Améliorer la compréhension d'un programme à l'aide de diagrammes dynamiques et interactifs

Mickaël Duruisseau

Université de Lille

Université de Lille

École doctorale **Sciences Pour l'Ingénieur**

Laboratoire **CRIStAL UMR 9189**

Thèse présentée par **Mickaël Duruisseau**

Soutenue le **16 juillet 2019**

En vue de l'obtention du grade de docteur de l'Université de Lille

Discipline **Informatique**

Améliorer la compréhension d'un programme à l'aide de diagrammes dynamiques et interactifs

Thèse dirigée par	Xavier Le Pallec	directeur
	Sébastien Gérard	co-directeur
	Jean-Claude Tarby	encadrant

Composition du jury

Rapporteurs	Marianne Huchard	professeure à l'Université de Montpellier	
	Philippe Palanque	professeur à l'Université de Toulouse 3	
Examinateurs	Bruno Dumas	professeur à l'Université de Namur	président du jury
	Jean-Claude Tarby	MCF à l'Université de Lille	
Directeurs de thèse	Xavier Le Pallec	MCF HDR à l'Université de Lille	
	Sébastien Gérard	directeur de recherche au CEA	

Université de Lille

Université de Lille

École doctorale **Sciences Pour l'Ingénieur**

Laboratoire **CRIStAL UMR 9189**

Thèse présentée par **Mickaël Duruisseau**

Soutenue le **16 juillet 2019**

En vue de l'obtention du grade de docteur de l'Université de Lille

Discipline **Informatique**

Améliorer la compréhension d'un programme à l'aide de diagrammes dynamiques et interactifs

Thèse dirigée par	Xavier Le Pallec	directeur
	Sébastien Gérard	co-directeur
	Jean-Claude Tarby	encadrant

Composition du jury

Rapporteurs	Marianne Huchard	professeure à l'Université de Montpellier	
	Philippe Palanque	professeur à l'Université de Toulouse 3	
Examinateurs	Bruno Dumas	professeur à l'Université de Namur	président du jury
	Jean-Claude Tarby	MCF à l'Université de Lille	
Directeurs de thèse	Xavier Le Pallec	MCF HDR à l'Université de Lille	
	Sébastien Gérard	directeur de recherche au CEA	

Université de Lille

Université de Lille

Doctoral School **Sciences Pour l'Ingénieur**

Laboratory **CRIStAL UMR 9189**

Thesis defended by **Mickaël Duruisseau**

Defended on **16th July, 2019**

In order to become Doctor from Université de Lille

Academic Field **Computer Science**

Enhance software comprehension with dynamic and interactive diagrams

Thesis supervised by	Xavier Le Pallec	Supervisor
	Sébastien Gérard	Co-Supervisor
	Jean-Claude Tarby	Monitor

Committee members

Referees	Marianne Huchard	Professor at Université de Montpellier	
	Philippe Palanque	Professor at Université de Toulouse 3	
Examiners	Bruno Dumas	Professor at Université de Namur	Committee President
	Jean-Claude Tarby	Associate Professor at Université de Lille	
Supervisors	Xavier Le Pallec	HDR Associate Professor at Université de Lille	
	Sébastien Gérard	Research Director at CEA	

Mots clés : ingénierie dirigée par les modèles (idm), interactions homme-machine (ihm), unified modeling language (uml), visualisation de programme, compréhension de programme, recherche d'informations

Keywords: model driven engineering (mde), human–computer interaction (hci), unified modeling language (uml), software visualization, program comprehension, information retrieval

Cette thèse a été préparée dans les laboratoires suivants.

CRIStAL UMR 9189
Université Lille
Bâtiment M3 extension
Avenue Carl Gauss
59655 Villeneuve d'Ascq Cedex
FRANCE

☎ 03 28 77 85 41
✉ olivier.colot@univ-lille.fr
Site https://www.cristal.univ-lille.fr/

CEA LIST
Centre d'intégration Nano-INNOV
8 avenue de la Vauve
91120 PALAISEAU
FRANCE

☎ 01 69 08 05 14
✉ info-list@cea.fr
Site http://www-list.cea.fr/

Améliorer la compréhension d'un programme à l'aide de diagrammes dynamiques et interactifs

Résumé

Les développeurs occupent une place prépondérante dans le développement logiciel. Dans ce cadre, ils doivent réaliser une succession de tâches élémentaires (analyse, codage, liaison avec le code existant...), mais pour effectuer ces tâches, un développeur doit régulièrement changer de contexte de travail (recherche d'information, lecture de code...) et analyser du code qui n'est pas le sien. Ces actions nécessitent un temps d'adaptation élevé et réduisent l'efficacité du développeur. La modélisation logicielle est une solution à ce type de problème. Elle propose une vue abstraite d'un logiciel, des liens entre ses entités ainsi que des algorithmes utilisés. Cependant, l'Ingénierie Dirigée par les Modèles (IDM) est encore trop peu utilisée en entreprise. Dans cette thèse, nous proposons un outil pour améliorer la compréhension d'un programme à l'aide de diagrammes dynamiques et interactifs. Cet outil se nomme VisUML et est centré sur l'activité principale de codage du développeur. VisUML fournit des vues (sur des pages web ou sur des outils de modélisation) synchronisées avec le code. Les diagrammes UML générés sont interactifs et permettent une navigation rapide avec et dans le code. Cette navigation réduit les pertes de temps et de contextes dues aux changements d'activités en fournissant à tout moment une vue abstraite sous forme de diagramme des éléments actuellement ouverts dans l'outil de codage du développeur. Au final, VisUML a été évalué par vingt développeurs dans le cadre d'une expérimentation qualitative de l'outil afin d'estimer l'utilité d'un tel outil.

Mots clés : ingénierie dirigée par les modèles (idm), interactions homme-machine (ihm), unified modeling language (uml), visualisation de programme, compréhension de programme, recherche d'informations

CRIStAL UMR 9189
Université Lille – Bâtiment M3 extension – Avenue Carl Gauss – 59655 Villeneuve d'Ascq Cedex – FRANCE

Enhance software comprehension with dynamic and interactive diagrams

Abstract

Developers dominate in software development. In this context, they must perform a succession of elementary tasks (analysis, coding, linking with existing code ...), but in order to perform these tasks, a developer must regularly change his context of work (search information, read code ...) and analyze code that is not his. These actions require a high adaptation time and reduce the efficiency of the developer. Software modeling is a solution to this type of problem. It offers an abstract view of a software, links between its entities as well as algorithms used. However, Model-Driven Engineering (MDE) is still underutilized in business. In this thesis, we propose a tool to improve the understanding of a program using dynamic and interactive diagrams. This tool is called VisUML and focuses on the main coding activity of the developer. VisUML provides views (on web pages or modeling tools) synchronized with the code.The generated UML diagrams are interactive and allow fast navigation with and in the code. This navigation reduces the loss of time and context due to activity changes by providing at any time an abstract diagram view of the elements currently open in the developer's coding tool. In the end, VisUML was evaluated by twenty developers as part of a qualitative experimentation of the tool to estimate the usefulness of such a tool.

Keywords: model driven engineering (mde), human–computer interaction (hci), unified modeling language (uml), software visualization, program comprehension, information retrieval

Remerciements

Je tiens à remercier Jean-Claude Tarby, Maître de Conférences à l'Université de Lille, qui m'a encadré tout au long de cette thèse et avec qui nous avons partagé de nombreuses idées et pistes à explorer. Je le remercie également pour sa disponibilité, ses encouragements et toute l'aide qu'il m'a prodiguée.

Je remercie Xavier Le Pallec, Maître de Conférences HDR à l'Université de Lille pour son soutien, ses retours et son humour pendant ces quatre années de thèse.

Je remercie également le CEA et notamment à Sébastien Gérard, directeur du LISE, qui m'a donné l'opportunité de réaliser cette thèse en co-direction.

J'adresse tous mes remerciements à Philippe Palanque, Professeur à l'Université Toulouse 3 Paul Sabatier, ainsi qu'à Marianne Huchard, Professeur à l'Université de Montpellier, de l'honneur qu'ils m'ont fait en acceptant d'être rapporteurs de cette thèse. Je remercie également Bruno Dumas, Professeur à l'Université de Namur, qui a accepté d'être examinateur.

Plus généralement, je remercie l'équipe Carbon pour son accueil au sein du laboratoire et pour la bonne ambiance qui y règne, et plus particulièrement je remercie Michel Dirix pour son soutien pendant toutes ces années et pour ses conseils avisés.

Enfin, je tiens à remercier ma famille qui m'a encouragé à faire cette thèse et qui m'a soutenu tout au long de ce travail, et également mes amis proches qui m'ont, à leurs manières, également soutenu pendant ces années.

Acronymes

A | B | E | F | H | I | J | M | S | U | W | X

A

API Application Programming Interface. 49, 66
AST Abstract Syntax Tree. 66

B

BPMN Business Process Model and Notation. 10

E

EDI Environnement de Développement Intégré. 8, 13, 16, 17, 23, 24, 27, 29, 34, 36–39, 42, 43, 50, 51, 61, 62, 64, 70, 71, 74, 81, 86, 88–95, 97–99, 101–104, 111, 126–135, 143–146, 149, 151–155, 190
ESN Entreprise de services du numérique. 19

F

FQN Fully Qualified Name. 44, 46, 47, 60, 93, 152

H

HTML HyperText Markup Language. 39
HTTP HyperText Transfer Protocol. 24

I

IA Intelligence Artificielle. 154, 155
IDM Ingénierie Dirigée par les Modèles. 10, 11
IHM Interaction Homme-Machine. 11

J

JS JavaScript. 39

JSON JavaScript Object Notation. 50, 51, 57, 190

M

MVC Modèle-Vue-Contrôleur. 1, 6, 155
MVP Modèle-Vue-Présentation. 1
MVVM Modèle-Vue-Vue Modèle. 1, 6

S

SysML System Modeling Language. 10

U

UML Unified Modeling Language. 2, 3, 10, 11, 17, 18, 21, 22, 24, 27–29, 31, 33, 34, 39–41, 44, 47, 61, 93, 106, 108, 115, 116, 129, 143, 144

W

WSE Web Server Event. 36, 37, 39, 42, 44, 91

X

XMI XML Metadata Interchange. 24, 25
XML Extensible Markup Language. 48

Sommaire

Liste des tableaux

Table des figures

Liste des codes

Introduction

L'informatique et le développement d'applications ne cessent d'évoluer. Des projets tels que Linux sont passés en une quarantaine d'années de moins d'un million de lignes de code à près de vingt millions, soit vingt fois plus. Ceci impacte bien évidemment le nombre de fichiers et le nombre de dossiers (*packages* ou modules) associés aux projets informatiques, mais cela est associé aussi à l'augmentation du nombre de développeurs concernés par les projets. De façon générale, le nombre moyen de développeurs par projet ne cesse d'augmenter. Cela est d'autant plus vrai dans les projets Open Source, pour lesquels le nombre de contributeurs peut atteindre plusieurs centaines de personnes. Par ailleurs, la façon d'organiser un projet a elle aussi changé. Les développeurs utilisent de plus en plus de patrons de conception et d'architecture (Modèle-Vue-Contrôleur (MVC), Modèle-Vue-Vue Modèle (MVVM), Modèle-Vue-Présentation (MVP)...), et de frameworks (Spring, Hibernate, Struts, Ruby on rails...). Ces nouvelles architectures de projets impliquent la création de nombreux dossiers et fichiers (spécifiques aux différents modèles), ce qui augmente la complexité d'un projet, et rend sa compréhension difficile, principalement lorsqu'un développeur fraîchement arrivé n'a pas connaissance du ou des patrons de conception utilisés.

