Analyse automatique de la maintenabilité logicielle basée sur une ontologie, vers une interaction dialogique avec un auditeur artificiel

Sébastien Bertrand, Pierre-Alexandre Favier, Jean-Marc André

Analyse automatique de la maintenabilité logicielle basée sur une ontologie, vers une interaction dialogique avec un auditeur artificiel

Sébastien Bertrand[1,2], Pierre-Alexandre Favier[1,3], and Jean-Marc André[1,3]

[1]IMS Laboratory, University of Bordeaux, UMR 5218 CNRS, France
[2]onepoint, Sud-Ouest, France
[3]ENSC, Bordeaux INP, France

s.bertrand@groupeonepoint.com, {pierre-alexandre.favier, jean-marc.andre}@ensc.fr

Résumé

La prédiction de la maintenabilité logicielle est particulièrement importante en milieu industriel. Afin d'améliorer l'adaptabilité et l'explicabilité des approches de régression existantes, notre objectif est de formaliser le discours sur la maintenabilité logicielle, d'abord entre experts humains pour aller vers une interaction dialogique interactive avec un agent d'analyse artificiel. Dans ce but, nous avons créé une ontologie de la maintenabilité afin de représenter les éléments d'un programme Java, et inférer les éléments de conception qui le structurent. Nous avons testé avec succès la détection d'une architecture en couches et d'un patron de conception singleton. Cette première étape nous permettra de structurer un système multi-agents qui devra interagir dialogiquement avec l'équipe de développement.

Mots-clés

Ontologie, maintenabilité logicielle, analyse de programme.

Abstract

Software maintainability prediction is especially important in an industrial context. In order to improve the adaptability and explicability of existing regression approaches, our objective is to formalize the discourse on software maintainability, first between human experts to move towards a dialogical interaction with an artificial analysis agent. To this end, we have created a maintainability ontology in order to represent the elements of a Java program, and to infer the design elements that structure it. We have successfully tested the detection of a layered architecture and a singleton design pattern. This first step will allow us to structure a multi-agent system that will have to interact dialogically with the development team.

Keywords

Ontology, software maintainability, program analysis.

1 Introduction

La maintenabilité logicielle est définie par le standard ISO/IEC 25010 comme la facilité d'évolution d'un logiciel en fonction de changements techniques et fonctionnels [16]. La prédiction de la maintenabilité logicielle aborde principalement le problème de régression consistant à prédire l'effort de maintenance en fonction d'un ensemble de métriques [3, 10]. L'effort de maintenance est souvent représenté par le nombre de lignes modifiées par classe, ou par l'indice de maintenabilité [14]. Les métriques utilisées comme variables explicatives sont majoritairement celles introduites par Li et Henry [20] et par Chidamber et Kemerer [7].

Cependant, certaines limites concernant les métriques ont été exposées. La taille d'une classe par exemple est un facteur confondant les métriques orientées objet [11, 8, 24]. Les grandes classes rassemblent plus de code, et donc l'effort de maintenance est mécaniquement plus important, mais de grandes classes ayant les mêmes responsabilités peuvent avoir une maintenabilité très différente en fonction de leurs implémentations. Le découpage interne des responsabilités, les cadriciels utilisés, l'utilisation des fonctionnalités offertes par le langage de programmation (par exemple les stream en Java ou LINQ en C#), ou même le nommage des éléments du programme vont influencer l'implémentation concrète d'une classe et donc sa maintenabilité. Il existe également un problème lié à l'utilisation du nombre de lignes modifiées par classe comme variable expliquée, car les grandes classes sont évidemment plus susceptibles d'être modifiées. De plus, l'indice de maintenabilité est également critiqué, car la formule semble être assez arbitraire, et seulement corrélé avec le nombre de lignes du code [14, 25]. Plus généralement, des doutes ont été émis sur l'efficacité des modèles de prédiction de la maintenabilité logicielle [23]. En s'attaquant aux limites des métriques, nous pourrions donc contribuer à améliorer la prédiction de la maintenabilité.

