Licence 3 Informatique parcours Info

Université de Lille

Inria

Rapport de stage

MooseCritics, un outil d'analyse statique de règles architecturales

Romain Degrave

Du 11/04/2022 au 23/06/2022

Tuteur entreprise : Mme Anne Etien

Parc scientifique de la Haute-Borne 40, avenue Halley - Bât A - Park Plaza 59650 Villeneuve d'Ascq - France

Tuteur universitaire : Mr Romain Rouvoy

Université de Lille - Département Informatique - Cité scientifique - Bâtiment M3 59655 Villeneuve-d'Ascq

Remerciements

Je tiens tout d'abord à remercier ma tutrice professionnelle Mme. Anne Etien, pour m'avoir accordé ce stage et pour son accompagnement et assistance tout au long de celui-ci.
Je remercie également mon tuteur universitaire M. Romain Rouvoy, pour ces précieux conseils au sujet de ce rapport de stage.
De même, je ne saurais oublier de remercier la partie de l'équipe de recherche travaillant sur Moose, tout particulièrement M. Nicolas Anquetil qui m'aura aussi supervisé tout le long de ce stage.
Toujours dans cette équipe, je remercie également Mme. Clotilde Toullec et M. Soufyane Labsari, tout deux ingénieurs travaillant sur Moose, dont l'assistance sur les aspects plus technique de ce stage auront su me débloquer plus d'une fois.

Résumé

Afin d'assurer qu'un projet ou logiciel dispose d'un code de qualité, plusieurs aspects sont à considérer. Parmi ceux-ci on peut citer les règles architecturales, appliquées pour vérifier que du code vérifie des propriétés architecturales de bonne qualité au sein du projet. Il est donc primordial de pouvoir vérifier le respect de ses règles, et pouvoir dans le cas contraire trouver les violations de ces règles afin de les corriger. Pour répondre à ce besoin, une solution serait de bénéficier d'un outil permettant l'écriture et la visualisation de règles à appliquer sur un projet, nous permettant de localiser et signaler chaque violation de celles-ci.
C'est pour implémenter cet outil que j'ai intégré durant ce stage l'équipe de recherche RMoD, spécialisée dans le génie logiciel, et que moi, Romain Degrave, étudiant en 3ème année de licence d'Informatique, ai travaillé aux côtés de ces chercheurs et ingénieurs utilisant la plateforme d'analyse logicielle Moose pour y développer MooseCritics, un outil d'analyse statique de règles architecturales. Ce rapport permet de montrer les différents aspects de l'implémentation de cette solution et les résultats de son application directe.
C'est sur cette plateforme que l'outil prévu a pu voir le jour et remplir ses fonctions :
- Pouvoir être lié au système de bus de Moose, afin de recevoir un modèle, représentation abstraite d'un projet à analyser, mais aussi pour propager le résultat de l'analyse
- Donner à l'utilisateur le moyen de représenter des règles architecturales
- Exécuter ses règles sur le modèle, pour dénicher et afficher les violations
Ces violations peuvent également être exportées pour être visualisées dans d'autres outils Moose. De même, les règles sont importables et exportables afin de pouvoir les transférer entre utilisateurs, ou les charger à chaque utilisation, comme dans un contexte d'intégration continue.
MooseCritics aura pu être également utilisé sur du code de l'industriel Berger-Levrault, nécessitant de vérifier le respect de règles architecturales dans le cadre d'une migration de certaines de leurs applications. Ainsi, le rapport présente également un retour d'expérience sur l'outil développé.
Pour conclure, l'outil est déjà utilisable par des industriels et reste en développement pour améliorer encore ces fonctionnalités.

Abstract

To insure that a project or software is built with quality, several aspects must be considered. Among those are architectural rules, applied on the code to check whether or not it verifies good architectural properties, within a project. It is therefore essential to be able to check if those rules are respected, and be able to find every violations to those rules otherwise, in order to fix them. To answer this need, a solution would be to have a tool allowing us to write and visualize architectural rules to apply on a project, enabling us to locate and raise attention on every existing violation of the rule.

It is with that purpose that I joined the research team RMoD during this internship, which is specialized in software engineering and that I, Romain Degrave, studying in my final year of bachelor in computer science, have worked alongside researchers and engineers using Moose, a platform of software analysis, to develop MooseCritics, a tool for static analysis of architectural rules. This report is made to show the differents sides of the implementation of this solution, and the results of its direct application.

It is on this platform that the expected tool took shape and fulfilled its responsabilities :

- Being linked to Moose's bus system, in order to receive a model, an abstract representation of a project to analyse, but also to propagate the results of an analysis
- Allow the user to represent architectural rules
- Execute those rules on the model, to be able to locate and print the violations

Those violations can also be exported, as a means to visualize them using other tools from Moose. Additionaly, rules are importable and exportable to allow transfer between users, or even to load them at every use, making it practical in continuous integration.

MooseCritics was also used on actual code from the industrial Berger-Levrault, requiring adherence to their architectural rules in regards to a migration of several of their applications. Therefore, the report also presents feedback on the designed tool.

Finally, the tool is now usable by industrials and remains in development to improve its functionalities.

DPP : Recherche de stage

L'option de détermination du projet professionnel que j'ai choisie a été la recherche de stage. En effet, après avoir fait une première L3 l'année dernière au cours de laquelle je n'avais pas obtenu de stage, j'ai considéré que cette option était primordiale pour moi, afin de pouvoir identifier les potentielles erreurs que j'ai pu commettre dans mes démarches jusqu'alors et ainsi les corriger à l'aide des divers ateliers proposés par le BAIP.

1) Atelier : Trouver son stage à l'international

Cet atelier aura été à mon sens très intéressant, n'étant absolument pas renseigné sur les stages à l'étranger de quelque façon que ce soit.
L'atelier aura également été très complet, en nous apportant de nombreuses informations sur tout les aspects d'un stage à l'étranger, en passant par les différents sites susceptibles de contenir des offres de stage dans d'autres pays mais également le mode de recrutement favori selon le pays.
Les à côtés peut-être moins réfléchis au premier abord auront également été couverts, comme les démarches concernant l'assurance, les visas, le logement ou d'autres plus particulières selon les pays d'accueil.
N'étant pas opposé à l'idée d'un stage à l'étranger, cet atelier m'aura permis de bien comprendre la réalité concernant ceux-ci, l'importance d'y mettre le temps nécessaire mais aussi tout ce qu'il faut pour pouvoir commencer les démarches nécessaires.

2) Atelier : Rédiger sa lettre de motivation

Pour cet atelier et celui qui a suivi, le contexte aura légèrement changé, étant donné que j'avais à ce moment déjà eu la chance d'obtenir ce stage, ce qui a donc légèrement changé mes motivations.
Si j'ai pris cet atelier, c'était pour pouvoir avoir plus d'informations et de conseil sur les différents « formats » de lettres de motivation, étant personnellement prompt à m'enfermer dans l'idée « vous / moi / nous » pour structurer mes lettres, ce qui m'a peut-être porté préjudice par le passé, étant incapable d'atteindre les attentes de certains recruteurs avec ce format.

L'atelier aura donc été lui aussi très pertinent, me permettant de découvrir et préciser d'autres formats de lettres comme les lettres « matching » qui consistent à faire correspondre les besoins du recruteur et de son offre à nos compétences, ou encore les lettres « lambda », moins cordiales et formatées pour laisser place à la spontanéité et à la personnalité, potentiellement plus pertinentes pour certaines petites structures à la recherche de ce genre de profil.
Cet atelier à lui aussi permis de considérer des aspects insoupçonnés, comme sur les lettres dans le contexte d'une candidature spontanée ou sur les mails rédigés pour transmettre cette lettre.
J'ai donc beaucoup appris pour de futures candidatures, mais ai également pu mettre à profit ces acquis dans un contexte différent : les lettres de motivations pour les candidatures de Master.

3) Atelier : Trouver son stage

Ce dernier atelier a été lui aussi effectué après l'obtention de mon stage. Ayant obtenu celui-ci grâce à l'Université, après avoir contacté Mme. Anne Etien par mail à la suite d'un cours magistral de génie logiciel en nous informant que son équipe recherchait des stagiaires, et en ayant ensuite suivi ses directives qui m'auront amené à apprendre Pharo et rendre un test technique composé d'une implémentation de liste chaînée et d'un constructeur de documentation.
Dans ce contexte assez particulier, j'étais intéressé par cet atelier afin d'apprendre peut-être d'autres manières ou plateformes permettant de trouver un stage, et c'est avec cette envie que je m'y suis inscrit.
Celui-ci m'aura cependant fait réfléchir à une autre dimension de cette recherche, l'importance de pouvoir faire un bilan personnel avant même de commencer à rechercher son stage, afin de pouvoir concevoir une stratégie réfléchie pour encadrer cette recherche.
On nous a également présenté plus en détail certains outils proposés par le BAIP pour nous aider à réaliser ce bilan, tels que le « PEC » ou le questionnaire « PerformanSe », tout deux nous permettant de mieux se mettre en avant, en nous aidant à identifier nos expériences et compétences pour mieux les faire ressortir sur un CV ou sur une lettre de motivation.