Un plus grand nombre de développeurs sur un même projet permet une implémentation plus rapide des applications, mais accroît également la difficulté de compréhension globale d'un projet. Chaque développeur va implémenter sa partie sans forcément connaître les éléments créés par ses collègues, ce qui peut entraîner des erreurs ou nécessiter un temps de travail plus grand lors d'une maintenance ou d'une évolution. Cependant, ceci est modéré par l'évolution des méthodes de conception, par exemple avec les méthodes agiles qui permettent de mieux organiser les tâches de l'équipe, ainsi que la mise en commun des informations.

Les outils ont eux aussi beaucoup évolué ces dernières décennies et se concentrent de plus en plus sur l'humain en tentant d'aider le développeur dans ses tâches quotidiennes (communication, échange de documents et d'informations, complétion automatique, « snippets » de code...). Les interfaces graphiques de ces outils permettent d'afficher plus d'informations, et surtout

elles fournissent plusieurs représentations graphiques possibles pour une même information (par exemple : arborescence d'un projet ou structure d'une classe). Cet apport d'informations modifie la façon dont les développeurs travaillent. Par exemple des plugins de qualité de code indiquent au développeur les améliorations possibles ou les risques de sécurité, l'intégration continue met en évidence les tests ou *builds* non validés.

Enfin, le marché actuel en informatique étant très prolifique, un développeur ne va rester que quelques années (voire quelques mois) dans une entreprise avant de changer de travail. Ce « turn-over » important peut entraîner une perte d'informations sur un projet s'il n'y a pas de transfert de connaissances. Généralement, celui-ci est réalisé uniquement grâce au code (via du *pair-programming* ou équivalent) alors que l'usage de diagrammes permettrait d'augmenter le niveau d'abstraction et par conséquent faciliterait la lecture du code.

Certaines pratiques limitent les pertes d'informations, notamment grâce aux documentations (générées et écrites) de projets. Cependant ces dernières restent encore rares, incomplètes ou obsolètes (non mises à jour) car d'une part elles demandent beaucoup de temps pour être produites et mises à jour, et d'autre part elles ne font pas partie des priorités des entreprises de manière générale (non demandé par le client).

Le « turn-over » implique également que tout développeur va forcément avoir à modifier, et donc comprendre, un projet qui n'est pas le sien. C'est cette phase de compréhension que nous souhaitons améliorer principalement.

Même si les méthodes de conception ont permis un usage plus aisé de la modélisation logicielle, celle-ci reste encore très peu pratiquée dans le monde professionnel. Pourtant, des langages de modélisation tels que Unified Modeling Language (UML) existent maintenant depuis plusieurs décennies et sont enseignés aux étudiants dans les Universités. On constate malheureusement cependant que son usage reste rare, partiel (seuls deux ou trois diagrammes sont généralement utilisés parmi les quatorze disponibles pour la version 2.3 [1] et souvent de façon détournée (diagrammes non formels, voire avec des modifications personnelles ou liées aux entreprises). Ceci peut paraître surprenant quand on connaît les outils informatiques de modélisation, comme Papyrus ou GenMyModel, qui proposent nombre de fonctionnalités pour dessiner les diagrammes UML voire pour vérifier la syntaxe des modèles ou bien encore générer du code.

Sur un tout autre aspect, la Recherche s'intéresse depuis longtemps à la psychologie des développeurs (voir par exemple le « Psychology of Programming Interest Group » créé en 1987 [2]. Une étude récente sur plus de 1800 dévelop-

1. 25 diagrammes en UML 1.3
2. PPIG : `http://www.ppig.org/`

peurs [19] montre que 42,75% des développeurs sont malheureux à cause des *artefacts* manipulés dans leurs tâches dont 52% à cause du code et des bugs. Quand on regarde les dix premières causes principales du mal-être des développeurs, la mauvaise qualité du code et les mauvaises pratiques de codage arrivent en 3ème position, et le code sans commentaire qui provoque des bugs arrive en 7ème position. La modélisation pourrait apporter une réponse à ces problèmes, par exemple en permettant des visualisations pertinentes du code source en fonction des tâches du développeur (recherche de bugs, vérification des bonnes pratiques, etc.). C'est dans cet esprit que nous avons travaillé dans cette thèse.

Problématique

Ce manuscrit tente de répondre à la problématique suivante : « Comment améliorer la compréhension d'un programme à l'aide de diagrammes ».

Il présente VisUML, un outil de visualisation dynamique et interactif, permettant aux développeurs d'obtenir un ou plusieurs diagrammes UML correspondants à leur activité en cours. Les diagrammes sont synchronisés avec le code et permettent une navigation rapide et efficace entre les différents éléments du projet. Les diagrammes actuellement gérés sont le diagramme de classe et le diagramme de séquence. Un prototype de diagramme d'activité a été développé, mais ne sera pas présenté dans ce document (cf. Annexe C page 185). VisUML est architecturé de façon à pouvoir ajouter facilement de nouvelles fonctionnalités, sans avoir à restructurer le code des visualisations (voir section 4.1.2 page 65 pour un exemple).

Le chapitre 1 présente le contexte de la thèse, les évolutions du métier de développeur ainsi que la façon dont un développeur travaille. Afin de situer VisUML dans l'existant, le chapitre 2 regroupe et compare les différents outils permettant une rétro-ingénierie d'un projet. Cette comparaison est orientée sur les fonctionnalités facilitant l'usage, ainsi que sur les interactions proposées. Le chapitre 3 présente VisUML, son architecture ainsi que les structures des entités utilisées, et sert d'introduction à l'outil. Les chapitres 4 et 5 présentent chacun un aspect de VisUML, respectivement les fonctionnalités et les choix centrés humains, et les interactions et navigations implémentées dans l'outil. Le chapitre 6 détaille l'expérimentation menée pour valider l'outil et présente les résultats obtenus. Enfin le dernier chapitre présente une conclusion ainsi qu'une perspective des travaux futurs.

Chapitre 1

Place de l'IDM dans la conception logicielle

Sommaire

Ce chapitre pose le contexte de cette thèse. Nous présentons l'évolution des métiers du développement informatique, dans l'organisation des projets, la façon dont les développeurs travaillent, ainsi que les méthodes et outils utilisés. Pour chacune de ces parties, nous mettrons en avant les avantages de ces évolutions, mais également les points en retrait. Enfin, la dernière section présente les mécanismes de psychologie du développeur utilisés dans cette thèse, et les choix qui ont été faits grâce à eux.

1.1 Évolution des projets et des métiers du développement informatique

L'informatique et le développement d'applications ont fortement évolué ces dernières années. Plusieurs études montrent que la taille des projets a augmenté de manière significative. Par exemple [40] montre que le code source du projet Unix est passé en quarante quatre ans (de 1972 à 2015) de moins d'un million de lignes de code à près de vingt millions, soit vingt fois plus. Cette augmentation se répercute bien évidemment à la fois en nombre de lignes de code, mais également dans le nombre de fichiers et le nombre de dossiers (packages ou modules) associés à un projet.

La façon d'organiser un projet a elle aussi changé. Les développeurs utilisent de plus en plus de patrons de conception et de frameworks. Par exemple, les applications modernes utilisent souvent le modèle MVC ou MVVM afin de respecter une architecture particulière [26]. Cette architecture implique la création de nombreux dossiers et fichiers (spécifiques aux différents modèles), ce qui augmente la complexité d'un projet, et rend sa compréhension difficile, principalement lorsqu'un développeur fraîchement arrivé n'a pas connaissance du ou des patrons de conception utilisés.

Cette augmentation de la taille des projets entraîne également une hausse du nombre de personnes impliquées. De façon générale, le nombre moyen de développeurs par projet ne cesse d'augmenter. Cela est d'autant plus vrai dans les projets Open Source, pour lesquels le nombre de contributeurs peut atteindre plusieurs centaines de personnes (voir figure 1.1 [1]).

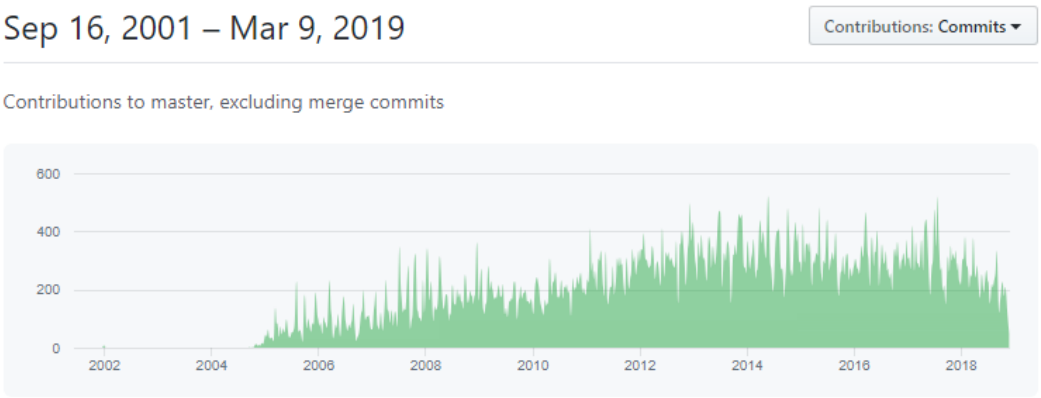


FIGURE 1.1 – Evolution des contributions au projet Linux open-source en 17 ans

1. Source : https://github.com/torvalds/linux/graphs/contributors

Un plus grand nombre de développeurs sur un même projet permet une implémentation plus rapide des applications, mais accroît également la difficulté de compréhension globale d'un projet. En effet, chaque développeur va implémenter sa partie sans forcément connaître les éléments créés par ses collègues, ce qui peut entraîner des erreurs ou nécessiter un temps de travail plus grand lors d'une maintenance ou d'une évolution [18, 34].

Cependant cette dernière affirmation peut être modulée par l'évolution des outils et des méthodes de conception. Aujourd'hui les équipes de développement adoptent facilement des méthodes agiles permettant de mieux organiser les tâches de l'équipe, ainsi que la mise en commun des informations.

De la même façon, les outils se concentrent de plus en plus sur l'humain, et visent à aider le développeur dans ses tâches quotidiennes, que ce soit dans le travail d'équipe (communication, échange de documents et d'informations avec des outils tels que Slack [2]) ou dans l'activité de codage (complétion automatique, « snippets » de code…). Les interfaces graphiques ayant beaucoup évolué, elles permettent aux outils de développement d'afficher plus d'informations, et surtout elles fournissent plusieurs représentations graphiques possibles pour une même information (par exemple : arborescence d'un projet ou structure d'une classe). Cet apport d'informations modifie la façon dont les développeurs travaillent. Par exemple des plugins de qualité de code indiquent au développeur les améliorations possibles ou les risques de sécurité, l'intégration continue met en évidence les tests ou *builds* non validés ; dans ces deux cas, le développeur prendra souvent immédiatement en considération les améliorations à apporter, alors qu'autrefois il aurait fallu attendre une revue de code ou la production manuelle d'une analyse du projet. Ces deux aspects (méthodes de conception et outils) sont détaillés dans la section 1.2 page suivante.

Enfin, le marché actuel en informatique étant très prolifique, un développeur ne va rester que quelques années dans une entreprise avant de changer de travail. Ce « turn-over » important peut entraîner une perte d'informations sur un projet s'il n'y a pas de transfert de connaissances. Dans le cas où un transfert est effectué, il est le plus souvent réalisé uniquement grâce au code (via du pair-programming ou équivalent) alors qu'il est plus simple d'utiliser des diagrammes pour augmenter le niveau d'abstraction et par conséquent de faciliter la lecture du code.

