un capital de 500DJF sur un compte rémunéré à 3% par an. Ecrire un algorithme qui permet de calculer le nombre d'années au bout desquelles le capital sera doublé.
Niveau taxonomie :	Application
Niveau difficulté :	1
Exercice N° :	1

Enoncé :	Écrire un algorithme qui demande deux entiers A et N. Ensuite l'algorithme lit la liste de N nombres entiers saisie par l'utilisateur. L'algorithme doit afficher combien de fois l'entier A apparaît dans cette liste.
Niveau taxonomie :	Application
Niveau difficulté :	1
Exercice N° :	2

Bibliographie

ACM, AIS, & IEEE. (2005). *Computing Curricula 2005, the overview Report*. New York.

Ala-Mutka, K. (2005). A survey of automated assessment approaches for programming assignments. *Computer Science Education*, *15*(2), 83–102.

Ala-Mutka, K., Uimonen, T., & Järvinen, H.-M. (2004). Supporting students In C++ Programming Courses with Automatic Program Style Assessment. *Journal of Information Technology Education*, *3*(1), 245–262.

Altadmri, A., & Neil, C. B. (2015). 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 522–527).

Amir, O., & Ya'akov, K. (2013). Plan Recognition and Visualization in Exploratory Learning Environments. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, *3*(3), 16.

Anderson, L. W., Krathwohl, D. R., Airasian, P. W., Cruikshank, K. A., Mayer, R. E., Pintrich, P. R., … Wittrock, M. C. (2001). *A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives*. New York: Longman.

Arifi, S. M., Abdellah, I. N., Zahi, A., & Benabbou, R. (2015). Automatic program assessment using static and dynamic analysis. In *Third world Conference on Complex Systems(WCCS)* (pp. 1–6).

Bangor, A., Kortum, P., & Miller, J. (2009). Determining what individual SUS scores mean: Adding an adjective rating scale. *Journal of Usability Studies*, *4*(3), 114–123.

Beaubouef, T., Lucas, R., & Howatt, J. (2001). The UNLOCK system: enhancing problem solving skills in CS-1 students. *ACM SIGCSE Bulletin*, *33*(2), 43–46.

Beaubouef, T., & Mason, J. (2005). Why the High Attrition Rate for Computer Science Students: Some Thoughts and Observations. *ACM SIGCSE Bulletin*, *37*(2), 103–106.

Bennedsen, J., & Caspersen, M. E. (2007). Failure rates in introductory programming. *ACM SIGCSE Bulletin*, *39*(2), 32–36.

Bergin, S., & Reilly, R. (2006). Predicting introductory programming performance: A multi-institutional multivariate study. *Computer Science Education*, *16*(4), 303–323.

Bey, A., & Bensebaa, T. (2011). ALGO+, an assessment tool for algorithmic competencies. In *Global Engineering Education Conference (EDUCON). IEEE* (pp. 941–946).

Bloom, B. S. (1956). Taxonomy of educational objectives. Vol. 1: Cognitive domain. *New-York: Mckay*, 20–24.

Bloom, B. S. (1971). Mastery learning: Theory and practice. *New York: Holt, Rinehart & Winston.*, 47–63.

Bosc-Miné, C. (2014). Caractéristiques et fonctions des feed-back dans les apprentissages. *L'Année Psychologique*, *114*(2), 315–353.

Boud, D., & Molloy, E. (2013). *Feedback in higher and professional education: understanding it and doing it well*. (eds, Ed.). London : Routledge.

Bouhineau, D. (2013). Utilisation de traits sémantiques pour une méthodologie de construction d'un système d'aide dans un EIAH de l'algorithmique. In *6e conférence sur les Environnements Informatiques pour l'Apprentissage Humain* (pp. 141–152). IRIT Press.

Carless, D. (2006). Differing perceptions in the feedback process. *Studies in Higher Education*, *31*(2), 219–233.

Chan, C. C., Tsui, M. S., Mandy, Y. C., & Hong, J. H. (2002). Applying the Structure of the Observed Learning Outcomes (SOLO) Taxonomy on Student's Learning Outcomes: An empirical study. *Assessment & Evaluation in Higher Education*, *27*(6), 511–527.

Chatzopoulou, D. I., & Economides, A. A. (2010). Adaptive assessment of student's knowledge in programming courses. *Journal of Computer Assisted Learning*, *26*(4), 258–269.