4) Entretiens JLMI

J'ai également pu obtenir trois entretiens dans le cadre des Journées Lilloises de la MIAGE et de l'Informatique, qui même si ils n'auront pu se conclure avec d'autres entretiens par la suite ou une offre de stage, m'auront permis de m'entretenir avec des recruteurs très tôt dans l'année, dont certains qui auront fait des remarques intéressantes sur mon CV (comme l'absence des dates sur mon cursus que j'ai ainsi pu corriger).

5) Conclusion

Malgré le fait que la moitié de ces ateliers auront été effectués après avoir trouvé mon stage, ceux-ci auront pu me permettre de remettre en question de nombreux aspects de ma recherche de stage et me donner des conseils précieux et souvent insoupçonnés qui sauront m'aider pour de futures démarches. D'autres conseils auront même pu m'être utile et mis à profit dans des contextes différents comme pour les candidatures de Master, ce qui est là également un gain inattendu de cette option que j'ai pu suivre en DPP.

Sommaire

I. Introduction

Plusieurs millions de lignes de code, représentant des dizaines de milliers de classes, comptant ensemble des centaines de milliers de méthodes : c'est aujourd'hui l'envergure (minimale) de nombreux projets et logiciels orientés objet que l'on peut retrouver dans l'industrie. Et lorsque que parmi tant de contenu se cachent des violations de règles architecturales (par exemple si l'on veut que chaque interface Java soit implémentée, avec un nom d'implémentation particulier), non détectées par les analyseurs syntaxiques, une question se pose très rapidement. Comment pouvoir représenter ces règles, afin de détecter ces violations et mettre en place les correctifs nécessaires ?

Ce besoin est aujourd'hui celui de l'industriel Berger-Levrault, qui dans le cadre d'une migration de frameworks graphiques de certaines de leurs applications, a besoin de vérifier le respect de plusieurs règles architecturales, et surtout pouvoir corriger le code contenant déjà des violations.

Intéressé par les problématiques du génie logiciel et de la qualité logicielle, j'ai intégré l'équipe RMod et en particulier les chercheurs et ingénieurs travaillant sur Moose, une plateforme d'analyse logicielle basée sur le langage Pharo.

Afin de répondre à la problématique posée, la solution entrevue est la suivante : développer un outil, MooseCritics, permettant de créer de manière simple et intuitive des règles architecturales à appliquer sur une représentation abstraite de la structure d'un logiciel. Cet outil doit également pouvoir exécuter ces règles sur cette représentation, afin de relever les entités qui violent ces règles, pour les afficher à l'utilisateur. Cet outil s'intègre à Moose et donc à son écosystème. En conséquence, l'outil doit pouvoir récupérer la représentation du logiciel, exécuter les règles sur celui-ci et fournir le résultat de l'exécution de ces règles, exportable ou manipulable par d'autres outils déjà existants dans Moose.

Parmi les besoins de Berger-Levrault se trouve aussi celui de l'intégration continue, impliquant donc des fonctionnalités d'importation et d'exportation de règles.

Mais ces besoins permettent également une application concrète et immédiate de cet outil dans le contexte sur lequel il est voué à s'appliquer, accordant ainsi l'occasion de représenter le gain de temps effectif que son utilisation apporte, ainsi que les réponses aux besoins de différents utilisateurs auxquels celui-ci peut s'adresser.

II. Contexte

1) Inria

Ce stage a pris place dans le centre Inria de l'Université de Lille, situé sur la Haute-Borne. L'implantation d'Inria à Lille, désormais présent sur deux sites différents et comptant 15 équipes-projets, s'insère dans une dynamique régionale et nationale avec des acteurs de la recherche comme les laboratoires CRIStAL, des acteurs académiques tels que l'Université de Lille et des écoles d'ingénieurs, mais aussi des start-ups et d'importants acteurs de l'industrie, tout cela sur des axes de recherches variés, les sciences des données, les systèmes cyberphysiques mais également le génie logiciel.

C'est dans ce dernier domaine que s'inscrit RMod, l'équipe-projet qui m'a accueilli ces derniers mois. Fondée en 2009 et dirigée par M. Stéphane Ducasse, celle-ci est derrière la création et le développement du langage Pharo et de la méta-plateforme dédiée à l'analyse logicielle ; Moose.

2) Pharo et Moose

Le langage Pharo dérive lui même d'un autre langage, Smalltalk et d'une de ces implémentations, Squeak. Il se démarque par sa simplicité et son interactivité, laissant libre place et facilitant grandement l'expérimentation grâce notamment à ses outils qui offrent aux développeurs un retour direct sur les objets et la possibilité de les manipuler après exécution.

Ce langage est aujourd'hui utilisé par de nombreuses entreprises de tout horizon, comme le prouve le Pharo Consortium regroupant plus de 50 acteurs industriels et académiques.

Ma tâche relevant de l'analyse logicielle, celle-ci a été effectuée sur Moose, et j'ai été encadré par les chercheurs et ingénieurs de l'équipe travaillant sur cette plateforme. Moose étant une méta-plateforme d'analyse basée sur Pharo, permettant une représentation de la structure de logiciels et leur analyse à l'aide d'outils personnalisables et liés entre eux.

3) Contexte de la mission

La mission de ce stage s'inscrit dans un besoin du monde industriel, celui de l'application et plus particulièrement du respect des règles architecturales. Le respect de celles-ci permet d'avoir un logiciel plus facile à maintenir et laisse aux développeurs la capacité de le faire évoluer avec plus de facilité. Par exemple, on peut décider sur un projet Java, d'avoir dans le nom de chaque interface le suffixe « Intf », et que celles-ci doivent obligatoirement être implémentées, par une classe partageant le même nom mais avec comme suffixe « Impl ».

L'un des points les plus insidieux ici est que, dans l'alternative où, dans certaines parties du projet, ces règles ne sont pas respectées, les problèmes peuvent ne pas nécessairement générer d'erreurs à la compilation ou dans un analyseur syntaxique classique. On peut de cette manière se retrouver avec du code « fonctionnel », mais malgré tout cachant des problèmes sous-jacents qui peuvent ainsi s'accumuler. Dès lors, ceux-ci engendreront un problème d'ordre bien plus important lorsque que le code sera voué à évoluer, ce qui est le propre de tout projet ou application qui est utilisé par une entreprise ou ses clients, encore plus lorsque ces projets existent depuis un certain temps.

C'est le cas aujourd'hui de l'entreprise Berger-Levrault, cherchant à effectuer une migration de certaines de leurs applications d'un framework Java à un autre, GWT vers Angular. Une migration pour laquelle une solution automatisée a été développée, mais rendue actuellement difficile par les nombreuses violations de règles architecturales au sein du code de ces applications. En effet, l'outil recherche un pattern dans le code d'origine pour le traduire dans le langage cible. Si le pattern recherché ne se retrouve pas dans le code, parce que la règle architecturale n'a pas été respectée, le code ne sera pas automatiquement traduit et devra donc l'être manuellement. Cette migration prend environ 30 minutes pour être complétée, et celle-ci doit être relancée après chaque correction si les erreurs sont corrigées manuellement, un processus de correction qui est ainsi très coûteux.

4) Problématique

Comment développer et intégrer des solutions permettant de représenter de telles règles, mais également pouvoir les appliquer sur un logiciel et vérifier que celles-ci sont respectées, et, le cas échéant, pouvoir obtenir la nature et l'emplacement où ce situe ces violations ?

5) Solution envisagée

La solution envisagée prendrait la forme d'un outil. Le but de cet outil serait de pouvoir écrire et représenter de manière intuitive des règles architecturales, pour pouvoir ensuite les exécuter sur le logiciel, et ainsi être capable de détecter les violations potentielles et les afficher à l'utilisateur.

L'outil devra être relié à l'écosystème de Moose. Il s'appuiera donc sur une représentation abstraite du logiciel (i.e. un modèle). Il doit aussi pouvoir propager les violations détectées pour offrir la possibilité de les utiliser au sein d'autres outils de la plateforme, insérant ainsi cet outil dans la même philosophie que les autres déjà présents.