Notre proposition est d'exploiter l'analyse de programme basée sur ontologie pour la prédiction de la maintenabilité logicielle. À notre connaissance, cette approche est nouvelle en ce qui concerne la prédiction de maintenabilité, nos travaux sont donc à comparer aux travaux d'analyse de programme basée sur ontologie, qui ont pour objectifs d'aider à la détection automatique de patrons de conception, de bugs,

ou plus généralement au support des activités de développement [1, 2, 18, 26, 27]. Une analyse de programme basée sur une ontologie représente ce programme dans une base de connaissances, c'est à dire une ontologie peuplée par des individus, pour permettre la réalisation d'analyses de programme manuelles ou automatiques. En alimentant une ontologie écrite en Web Ontology Language (OWL) avec une représentation du langage de programmation ciblé, des patrons de conception pertinents et des sous-caractéristiques élémentaires de la maintenabilité, nous devrions être en mesure d'affiner notre analyse en fonction du contexte et de l'équipe de développement et de fournir des informations utiles sur la maintenabilité, sur la base d'une évaluation élémentaire de ces sous-caractéristiques. Pour mettre en œuvre cette approche, nous avons développé *Javanalyser*, un outil qui analyse un projet Java pour produire une ontologie de programme correspondante. L'objectif de cette étape est de formaliser la manière dont nous discutons de maintenabilité.

Ces travaux sont effectués en collaboration avec les équipes de la société onepoint, en particulier les architectes logiciels qui nous fournissent un retour critique sur notre approche en vue de son exploitation sur des projets de développement.

2 Proposition

Pour permettre la discussion sur la maintenabilité, nous proposons de développer une ontologie de la maintenabilité organisée en trois concepts fondamentaux : les éléments de programme, les éléments de conception et les éléments de maintenabilité. Les éléments de programme sont conçus pour représenter la hiérarchie des concepts dans le code source. Les éléments de conception sont des concepts liés à la macrostructure du code, comme une couche d'une architecture en couches, ou comme un adaptateur du patron de conception éponyme. Ces éléments de conception sont définis sur la base des éléments du programme. Enfin, les éléments de maintenabilité représentent des sous-caractéristiques élémentaires et composées de la maintenabilité.

Le premier aspect de notre proposition est de définir et de décrire au sein d'une ontologie la maintenabilité et ses sous-caractéristiques afin de pouvoir adapter notre analyse au contexte et à l'équipe de développement. Le contexte peut en effet justifier des structures particulières pour des raisons techniques ou fonctionnelles, ce dernier cas correspond à la complexité essentielle du problème à résoudre [6], par opposition à la complexité accidentelle qui peut potentiellement caractériser un problème de maintenabilité. L'équipe de développement peut également avoir des préférences liées à sa culture ou à ses compétences, par exemple sa langue maternelle influencera le nommage des variables, ou certains patrons de conception pourraient être préféré parce qu'ils sont déjà bien connus par l'équipe.

Les ontologies ont été initialement proposées pour résoudre des problèmes liés à l'intégration des données et des systèmes d'information en fournissant un vocabulaire commun partagé et permettre l'utilisation d'un raisonneur automatisé pour déduire de nouvelles connaissances [17]. Elles peuvent être utilisées au niveau du schéma ou des données pour assurer l'interopérabilité entre les systèmes, les applications et les bases de données [17]. Les ontologies constituent également une partie importante de la solution à d'autres problèmes où la représentation des connaissances peut être d'une grande aide, par exemple dans des domaines tels que la formation en ligne, l'intelligence artificielle ou l'exploration de données [17]. Ainsi, l'utilisation d'une ontologie de la maintenabilité permettra la discussion sur la maintenabilité d'une manière formelle, avec le bénéfice d'un raisonneur automatisé et la compréhensibilité puisque l'évaluation de la maintenabilité sera associée à l'évaluation de ses sous-caractéristiques.

Éléments de maintenabilité. La maintenabilité est définie comme l'efficacité avec laquelle un logiciel peut être corrigé, amélioré ou adapté aux modifications du système environnant ou des spécifications, soit techniques, soit fonctionnelles. Selon la spécification ISO, la maintenabilité est composée de cinq sous-caractéristiques [16] : modularité, réutilisabilité, analysabilité, modifiabilité et testabilité. Selon Boehm et al., la maintenabilité joue un rôle clé pour la qualité finale du système [4]. Les caractéristiques de la qualité des logiciels entretiennent des relations plusieurs-à-plusieurs, la maintenabilité, telle que définie par la norme ISO, est présentée sous forme d'arbre et ne décrit donc pas la complexité réelle des relations entre ses sous-caractéristiques. Par exemple, la norme ISO définit la testabilité comme une sous-caractéristique de la maintenabilité alors que la testabilité contribue également à de nombreuses autres caractéristiques, allant de l'adéquation fonctionnelle aux performances [5]. De plus, par rapport à d'autres sous-caractéristiques, la modularité joue un rôle central lors de la phase de maintenance, où il est essentiel de comprendre, d'analyser, de modifier et de tester le logiciel en cours de maintenance, avec un impact minimal entre les composants [13]. Ainsi, la maintenabilité peut être décrite comme une propriété émergente de ces sous-caractéristiques, qui ont entre elles un réseau de relations, certaines plus influentes que d'autres.