Certaines pratiques limitent les pertes d'informations, notamment grâce aux documentations (générées et écrites) de projets. Cependant ces dernières restent encore rares, incomplètes ou obsolètes (non mises à jour) car d'une part elles demandent beaucoup de temps pour être produites et mises à jour, et d'autre part elles ne font pas partie des priorités des entreprises de manière générale

2. Slack : https ://slack.com/intl/fr-fr/

(non demandé par le client).

Ce « turn-over » implique également que tout développeur va forcément avoir à modifier, et donc comprendre, un projet qui n'est pas le sien. C'est cette phase de compréhension que nous souhaitons améliorer principalement.

1.2 Méthodes et outils de conception

La façon dont les développeurs travaillent a évolué grâce aux méthodes de conception (méthodes Agile, Lean, Extreme programming...), ainsi qu'aux outils associés (forges logicielles, qualité de code...). Ces deux aspects améliorent aussi bien le développement que la gestion de projets.

Les méthodes de conception, telles que les méthodes Agiles [9], permettent une meilleure organisation de l'équipe de développement. Ces méthodes visent à augmenter la flexibilité de l'équipe, en privilégiant les échanges et les individus plutôt que de suivre un processus fixe. En revanche, une des contreparties des méthodes agiles est qu'elles ne privilégient pas la documentation, mais plutôt les « logiciels fonctionnels ». La documentation est donc souvent oubliée, très incomplète ou jamais mise à jour. Ces méthodes permettent cependant de meilleures relations avec le client, en livrant rapidement des versions fonctionnelles, en acceptant les modifications du cahier des charges et en collaborant de manière permanente avec celui-ci. Les méthodes agiles changent la façon dont les développeurs s'organisent mais aussi leur façon de coder, puisqu'elles ajoutent des outils de suivi de projets, des outils de suivi de tickets ainsi que des mécanismes d'intégration continue que nous détaillons ci-après.

Les Environnements de Développement Intégrés (EDI) ont bénéficié de nombreuses évolutions depuis leur création et peuvent afficher plusieurs fichiers simultanément, distinguer des éléments importants du code, colorier syntaxiquement n'importe quel langage de programmation, etc. Ils intègrent aussi de nombreuses fonctionnalités de recherche, de refactorisation et d'analyse du code, qui permettent aux développeurs de mieux architecturer leurs programmes ou de respecter certains standards de notation. Cependant, la façon dont ces informations sont affichées par défaut n'est pas toujours la plus efficace. Des études montrent que le texte n'est pas toujours adapté pour représenter les éléments connectés entre eux [28], ce qui peut limiter l'intérêt de ces fonctionnalités.

Certains outils fournissent également des visualisations du code source sous forme de diagrammes, ces derniers sont présentés dans le chapitre 2 page 16.

Enfin, de nombreux outils de suivi de projets ont vu le jour. Ces outils, appelés « forges logicielles », regroupent un ensemble de fonctionnalités de gestion de projet.

La première fonctionnalité est un gestionnaire de version, comme git [3] ou SVN [4], permettant aux développeurs de travailler en parallèle sans se gêner, ainsi qu'en conservant un historique de toutes les modifications effectuées.

La seconde est un outil de suivi des bugs, ou *ticketing*, permettant aux membres de l'équipe de développement de documenter les bugs, les tâches à effectuer, les améliorations possibles, tout en liant ces tickets aux différents « commits » [5] du gestionnaire de version. Ces outils possèdent généralement des systèmes de documentation (sous la forme de Wiki) et des forums de discussion, qui permettent par exemple de décrire les bonnes pratiques ou les trucs et astuces d'une entreprise ou d'une technologie.

Plus récemment, l'intégration continue a fait son apparition dans ces outils. Cette pratique consiste à vérifier, à chaque modification du code, que le résultat des modifications ne produit pas de régression. Cette régression est calculée à partir des tests (unitaires ou fonctionnels) définis dans l'application, bien qu'une partie de l'intégration continue (génération automatique, tickets…) puisse fonctionner sans tests, ces derniers ajoutent un objectif qualitatif et quantitatif important. Elle permet également la génération automatique d'artefacts (application générée, fichiers de logs…) ainsi que l'envoi automatique d'informations à d'éventuels outils liés (par exemple à chaque nouveau commit, lors d'une génération réussie, si un test n'est pas validé…).

Pour finir, ces forges logicielles possèdent généralement un système d'intégration pour d'autres outils, permettant d'ajouter des informations aux éléments présents dans la forge. Par exemple, il existe des outils calculant la qualité d'un code source à partir de plusieurs critères, l'outil SonarQube [6] est un des outils les plus utilisés dans ce cadre. Ce calcul peut être déclenché automatiquement par la forge, et les résultats affichés dans l'interface de cette dernière.

Pour résumer, les outils ont énormément évolué ces dernières années, et permettent d'obtenir un très grand nombre d'informations sur un projet. Les outils récents regroupent souvent plusieurs petits outils au sein d'une même interface, limitant ainsi le nombre d'interactions et de transferts de données nécessaires pour obtenir les mêmes informations avec d'autres outils.

Ces informations peuvent être affichées automatiquement par l'outil, avec

3. git : https ://git-scm.com/
4. SVN : https ://subversion.apache.org/
5. commit : validation d'un ensemble de modifications
6. https ://www.sonarqube.org/

cependant un risque de surcharge cognitive (trop de données, informations non essentielles...), ou bien le développeur peut sélectionner lui-même les informations souhaitées ce qui résulte en une perte de temps et en une perte de contexte de travail (tâche de codage interrompue).

Cette thèse apporte une solution sur ces deux problématiques en fournissant automatiquement un ensemble réduit d'informations parmi lesquelles le développeur peut naviguer et choisir ce qu'il désire afficher.

1.3 Modélisation logicielle

De même que les méthodes de conception, les méthodologies d'ingénierie ont également fortement évolué en parallèle. Parmi celles-ci on trouve par exemple le développement piloté par les tests [7] et l'Ingénierie Dirigée par les Modèles (IDM).

Cette dernière utilise des technologies permettant de décrire au travers de modèles, concepts, et langages, à la fois le problème posé (le besoin) et sa solution. La modélisation permet une abstraction du problème et de la solution, sous la forme de différents diagrammes. Il existe plusieurs langages de modélisation, chacun permettant de décrire un aspect particulier d'un logiciel. Parmi les plus connus se trouvent System Modeling Language (SysML), spécifique au domaine de l'ingénierie système, Business Process Model and Notation (BPMN), permettant l'analyse et la conception des processus métiers ainsi qu'UML qui se spécialise pour visualiser la conception d'un système logiciel. Celui-ci est le plus couramment utilisé par les développeurs pour définir leurs logiciels, et c'est donc tout naturellement que nous avons choisi de nous y intéresser dans cette thèse.

Langage UML

UML est un langage de modélisation graphique utilisé pour représenter l'architecture et les processus d'un logiciel. Il regroupe quatorze diagrammes, répartis en trois catégories :

- **Diagrammes de structure** : représentations graphiques permettant de structurer les différents éléments du logiciel.
- **Diagrammes de comportement** : représentations graphiques permettant de représenter les scénarios des utilisateurs du logiciel.

7. TDD : Test-Driven Development

— **Diagrammes d'interaction** : représentations graphiques permettant de schématiser les processus de l'application.

Parmi ces quatorze diagrammes, tous ne sont pas utilisés régulièrement par les développeurs. Plusieurs études [14, 15, 16, 22, 27] montrent que les trois plus répandus sont les diagrammes de classe, de séquence, et de cas d'utilisation, soit un par catégorie. Les principales raisons évoquées pour ce choix de diagrammes sont que ce sont les plus étudiés à l'université lors d'un cursus informatique. Le diagramme de classe est également celui permettant de représenter un logiciel dans sa globalité, occasionnant une meilleure communication entre les participants du projet.

Utilisation d'UML en entreprise

D'autres études [5, 31, 38] posent des questions plus orientées sur les raisons de l'utilisation ou non d'un diagramme, ainsi que sur les façons dont ils sont utilisés. Leurs résultats montrent que, la plupart du temps, ceux-ci ne sont pas utilisés de façon formelle mais plutôt dans un objectif de compréhension ou de communication. Ainsi, le diagramme de classe ne respectera peut-être pas entièrement le standard UML, mais sera utilisé par des développeurs pour se mettre d'accord sur une idée.

De même, le coût d'une étape de modélisation est souvent surestimé par les entreprises ou les clients, ce qui freine l'adoption de l'IDM et donc la création de diagrammes. Pourtant, Dzidek [16] montre qu'avec une mise à jour des documentations UML, des développeurs ont été plus efficaces, lors d'une modification de code, qu'un groupe n'utilisant pas de diagrammes.

Chaudron et al. [5] insistent également sur le fait qu'UML n'est pas suffisamment bien intégré aux outils de programmation, et que les modèles devraient suivre l'évolution du code. Pour ce faire, les outils doivent implémenter des fonctions de rétro-ingénierie, permettant de passer du code source à un ou plusieurs diagrammes. Le chapitre 2 page 16 compare les outils permettant la rétro-ingénierie, en mettant en avant les fonctionnalités et interactions utilisées par ceux-ci.

C'est sur ce dernier point que nous souhaitons apporter une contribution.

1.4 Psychologie du développeur

Le principal objectif des Interaction Homme-Machine (IHM) est de faciliter l'usage des logiciels en fournissant des interactions simples aux utilisateurs. Dans cette optique, il est nécessaire de bien comprendre comment ces derniers travaillent, utilisent leurs applications et quelles sont les fonctionnalités les

plus utiles. Dans notre cas, les utilisateurs sont des développeurs et un axe de recherche se consacre à l'étude de la façon dont ils travaillent en abordant principalement des aspects psychologiques (psychologie du développeur, compréhension de programme, visualisations efficaces...).

Green [36] mentionne la notion de « sprint » pour souligner que la programmation est une série de petites étapes se référant chacune à un modèle mental. Il est donc logique que la principale préoccupation du développeur consiste à relier le résultat du sprint avec ce qui a été produit jusqu'à présent. En effet, les développeurs lisent et analysent souvent ce qui a été fait afin de « tricoter » correctement (link) ce qu'ils font avec le reste du code.

L'environnement de développement doit donc optimiser le cycle « lecture / production ». Il doit être adapté au sprint actuel tout en fournissant un accès rapide aux informations qui aideront les développeurs à lier leur code à celui existant [10].

Il n'est donc pas étonnant que la plupart des éditeurs de code actuels proposent des raccourcis pour aller rapidement à la définition de l'élément sélectionné ou pour lister toutes les invocations d'une méthode spécifique. Cependant, la navigation n'est pas le seul moyen de trouver des informations pertinentes. Changer les propriétés visuelles des éléments du code est une autre façon de mettre en évidence ce qui peut intéresser les développeurs dans leur tâche d'assemblage ; par exemple, l'indentation montre clairement les différentes structures de contrôle dans lesquelles la ligne de code courante est imbriquée, la variation de couleur de fond est parfois utilisée pour identifier les différents endroits où une variable est utilisée, etc.