III. Background

Afin de pouvoir avoir une pleine compréhension de ce rapport, il est nécessaire de présenter plus en détail la plateforme d'analyse logicielle utilisée au cours de celui-ci ; Moose.
Cette partie du rapport permet de présenter les différents aspects de Moose sur lesquels l'outil vient se rattacher, de même que les capacités d'analyse de la plateforme et ainsi la pertinence de développer notre solution sur Moose.

1) MooseIDE

Parmi les nombreux projets qui composent Moose, l'un d'entre eux possède une place importante et contient la totalité des outils de Moose disposant d'une interface utilisateur ; MooseIDE.

a) Description de l'existant

MooseIDE contient de nombreux navigateurs différents divisés selon trois rôles distincts permettant à un utilisateur d'effectuer une analyse complète et personnalisée de son projet. Certains outils sont dédiés aux requêtes, afin de réduire l'échelle de l'analyse aux entités qui nous intéressent (*Queries Browser* et l'outil développé durant ce stage, le *Model Critic Browser*). D'autres ont un rôle de visualisation afin de permettre une analyse plus abstraite des entités ciblées ou alors afin de représenter les résultats d'une analyse de manière plus visuelle (*Hierarchical Map, Distribution Map, Butterfly Map*, etc.). Finalement la possibilité d'annoter des entités, afin d'atteindre un niveau d'abstraction plus élevé encore, fonctionnalité souvent utilisée au sein d'une visualisation également (*Tag Browser*).

b) Processus d'utilisation d'un browser

Ces navigateurs sont tous inter-connectés à l'aide d'un système de bus, qui leur permet à la fois d'écrire et de lire des informations, octroyant ainsi à l'utilisateur la possibilité de coupler les résultats et fonctionnalités de ces navigateurs entre eux, et de faire ainsi fonctionner ensemble les mécanismes de requête, visualisation et annotation.
Chacun de ces navigateurs doit hériter de la même classe, *MiAbstractBrowser.* Cela permet à ces outils d'être enregistrés auprès d'une classe et de son unique instance, *MiApplication*.

C'est par ce biais que toutes les composantes de MooseIDE peuvent interagir ensemble, en utilisant les bus qui sont gérés par l'application.

Pour les utiliser, il est nécessaire d'implémenter plusieurs méthodes de *MiAbstractBrowser* :

- « *canFollowEntity*: *anEntity*», pour vérifier si les entités circulant sur le bus sont utilisables par l'outil
- « *followEntity*: *anEntity*», afin de récupérer les entités et les stocker si celles-ci sont utilisables
- « *miSelectedItem* », pour renvoyer les entités que l'outil devrait propager sur le bus

L'implémentation de ces méthodes permet de faire fonctionner une barre d'outils intégrée à l'interface, permettant à l'utilisateur de changer de bus, de choisir de récupérer ou non les données qui y circulent et de propager les données produites par l'outil sur le bus. On peut aussi mettre des entités en surbrillance et ajouter des options au navigateur dans la barre d'outils, deux possibilités facultatives néanmoins.

Ces browsers ont aussi une classe « *model* » dans laquelle les données sont manipulées et stockées, assurant ainsi une division logique suivant le *design pattern MVC* (*Model View Controller*) entre les classes dédiées à l'interface et aux fonctionnalités, et celle contenant les données présentées par le navigateur. Toute cette structuration est représentée par le diagramme UML sur la figure 1.

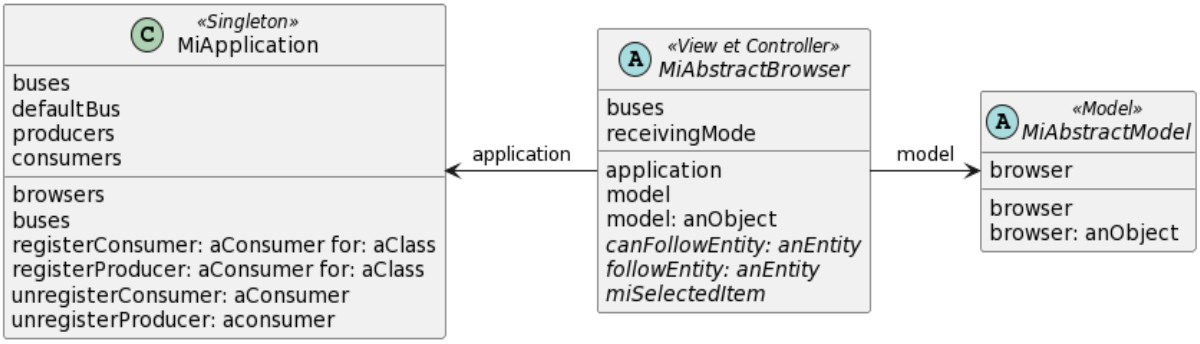

Figure 1 : Diagramme UML simplifié représentant la structure abstraite sur laquelle se base les outils de MooseIDE.

2) Méta-modèles Famix

Cependant, une analyse ne nécessite pas uniquement des outils adéquats afin d'être effectuée, elle demande avant tout de s'appliquer sur des données analysables, représentant fidèlement leur origine, le code source d'un logiciel.

Comme évoqué précédemment, Moose s'appuie sur une représentation abstraite des logiciels pour faire les analyses. Ces modèles sont conformes au méta-modèle Famix.

a) Définition

Un méta-modèle décrit les éléments d'un modèle et les relations qui peuvent exister entre eux de façon similaire à la grammaire d'un langage qui décrit les éléments du langage ou la légende d'une carte. Famix est un méta-modèle générique de langage qui peut être spécifié pour chaque langage de programmation.

b) Traits

Le mécanisme des *traits* permet à une classe de bénéficier d'héritage multiple. Concrètement, les traits contiennent un ensemble de méthodes, voir des attributs dans la version *stateful*, et ainsi permettent aux classes qui les utilisent de bénéficier de ces méthodes et / ou attributs comme si ils faisaient partie d'une super-classe.
Un exemple sur les traits utilisés par Famix est le trait *FamixTNamedEntity*, qui octroie aux classes qui l'utilisent un attribut *name*, et un ensemble de méthodes permettant des manipulations communes sur le nom. Ce trait est utilisé par des classes représentant un nombre important d'entités, comme les packages, les classes, les méthodes, les paramètres, les attributs, etc. Toutes les entités de la syntaxe d'un langage qui sont nommées vont utiliser ce trait.

c) Spécification

Un méta-modèle est donc ainsi composé de classes bénéficiant des traits écrits pour Famix, représentant chacune une entité spécifique, un élément du langage de programmation auquel il se rapporte. Le méta-modèle définit également les relations entre ces entités, par exemple une classe peut contenir des méthodes, et qu'une méthode est contenue dans une classe. Mais ces relations peuvent aussi être réifiées (héritage, accès, invocation et référence) et dans ce cas seront elles aussi gérées par des traits.
De plus, Famix supporte également l'ajout de propriétés annexes sans passer par le système de traits, si le besoin d'ajouter un comportement spécifique est présent mais que ce comportement ne se retrouve pas dans d'autres entités ou d'autres langages de programmation, et donc ne nécessite pas d'être spécifié seul.

Avec toute cette décomposition effectuée à l'aide des traits, nous sommes capables de composer un méta-modèle personnalisé et spécifique à un langage en utilisant les briques élémentaires fournies par Famix. Ainsi, chaque nouveau méta-modèle s'inscrit dans une cohérence globale avec les autres méta-modèles déjà existants, mais également avec MooseIDE défini précédemment.

3) Justification de l'utilisation de la plateforme Moose

On peut s'interroger sur la pertinence d'utiliser la plateforme Moose pour développer notre outils de vérification de règles architecturales. Cette question ne s'est évidemment pas posée en intégrant une équipe travaillant et développant cet outil, mais cela ne veut pas dire que Moose ne dispose pas des arguments en sa faveur pour défendre le fait que c'est bien la plateforme idéale pour cela.

Moose présente de ce fait deux avantages de taille. L'intégration de l'outil au sein de cette méta-plateforme nous permet de l'utiliser et d'utiliser les résultats de nos analyses dans d'autres outils de MooseIDE, pour pouvoir par exemple produire des visualisations graphiques de nos résultats.
Le méta-modèle Famix apporte quant à lui l'avantage de la généricité. En effet, il est très facile grâce à lui d'analyser différents logiciels (en y appliquant des règles différentes, ou non) mais aussi d'analyser des logiciels programmés dans des langages différents, si ceux-ci sont représentés par un méta-modèle personnalisé.