Éléments de programme. Pour évaluer les sous-caractéristiques de la maintenabilité, nous avons choisi de nous appuyer sur une analyse de programme qui s'articule autour d'une ontologie qui contient une représentation du langage ou du paradigme de programmation visé. Une base de connaissances est alimentée par des assertions sur des individus représentant les éléments constitutifs du code source du programme ciblé et les analyses sont effectuées par des requêtes sur la base de connaissances [18, 26, 27]. L'utilisation d'une ontologie pour modéliser les programmes aide à fournir une abstraction commune pour l'extraction des caractéristiques et des relations internes du programme et aide à faciliter la coopération entre les outils d'analyse de programme [27]. De plus, les ontologies sont facilement extensibles par l'ajout contextuel de nouveaux concepts et de nouvelles relations, et un raisonneur enrichira et adaptera alors automatiquement

l'analyse de programme [27]. Enfin, une ontologie d'analyse de programme aide à intégrer et à modéliser les informations provenant de diverses sources [27]. En représentant le code source d'un programme dans une base de connaissances dérivée de l'ontologie de maintenabilité, nous serons en mesure d'évaluer les sous-caractéristiques de maintenabilité à partir de son code source.

Éléments de conception. Dans le cadre de notre collaboration avec onepoint, les architectes logiciels ont exprimé l'importance de l'enrichissement sémantique de l'évaluation de la maintenabilité. Le premier point était de suggérer d'inclure des concepts sur les patrons de conception, tels que l'architecture en couches ou des patrons logiciels classiques (singleton, adaptateur, ...), afin que l'évaluation de la maintenabilité prenne en compte l'objectif du code source. Le deuxième point était de suggérer d'inclure des relations sur le flux de données, car l'analyse des flux de données est importante pour comprendre la finalité et évaluer la complexité du code source. Ces deux points ont été étudiés par des approches précédentes d'analyse de programmes basées sur ontologie [18, 2, 1] et, malgré certaines limites, elles ont démontré leur efficacité. En ajoutant à l'ontologie de la maintenabilité les concepts et les relations modélisant les patrons de conception logiciels et les flux de données, nous devrions être en mesure de vérifier que l'évaluation de la maintenabilité est plus significative et fonction de la macro-structure du code source.

Métriques. Enfin, afin de prédire la maintenabilité, nous devrons évaluer concrètement le code source. Cela se fait classiquement par le traitement d'un ensemble de métriques, les plus utilisées étant celles introduites par Chidamber et Kemerer et par Li et Henry [3, 10]. En 1991, Chidamber et Kemerer ont introduit 6 métriques de programmation orientées objet [7] :

- le nombre de méthodes pondérées par classe est la somme des complexités des méthodes locales, où ces dernières sont mesurées par une autre métrique (comme par exemple la complexité cyclomatique de McCabe [21], ou le nombre de lignes de code) ;
- la profondeur de l'arbre d'héritage est le nombre de parents dont la classe hérite ;
- le nombre d'enfants est le nombre de classes qui héritent directement de la classe considérée ;
- le couplage entre les objets compte le nombre de classes non liées par héritage, c'est-à-dire les classes sur lesquelles la classe agit ou qui agissent sur la classe (e.g. : appel de méthodes ou maintien de variables d'instance) ;
- le nombre de réponses pour une classe est la cardinalité de l'ensemble des méthodes locales et des méthodes appelées par celles-ci ;
- le manque de cohésion des méthodes est le nombre d'ensembles indépendants de méthodes liées par au moins une variable d'instance commune.

En 1993, Li et Henry ont ajouté pour leur étude 5 nouvelles métriques [20] :

- le couplage de transmission de messages est le nombre d'appels d'une classe à d'autres classes ;
- le couplage d'abstraction de données est le nombre de variables définies dans la classe et ayant un type abstrait ;
- le nombre de méthodes est le nombre de méthodes déclarées localement dans la classe ;
- la première métrique de la taille est le nombre de lignes de code de la classe ;
- la deuxième métrique de la taille est le nombre d'attributs locaux et de méthodes déclarés dans la classe.