Les développeurs exécutent constamment des opérations de lecture de code pour trouver des informations afin de leur permettre de modifier leur code. Nous souhaitons raccourcir ces opérations de lecture en donnant un accès rapide aux éléments souvent utilisés ou visualisés par les développeurs. Ces éléments sont le plus souvent représentés comme des entités reliées par des liens. Le support textuel n'est cependant pas très efficace pour représenter un système avec des éléments interconnectés ; les représentations schématiques sont beaucoup plus efficaces dans cette tâche [28]. En outre, les travaux sur la psychologie des développeurs montrent que ces derniers ont principalement des problèmes pour comprendre l'algorithme plutôt que les briques de base d'un langage (par exemple les noms de variables) [7]. Ainsi, en plus des deux aspects que l'on devrait afficher (c'est-à-dire les entités et les liens), on peut ajouter l'algorithme utilisant les entités associées à la tâche de codage active (pouvant par exemple être représenté par un diagramme d'activités).

Pour que l'accès à l'information par le biais de la représentation graphique soit aussi rapide que possible, la lecture/décodage de cette représentation doit

prendre le moins de temps possible. L'ajustement cognitif est une préoccupation majeure pour notre outil, et le temps nécessaire pour passer d'une représentation à une autre, ou pour passer d'une tâche à l'autre en général (par exemple, révision de code vs. création de diagramme, recherche de dépendances de classe vs. débogage), est donc très important.

Cette préoccupation est au cœur des dimensions cognitives [21] et peut être trouvée dans la règle *intégration cognitive* dans la physique des notations [35].

Dans notre cas, lorsque l'on passe d'un éditeur de code textuel à une représentation graphique, il est nécessaire que les développeurs conservent leurs références. Par exemple, il est important que ces derniers reconnaissent les entités qu'ils viennent de manipuler. Par conséquent, la représentation graphique doit être proche de leur modèle mental et afficher des informations sur :

— les entités liées à la tâche de codage active, qu'elles soient ouvertes ou non dans l'EDI, et quel élément est actuellement actif.
— les données de voisinage, accessibles via les variables d'une méthode, les attributs d'un objet, l'héritage.....
— l'algorithme : le contenu d'une méthode actuellement consultée, ou à l'origine d'une recherche ou d'une navigation.

Dzidek [16] et Lethbridge [31] attestent que deux des trois diagrammes UML les plus utilisés sont ceux de classe et de séquence. C'est pourquoi nous les avons choisis dans VisUML. Le **diagramme de classe** est important parce que les développeurs peuvent reconnaître les entités qu'ils manipulent, ainsi que consulter d'autres entités connexes grâce aux différents types de lien. Afin d'afficher l'algorithme, en particulier pour le corps d'une méthode, nous avons opté pour le **diagramme de séquence**. Nous supposons que ce diagramme est un support visuel complémentaire pour les développeurs qui travaillent sur l'implémentation d'une méthode. Ces derniers pourront voir toutes les classes impliquées dans cette méthode, et surtout les différents échanges (et donc les liens) entre elles.

De plus, nous avons choisi d'afficher nos diagrammes dans des pages web afin de pouvoir facilement connecter notre outil à n'importe quel EDI. Cet aspect peut sembler purement technique mais il ne l'est pas : l'une des dimensions cognitives est la visibilité dans laquelle la juxtaposition de deux points de vue est un moyen de passer facilement de l'un à l'autre. Si l'EDI ne permet pas d'afficher deux grandes fenêtres l'une à côté de l'autre (chacune sur un écran ou les deux sur le même écran), c'est nativement possible avec notre approche, même en utilisant une tablette ou tout autre dispositif d'affichage avec un système d'exploitation contenant un navigateur web.

Enfin, nous avons utilisé la sémiologie graphique et ses variables visuelles [17, 1] afin d'améliorer la lecture et la compréhension des informations affichées.

Ainsi, le diagramme de classe et de séquence utilisent chacun un ensemble de variables visuelles (par exemple, la couleur de fond, la taille de la bordure...) pour mettre en avant ou ajouter du sens à une information précise. Cette utilisation est détaillée dans la section 4.2 page 68.

Chapitre 2

Fonctionnalités et interactions de rétro-ingénierie dans les outils de développement et de modélisation

Sommaire

Ce chapitre présente les différents critères que nous avons choisis pour comparer notre proposition aux outils existants fournissant des fonctionnalités similaires.

Pour chaque section, nous présentons le critère choisi ainsi que les valeurs que ce dernier peut prendre. Nous discutons également de l'importance de ce critère dans le contexte de la thèse. Certaines comparaisons de la littérature seront également citées lorsqu'elles se rapportent à un point particulier. Enfin, la dernière partie résume les différences observées et insiste sur les choix faits durant cette thèse.

2.1 Présentation des outils comparés

Les outils qui seront comparés dans ce chapitre sont des outils couramment utilisés et disponibles sur internet. La plupart de ces outils sont également maintenus régulièrement, d'autres proviennent de la recherche et sont des prototypes ou des preuves de concept, parfois très instables.

Outils	Type	Domaine
Papyrus	Modeleur UML	Commercial
Visual Paradigm	Modeleur UML	Commercial
UML Designer	Modeleur UML	Commercial
Modelio	Modeleur UML	Commercial
ObjectAid	Visualisation de code	Commercial
MaintainJ	Visualisation de code	Commercial
Architexa	Visualisation de code	Commercial
IntelliJ	EDI (visualisation)	Commercial
OctoBubble	Visualisation de code	Recherche
VisUML	Visualisation de code	Recherche

Tableau 2.1 – Présentation des outils comparés

Le tableau 2.1 regroupe les outils comparés dans ce chapitre. La plupart de ces outils viennent du domaine commercial et sont utilisés quotidiennement par des professionnels.

Papyrus [1], **Visual Paradigm** [2], **UML Designer** [3] et **Modelio** [4] sont des logiciels de modélisation (UML et autres). Ils possèdent les mêmes fonctionnalités de

1. Papyrus : `https://www.eclipse.org/papyrus/`
2. Visual Paradigm : `https://www.visual-paradigm.com/`
3. UML Designer : `http://www.umldesigner.org/`
4. Modelio : `https://www.modelio.org/`

génération de diagramme, de code associé et de validation syntaxique et formelle des modèles. Ils possèdent également un mécanisme de *reverse-engineering*, qui nous intéresse dans ce chapitre.

ObjectAid [5], **MaintainJ** [6] et **Architexa** [7] sont quant à eux des logiciels de visualisation de code ou de traces de code. Ils ne permettent pas la création manuelle de diagramme et se base donc essentiellement sur du *reverse-engineering*.

IntelliJ [8] est un EDI qui fournit des fonctionnalités de *reverse-engineering* intégrées à l'outil, qui génèrent des diagrammes de classe UML. Ces diagrammes sont connectés au code, il n'est donc pas possible de créer un diagramme manuellement.

OctoBubble[25] [9] est un outil de recherche permettant de visualiser du code et des diagrammes en parallèle. Les diagrammes sont générés automatiquement via du *reverse-engineering* et peuvent être modifiés manuellement (*round-trip engineering*.

Enfin, **VisUML** [10] est l'outil développé dans cette thèse. Il se présente sous la forme d'un plugin pour EDI et utilise le code source affiché dans ce dernier pour générer des diagrammes automatiquement. Il n'est pas possible de créer de diagramme manuellement.

2.2 Données en entrée

Le premier groupe de critères présentés concerne les données utilisées par l'outil pour générer des informations et/ou des visualisations. Il compare le type des données, le langage des données et la facilité avec laquelle l'outil peut gérer d'autres formats ou langages.

2.2.1 Type des données

Il existe plusieurs façons de transformer en données puis en modèles les informations fournies ou définies dans un logiciel. La plupart du temps, les outil se basent sur le **code source** de l'application pour générer une abstraction de celui-ci. Cependant, dans certains cas, il est possible d'utiliser les traces (« logs ») générées par l'utilisation d'une application. Ces traces sont appelées **StackTraces**. Enfin, il est également possible de générer des visualisations grâce

5. ObjectAid : `https://www.objectaid.com/`
6. MaintainJ : `http://www.maintainj.com/`
7. Architexa : `https://www.architexa.com/`
8. IntelliJ : `https://www.jetbrains.com/idea/`
9. OctoBubble : `http://www.rodijolak.com/`
10. VisUML : `http://these.mickaelduruisseau.fr/VisUML/`

aux données recueillies durant l'exécution d'un logiciel ; ce type de données est appelée **Runtime**.

Outils \Type	Code source	StackTrace	Runtime
Papyrus	✓		
Visual Paradigm	✓		
UML Designer	✓		
Modelio	✓		
ObjectAid	✓	✓	
MaintainJ		✓	✓
Architexa	✓		
IntelliJ	✓		
OctoBubble	✓		
VisUML	✓		

Tableau 2.2 – Critère 1 : Type des données en entrée

Pour VisUML, il est essentiel que le type des données en entrée soit le code source, puisque l'objectif est de fournir une vue en temps réel, à chaque modification du code. Les deux autres types ne conviennent donc pas à notre approche.

2.2.2 Langage des données

Chaque type de donnée utilisée en entrée est associé à un ou plusieurs langages. Ainsi, pour le code source le langage peut être Java, C++, Python, etc. Pour les Stacktraces, même si le principe de génération de ces traces est le même quel que soit le langage (par exemple récupération d'une erreur et des paramètres associés), le résultat produit dans les stacktraces diffère en syntaxe en fonction du langage sous-jacent (voir figure 2.1). Par conséquent, il est difficile de rendre générique le processus de récupération des Stacktraces en vue de leur exploitation par un outil pour alimenter un modèle.

Quant aux outils utilisant en temps réel les informations issues de l'exécution d'un programme (par exemple accès à la pile et aux variables), ceux-ci nécessitent un développement spécifique afin de récupérer et d'interpréter les données du débogueur.

Pour notre approche, le langage n'est pas très important, mais il est essentiel qu'il soit orienté objet afin d'être compatible avec UML, c'est-à-dire pouvant être représenté par des entités et des liens. Des langages tels que C++, C#, PHP, Swift, Kotlin... sont de très bons candidats. Pour notre étude, nous avons restreint le choix à C++/C#, PHP et Java avec un focus sur ce dernier car il fait partie des

```
Exception in thread "main" java.lang.NullPointerException
        at com.example.myproject.Book.getTitle(Book.java:16)
        at com.example.myproject.Author.getBookTitles(Author.java:25)
        at com.example.myproject.Bootstrap.main(Bootstrap.java:14)
```

(a) *Stacktrace* Java

```
Program terminated with signal 6, Aborted.
[New process 22857]
#0  0x00007f4189be5fb5 in raise () from /lib/libc.so.6
#0  0x00007f4189be5fb5 in raise () from /lib/libc.so.6
#1  0x00007f4189be7bc3 in abort () from /lib/libc.so.6
#2  0x00007f4189bdef09 in __assert_fail () from /lib/libc.so.6
#3  0x00000000004007e8 in recursive (i=5) at ./demo1.cpp:18
#4  0x00000000004007f3 in recursive (i=4) at ./demo1.cpp:19
#5  0x00000000004007f3 in recursive (i=3) at ./demo1.cpp:19
#6  0x00000000004007f3 in recursive (i=2) at ./demo1.cpp:19
#7  0x00000000004007f3 in recursive (i=1) at ./demo1.cpp:19
#8  0x00000000004007f3 in recursive (i=0) at ./demo1.cpp:19
#9  0x0000000000400849 in main (argc=1, argv=0x7fff2483bd98) at ./demo1.cpp:26
```

(b) *Stacktrace* C++

FIGURE 2.1 – Exemples de *Stacktraces* en Java et C++

trois langages les plus utilisés en Entreprise de services du numérique (ESN) [11], et rassemble à la fois des logiciels et des applications (avec Android). De par son enseignement en Université, ce langage nous permet également de tester rapidement la solution avec des étudiants.