4) Plan

L'outil a donc bien des raisons d'être implémenté dans Moose. Nous verrons ainsi son architecture et ses fonctionnalités, avant de voir comment celles-ci sont utilisées dans son interface utilisateur, et finalement son application concrète sur le code de Berger-Levrault, et les résultats de cette application.

IV. Contribution : MooseCritics

1) Architecture de l'outil

Pour commencer, une présentation générale de l'outil s'impose, afin de représenter l'ampleur du travail accompli et la structure de l'outil, divisé en plusieurs classes. Pour ce faire, la figure 2 représente une version simplifiée (ne contenant pas les méthodes utilisées par l'interface de l'outil pour la présentation, expliquées plus tard dans le rapport) d'un diagramme UML du package principal du code implémentant l'outil. Cela n'inclut également pas les classes de test dans un package différent, ainsi que les classes représentant les règles et les violations, formées sur la base d'un méta-modèle et donc elles aussi présentées dans la suite du rapport.

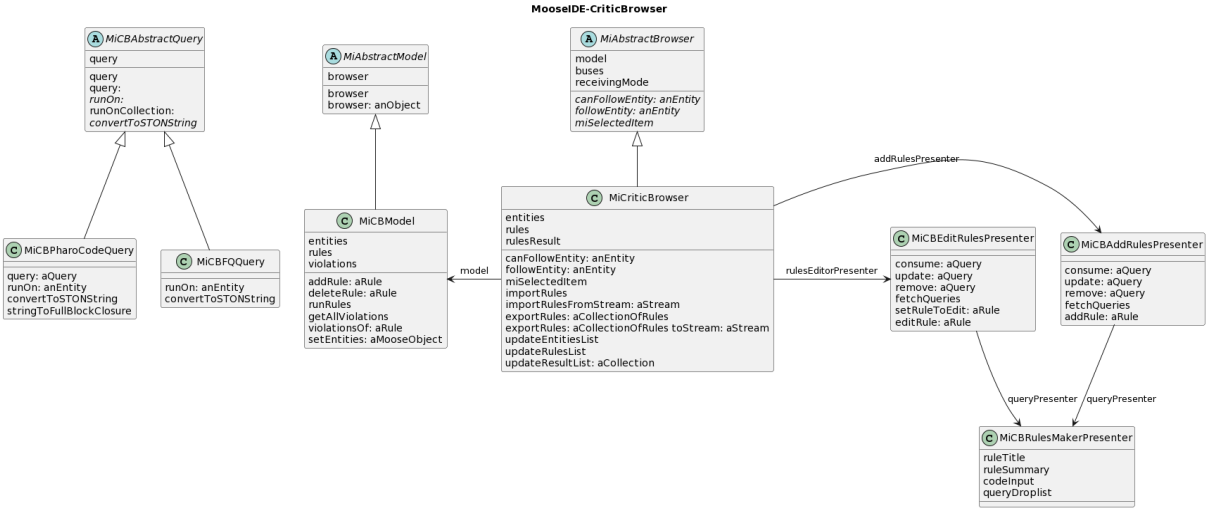

Figure 2 : Diagramme UML simplifié de la structure des classes de l'outil. Tout les classes sont préfixées par « Mi », convention pour les classes d'outils de MooseIDE.
(Dans le nom des classes, CB signifie Critic Browser et FQ signifie Famix Queries)

On distingue la classe principale, implémentant *MiAbstractBrowser* afin d'être reliée à MooseIDE, et son *model* représenté par la classe « *MiCBModel* », stockant les entités reçues depuis le bus, les règles et leurs violations une fois que les règles ont été vérifiées.

D'un point de vue interface graphique, malgré l'absence des attributs et méthodes correspondantes dans la classe principale de l'outil, on remarque les trois classes dédiées à l'écriture de règles, « *MiCBEditRulesPresenter* », « *MiCBAddRulesPresenter* » et « *MiCBRulesMakerPresenter* ». Ces deux premières classes représentent des fenêtres séparées de celle du navigateur, permettant l'ajout et l'édition de règles. La troisième

représente la majeure partie de l'interface au sein de ces deux fenêtres, afin de minimiser la duplication de code et ainsi faciliter la maintenabilité de l'outil.
Finalement, les classes « *MiCBAbstractQuery* », « *MiCBFQQuery* » et « *MiCBPharoCodeQuery* » représentent l'aspect principal d'une règle ; une requête booléenne permettant de détecter si une entité contient une violation de règle ou non. Celles-ci sont présentées plus tard dans le rapport, a côté du méta-modèle définissant les règles et les violations.

2) Connexion à MooseIDE

Comme introduit dans le background et visible sur le diagramme, la connexion de l'outil à MooseIDE passe par l'implémentation de la classe abstraite *MiAbstractBrowser.*
Celle-ci apporte ainsi à l'outil la connexion à l'instance de l'application centrale de MooseIDE et tout ce que cela implique, mais demande d'abord la définition des trois méthodes ; « *canFollowEntity*: », « *followEntity*: » et « *miSelectedItem* ».
Grâce à l'aide apporté par un blog post dédié à la création d'un outil lié à MooseIDE, mais aussi grâce aux code source d'autres outils MooseIDE, ces méthodes ont été relativement simples à implémenter.

a) Lecture du bus et récupération des entités

Pour pouvoir récupérer dans l'outil des entités circulant sur le bus, propagées par un autre outil de MooseIDE, il faut s'intéresser aux deux méthodes « *canFollowEntity*: *anEntity* » et « *followEntity*: *anEntity* ».
Elles ont toutes les deux une implémentation très basique, « *canFollowEntity*: *anEntity* » acceptant toute classe héritant de la classe « *MooseObject* » (qui est la superclasse de toutes les données analysables dans Moose, comme un modèle ou chaque entité contenu par celui-ci). La méthode « *followEntity*: *anEntity* » quant à elle transmet cela au *model* de l'outil à l'aide de la méthode « *setEntities: aMooseObject* », qui se charge de ne garder que les entités analysables par l'outil. Cette sélection n'est pas effectuée dans la méthode « *canFollowEntity*: *anEntity* » qui doit renvoyer un booléen, et donc dans le cas où un modèle est propagé sur le bus (cas d'utilisation de l'outil le plus fréquent) ne peut donc que choisir de l'accepter entièrement ou le rejeter entièrement.
Cependant, un modèle est composé de très nombreuses entités, et cela représente dès lors un souci d'optimisation. De ce fait les seules entités stockées par l'outil sont les entités

utilisant le trait *FamixTNamedEntity* (soit 21 classes sur 46 pour le méta-modèle représentant des projets Java), pour ne pas avoir à traiter un nombre trop important d'entités mais également pour simplifier le traitement de l'affichage. Cela reste néanmoins suffisant pour une analyse complète, car la plupart des entités non stockées dans l'outil reste accessibles à travers les propriétés et relations des entités stockées. Par exemple, même si les relations d'héritage ne sont pas stockées dans l'outil, chaque classe connaît ses super-classes et ses sous-classes, et des règles se basant sur cette hiérarchie peuvent donc fonctionner.

b) Propagation sur le bus

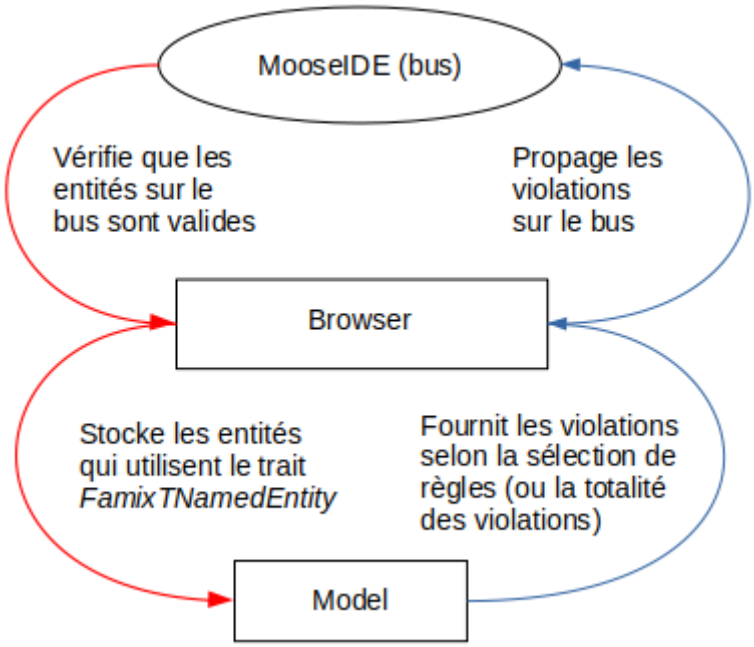

Figure 3 : Représentation des connexions entre le bus de MooseIDE, le navigateur de l'outil et son model

Rouge : lecture sur le bus / Bleu : écriture sur le bus

La méthode « *miSelectedItem* », renvoyant les entités à transmettre sur le bus, permet donc à l'outil de propager les éventuelles violations détectées sur le bus pour que d'autres outils de MooseIDE puissent les manipuler. L'outil offrant la possibilité de sélectionner une ou plusieurs règles en particulier afin d'afficher uniquement les violations de celles-ci dans l'interface graphique, il était intéressant de traiter les violations à propager de façon identique. Ainsi, si des règles ont été sélectionnées dans l'interface graphique,

« *miSelectedItem* » retourne les violations des règles sélectionnées, ou alors retourne l'ensemble des violations détectée si aucune règle n'a été sélectionnée.
Avec ces fonctionnalités, les connexions fondamentales avec MooseIDE sont complétées, comme représenté sur la figure 3.