La plupart des métriques liées à l'architecture du programme sont essentiellement des compteurs spécialisés d'une sorte ou d'une autre. Par exemple, la complexité cyclomatique compte le nombre de prédicats, la profondeur de l'arbre d'héritage compte le nombre de classes parentes, ou le couplage de transmission de messages compte le nombre d'appels "étrangers". Il est intéressant de noter que le manque de cohésion des méthodes est plus complexe, car il est associé à des sous-ensembles indépendants de méthodes au sein d'une classe. La plupart des métriques seraient alors facilement calculables à partir des éléments du programme permettant à l'ontologie de maintenabilité de servir de seul substrat pour la prédiction de la maintenabilité.

Résumé. L'ontologie de la maintenabilité est organisée autour de trois concepts fondamentaux : les éléments de programme, les éléments de conception et les éléments de maintenabilité.

3 Résultats

Pour implémenter cette approche, la première étape a été de développer une ontologie de la maintenabilité, et la seconde étape consistait à l'encapsuler dans un analyseur syntaxique personnalisé, appelé *Javanalyser*. Nous avons choisi de traiter spécifiquement des programmes Java. En effet, une grande quantité de code industriel est à notre disposition dans ce langage, car largement utilisé chez onepoint. Il convient de noter que l'ontologie de la maintenabilité et *Javanalyser* sont toujours en cours de développement.

3.1 L'ontologie de la maintenabilité

L'ontologie de la maintenabilité a été conçue pour décrire le code source d'un programme en termes d'éléments de programme qui sont des concepts et des rôles au sens ontologique. Comme nous avons choisi *Spoon* [22] pour l'analyse du code Java, nous avons fait correspondre précisément ces éléments de programme au méta-modèle de *Spoon*, ce dernier étant conçu pour modéliser du code Java grâce à des arbres de syntaxe abstraits à la fois complets et très compréhensibles pour des développeurs Java. Ainsi, les éléments du programme peuvent être décrits comme une implémentation dans une logique de description du méta-modèle *Spoon*. La figure 1 présente une extraction partielle des éléments de programme : une `Method` est un genre d'`Executable`, qui est un genre de `TypedElement`, qui est typé selon un `Type`.