On constate d'une part que tous les outils étudiés supportent Java et que seuls trois (hors VisUML) gèrent plusieurs langages. D'autre part, seul IntelliJ supporte l'ensemble des langages présentés. Il propose également une implémentation générique des éléments d'un langage (classe, bloc, exécution...), qui permettra à VisUML d'intégrer d'autres langages (voir section 6.6 page 143).

2.2.3 Facilité d'intégration d'autres langages

En plus des langages supportés, l'un des critères mis en avant dans cette comparaison est la possibilité et la facilité avec laquelle un langage peut être ajouté à un outil. Ce critère est défini en fonction des langages déjà supportés, des formats gérés ainsi que des données produites. Enfin, ce critère dépend également de la façon dont la partie analysant des données est intégrée dans l'outil. Il est représenté par un niveau de facilité allant de 1 (le plus facile) à 4 (le

11. IEEE - The Top Programming Languages 2018 : http ://spectrum.ieee.org/computing/software/the-2017-top-programming-languages

Outils \Langages	Java	C++ / C#	PHP
Papyrus	✓	✓	
Visual Paradigm	✓	✓	
UML Designer	✓		
Modelio	✓		
ObjectAid	✓		
MaintainJ	✓		
Architexa	✓		
IntelliJ	✓	✓	✓
OctoBubble	✓		
VisUML	✓		✓ (partiel)

Tableau 2.3 – Critère 2 : Langage des données en entrée

plus difficile).

Outils	Facilité d'intégration
Papyrus	+++
Visual Paradigm	+++
UML Designer	++
Modelio	++
ObjectAid	+
MaintainJ	+
Architexa	++
IntelliJ	++++
OctoBubble	+++
VisUML	++++

Tableau 2.4 – Critère 3 : Intégration d'autres langages

Le **niveau ++++** correspond aux outils pour lesquels intégrer un langage est le plus facile.

— IntelliJ possède une abstraction d'un très grand nombre de langages de programmation, et l'analyse du code de ces langages est donc très rapide. Cette abstraction consiste en un ensemble d'éléments représentant les concepts des langages orientés objets comme les classes, les interfaces, les méthodes, les attributs, les annotations…, ainsi que les blocs (for, if, else…) et instructions du code (appel de méthode). Pour chacun de ces éléments, IntelliJ récupère également sa position dans le fichier et les commentaires associés. Enfin IntelliJ fournit un mécanisme permettant de mettre en évidence dans le code source un élément en particulier.

— VisUML utilise deux mécanismes en parallèle. Le premier est fourni par IntelliJ comme décrit ci-dessus et permet d'une part d'identifier un élément à partir de la position du curseur et d'autre part de mettre en évidence l'élément désigné par l'utilisateur depuis les diagrammes UML. Le second repose sur la librairie Spoon [37] qui produit un arbre syntaxique abstrait du projet.
Spoon nous permet de gérer Java à la fois sur Eclipse et IntelliJ, tandis qu'un prototype pour PHP a été développé via l'abstraction proposée par IntelliJ. Il est possible d'étendre VisUML grâce à ce mécanisme.

Le **niveau +++** regroupe des outils gérant déjà plusieurs langages, mais pour lesquels l'analyse du code n'est pas connue ou effectuée via des plugins. Implémenter un nouveau langage dans ces outils nécessite donc plus de travail que pour les outils de niveau ++++ et l'utilisation de *parser* spécifique à ce langage.

Le **niveau ++** contient les outils ne gérant qu'un langage (Java) et dont l'implémentation de l'analyse est cachée ou fait partie intégrante de l'outil.

Le **niveau +** contient les outils utilisant les données au format *StackTrace* ou *Runtime*, très dépendants des langages et de la façon dont ceux-ci enregistrent leurs traces ou permettent un accès aux données pendant l'exécution.

2.3 Données et visualisations en sortie

Ce second groupe de critères rassemble les informations et comparaisons relatives aux données générées par les outils, ainsi que les visualisations associées. Il contient quatre critères : le type des données, les diagrammes (visualisations) gérés, le support de ces diagrammes, ainsi que l'exportation et l'accès aux données.

2.3.1 Type des données en sortie

De la même façon que les données en entrée, les données générées possèdent plusieurs types. Ce critère se concentre sur les données et non sur les visualisations qui seront traitées dans le critère suivant.

Dans le cas d'un outil de rétro-ingénierie, les données sont souvent stockées sous la forme d'un modèle, mais il arrive que ce modèle ne soit pas accessible ou soit caché. De plus, celui-ci doit respecter un langage standard parmi ceux existants. Ce critère compare donc également les langages de modélisation supportés.

Quand un modèle est visible (par exemple dans Papyrus), l'utilisateur a accès aux éléments de ce dernier et peut donc les modifier si besoin. Il peut ainsi

Outils	Modèle visible	UML	BPMN	SysML
Papyrus	oui	✓	✓	✓
Visual Paradigm	oui	✓	✓	✓
UML Designer	oui	✓		
Modelio	oui	✓	✓	✓
ObjectAid	non	✓		
MaintainJ	non	✓		
Architexa	non	✓		
IntelliJ	non	✓		
OctoBubble	non	✓		
VisUML	non	✓		

Tableau 2.5 – Critère 4 : Type des données en sortie

générer plusieurs diagrammes pour représenter différent points de vues d'un même projet. Généralement, lorsque le modèle n'est pas visible graphiquement, il est souvent inaccessible et donc ne permet aucune interaction avec lui.

VisUML, quant à lui, ayant pour but de faciliter la compréhension d'un logiciel, nous avons logiquement opté pour UML, qui permet de représenter une application sous forme de diagrammes. Nous n'avons pas souhaité fournir de modèle à l'utilisateur puisque nous considérons le code comme notre modèle lors de la génération des visualisations en temps réel. A titre de démonstration d'intégration avec d'autres outils, nous avons cependant implémenté une solution pour rendre ce modèle visible dans des outils de modélisation tels que Papyrus et GenMyModel.

Ainsi, à partir de son code source, un développeur peut voir à la fois des diagrammes générés automatiquement ainsi que le modèle sous-jacent dans un outil de modélisation, ce qui lui permet de générer autant de points de vue que nécessaire.

2.3.2 Visualisations gérées

Ce critère vise à comparer les diagrammes gérés par les outils comparés dans ce chapitre. Nous nous concentrerons sur les diagrammes du langage UML, puisque c'est le langage de modélisation commun à tous ces outils.

De façon générale, le modèle associé aux données ne comporte aucune représentation visuelle de ces dernières en dehors d'une liste arborescente des éléments générés. Cependant, bien que certains outils gèrent un grand nombre de diagrammes, la fonctionnalité de rétro-ingénierie n'est pas toujours disponible pour chacun de ceux-ci. Le tableau suivant ne met en avant que les diagrammes

qui peuvent être créés automatiquement à partir de données en entrée.

Outils \ Diagrammes	Diag. classe	Diag. séquence	Diag. activité
Papyrus	✓		
Visual Paradigm	✓	✓	
UML Designer	✓		
Modelio	✓		
ObjectAid	✓	✓	
MaintainJ	✓	✓	
Architexa	✓	✓	
IntelliJ	✓	✓ (avec plugin)	
OctoBubble	✓		
VisUML	✓	✓	✓

Tableau 2.6 – Critère 5 : Diagrammes pris en charge

VisUML permet de générer automatiquement trois types de diagrammes que nous avons sélectionnés par rapport à leur utilisation au sein des entreprises. Le diagramme de classe est celui le plus utilisé pour représenter la structure et les liens entre les éléments d'un projet. Le diagramme de séquence permet quant à lui de visualiser le détail d'une méthode. Le diagramme d'activité de VisUML représente les mêmes informations que le diagramme de séquence, mais sous une vue différente. Ce dernier a été réalisé dans le cadre d'une preuve de concept et est présenté en annexe C page 185.

2.3.3 Support des visualisations

La plupart des outils visualisent les données de façon interne au travers de vues intégrées à ceux-ci, par exemple par l'intermédiaire de plugins pour EDI. Cette intégration permet d'afficher les visualisations soit juxtaposées au code (onglet différent par exemple), soit à la place du code.

Notre approche étant basée sur une extension de l'EDI du développeur, VisUML ne possède pas d'interface propre. Les visualisations sont disponibles sur des pages web, ou directement dans certains outils au travers de plugins (Papyrus et GenMyModel). Nous avons fait ce choix afin de permettre une plus grande flexibilité pour l'utilisateur. Il peut choisir d'ouvrir les pages web sur un même écran ou sur une tablette, pour disposer au mieux de ces informations. De plus, ces pages web sont partageables n'importe où dans le monde et permettent une visualisation synchrone avec des collaborateurs distants tels que des équipes off-shore.

Outils \ Visualisation	Interne à l'outil	Intégré aux EDI	Vues web
Papyrus	✓		
Visual Paradigm	✓	✓	
UML Designer	✓		
Modelio	✓		
ObjectAid	✓		
MaintainJ	✓		
Architexa	✓	✓	
IntelliJ	✓	✓	
OctoBubble	✓		
VisUML		✓	✓

Tableau 2.7 – Critère 6 : Support des visualisations

On constate que dans le tableau 2.7 seul VisUML propose un affichage en mode web à partir d'un projet existant. OctoBubbles fournit une interface dédié mais nécessite l'intégration manuelle du code source dans l'outil, et ne permet donc pas de se connecter de façon synchrone à un projet existant. Il est à noter cependant que d'autres outils, tels que GenMyModel, permettent une visualisation UML sur le web à partir de code source existant, mais sans fournir de synchronisation (par exemple, GenMyModel peut générer un modèle à partir d'un dépôt Git, mais uniquement de façon manuelle et à l'initialisation du projet).

2.3.4 Accès et exportation des données générées

Il peut être intéressant d'utiliser plus tard les données et visualisations générées par les outils, c'est-à-dire les modèles et les diagrammes. C'est pourquoi la plupart des outils proposent un système de sauvegarde dans un fichier. Il existe aussi plusieurs mécanismes d'exportation sous des formats divers (par exemple au format XML Metadata Interchange (XMI)), aussi bien pour le modèle que pour les visualisations. Ces exports constituent donc des instantanés créés au moment où l'utilisateur le décide.

De plus, il existe également des outils qui transmettent leurs informations en temps réel via plusieurs interfaces (par exemple, via des requêtes HyperText Transfer Protocol (HTTP) ou par un bus de données). C'est le cas de Papyrus grâce au plugin « ModelBus® Adapter for Papyrus » [12].

Pour VisUML, le bus de données a été un choix nécessaire afin de permettre aux visualisations d'être mises à jour en temps réel pour refléter les dernières modifications du code. Il est également possible d'exporter un diagramme sous

12. ModelBus® Adapter for Papyrus : https ://www.modelbus.org/en/papyrus.html

Outils \ Export	Modèle		Diagrammes	
	Fichier (XMI)	Bus de données	XML	Image
Papyrus	✓	✓ (avec plugin)	✓	✓
Visual Paradigm	✓			✓
UML Designer	✓			✓
Modelio	✓			✓
ObjectAid			✓	✓
MaintainJ				✓
Architexa				✓
IntelliJ				✓
OctoBubble			✓	✓
VisUML		✓		✓

Tableau 2.8 – Critère 7 : Export des données

forme d'image (PNG ou SVG) pour obtenir un instantané de ce dernier. Cette fonctionnalité répond ainsi au problème de documentation rarement mise à jour ou incomplète. L'export en XMI pour VisUML serait possible mais n'est pas prioritaire pour notre outil, celui-ci étant orienté visualisation temps réel et non export/import de modèles entre outils. Comme nous le montrerons dans la section 3 page 33, VisUML est facilement connectable à d'autres outils (par exemple GenMyModel et Papyrus) via son bus et permet une synchronisation en temps réel des modèles.