Chookaew, S., Panjaburee, P., Wanichsan, D., & Laosinchai, P. (2014). A Personalized E-Learning Environment to Promote Students' Conceptual Learning on Basic Computer Programming. *Procedia - Social and Behavioral Sciences*, *114*, 815–819.

Chrysafiadi, K., & Virvou, M. (2013). Student modeling approaches : A literature review for the last decade. *Expert Systems with Applications*, *40*(11), 4715–4729.

De Ketele, J.-M. (2010). Ne pas se tromper d'évaluation. *Revue Française de Linguistique Appliquée*, *15*(1), 25–37.

Delorme, F., Delestre, N., & Pécuchet, J.-P. (2004). Évaluer l'apprenant à l'aide de cartes conceptuelles. In *Technologies de l'Information et de la Connaissance dans l'Enseignement Supérieur et l'Industrie* (pp. 25–31).

Djelil, F. (2016). *Conception et évaluation d'un micromonde de Programmation Orientée-Objet fondé sur un jeu de construction et d'animation 3D*. Thèse de doct. Université Blaise Pascal - Clermont II.

Durand, G. (2006). *La scénarisation de l'évaluation des activités pédagogiques utilisant les Environnements Informatiques d'Apprentissage Humain*. Thèse de doct. Université de Savoie.

Eitelman, S. M. (2006). Computer tutoring for programming education. In *proceedings of the 44th annual Southeast regional conference* (pp. 607–610).

Engel, G., & Roberts, E. (2001). *Computing curricula 2001. Computer science*. *The Joint Task Force on Computing Curricula, ACM Press. and IEEE Computer Society Press, New York*.

Flowers, T., Carver, C. A., & Jackson, J. (2004). Empowering students and building confidence in novice programmers through Gauntlet. In *34th Annual Frontiers in Education, 2004. FIE 2004.* (pp. 10–13). https://doi.org/10.1109/FIE.2004.1408551

Fonte, D., Da Cruz, D., Gançarski, A. L., & Henriques, P. R. (2013). A flexible dynamic system for automatic grading of programming exercises. *2nd Symposium on Languages, Applications and Technologies, SLATE 2013*, *29*, 129–144. https://doi.org/10.4230/OASIcs.SLATE.2013.129

Fuller, U., Johnson, C. G., Ahoniemi, T., Cukierman, D., Hernan-Losada, I., Jackova, J., … Thompson, E. (2007). Developing a Computer Science-specific Learning Taxonomy. *ACM SIGCSE Bulletin*, *39*(4), 152–170.

Gardner-Medwin, A. R. (2006). Confidence based marking: towards deeper learning and better exams. In C. Bryan & K. Clegg (Eds.) (pp. 141–149). Innovative assessment in higher education, London : Routledge.

Geib, C. W., & Goldman, R. P. (2009). A probabilistic plan recognition algorithm based on plan tree grammars. *Artificial Intelligence*, *173*(11), 1101–1132.

Gerdes, A., Heeren, B., & Jeuring, J. (2017). Ask-Elle: an adaptable programming tutor for Haskell giving automated feedback. *International Journal of Artificial Intelligence in Education*, *27*(1), 65–100.

Gomes, A., & Mendes, a. J. (2007). Learning to program - difficulties and solutions. In *International Conference on Engineering Education – ICEE*.

Gouli, E., Gogoulou, A., Papanikolaou, K., & Grigoriadou, M. (2004). COMPASS: an adaptive Web-based concept map assessment tool. In *Proceedings of the First International Conference on Concept Mapping* (pp. 295–302).

Granet, V. (2014). *Algorithmique et programmation en Java*. (Dunod, Ed.) (4 édition:).

Greiff, S., Scheiter, K., Scherer, R., Borgonovi, F., Britt, A., Graesser, A., … Rouet, J.-F. (2017). Adaptive problem solving: Moving towards a new assessment domain in the second cycle of PIAAC. *OECD Education Working Papers 156*.

Guskey, T. R. (2007). Closing achievement gaps : revisiting Benjamin S.Bloom's "Learning for Mastering." *Journal of Advanced Academics*, *19*(1), 8–31.

Hadji, C. (1990). *L'évaluation, règles du jeu* (ESF).