3) Connexion avec le Queries Browser

Afin de donner à l'utilisateur la possibilité de construire ses règles de la façon la plus intuitive possible, notre outil doit être connecté à un autre outil de MooseIDE, le *Queries Browser.*
Cet outil permet la création de requêtes à travers une interface utilisateur, sans avoir besoin de coder celles-ci. On peut donc s'en servir pour formuler les requêtes cherchant les violations d'une règle. Le *Queries Browser* est affiché dans la figure 4.

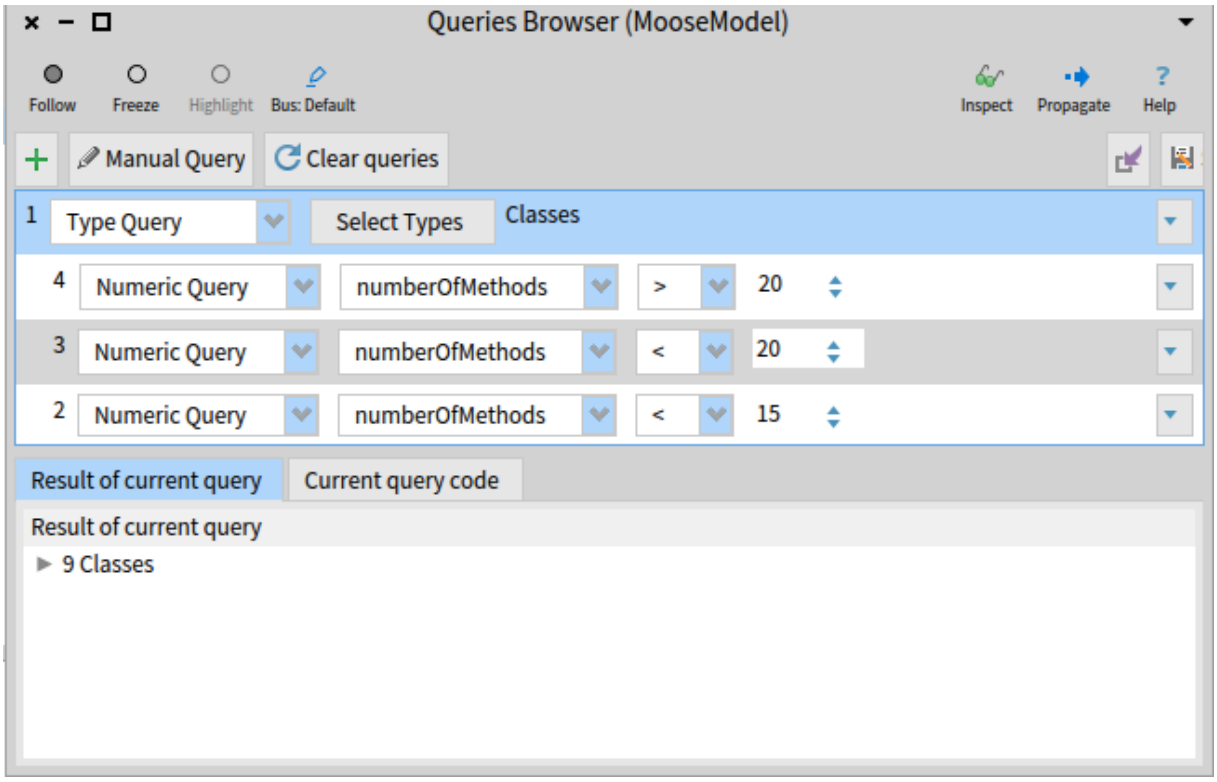

Figure 4 : Queries Browser contenant les requêtes utilisées pour les tags de l'analyse en première partie de rapport

Pour effectuer cette connexion, une nouvelle fonctionnalité de la classe *MiApplication* nous est utile ; le système de *consumer / producer*. Celui-ci permet de déclarer qu'une classe est productrice d'instances d'une certaine classe (dans notre exemple, la classe principale du *Queries Browser* est enregistrée auprès de l'application comme productrice de requêtes, de

la classe *FQAbstractQuery,* la classe abstraite et super-classe de toutes les classes représentant une requête dans le *Queries Browser*).

De plus, un producteur doit également définir la méthode « *itemsFor:* » qui doit renvoyer un ensemble contenant toutes les instances produites par la classe en question.

Le *Queries Browser* étant déjà enregistré à ce système en tant que producteur (pour être utilisé par la *Distribution Map* ou le *Tag Browser*, entre autres), il n'y a pas besoin de modifier celui-ci. Il suffit de définir notre outil comme un consommateur de *FQAbstractQuery* et de définir les méthodes nécessaires à la récupération de ses objets.

Ce mécanisme n'était malheureusement pas couvert par un blog post, j'ai donc uniquement pu me baser sur le code d'autres outils de MooseIDE définissant un système de consumer / producer, ainsi que l'aide apportée par l'un des ingénieurs de l'équipe, M. Soufyane Labsari.

Cela m'aura permis de faire de mon outil un consommateur de requêtes du *Queries Browser*, en ajoutant dans sa méthode d'initialisation une instruction similaire, mais appelant dans ce cas la méthode « *registerConsumer: for: ».*

Afin de compléter cette connexion, trois méthodes doivent être implémentées : « *consume:* », « *update:* » et « *remove:* ». Ces trois méthodes servent respectivement à obtenir une nouvelle instance, mettre à jour une instance reçue précédemment si elle subit des modifications du côté du producteur, et finalement supprimer une instance si celle-ci à été supprimée du côté du producteur.

Suivant l'exemple de la *Distribution Map* et de la *Hierarchical Map* (également enregistrées comme consommateurs auprès de *MiApplication*, ces trois méthodes ont la même implémentation, qui permet ainsi à chaque modification (nouvelle requête, requête mise à jour ou requête supprimée) de récupérer auprès de l'application toutes les instances mises à disposition par le ou les producteurs. Cela permet de simplifier la façon de traiter les suppression ou les mises à jour de requêtes (qui demanderait de modifier ou retirer une requête au sein d'une liste déroulante dans une interface graphique). L'ensemble des requêtes stockées est recréé à chaque modification pour ne pas avoir à traiter ces cas particuliers.

4) Représentation et traitement des règles

a) Méta-modèle des règles

Nos règles nécessitent elles-aussi une représentation particulière. Initialement, elles sont une requête booléenne, qui considère qu'une violation est présente au sein d'une entité lorsque la requête renvoie *true* avec cette entité en paramètre. Dans le cas ou la requête génère une erreur « *MessageNotUnderstood* » (par exemple si l'on interrogeait un attribut présent dans notre modèle pour connaître le nombre de méthodes qu'il contient, incohérent mais une possibilité fréquente, étant donné que l'on travaille souvent sur le modèle complet), l'erreur est ignorée et la requête renvoie *false*, afin d'ignorer l'entité.