2.4 Fonctionnalités des outils

Ce troisième et dernier groupe de critères inclut tous les critères relatifs aux fonctionnalités fournies par les outils, ainsi que les interactions implémentées par ces derniers.

2.4.1 Génération et mise à jour de modèles

L'un des aspects les plus importants d'un outil de rétro-ingénierie est la génération du modèle (qu'il soit affiché ou non). Celle-ci s'effectue au moins une fois, au début du processus de rétro-ingénierie. Cette première génération est obligatoire pour afficher une visualisation.

Par la suite, certains outils permettent de régénérer ou de mettre à jour la totalité ou un sous-ensemble du modèle de façon manuelle en sélectionnant les éléments à mettre à jour. Enfin, d'autres outils régénèrent automatiquement les

éléments qui ont subi une modification (écoute des fichiers sources, traces…), ce qui réduit le nombre d'actions utilisateur nécessaires.

Outils	Génération automatique	Mise à jour	
		Manuelle	Automatique
Papyrus			
Visual Paradigm		✓	
UML Designer		✓	
Modelio			
ObjectAid		✓	
MaintainJ		✓	
Architexa			
IntelliJ	✓		✓
OctoBubble		✓	✓
VisUML	✓		✓

Tableau 2.9 – Critère 8 : Génération et mises à jour du modèle

Dans ce tableau, la première colonne (Génération automatique) indique que les outils ne nécessitent pas d'intervention de l'utilisateur pour afficher des visualisations. A l'inverse, pour une génération manuelle, l'utilisateur doit sélectionner les éléments qu'il veut transformer en modèle avant de pouvoir les afficher dans un ou plusieurs diagrammes. Cette étape n'existe pas pour IntelliJ et VisUML, puisque le modèle n'est pas visible pour l'utilisateur et que les données en entrée sont recueillies automatiquement.

Les deux colonnes suivantes indiquent si l'outil met à jour automatiquement un élément modifié parmi les données en entrée, ou si l'utilisateur doit demander systématiquement la mise à jour.

2.4.2 Choix des éléments du modèle à afficher

La génération du modèle nécessite une liste d'éléments en entrée. Cette liste est souvent gérée par l'utilisateur, et ce dernier doit alors ajouter ou retirer les éléments qu'il souhaite afficher. Il est cependant possible que l'outil ne permette pas la sélection précise d'éléments, mais utilise un dossier de projet complet, ce qui inclut automatiquement un grand nombre de fichiers.

D'autres, cependant, ajoutent automatiquement des éléments par rapport à certains critères, par exemple les données recueillies au *Runtime*, ou les *Stack-Traces* importées.

Enfin, tous les outils permettent l'ajout, après la première génération, de nouveaux éléments à l'intérieur du modèle.

Outils \ Choix des éléments	Manuel	Auto. (selon critères)
Papyrus	✓	
Visual Paradigm	✓	
UML Designer	✓	
Modelio	✓	
ObjectAid		✓
MaintainJ		✓
Architexa	✓	
IntelliJ	✓	
OctoBubble	✓	
VisUML		✓

Tableau 2.10 – Critère 9 : Choix des éléments

Pour ObjectAid et MaintainJ, les éléments sélectionnés correspondent soit aux points d'arrêt placés par le développeur, soit à l'origine de l'exception récupérée dans la *StackTrace*. VisUML quant à lui se base sur les onglets ouverts dans l'EDI tout en fournissant des informations complémentaires telles que les éléments associés et la position du curseur.

2.4.3 Informations affichées sur les diagrammes

Les diagrammes UML possèdent chacun une liste d'éléments et d'informations qu'ils peuvent gérer et afficher. Les outils de visualisation ne gèrent cependant pas tout le temps l'ensemble de ces éléments. Le diagramme de séquence, par exemple, utilise des éléments qui ne sont pas toujours supportés. C'est le cas des « Fragments » [13] (« alt », « loop », « opt ») et également de certaines informations utiles telles que les « try/catch » ou les appels parents (endroits où une méthode analysée est appelée).

Sur le diagramme de classe, il est possible d'afficher ou non certaines informations sur un élément (ses attributs, méthodes…). Cette fonctionnalité est gérée par tous les outils.

Il est également possible de créer des éléments qui n'ont pas été définis ou sélectionnés mais qui possèdent une ou plusieurs relations avec au moins un élément présent.

Cet ajout permet d'obtenir une vue plus précise d'un élément, sans surcharger le diagramme.

Dans VisUML, nous avons voulu être au plus proche du code, c'est pourquoi nous supportons un grand nombre d'éléments dans le diagramme de séquence.

13. En UML, un Fragment représente un ensemble d'exécutions

Outils \ Informations	Fragments	Appels parents	Éléments reliés
Papyrus	✓		
Visual Paradigm	✓		
UML Designer	✓ (partiel)		
Modelio	✓		
ObjectAid			
MaintainJ			
Architexa	✓ (partiel)		✓ (partiel)
IntelliJ	✓		✓ (partiel)
OctoBubble			✓ (partiel)
VisUML	✓	✓	✓

Tableau 2.11 – Critère 10 : Informations des diagrammes

Ainsi, contrairement à la majorité des autres outils, VisUML met en avant les « try/catch », les « switch », les fragments classiques et gère le niveau de « profondeur » d'un appel. Nous renforçons visuellement cette structure en blocs par l'utilisation de couleurs tel que présenté section 4.2.4 page 72. Le développeur peut donc appréhender plus rapidement l'architecture du code.

Nous ajoutons également des informations sur les appels parents d'une méthode en affichant la signature des méthodes appelantes ainsi que les numéros de lignes concernés. Ces appels parents sont interactifs et permettent une navigation dans le code et entre diagrammes tel que présenté section 5.3.3 page 99.

Sur le diagramme de classe, pour afficher l'ensemble des informations d'une classe, nous avons choisi d'afficher non seulement la classe mais également les éléments reliés non ouverts, par exemple une classe mère ou les classes filles. Cette fonctionnalité est présentée section 4.1.1 page 64.

2.4.4 Liens entre source et visualisations

Puisque les outils de rétro-ingénierie utilisent une source de données (le code source par exemple) afin de générer des visualisations, il est possible de créer des liens entre un élément en entrée et un élément en sortie.

Les liens ainsi créés peuvent être utilisés pour mettre en avant des informations ou mettre à jour automatiquement l'entrée et/ou la sortie. Ainsi, la modification d'un élément d'entrée (par exemple du code) va automatiquement mettre à jour la visualisation associée (par exemple la classe UML associée).

Les outils permettant une mise à jour des éléments en entrée et en sortie implémentent le « Round-Trip Engineering », permettant une synchronisation complète et bidirectionnelle entre la source et la sortie de l'outil.

Outils	Lien source-visu.	Mise à jour visu.	Round-Trip
Papyrus			
Visual Paradigm	✓	✓	✓
UML Designer			
Modelio	✓		
ObjectAid	✓	✓	
MaintainJ			
Architexa	✓		
IntelliJ	✓	✓	✓
OctoBubble	✓		✓ (partiel)
VisUML	✓	✓	

Tableau 2.12 – Critère 11 : Liens entre source et visualisations

VisUML est un outil de visualisation dynamique du code. Il est donc logique de conserver un lien entre le code source et ses visualisations et d'exploiter ce lien pour mettre à jour les diagrammes à chaque modification du code.

Le « Round-Trip Engineering » permet également de modifier le code source à partir de diagrammes. Par exemple en ajoutant une classe mère dans le diagramme de classe UML, l'EDI proposera une mise à jour du code. OctoBubble permet de modifier à la fois le code et le diagramme, mais ne permet cependant pas de sauvegarder le code généré. C'est une méthode que nous avons implémentée sous forme de preuve de concept avec un autre outil connecté à VisUML (que nous avons appelé « Télécommande contextuelle », voir section suivante et annexe D page 189). Le « Round-Trip Engineering » dans son ensemble fait partie des évolutions envisagées, voir section 6.6 page 143.

2.4.5 Interactions avec les diagrammes

Une fois les diagrammes générés, il est possible d'interagir avec ceux-ci. Les outils générant des liens entre la source et les visualisations proposent nécessairement une navigation rapide, parfois bidirectionnelle. Il devient alors possible pour un utilisateur de passer du diagramme au code via une interaction (plus ou moins simple selon l'outil), mais également du code au diagramme de la même façon (mise en valeur d'informations, ajout d'éléments…).

VisUML quant à lui ajoute également des informations sur les diagrammes, indiquant à l'utilisateur où se trouve son curseur dans l'EDI. Ainsi lorsque ce dernier se trouve dans une méthode, celle-ci est mise en valeur dans le diagramme de classe (par un changement de la couleur de son nom) et le message relié sera également plus prononcé dans le diagramme de séquence (ligne plus

épaisse et coloriée), ce qui permet à l'utilisateur de se repérer rapidement mais également à un collaborateur de connaître le code actuellement *revu* (*review* de code).

Outils	Navigation code -> diag.	Navigation diag. -> code	Informations curseur
Papyrus			
Visual Paradigm		> 1 clic	
UML Designer			
Modelio		> 1 clic (partiel)	
ObjectAid		> 1 clic	
MaintainJ			
Architexa		double clic	
IntelliJ	✓	> 1 clic	
OctoBubble	✓		
VisUML	✓	1 clic	✓

Tableau 2.13 – Critère 12 : Interactions sur les diagrammes

Comme présenté dans ce tableau, VisUML gère plus d'interactions que les autres outils. Elles sont étroitement liées à l'aspect visualisation en temps réel de VisUML ; cette absence de visualisation en temps réel dans les autres outils explique peut être le manque d'interactivité avec les diagrammes.

Par rapport aux outils proposant des interactions, celles de VisUML sont plus simples et plus rapides. C'est notamment le cas lors d'une navigation depuis un élément du diagramme et vers le code. Pour VisUML, il suffit d'un simple clic sur cet élément pour voir le code associé. Architexa implémente quant à lui le double clic sur un élément, ce qui est également assez rapide. Les autres outils proposent toujours un menu contextuel ce qui augmente la complexité et le temps nécessaire pour effectuer l'interaction. Ces menus possèdent par ailleurs plusieurs niveaux (parfois jusqu'à cinq niveaux), ce qui oblige l'utilisateur à parcourir tous les sous-menus avant de trouver l'interaction désirée, comme le montre la figure 2.2.

2.5 Conclusion

Ce chapitre a présenté les critères utilisés pour comparer notre outil aux outils existants. Il s'est concentré sur des fonctionnalités et éléments spécifiques à la rétro-ingénierie.