L'implémentation de l'ontologie est faite en Web Ontology

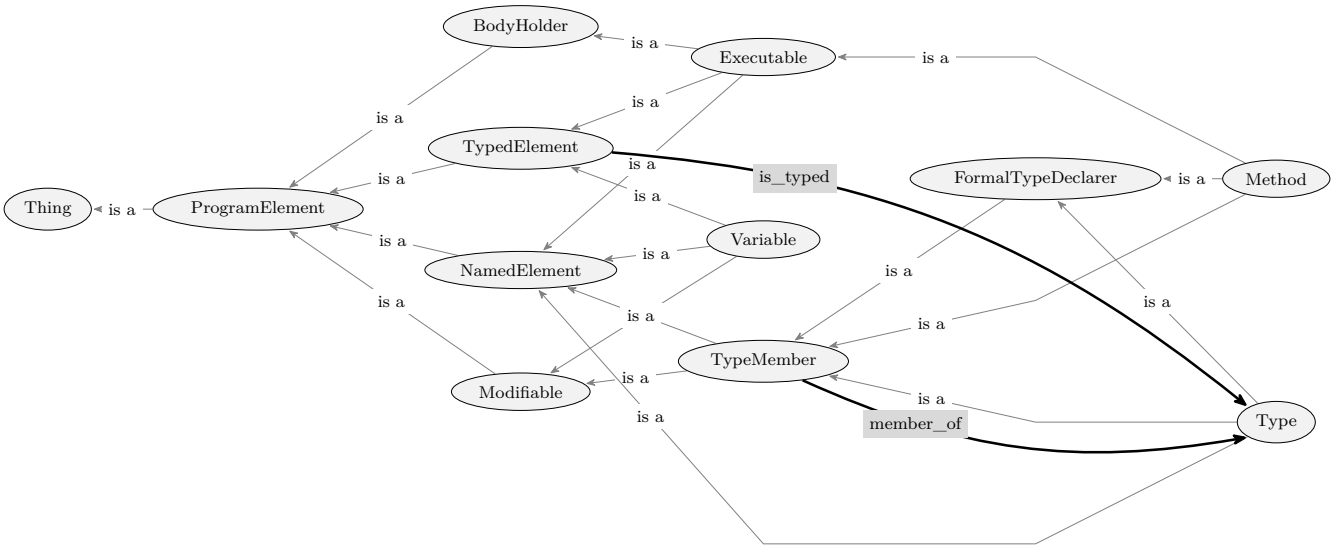

FIGURE 1 – Une extraction de l'ontologie de la maintenabilité

Language (OWL), et nous avons choisi d'utiliser une logique de description SROIQ (OWL 2 DL) complété par des règles Semantic Web Rule Language (SWRL) lorsque OWL ne nous permettait pas de modéliser les concepts nécessaires. Des rudiments d'architecture en couches ont été implémentés dans les éléments de modèle, et pour la version actuelle les caractéristiques de la norme ISO 25010 [16] ont été extraites dans les éléments de maintenabilité sous la forme d'une liste plate. Dans cette phase de prototypage, nous travaillons à l'implémentation de certains patrons de conception couramment utilisés, mais tous les patrons existants ne seront pas implémentés car ils encombreraient l'ontologie de la maintenabilité, ce qui pourrait avoir un impact négatif sur sa compréhensibilité et ses performances. Les inférences de l'ontologie de la maintenabilité ont été testées en Python grâce à *Owlready2*. Cette librairie est très pratique pour effectuer les déclarations et les contrôles sur l'ontologie, car elle permet des manipulations de haut niveau et d'adopter une hypothèse locale de monde clos [19].

3.2 Javanalyser

L'objectif de *Javanalyser* est d'alimenter l'ontologie de la maintenabilité pour un projet Java afin de caractériser la structure de son code source. La figure 2 présente son actigramme SADT.

Le traitement d'un projet Java comporte trois étapes principales : (1) l'analyse du projet Java, (2) les assertions de son arbre syntaxique abstrait, et (3) l'alimentation de l'ontologie du projet Java. Nous avons choisi d'utiliser *Spoon* [22] pour l'analyse syntaxique d'un projet Java. Son utilisation a été simple et efficace. En outre, la documentation Java de son méta-modèle a été d'une grande aide pour développer les éléments de programme de l'ontologie de maintenabilité. Les assertions des éléments de l'arbre syntaxique abstrait sont réalisées en implémentant la classe abstraite *CtScanner* de *Spoon*, qui parcourt hiérarchiquement chaque nœud de l'arbre syntaxique abstrait. Cela implique de créer un individu pour chaque nœud et de déclarer ses relations avec tous les individus déjà présents dans l'ontologie. De plus, comme l'ont fait les auteurs de *Graph4Code* [1], il est essentiel d'utiliser systématiquement des noms pleinement qualifiés pour chaque nœud, afin de les identifier de manière unique (paquets, classes, variables, ...). Enfin, un gestionnaire d'ontologie a été implémenté avec l'aide l'OWL API [15]. Sa responsabilité concerne les opérations concrètes sur l'ontologie, comme le chargement de l'ontologie de la maintenabilité, l'alimentation de l'ontologie du programme Java comme demandé par le scanneur et la fermeture des individus pour permettre au raisonneur de travailler efficacement sur l'ontologie du projet Java. Enfin, l'ontologie du programme Java est produite et permet l'accès et le raisonnement automatique sur la base des connaissances qu'elle contient.

Voici deux exemples d'ontologie de programme Java produite par *Javanalyser* :

- le premier, figure 3, concerne la détection d'un Singleton, le raisonneur de l'ontologie a automatiquement classé les individus (losange violet) dans les bons concepts en ajoutant les relations "`has_individual`" adéquates jusqu'à inférer que la classe A est un singleton ;
- le second, figure 4, présente la détection d'un ensemble de couches, où le raisonneur de l'ontologie a classé avec "`has_individual`" les classes et méthodes de ces classes en se basant sur les relations qu'elles ont entre elles, c'est-à-dire "`has_member`" et "`invoke`".