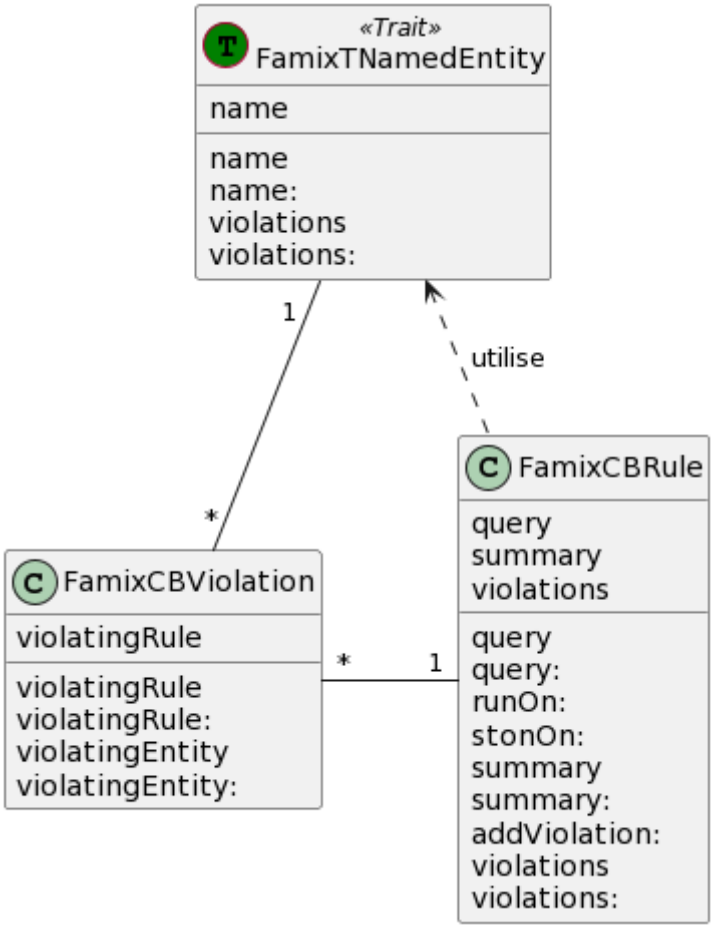

Figure 5 : Diagramme UML simplifié du méta-modèle de règles

Ces règles ont ensuite eu un titre et un résumé, puis les violations sont devenues des objets à part entière, contenant une entité et la règle violée par celle-ci.

On a ainsi des relations, des propriétés, formant des objets qui permettent de représenter un aspect d'un logiciel, les bases parfaites pour en créer un méta-modèle, tel que représenté sur la figure 5. Cela permet également de développer encore cette cohérence entre l'outil et le reste de Moose.

b) Classe génératrice du méta-modèle

Pour pouvoir instancier ce méta-modèle, en m'inspirant d'autres classes définissant des méta-modèles dans Moose et avec l'aide de M. Nicolas Anquetil en m'expliquant la formation de ces classes, j'ai défini une classe « générateur », héritant de la classe « *FamixMetamodelGenerator* » permettant de définir un méta-modèle simplement à l'aide de quelques méthodes.

Les quatre méthodes principales étant ; « *defineClasses* », « *defineHierarchy* », « *defineProperties* » et « *defineRelations* », définissant les classes du méta-modèle, leur hiérarchie (les traits qu'elles utilisent et les classes dont elles héritent), leurs propriétés non définies par des traits (ou les propriétés des traits d'un méta-modèle, si celui-ci en définit), et finalement les relations entre chacune de ces classes. Des méthodes statiques sont également utilisées, pour le nom du package à construire, le préfixe nommant les classes et les méta-modèles externes auquel il est fait référence (en héritage ou en relation, par exemple).

À l'aide de ces méthodes il est donc très facile de représenter toutes les caractéristiques du méta-modèle citées précédemment ; une règle à un nom, elle hérite donc du trait *FamixTNamedEntity.* Elle possède un résumé pour expliquer la vérification effectuée par la règle, définie comme propriété en tant que chaîne de caractère.

Les relations sont le point le plus développé dans le méta-modèle. Une règle peut être violée par plusieurs entités et peut donc avoir plusieurs violations, tandis qu'une violation ne pointe que sur une seule règle. Une entité (définie par le trait *FamixTNamedEntity*, soit l'ensemble des entités que l'outil va stocker et analyser) peut violer plusieurs règles et peut donc faire l'objet de plusieurs violations, tandis qu'une violation ne peut concerner qu'une entité à la fois. Cela impliquait initialement qu'une règle pouvait en violer d'autres (de par son utilisation de *FamixTNamedEntity*) mais ce problème à été résolu grâce aux noms utilisés pour les relations, qui ont permis de redéfinir les méthodes (« *violations* » et « *violations*: » générant ce comportement.

Une fois ces méthodes définies, un simple appel à la méthode « *generate* » permet de créer le package et d'y ajouter les classes définies au sein des méthodes du générateur.
Un autre comportement à été redéfini après génération, celui des requêtes. Élément central des règles et nécessitant une méthode supplémentaire ; « *runOn*: », pour exécuter celles-ci sur les entités fournies en paramètre. La méthode « *stonOn*: » également définie après la génération, est utilisée pour l'export de règles et est donc expliquée dans la partie correspondante.

c) Requêtes

Les classes représentant les premières versions des règles dans l'outil, brièvement présentées dans le début de cette partie et dans la figure 1, sont toujours utilisées au sein des règles. Elles représentent désormais la partie « programmée » des règles, devenant un attribut des règles tout comme le titre, le résumé de celles-ci et ses violations.
Représentées par trois classes, une abstraite ; « *MiCBAbstractQuery* » ainsi que deux implémentations de celles-ci ; « *MiCBPharoCodeQuery* » et « *MiCBFQQuery* », qui permettent chacune de représenter ces requêtes booléennes en code Pharo, ou bien en utilisant les requêtes du *Queries Browser* obtenues grâce à la connexion à celui-ci. Ce design utilisant une classe abstraite permet également d'ajouter si besoin est, de nouvelles façons de représenter ses requêtes.
Pour cela, deux méthodes sont à redéfinir. La première, « *runOn*: », prend une entité comme paramètre, et renvoie le résultat de la requête appliquée à cette entité. La seconde quant à elle, « *convertToSTONString* », convertit cette requête en une chaîne au format STON, afin de pouvoir l'exporter avec le reste de la règle dans un fichier.

d) Import et export de règles

La dernière fonctionnalité importante concernant les règles présentée dans ce rapport était une demande directe du chercheur employé par Berger-Levrault, M. Benoît Verhaeghe, avec qui j'étais en contact au cours de ce stage : la possibilité d'importer et d'exporter des règles, avec comme objectif de s'en servir dans un contexte d'intégration continue. En d'autres termes, les règles devaient être enregistrées dans un fichier, fichier qui doit par la suite être lisible par l'outil afin de récupérer les règles enregistrées précédemment.

Le format utilisé pour ce fichier à été le format STON, « *Smalltalk Object Notation* », format similaire à JSON. Celui-ci permet de représenter facilement une instance, par le nom de sa classe et la valeur des attributs qui le composent.

Parmi les classes manipulant le format STON, les deux classes « *SCExporter* » et « *STONReader* » auront été particulièrement utiles. Cependant, ces classes seules ne suffisent pas à interpréter chaque objet, selon la complexité des attributs à représenter. La requête étant dans notre cas, l'aspect demandant le plus de travail.

Ainsi vient pour l'export la définition des méthodes « *stonOn*: » et la méthode statique « *stonAllInstVarNames* » pour l'objet représentant les règles. La première méthode sert à convertir une instance en chaîne STON et est appelée par la classe d'export, et la deuxième sert à déterminer les attributs exportés (et ainsi permet de ne pas exporter les violations d'une règle par exemple, ce qui prendrait trop de place et serait complètement superflu). Cette première méthode convertit simplement le nom et le résumé de la règle, les chaînes de caractères étant converties par défaut par la classe d'export, tandis que la requête est convertie grâce à la méthode « *convertToSTONString* », définie au sein de la classe abstraite représentant les requêtes, afin de pouvoir adopter un comportement différent selon le type de requête.

Quant à l'import, le parser de la classe *STONReader* est capable par lui-même de recréer une classe à l'aide de son nom, écrit lors de l'export, ainsi que ses attributs lorsqu'ils ont une valeur « simple », typiquement le titre et le résumé de la règle. La requête, récupérée sous forme de chaîne de caractère, est interprétée comme une ligne de commande par le compilateur de Pharo, afin d'en déterminer la classe et de l'encapsuler dans la requête de classe correspondante.

La classe principale de l'outil, le navigateur, a elle aussi plusieurs méthodes servant à l'import et à l'export. Elles sont au nombre de quatre, deux pour l'import et deux pour l'export, afin de diviser la partie de la fonctionnalité écrivant ou lisant sur le flux et celle ouvrant une fenêtre déjà définie dans Pharo, permettant de choisir le nom du fichier à exporter ou bien le fichier à importer. En plus de diviser le code et d'apporter de la clarté, cela reste nécessaire dans le contexte de l'intégration continue (on doit pouvoir faire ses manipulations sans avoir à passer par l'interface).