VisUML propose des visualisations générées grâce à la rétro-ingénierie. Cette

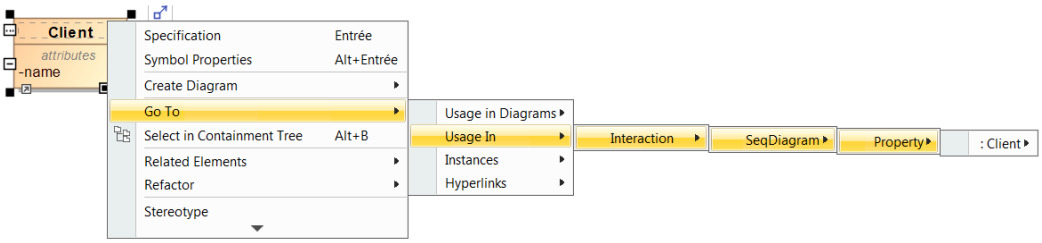


FIGURE 2.2 – Sous-menus imbriqués

dernière est effectuée à partir du code source, ce qui permet une (re)génération à chaque modification ou suppression d'un ou plusieurs fichiers du projet.

L'outil actuel supporte le langage Java (un prototype PHP est également en partie développé), afin d'être utilisable par un grand nombre de développeurs. L'architecture, présentée dans le chapitre suivant, est également conçue de sorte à permettre une intégration d'autres langages dans le futur (voir section 6.6 page 143).

Nous avons choisi le langage UML afin de nous adapter aux habitudes des développeurs, tout en bénéficiant des diagrammes associés. Ces derniers permettent de représenter à la fois la structure d'un projet et les détails d'une implémentation. Les diagrammes de classe et de séquence sont les deux plus utilisés en entreprise, ce qui permet de réduire les efforts de compréhension lors de l'utilisation de VisUML.

Afin de fournir plus de flexibilité aux utilisateurs, ces visualisations sont disponibles sur des pages web. Elles peuvent être ouvertes sur un ordinateur, sur tablette ou sur mobile, la vue s'adaptant au périphérique. L'utilisation du bus de données permet également d'afficher un diagramme à distance et de partager sa visualisation avec d'autres utilisateurs.

Notre approche de visualisation en temps réel nécessite également de maintenir les diagrammes à jour, ce qui est grandement simplifié par l'utilisation d'un bus.

Les informations affichées et les interactions avec les diagrammes sont présentées dans les chapitres 4 page 63 et 5 page 89. Ces dernières ont toujours été réfléchies de sorte à faciliter l'usage du développeur, en lui fournissant les informations utiles au bon moment, et en permettant une navigation rapide et bidirectionnelle entre le code et les diagrammes.

Chapitre 3

VisUML : Présentation et architecture

Sommaire

VisUML est un outil pour les développeurs permettant la création en temps réel de diagrammes UML. Ces diagrammes sont interactifs et synchronisés avec le code, et permettent d'avoir une représentation graphique de celui-ci sans accroître la charge cognitive du développeur, ni lui demander du temps pour la production de diagrammes.

Ce chapitre présente l'outil de manière générale puis se concentre sur l'architecture de VisUML, ses différentes parties ainsi que leur système de communication. Il présente ensuite la structure des entités utilisées pour représenter les éléments du code, puis la structure des messages envoyés permettant d'établir un lien entre le code et les représentations graphiques.

3.1 Présentation générale

VisUML est un outil de visualisation de code source. Il permet la génération automatique de diagrammes UML à partir d'informations récupérées depuis le code source d'un projet. Il consiste d'un part en l'ajout d'un plugin dans l'EDI de l'utilisateur, qui se charge d'analyser le code, et d'autre part de représentations graphiques. Ces dernières sont mises à jour en temps réel, à chaque modification dans le code. Elles sont également interactives, permettant une navigation simple et efficace entre le code et celles-ci.

Actuellement, deux types de diagrammes sont générés [1]. Le premier est un diagramme de classe basé sur les onglets ouverts dans l'EDI, ainsi que les éléments associés (classe mère, classes filles...), et éventuellement non encore ouverts dans l'EDI. Ce choix d'affichage permet d'obtenir une vue simplifiée interactive qui correspond au modèle mental du développeur.[24, 29] Il est en effet possible d'interagir avec les éléments du diagramme simplement en cliquant dessus. Un clic sur une classe, une méthode ou un attribut va aussitôt mettre à jour l'EDI pour qu'il affiche cet élément, en changeant d'onglet si nécessaire, en défilant le code jusqu'à l'élément et en mettant en surbrillance celui-ci.

Le deuxième type de diagramme est un diagramme de séquence, affichant la méthode actuellement parcourue. De même que sur le diagramme de classe, chaque élément est cliquable, permettant une navigation rapide depuis le diagramme. De plus, la position du curseur de l'utilisateur est aussi écoutée. Ainsi, s'il est déplacé dans le corps d'une autre méthode, cette dernière sera alors affichée sur le diagramme de séquence.

Chacune de ces représentations graphiques possèdent également des filtres et autres outils simplifiant leur utilisation, ces informations sont présentées dans le chapitre 4 page 63.

La figure 3.1 page suivante montre l'ensemble des applications et prototypes développés pour VisUML.

VisUML offre plus de 65 actions possibles (liste complète disponible en annexe G page 223) au travers de l'EDI, des deux diagrammes (classe et séquence) ainsi que du panneau de lancement. Ces actions sont listées ci-dessous. Cette liste ne donne que les actions (avec le clavier et la souris, ou le doigt sur une tablette) que l'utilisateur peut exécuter, mais pas les conséquences qu'elles entraînent (modifications des diagrammes, navigation dans le code source, etc.) ; ces conséquences seront présentées tout au long de la suite de ce document.

1. Le diagramme d'activité a été réalisé sous la forme d'une preuve de concept, et est pour le moment à l'étape de prototype, voir annexe C page 185.

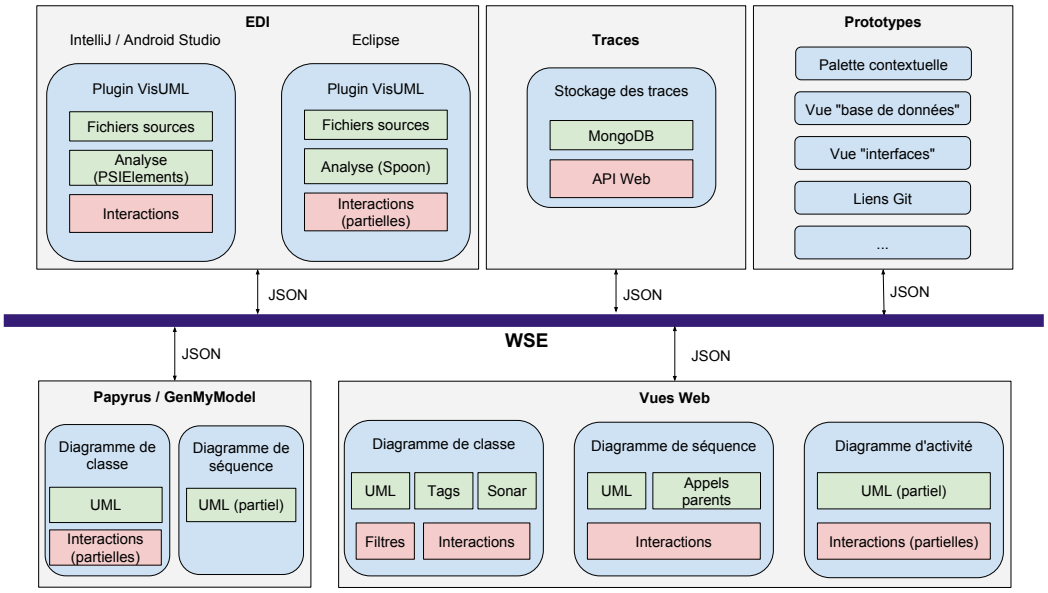


FIGURE 3.1 – Écosystème de VisUML (en bleu les parties développées)

Diagramme de classe (+30 actions)

1. Filtrage sur les éléments affichés (avec le menu gauche ou avec le langage de requête (voir sections 4.3.4 page 77 et 4.3.6 page 80))
2. Modification de la représentation UML (changement de bordure, de couleur de fond, rotation, échelle…)
3. Navigation et zoom sur le diagramme
4. Interactions sur le diagramme (ouvrir une classe, sélectionner une méthode…)
5. Exportation du diagramme de classe (non encore implémentée)

Diagramme de séquence (+20 actions)

1. Sélection de la classe et de la méthode à visualiser
2. Navigation et zoom sur le diagramme
3. Navigation dans les séquences UML
4. Interactions sur le diagramme (sélectionner une séquence, un appel parent…)
5. Exportation du diagramme de séquence en PNG/SVG
6. Affichage en mode « Code Bubbles »

Interactions dans l'EDI (+3 actions)

1. Ouverture/fermeture de fichiers code source
2. Changement d'onglet
3. Navigation dans un fichier (par clic ou clavier)

« Landing page » (4 actions)

1. Ouverture du diagramme de classe
2. Ouverture du diagramme de séquence
3. Ouverture de la télécommande
4. Informations sur le projet (nom de la session, URL WSE, nom du projet)

Palette contextuelle (+8 actions)

1. Créer un élément (classe, énumération, interface, paquetage)
2. Créer une classe mère
3. Déplacer dans un package
4. Supprimer un ou plusieurs éléments (de façon sécurisée ou forcée)

3.2 Architecture

L'architecture de VisUML se compose de deux parties qui communiquent ensemble grâce à un bus de données, nommé Web Server Event (WSE), présenté dans la section 3.2.3 page 43. Ces deux parties se concentrent chacune sur un aspect de l'outil :

1. Analyse du code et envoi des informations
2. Représentation graphique de ces informations

Chaque partie contient un ensemble d'applications détaillées respectivement sections 3.2.1 page ci-contre et 3.2.2 page 39.

La figure 3.2 représente les liens entre ces parties et les différentes applications.

L'approche suivie pour VisUML a été de répartir les fonctionnalités sur différentes applications plutôt que de créer un nouvel outil les regroupant toutes. Le but de cette démarche est de s'adapter aux habitudes des développeurs tout en limitant au maximum le nombre d'actions nécessaires à l'incorporation de VisUML dans leur écosystème. En effet, l'outil étant basé sur un plugin pour EDI, il n'oblige pas les développeurs à changer d'outil ; il suffit d'y ajouter cette extension pour profiter, à la demande, des fonctionnalités de VisUML. Cette approche se base sur la vision de Michel R.V. Chaudron [32], et la notion de « *next generation software design environment* ».

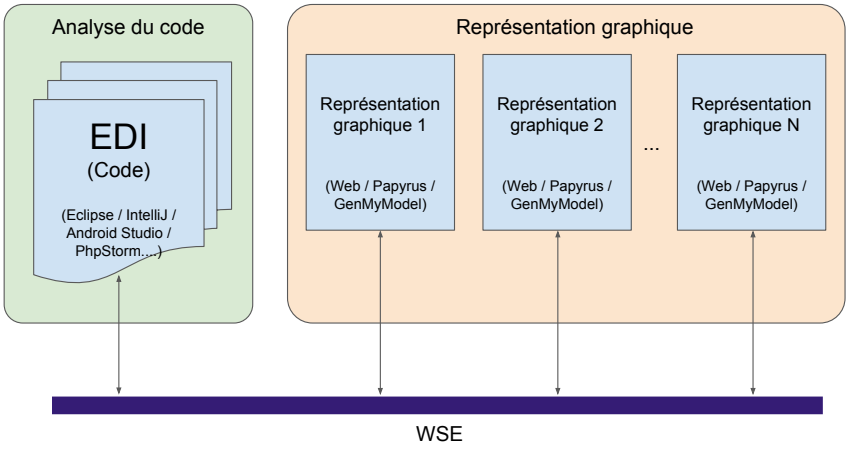


FIGURE 3.2 – Schéma de la répartition des outils de VisUML

3.2.1 Plugins pour les environnements de développement intégré

La partie de VisUML qui analyse le code fonctionne via un plugin pour EDI. Nous avons implémenté ce plugin pour deux types d'EDI : Eclipse [2] et IntelliJ [3]. Le plugin IntelliJ fonctionne à la fois sur IntelliJ IDEA et sur Android Studio [4]. Comme expliqué précédemment, l'intérêt de gérer ces deux grands EDI, ainsi que Android Studio qui est le seul EDI pour Android supporté officiellement par Google, est de s'adapter aux habitudes des développeurs afin de faciliter au maximum l'adoption de cet outil.