Harlen, W. (2007). *Assessment of Learning*. London: SAGE.

Hattie, J., & Timperly, H. (2007). The Power of Feedback. *Review of Education Research*, *77*(1), 81–112.

Haydar, C. A. (2014). *Les systèmes de recommandation à base de confiance*. Thèse de doct. Université de Lorraine.

HO, C.-H. (2001). Some phenomena of problem decomposition strategy for design thinking: differences between novices and experts. *Design Studies*, *22*(1), 27–45.

Ihantola, P., Ahoniemi, T., Karavirta, V., & Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli calling international conference on computing education research* (pp. 86–93).

Inventado, P. S. (2019). Promoting Mastery Learning in an Introductory Programming Course. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (pp. 1285–1285).

Jackson, D., & Usher, M. (1997). Grading Student Programs using ASSYST. *ACM SIGCSE Bulletin*, *29*(1), 335–339.

Jenkins, T. (2002). On The Difficulty of Learning to Program. In *Proceedings of the 3rd Annual Conference of the LTSN Centre for information and Computer Sciences.* (pp. 53–58). Retrieved from http://www.psy.gla.ac.uk/~steve/localed/jenkins.html

Jonassen, D. H. (2010). *Learning to solve problems: A handbook for designing problem-solving learning environments*. New York: Routledge.

Juwah, C. (2003). Using peer assessment to develop skills and capabilities. *United States Distance Learning Association*, *17*(1), 39–50.

Kaasbøll, J. (2002). Learning Programming. *University of Oslo*.

Kelly, T., & Buckley, J. (2006). A Context-Aware Analysis Scheme for Bloom's Taxonomy. In *14th IEEE International Conference on Program Comprehension(ICPC'06)* (pp. 275–284).

Keuning, H., Jeuring, J., & Heeren, B. (2018). A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *ACM Transactions on Computing Education (TOCE)*, *19*(1), 3.

Konecki, M. (2014). Problems in programming education and means of their improvement. *DAAAM International Scientific Book, 2014*, 459–470.

Labat, J.-M. (2002). EIAH: Quel retour d'informations pour le tuteur? In *Technologies de l'Information et de la Communication dans les Enseignements d'ingénieurs et dans l'industrie* (pp. 81–88).

Lahtinen, E., Ala-Mutka, K., & Järvinen, H.-M. (2005). A study of the difficulties of novice programmers. *ACM SIGCSE Bulletin*, *37*(3), 14–18. https://doi.org/10.1145/1151954.1067453

Lane, H. C., & VanLehn, K. (2005). Teaching the tacit knowledge of programming to novices with natural language tutoring. *Computer Science Education*, *15*(3), 183–201. https://doi.org/10.1080/08993400500224286

Le, N.-T. (2016). A classification of adaptive feedback in educational systems for programming. *Systems*, *4*(2), 22.

Le, N.-T., & Pinkwart, N. (2014). Towards a classification for programming exercises. In *Proceedings of the 2nd Workshop on AI-supported Education for Computer Science.*

Lefevre, M. (2009). *Processus unifié pour la personnalisation des activités pédagogiques : méta-modèle, modèle et outils*. Thèse de doct. Université Claude Bernard Lyon 1.

Lishinski, A., Yadav, A., Enbody, R., & Good, J. (2016). The Influence of Problem Solving Abilities on Students' Performance on Different Assessment Tasks in CS1. In *proceedings of the 47th ACM technical symposium on computing science education* (pp. 329–334).

Lister, R., & Leaney, J. (2003). Introductory programming, criterion-referencing, and bloom. In *ACM SIGCSE Bulletin* (pp. 143–147).

Luck, M., & Joy, M. (1999). A secure on-line submission system. *Software: Practice and Experience*, *29*(8), 721–740.

Luxton-Reilly, A., Simon, Beker, B. A., Albluwi, I., Giannakos, M., Amruth N., K., … Szabo, C. (2018). Introductory Programming: A Systematic Literature Review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (pp. 55–106).

Mayer, R. E., Dyck, J. L., & Vilberg, W. (1986). Learning to program and learning to think: what's the connection? *Communications of the ACM*, *29*(7), 605–610.

Mcgettrick, A., Boyle, R., Ibbet, R., LIoyd, J., Lovegrove, G., & Mander, K. (2005). Grand challenges in computing : Education—a summary. *The Computer Journal*, *48*(1), 42–48.