5) Navigateur

L'aspect interface de l'outil, codé à l'aide de Spec, framework utilisé pour l'ensemble des interfaces graphiques de Pharo et de Moose représente également une part importante de ce stage. Cette partie sera cependant moins détaillée sur l'aspect implémentation, faute de place et pour privilégier les fonctionnalités permettant de résoudre la problématique initiale, présentées dans la partie précédente. Ce travail aura été fait en se basant principalement sur « *Spec UI Framework Demo* », une démonstration interactive de Spec et des possibilités qu'il offre au sein de Pharo, montrant l'affichage de chaque aspect développé ainsi que son code source. Le code d'autres navigateurs aura aussi été utile, ainsi que des conseils offerts par de nombreux membres de l'équipe, travaillant sur Moose ou non.

a) Fenêtre principale

L'outil dispose premièrement d'un navigateur comme interface utilisateur, contenant toutes les fonctionnalités présentées jusqu'ici. Visualiser les dernières entités récupérées sur le bus et stockées dans le *model* de l'outil, créer des règles mais aussi les supprimer ou les éditer, les importer ou alors exporter celles actuellement chargées dans l'outil.

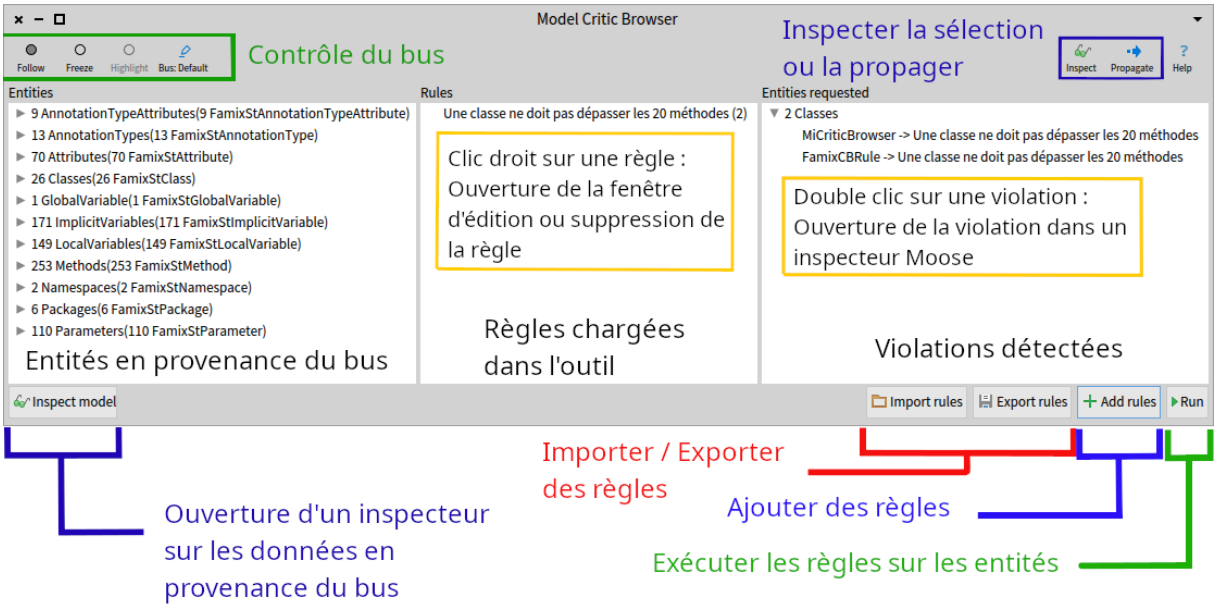

Figure 6 : Fenêtre du navigateur annotée, après avoir récupéré des entités sur le bus, écrit une règle et l'avoir exécutée sur les entités

Un bouton permet l'exécution des règles sur les entités stockées, affichant dans une autre liste les violations des règles chargées dans l'outil si elles existent. Après cette étape, sélectionner une ou plusieurs règles affiche uniquement les violations de cette sélection. Les violations peuvent également être ouvertes dans un inspecteur (afin d'examiner l'entité contenant la violation par exemple) en cliquant deux fois sur celle-ci.
Finalement, l'entièreté des violations ou bien celles des règles sélectionnées peuvent être propagées sur le bus, pour les utiliser dans un autre outil de MooseIDE.
Tout cela est visible sur la figure 6, montrant le navigateur de l'outil contenant l'ensemble des packages de MooseCritics, transférés grâce au bus. Sur ces données vient s'appliquer une règle, spécifiant qu'une classe ne peut avoir plus de 20 méthodes.

b) Rules Maker, ajout et édition de règles

L'interface contient également des fenêtres supplémentaires, toutes deux très similaires car contenant la même classe représentant le contenu de ces fenêtres. Ces deux fenêtres sont celles utilisées pour la création de règles ou pour l'édition de celles-ci.
Cette partie aura connue de nombreuses évolutions également au fil du développement. La première version permettant d'ajouter une règle n'était pas une fenêtre séparée, mais remplaçait temporairement la partie de la fenêtre principale représentant les violations pour permettre à l'utilisateur d'écrire ses règles. Elle est ensuite devenue une fenêtre séparée de l'outil, et à ce titre est devenue une classe à part.

Lors de l'ajout de la fonctionnalité d'édition de règles, afin de ne pas avoir à dupliquer du code, ou modifier la classe d'ajout de règles au risque de lui assigner trop de fonctionnalités et d'en faire une God Class, la division en trois classes utilisée désormais par l'outil a eu lieu.

Celle-ci est permise par l'architecture des interfaces sur Spec, divisant le contenu des fenêtres sous forme de classes « *Presenter* », tandis que la répartition de ce contenu est laissé à la charge des fenêtres, grâce à une méthode « *defaultLayout* », définissant la mise en page, grâce aux méthodes fournies par Spec. La fenêtre jusqu'ici utilisée pour l'ajout de règles est ainsi devenue un « *Presenter* », tandis que deux classes ont été créées, chacune gérant respectivement la responsabilité de l'ajout des règles, ou bien leur édition.

V. MooseCritics en application

Comme présenté dans l'introduction, ce stage est né du besoin de l'industriel Berger-Levrault de pouvoir vérifier le respect des règles architecturales de certaines de leurs applications, mais surtout de pouvoir trouver les violations de celles-ci déjà présentes dans leur code.
Après une démonstration de l'avancée du développement de l'outil, un peu moins d'un mois après mon arrivée à RMoD, j'ai reçu de la part de M. Benoît Verhaeghe, chercheur chez Berger-Levrault ayant précédemment effectué sa thèse chez RMoD, un modèle d'application de Berger-Levrault, ainsi qu'une description des règles architecturales à vérifier.

1) Structure générale des règles

Ces règles sont définies en deux parties : d'abord le contexte, définissant les entités sur lesquelles les règles s'appliquent (au nombre de trois, les « *Services* », les « *Endpoints* » et les « *DTO* », pour *Data Transfer Object*), et ensuite les différentes conditions à appliquer.
En terme de contexte, nous avons :
- Services : Toute interface qui étend l'interface « *java.rmi.Remote* »
- Endpoints : Toutes les méthodes des Services
- DTOs : Tout les types de paramètres et types de retour des Endpoints, ainsi que les attributs des DTOs (récursivement). Cela exclut néanmoins les Endpoints et Services, les types primitifs, les types paramétrés, et certaines autres classes particulières

Les conditions étaient elles diverses et variées, quelques exemples :
- Le nom d'un service doit avoir comme suffixe « Intf »
- Un service doit avoir au moins une implémentation, non abstraite, avec le même nom à l'exception du suffixe de la classe, qui doit devenir « Impl »
- Un DTO doit avoir un constructeur qui soit public, et sans paramètres

Ainsi, j'ai implémenté chaque condition comme une règle unique, en implémentant la recherche du contexte et la condition à vérifier dans une seule et même requête. Cela aura été conseillé par ma tutrice Mme. Anne Etien et M. Nicolas Anquetil, afin de les écrire le plus simplement possible par rapport à la façon dont les règles sont représentées dans l'outil.
Celles-ci auront été implémentées petit à petit le long du stage, en bénéficiant d'échanges et de retour de la part de M. Benoît Verhaeghe à plusieurs reprises pour vérifier leur validité ou pour m'aider dans la compréhension du contexte, et sont aujourd'hui toutes réalisées.

2) Règles particulières sur les DTO

Une exception concerne cependant les règles portant sur les DTO. Le problème étant que récupérer le contexte au sein d'une requête booléenne n'était pas, ou très difficilement réalisable, à cause de la dimension récursive de ce contexte.