3.3 Discussion

Pour résumer les résultats, *Javanalyser* traduit le code source d'un programme Java dans le langage de l'ontologie qui a été délibérément construite pour être proche de la syntaxe du langage de programmation ciblé. L'ontologie

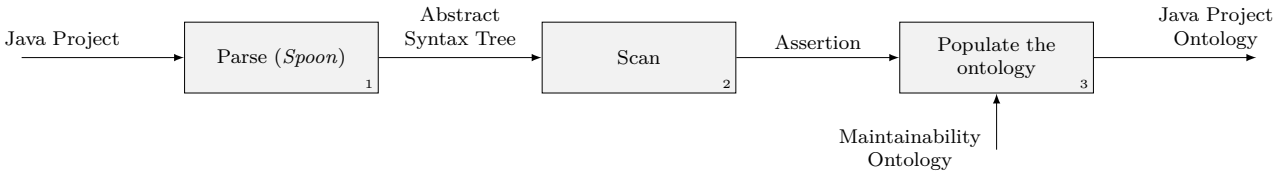

FIGURE 2 – Actigramme de *Javanalyser* (A0)

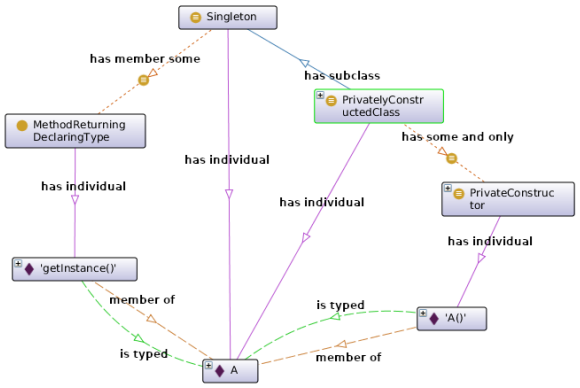

FIGURE 3 – Exemple d'analyse d'un Singleton

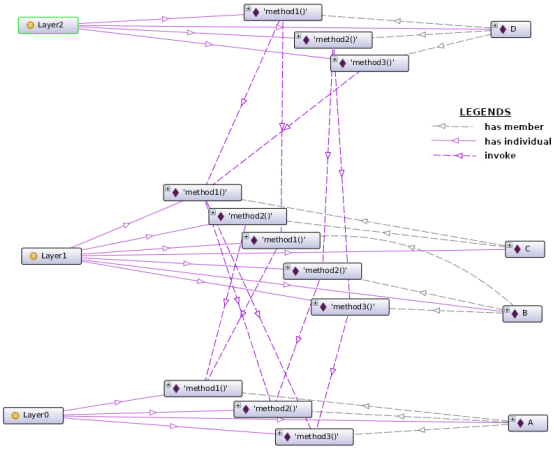

FIGURE 4 – Exemple d'analyse d'une architecture en couche

de la maintenabilité joue un rôle central en définissant les concepts et les relations des éléments de programme, des éléments de conception modèle et des éléments de maintenabilité, et donc enrichissant le méta-modèle de *Spoon* [22] avec des concepts de programmation de plus haut niveau. L'ontologie du programme Java produite contient une représentation complète de ce programme, sur laquelle un raisonneur peut effectuer un premier niveau d'analyses. Cette représentation peut également servir de substrat sur lequel d'autres agents peuvent effectuer et partager leurs analyses. L'ontologie de la maintenabilité formalise donc un langage pour discuter du code source du point de vue de la maintenabilité.

Méthodologiquement, nos travaux s'appuient sur l'analyse des programmes basée sur ontologie, cette dernière définissant le méta-modèle du langage ciblé. La reconnaissance de patrons de conception avec l'aide d'une ontologie, appliquée au singleton, a été mise en œuvre par [18] et démontre quelques limitations intéressantes concernant la modélisation avec une logique de description. La détection de nombreux autres patrons de conception a également été implémenté en SWRL par [2], les auteurs se comparent favorablement à des outils analogues testés sur JUnit, JHotDraw et JRefactory. La détection automatique des bogues sur des programmes Java a été réalisée par [26] et se compare favorablement au bien-connu *FindBugs* [1]. Les auteurs de [27] utilisent le moteur de raisonnement de SWI-Prolog et des API personnalisées pour exploiter et éventuellement mettre à jour la base de connaissances, cette dernière a été testée dans plusieurs situations telles que la détection de boucles canoniques ou l'analyse de pointeurs. Par ailleurs, *Graph4Code* [1] a une approche plus globale en déclarant tous les programmes ciblés dans la même base de connaissances et en reliant l'analyse des codes à des ressources non structurées tels que de la documentation ou des fils de discussion *StackOverflow*. *Graph4Code* exploite des cas d'utilisation intéressants concernant la mise en place de bonnes pratiques ou le débogage avec *StackOverflow*. L'analyse du code est effectuée avec l'aide de WALA [2], principalement en extrayant deux types de relations : "`flowsTo`" qui capture le flux de données du code en traçant l'usage des variables, et "`immediatelyPrecedes`" qui capture le flux de contrôle du code. Notre approche couvre de nombreux aspects de ces travaux en introduisant les éléments du modèle qui enrichissent les éléments de programme du méta-modèle détenu par l'ontologie. Cependant, nous nous différencions par l'utilisation du méta-modèle de *Spoon* pour décrire les éléments du programme, nous permettant de saisir toute la structure du code dans l'ontologie.