Mead, J., Simon, G., John, H., Richard, J., Juha, S., Caroline, S. C., & Linda, T. (2006). A Cognitive Approach to Identifying Measurable Milestones for Programming Skill Acquisition. *ACM SIGCSE Bulletin*, *38*(4), 182–194.

Milne, I., & Rowe, G. (2002). Difficulties in Learning and Teaching Programming — Views of Students and Tutors. *Education and Information Technologies*, *7*(1), 55–66. https://doi.org/10.1023/A:1015362608943

Mufti-Alchawafa, D. (2008). *Modélisation et représentation de la connaissance pour la conception d'un système décisionnel dans un environnement informatique d'apprentissage en chirurgie.* Thèse de doct. Université Joseph-Fourier-Grenoble.

Muratet, M. (2010). *Conception, réalisation et évaluation d'un jeu sérieux de stratégie temps réel pour l'apprentissage des fondamentaux de la programmation*. Thèse de doct. Université Toulouse III - Paul Sabatier. Retrieved from http://tel.archives-ouvertes.fr/tel-00554287/

Narciss, S. (2008). Feedback strategies for interactive learning tasks. *Handbook of Research on Educational Communications and Technology*, *3*, 125–144.

Nicol, D. (2007). E-assessment by design: using multiple-choice tests to good effect. *Journal of Further and Higher Education*, *31*(1), 53–64.

Nijimbere, C. (2015). *L'enseignement de savoirs informatiques pour débutants, du second cycle de la scolarité secondaire scientifique à l'université en France. Une étude comparative*. Thèse de doct. Université Paris Descartes.

Novak, J. D., & Gowin, D. B. (1984). *Learning how to learn.* (C. U. Press, Ed.).

Ott, C., Robins, A., & Shephard, K. (2016). Translating Principles of Effective Feedback for Students into the CS1 Context. *ACM Transactions on Computing Education (TOCE)*, *16*(1), 1.

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., … Paterson, J. (2007). A survey of literature on the teaching of introductory programming. *Acm Sigcse*, *39*(4), 204–223. Retrieved from http://dl.acm.org/citation.cfm?id=1345441

Piton, T. (2011). *Une Méthodologie de Recommandations Produits Fondée sur l'Actionnabilité et l'Intérêt Économique des Clients : Application à la Gestion de la Relation Client du groupe VM Matériaux*. Thèse de doct. Université de Nantes.

Polya, G. (1957). How to solve it. *Prentice University Press*.

Prévit, D. (2008). *Génération d'exercices et analyse multicritère automatique de réponses ouvertes: PépiGen, un système auteur en algèbre élémentaire*. Thèse de doct. Université du Maine.

Py, D. (1998). Quelques méthodes d'intelligence artificielle pour la modélisation de l'élève. *Sciences et Techniques Educatives*, *5*(2).

Qian, Y., & James, L. (2017). Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Transactions on Computing Education (TOCE)*, *18*(1), 1.

Rahman, khirulnizam abd, & Nordin, M. J. (2007). A Review on the Static Analysis Approach in the Automated Programming Assessment Systems. In *Proceedings of the national Conference on Programming, 07,*. Kuala Lampur, Malaysia.

Rahman, khirulnizam abd, Nordin, M. J., & Che, W. S. B. C. (2008). Automated programming assessment using the pseudocode comparison technique: Does It Really Work? In *International Symposium in Information Technology (ITSim)* (pp. 1–4).

Raîche, G. (2004). L'évaluation des compétences à l'enseignement supérieur. Vers une vision intégratrice de l'évaluation des apprentissages. *Montréal, Québec : Université Du Québec à Montréal.*

Ramousse, R., Le Berre, M., & Le Guelte, L. (1996). Introduction aux statistiques.

Reek, K. A. (1989). The TRY System - or - How to Avoid Testing Student Programs. In *ACM SIGCSE Bulletin* (pp. 112–116).

Renumol, V. G., Jayaprakash, S., & Janakiram, D. (2009). Classification of cognitive difficulties of students to learn computer programming. *Indian Institute of Technology*, 12. Retrieved from http://dos.iitm.ac.in/publications/LabPapers/techRep2009-01.pdf

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, *13*(2), 137–172. https://doi.org/10.1076/csed.13.2.137.14200

Romli, R., Sulaiman, S., & Zamli, K. (2013). Designing a test set for structural testing in automatic programming assessment. *International Journal of Advanced Soft Computing and Application*, *5*(3), 1–24.