J'ai alors mis en place une solution annexe, suggérée par M. Nicolas Anquetil, qui consiste à faire une sous-classe de la classe représentant les règles, sous-classe qui ne s'exécute pas directement sur les entités stockées par l'outil, mais va tout d'abord créer un sous-ensemble contenant les Services, s'en servir pour obtenir les Endpoints, et finalement se servir de ce dernier ensemble pour obtenir les DTO. Les règles sont évaluées sur ce dernier ensemble. Pour illustrer ce procédé, la figure 7 présente un diagramme d'activité de ces règles.

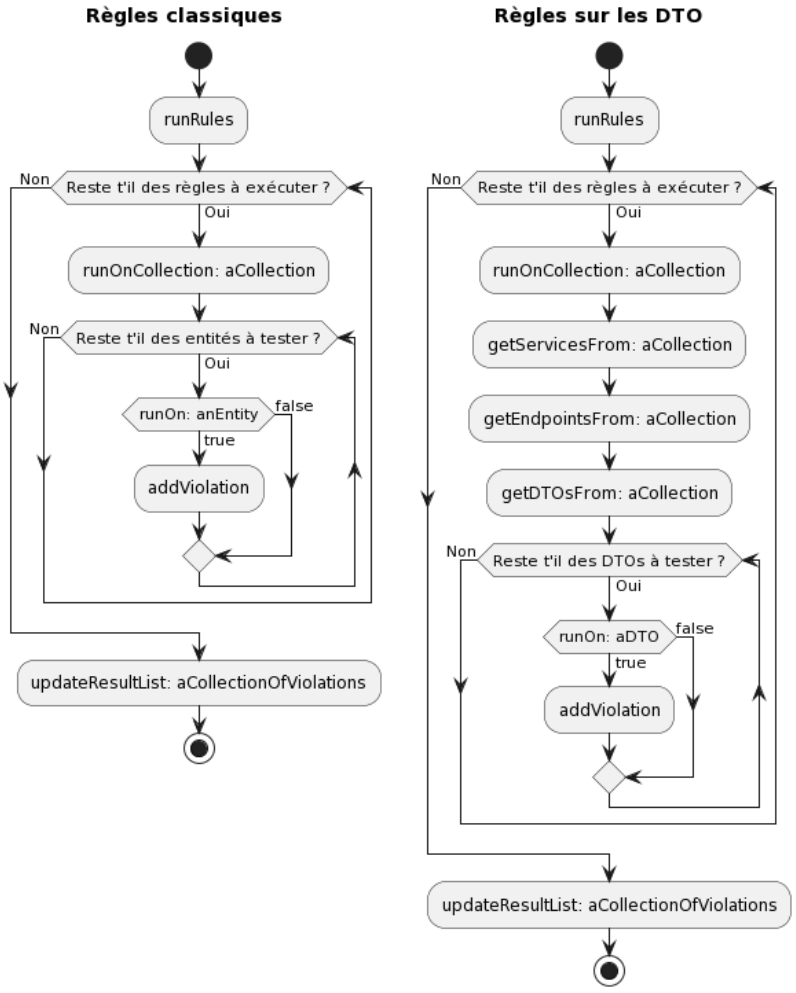

Figure 7 : Diagramme d'activité des règles selon leur contexte

VI. Conclusion

1) Contribution et résultats

Après maintenant un peu plus de la moitié de la durée de ce stage derrière moi, j'ai pu répondre à une majorité des besoins initiaux pour répondre à la problématique posée.

Cela est d'ailleurs représenté par MooseCritics intégré à Moose depuis le 1er juin, désormais chargé par défaut lorsque l'on télécharge une nouvelle image Moose 10. Celui-ci est bien connecté à MooseIDE mais aussi au Queries Browser, et un utilisateur est capable de créer des règles de façon intuitive que l'outil pourra exécuter sur le modèle d'un logiciel.

Le restant de ce stage sera l'occasion de rendre ses résultats utilisables dans d'autres outils Moose. Des modifications seront donc nécessaires dans l'outil, et très probablement aussi sur des outils de MooseIDE, comme le *Tag Browser* pour l'aspect visualisation, ou alors l'*Entities Exporter* pour obtenir un fichier CSV contenant des entités (ici, nos violations).

Un bilan peut cependant être tiré, grâce à l'expérimentation sur les données de Berger-Levrault. Sur le modèle qui m'a été fourni, représentant une application Java dont le code source est composé de 2 071 275 lignes de code, contenu au sein de 19 846 classes qui définissent ensemble 136 195 méthodes, j'ai pu trouver 632 violations au total. En considérant que plusieurs applications doivent subir une migration (et que des violations se cachent parmi chacune), un gain de temps considérable pourrait être obtenu grâce à cet outil, estimé par M. Benoît Verhaeghe à plus d'un mois par application, si celles-ci devaient être corrigées manuellement, en effectuant une migration partielle et l'utiliser pour obtenir un message d'erreur, effectuer la correction et relancer la compilation, durant une demie-heure.

2) Enseignements et apports

Ce stage m'aura permis de confirmer et approfondir de nombreuses notions abordées lors de mon cursus universitaire et aussi de les mettre dans une perspective plus professionnelle. L'importance du développement dirigé par les tests et ce qu'il apporte de confiance et de liberté à un développeur n'aura jamais été aussi flagrant à mes yeux et les principes et concept de la conception orientée objet ainsi que les problématiques du génie logiciel auront dirigé mon travail et permis de le réaliser dans les meilleures conditions, et sont maintenant bien plus claires et précisées après cette application professionnelle de ceux-ci.

Finalement, les échanges avec l'équipe mais aussi avec différents industriels intéressés par les perspectives de l'outil m'auront enseigné la communication de mon travail sous divers aspects que je n'avais encore jamais eu la possibilité d'expérimenter.

Bilan

Ce stage, en accord avec ma formation en amont mais aussi avec le master Génie Logiciel que j'intégrerais à la rentrée prochaine, aura été une formidable première expérience à mes yeux qui m'aura beaucoup apporté.

La possibilité d'ancrer dans le concret les enseignements de cette licence, et de découvrir l'univers de la recherche dans une équipe multi-culturelle regorgeant d'individus talentueux et passionnés composant un cadre idéal et épanouissant pour cette première expérience professionnelle. Je tire également la satisfaction d'avoir atteint les objectifs fixés au début de ce stage, grâce au travail fortement inspiré par ce cadre et l'assistance précieuse que les différents membres de l'équipe ont pu m'apporter tout au long de ce procédé, et ai hâte de continuer le développement de cet outil pour la durée restante de ce stage.

Cette expérience de la recherche et de l'analyse logicielle, deux perspectives déjà attirantes pour la suite de mon projet professionnel, auront également contribué à confirmer ce sentiment et m'apporte davantage encore de motivation pour la suite de mon cursus à l'Université de Lille.

Bibliographie

Modular Moose: A new generation software reverse engineering environment (ICSR'20), *Nicolas Anquetil, Anne Etien, Mahugnon H. Houekpetodji, Benoit Verhaeghe, Stéphane Ducasse, Clotilde Toullec, Fatiha Djareddir, Jerôme Sudich, and Moustapha Derra*

Sitographie :

- Blog post sur la construction de navigateurs Moose
https://modularmoose.org/2021/05/04/how-to-build-a-new-moose-tool.html

Annexe

Représentation de deux règles de Berger-Levrault dans MooseCritics :

- Règle sur les services :

Rule name :

Services implementations must have the same name, except an 'Impl' suffix

Input for code query :

```
[ :entity |
    (entity isInterface and: [
        entity superclassHierarchy anySatisfy: [ :class |
            class name = 'Remote' ] ]) and: [
        entity directSubclasses anySatisfy: [ :class |
            (class isInterface not and: [
                ((class name allButLast: 4) = (entity name allButLast: 4)) not ])
                and: [ (class name endsWith: 'Impl') not ] ] ] ]
```

Summary of the rule :

A service has to be implemented, and that implementation must share the same name, excepted the suffix "Intf" becoming "Impl".

En bleu : Contexte / En rouge : Condition qui génère une violation

- Règle sur les DTOs :

Rule name :

A DTO must have an empty public constructor

Input for code query :

```
[ :entity |
    entity methods noneSatisfy: [ :m |
        m name = entity name and: [ m parameters isEmpty ] ] ]
```

Summary of the rule :

A DTO must have at least one constructor that is :

- Public
- Without any parameter

Contexte obtenu dans la classe ; ne s'applique qu'aux DTOs