En outre, notre approche se distingue notamment de l'approche classique de prédiction de la maintenabilité [20, 3, 10]. Les études classiques construisent un modèle de maintenabilité qui rassemble et organise un ensemble de métriques pour servir de prédicteurs de la maintenabilité logicielle, afin de réduire les coûts de maintenance [9]. L'on-

1. `http://findbugs.sourceforge.net/`

2. `https://github.com/wala/WALA`

tologie de la maintenabilité n'a pas pour but de prévoir directement la maintenabilité, mais plutôt de prédire ses sous-caractéristiques, et en croisant ces évaluations avec la caractérisation des modèles, de permettre une prédiction plus précise de la maintenabilité du logiciel. En déduisant la maintenabilité globale de ses sous-caractéristiques, nous souhaitons permettre aux équipes de développement de pouvoir personnaliser leur propre définition de ce qu'est une bonne maintenabilité dans le contexte spécifique de chaque projet et à terme de pouvoir suggérer comment l'améliorer. Perspective plus lointaine, cette approche permettra aux développeurs de s'inscrire dans une interaction dialogique avec un auditeur artificiel, en "discutant" avec lui de la maintenabilité, de sa mesure et des bonnes pratiques en vigueur dans l'équipe. L'ajout de sens et la personnalisation ne sont possibles que s'il y a une sémantique attachée à l'analyse de chaque sous-caractéristique. Ce préalable à l'interaction dialogique et à l'apprentissage est impossible par une approche basée sur des métriques globales, et nous l'espérons accessible par une approche basée sur une ontologie comme celle que nous proposons.

4 Conclusions

Dans nos travaux, nous avons construit une ontologie de la maintenabilité pour structurer et décrire le code source d'un programme Java en trois niveaux :

- les éléments de programme qui représentent le méta-modèle du langage ciblé ;
- les éléments de conception qui représentent la structure du programme ;
- les sous-caractéristiques de la maintenabilité, pour permettre des prévisions plus précises et plus personnalisables.

Cette ontologie de la maintenabilité est alimentée par *Javanalyser* un programme que nous avons créé pour produire une ontologie d'un programme Java, qui peut servir de substrat pour le travail d'un ensemble d'agents.

L'ontologie de la maintenabilité découple l'analyse de la maintenabilité de la compréhension du programme en formalisant la manière d'évaluer la maintenabilité à partir de ses sous-caractéristiques et de la structure du programme. Ainsi, l'ontologie de la maintenabilité constitue une première étape dans la modélisation de la manière dont nous discutons de la maintenabilité.

Nos futurs travaux concerneront la finalisation d'une version plus complète de l'ontologie de la maintenabilité, décrivant des patrons de conception classiques tels que ceux décrits par le Gang Of Four [12] et le traitement d'une quantité importante de projets Java pour garantir la fiabilité de notre approche. Ensuite, nous devrons comparer notre approche au modèle classique de prédiction de la maintenabilité logicielle pour évaluer son efficacité et, à terme, aller vers une véritable interaction dialogique entre les agents artificiels et humains, basée sur l'ontologie de la maintenabilité.

Remerciements. Nous remercions les équipes de onepoint pour leur collaboration, et en particulier Ahmed Alami, Damien Bonvillain, Jérôme Fillioux, Dan Gugenheim et Jérôme Lelong pour leurs conseils avisés.

Références

[1] Ibrahim Abdelaziz, Julian Dolby, James P. McCusker, and Kavitha Srinivas. Graph4Code : A Machine Interpretable Knowledge Graph for Code. *arXiv :2002.09440 [cs]*, May 2020.

[2] Awny Alnusair, Tian Zhao, and Gongjun Yan. Rule-based detection of design patterns in program code. *International Journal on Software Tools for Technology Transfer*, 16(3) :315–334, June 2014.

[3] Hadeel Alsolai and Marc Roper. A systematic literature review of machine learning techniques for software maintainability prediction. *Information and Software Technology*, 119 :106214, March 2020.