Rongas, T., Kaarna, A., & Kalviainen, H. (2004). Classification of computerized learning tools for introductory programming courses: Learning approach. *Proceedings - IEEE International Conference on Advanced Learning Technologies, ICALT 2004*, 678–680. https://doi.org/10.1109/ICALT.2004.1357618

Santos, Á., Gomes, A., & Mendes, A. (2013). A taxonomy of exercises to support individual learning paths in initial programming learning. In *Frontiers in Education Conference, IEEE* (pp. 87–93). https://doi.org/10.1109/FIE.2013.6684794

Scallon, G. (2000). L'évaluation formative. *Éditions Du Renouveau Pédagogique*.

Schorsch, T. (1995). An Automated Self-Assessment Tool To Check Pascal Programs For Syntax, Logic And Style Errors. In *ACM SIGCSE Bulletin* (pp. 168–172).

Selby, C. C. (2015). Relationships: computational thinking, pedagogy of programming, and Bloom's Taxonomy. In ACM (Ed.), *Proceedings of the workshop in primary and secondary computing education* (pp. 80–87).

Shaffer, D., Doube, W., & Tuovinen, J. (2003). Applying Cognitive Load Theory to Computer Science Education. In *15th Workshop of the Psychology of programming Interest Group* (pp. 333–346). Keele UK. Retrieved from http://www.ppig.org/sites/ppig.org/files/2003-PPIG-15th-shaffer.pdf

Shuhidan, S., Hamilton, M., & D'Souza, D. (2009). A Taxonomic Study of Novice Programming Summative Assessment. In *Proceedings of the Eleventh Australasian Conference on Computer Education-Volume 95* (pp. 147–156).

Sweller, J. (1988). Cognitive Load During Problem Solving: Effects on Learning. *Cognitive Science*, *12*(2), 257–285.

Tan, P. H., Ting, C. Y., & Ling, S. W. (2009). Learning difficulties in programming courses: Undergraduates' perspective and perception. In IEEE (Ed.), *ICCTD 2009 - International Conference on Computer Technology and Development* (pp. 42–46). https://doi.org/10.1109/ICCTD.2009.188

Tchounikine, P. (2009). Précis de recherche en ingénierie des EIAH. Retrieved October 20, 2016, from http://membres-liglab.imag.fr/tchounikine/Precis.html

Tribollet, B., Françoise, L., & Laurence, J. (2000). Protocoles d'emploi des cartes conceptuelles au lycée et en formation des maîtres. *Trema*, *18*, 61–78.

Truong, N., Roe, P., & Bancroft, P. (2004). Static Analysis of Students' Java Programs. In *Proceedings of the Sixth Australasian Conference on Computing Education-Volume 30* (pp. 317–325).

Tullis, T. S., & Stetson, J. N. (2004). A comparison of questionnaires for assessing website usability. In *Usability professional association conference* (pp. 1–12).

Wang, F. L., & Wong, T.-L. (2008). Designing Programming Exercises with Computer Assisted Instruction. In Springer (Ed.), *International Conference on Hybrid Learning and Education* (pp. 283–293). Berlin, Heidelberg.

Wang, T., Su, X., Wang, Y., & Ma, P. (2007). Semantic similarity-based grading of student programs. *Information and Software Technology*, *49*(2), 99–107.

Whalley, J., Clear, T., Robbins, P., & Thompson, E. (2011). Salient elements in novice solutions to code writing problems. In *Proceedings of the thirteenth Austalasian Computing Education Conference- Volume 114* (pp. 37–46). Australian Computer Society, Inc.

Winslow, L. E. (1996). Programming pedagogy---a psychological overview. *ACM SIGCSE Bulletin*, *28*(3), 17–22. https://doi.org/10.1145/234867.234872

Zamin, N., Mustapha, E. E., Sugathan, S. K., Mehat, M., Ellia, & Anuar. (2006). Development Of A Web-Based Automated Grading System For Programming Assignments Using Static Analysis Approach. In *Proceedings of the international Conference on Technology and Operations Management*.