[4] Barry Boehm, Celia Chen, Kamonphop Srisopha, and Lin Shi. The Key Roles of Maintainability in an Ontology for System Qualities. In *INCOSE International Symposium*, volume 26, pages 2026–2040, 2016.

[5] Barry Boehm and Nupul Kukreja. An Initial Ontology for System Qualities. *INSIGHT*, 20(3) :18–28, September 2017.

[6] Brooks. No Silver Bullet Essence and Accidents of Software Engineering. *Computer*, 20(4) :10–19, April 1987.

[7] Shyam R. Chidamber and Chris F. Kemerer. Towards a Metrics Suite for Object Oriented Design. In *OOPSLA '91 : Conference Proceedings on Object-Oriented Programming, Systems, Languages, and Applications*, volume 26, pages 197–211, Phoenix, Arizona, USA, November 1991. Association for Computing Machinery.

[8] Melis Dagpinar and Jens H. Jahnke. Predicting maintainability with object-oriented metrics - an empirical comparison. In Arie van Deursen, Eleni Stroulia, and Margaret-Anne D. Storey, editors, *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.*, pages 155–164. IEEE Computer Society, 2003.

[9] Sara Elmidaoui, Laila Cheikhi, and Ali Idri. Towards a Taxonomy of Software Maintainability Predictors. In Álvaro Rocha, Hojjat Adeli, Luís Paulo Reis, and Sandra Costanzo, editors, *New Knowledge in Information Systems and Technologies*, volume 930 of *Advances in Intelligent Systems and Computing*, pages 823–832. Springer, 2019.

[10] Sara Elmidaoui, Laila Cheikhi, Ali Idri, and Alain Abran. Empirical Studies on Software Product Maintainability Prediction : A Systematic Mapping and Review. *e-Informatica Software Engineering Journal*, 13(1) :141–202, 2019.

[11] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics. *IEEE Transactions on Software Engineering*, 27(7) :630–650, July 2001.

[12] Erich Gamma, Richard Helm, Ralph E Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995. OCLC : 961356420.

[13] Ahmad Haboush, Mohammad Alnabhan, Anas Al-Badareen, Mohammad Al-nawayseh, and Bassam El-Zaghmouri. Investigating Software Maintainability Development : A case for ISO 9126. *International Journal of Computer Science Issues*, 11(2) :18–23, March 2014.

[14] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A Practical Model for Measuring Maintainability. In Ricardo Jorge Machado, Fernando Brito e Abreu, and Paulo Rupino da Cunha, editors, *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, pages 30–39. IEEE Computer Society, 2007.

[15] Matthew Horridge and Sean Bechhofer. The OWL API : A Java API for OWL ontologies. *Semantic Web*, 2(1) :11–21, 2011.

[16] ISO/IEC. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. Standard ISO/IEC 25010 :2011, ISO/IEC, 2011.

[17] Maria Keet. *An Introduction to Ontology Engineering*. College Publications, 2020.

[18] Damir Kirasić and Danko Basch. Ontology-Based Design Pattern Recognition. In Ignac Lovrek, Robert J. Howlett, and Lakhmi C. Jain, editors, *Knowledge-Based Intelligent Information and Engineering Systems*, volume 5177, pages 384–393, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[19] Jean-Baptiste Lamy. Owlready : Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies. *Artificial Intelligence in Medicine*, 80 :11–28, July 2017.

[20] Wei Li and Sallie Henry. Object-Oriented Metrics that Predict Maintainability. *Journal of Systems and Software*, 23(2) :111–122, November 1993.

[21] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4) :308–320, December 1976.

[22] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon : A library for implementing analyses and transformations of Java source code. *Software : Practice and Experience*, 46(9) :1155–1179, September 2016.

[23] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. A systematic review of software maintainability prediction and metrics. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 367–377, October 2009.

[24] Dag I. K. Sjøberg, Bente Anda, and Audris Mockus. Questioning Software Maintenance Metrics : A Comparative Case Study. In *ESEM '12 : Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 4, 2012.

[25] Arie van Deursen. Think Twice Before Using the "Maintainability Index", August 2014.

[26] Lian Yu, Jun Zhou, Yue Yi, Ping Li, and Qianxiang Wang. Ontology Model-Based Static Analysis on Java Programs. In *2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 92–99, July 2008.

[27] Yue Zhao, Guoyang Chen, Chunhua Liao, and Xipeng Shen. Towards Ontology-Based Program Analysis. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming*, volume 56 of *Leibniz International Proceedings in Informatics*, pages 26 :1–26 :25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.