Le plugin est lui-même divisé en plusieurs packages. Certains sont communs aux deux EDI, d'autres sont spécifiques aux langages analysés (cf. figure 3.3).

La partie commune aux deux EDI contient les Entités correspondantes aux représentations des objets (au sens Programmation Orienté Objet) du projet analysé : classe, attributs, méthodes... Ces objets sont créés par l'analyseur de code, détaillé plus loin, et envoyés aux représentations graphiques, via le bus de communication WSE. La figure 3.9 page 45 montre le diagramme de classe de ces entités. Le détail du contenu de ces entités peut être trouvé en annexe A.2, figure A.2. La sous-section 3.3 détaille la structure de ces entités.

Cette partie commune contient également la gestion de la communication avec le bus de données, ainsi que le traitement des messages reçus et envoyés.

2. https ://www.eclipse.org
3. `https://www.jetbrains.com/idea/`
4. `https://developer.android.com/studio/index.html`

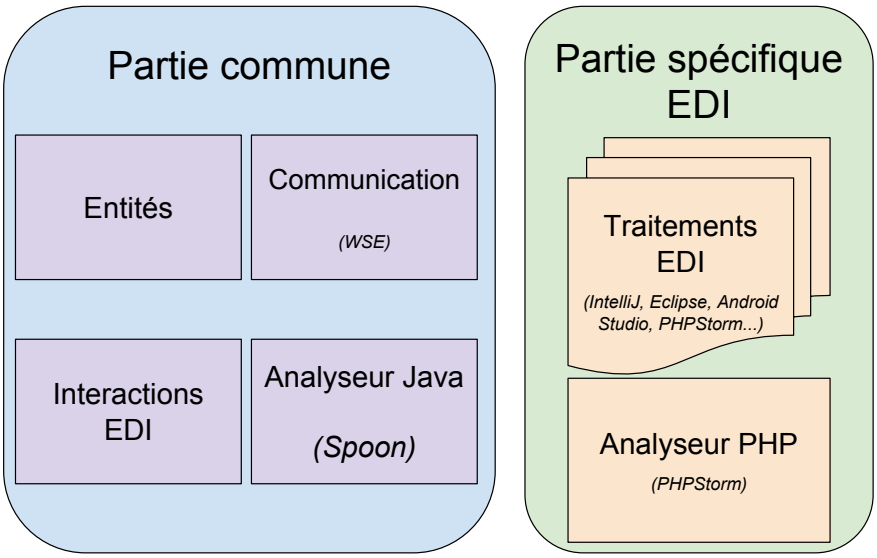


FIGURE 3.3 – Architecture des plugins de VisUML

Ces traitements pouvant nécessiter une action spécifique sur l'EDI, une interface (nommée *IDEUtils*) est utilisée et implémentée pour chaque EDI supporté, afin de généraliser un maximum l'implémentation. Cette interface contient des méthodes permettant par exemple de changer l'onglet actif, de naviguer dans le fichier vers une ligne précise, de mettre en valeur un morceau du code, etc.

La partie spécifique à l'EDI est l'interface graphique. Bien que l'interface de VisUML ne soit pas très développée (partie plugin), elle permet de lancer le plugin et affiche quelques informations, principalement pour du « debug », ainsi qu'un ou plusieurs boutons (selon l'EDI) pour ouvrir les représentations graphiques. La figure 3.4 montre cette interface sur IntelliJ.

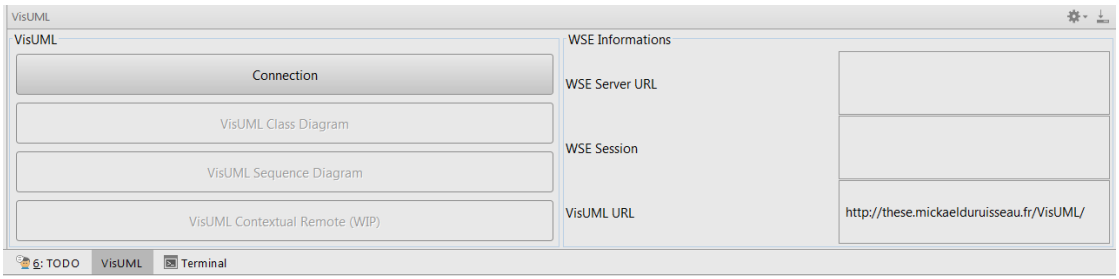


FIGURE 3.4 – Interface du plugin VisUML

Il existe également une partie **Analyse du code**, qui diffère en fonction du

code analysé. Actuellement deux langages sont supportés : Java et PHP. Ce dernier est supporté partiellement sur IntelliJ et PhPStorm [5]. L'analyseur de code Java utilise la bibliothèque Spoon [6] pour créer les entités présentées précédemment. Spoon fonctionne en analysant les fichiers source Java du projet ; il construit un arbre syntaxique abstrait (AST) des éléments et de leurs relations qui est facilement interrogeable avec les méthodes et accesseurs fournis par Spoon. Cet arbre est donc parcouru par des *processeurs* (au sens Spoon), et les données intéressantes pour VisUML sont extraites et enregistrées dans les entités. Ces entités sont par la suite envoyées sur le bus WSE. Spoon étant une librairie Java, son fonctionnement est identique sur IntelliJ et sur Eclipse ; cet analyseur de code est donc commun aux deux plugins (« Analyseur Java » sur la figure 3.3 page précédente).

En revanche, l'analyseur de code PHP utilise des fonctionnalités fournies par JetBrains dans PhpStorm et ne fonctionne donc que pour cet EDI (ainsi que pour IntelliJ avec le plugin PHP installé, partie « Analyseur PHP » sur la figure 3.3 page ci-contre). Cependant, l'analyseur crée le même type d'entités VisUML et donc le reste du plugin fonctionne exactement de la même façon.

3.2.2 Représentations graphiques proposées

VisUML fournit des représentations graphiques des projets analysés. Elles sont implémentées sur deux supports : en pages web (HyperText Markup Language (HTML) et JavaScript (JS)) et sur Papyrus [7]. Toutes les fonctionnalités présentées par la suite, dans ce chapitre et les suivants, fonctionnent à la fois sur le web et sur Papyrus (sauf mention contraire). De la même façon, un connecteur à GenMyModel [8], qui est un modeleur UML sur le web, a été développé. L'outil est construit de façon à ce que n'importe quel outil se connectant sur le bus avec les bonnes données de session (voir section 3.2.3 page 42) puissent afficher sa propre représentation graphique (ou autre) associée au code.

Les pages web utilisent la librairie graphique GoJS [9] pour afficher les diagrammes. GoJS est une librairie JavaScript permettant de générer des graphiques interactifs. Elle fonctionne avec un système de *Node* (noeud, élément) et de *Link* (lien, relation) qui constituent un modèle de données. Pour remplir ce modèle, les représentations graphiques écoutent les messages de type *createOrUpdateUML*, présenté section 3.4 page 50, qui contiennent toutes les informations d'une *Entity* (présentées dans la partie 3.3 page 44). Le modèle est ensuite affiché graphique-

5. https://www.jetbrains.com/phpstorm/
6. http://spoon.gforge.inria.fr/
7. Papyrus : https ://www.eclipse.org/papyrus/
8. GenMyModel : https ://genmymodel.com/fr
9. GoJS : https ://gojs.net/latest/index.html

ment grâce aux différents *templates* (GoJS) définis dans l'application. Grâce à ces derniers, des liens sont créés entre les données et leurs représentations graphiques. Les *templates* contiennent également une gestion des événements. Ainsi, il est possible de définir un comportement spécifique lors d'un clic, un double clic, un passage de souris, etc., en ayant les données associées à l'élément sur lequel l'utilisateur a interagi. Ces événements déclenchent l'envoi de messages possédant chacun un type spécifique, selon l'élément. Par exemple, un clic sur un attribut va déclencher l'envoi d'un message de type *highlightAttribute* [10]. Le fonctionnement de GoJS est illustré figure 3.5. GoJS fonctionne à la fois sur PC et sur tablette ou téléphone, les événements tactiles sont interprétés et gérés comme des clics, ou peuvent avoir un comportement différent ; pour VisUML, un *touch* aura le même comportement qu'un clic.

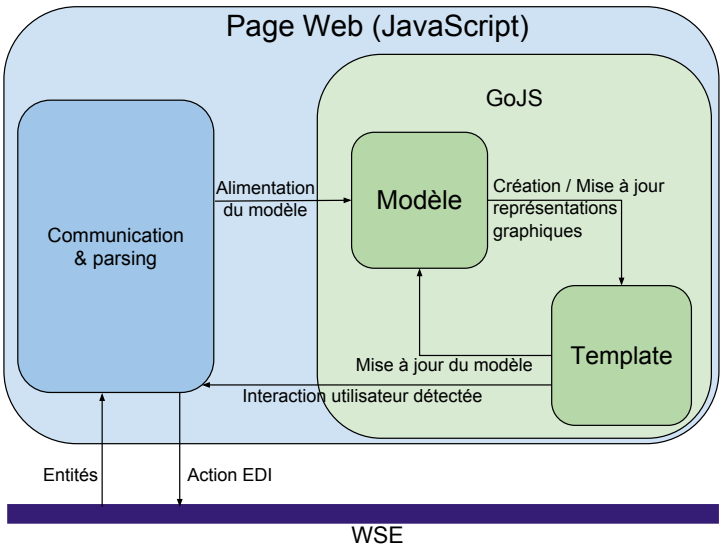


FIGURE 3.5 – Fonctionnement de GoJS dans VisUML

Le diagramme de classe affiche une représentation graphique des éléments *Entity* sous la forme de *classifier* UML (un rectangle, avec une partie nom de l'élément, une partie attributs et une partie méthodes, les classes internes ne sont pas affichées dans cette représentation). Cette vue est basée sur UML mais n'a pas pour vocation de respecter totalement le standard. Ce choix est expliqué dans le chapitre 4 page 63.

La vue diagramme de séquence se comporte de la même façon, et affiche une représentation d'une méthode (*Method*) sous forme de séquences, en utilisant les blocs (*Block*) contenant l'exécution complète de la méthode. De même que pour

10. plus d'informations sur les messages dans la section 3.4 page 50 et sur les fonctionnalités de l'outil dans le chapitre 4 page 63

le diagramme de classe, nous étendons le standard UML par différent artifices graphiques. Ces représentations sont détaillées dans le chapitre suivant.

Ces vues étant implémentées sur des pages web, il est possible de les ouvrir sur n'importe quel type d'appareil. Comme dit précédemment, GoJS gère les interactions tactiles et est elle-même gérée sur tous les navigateurs web récents, même mobiles. Il est également possible d'ouvrir autant de représentations graphiques que souhaité, permettant par exemple l'affichage de deux diagrammes de séquences possédant des options différentes, ou plusieurs diagrammes de classes utilisant des filtres différents. Les paramètres des applications sont également présentés au chapitre suivant. La figure 3.6 montre plusieurs dispositions possibles.

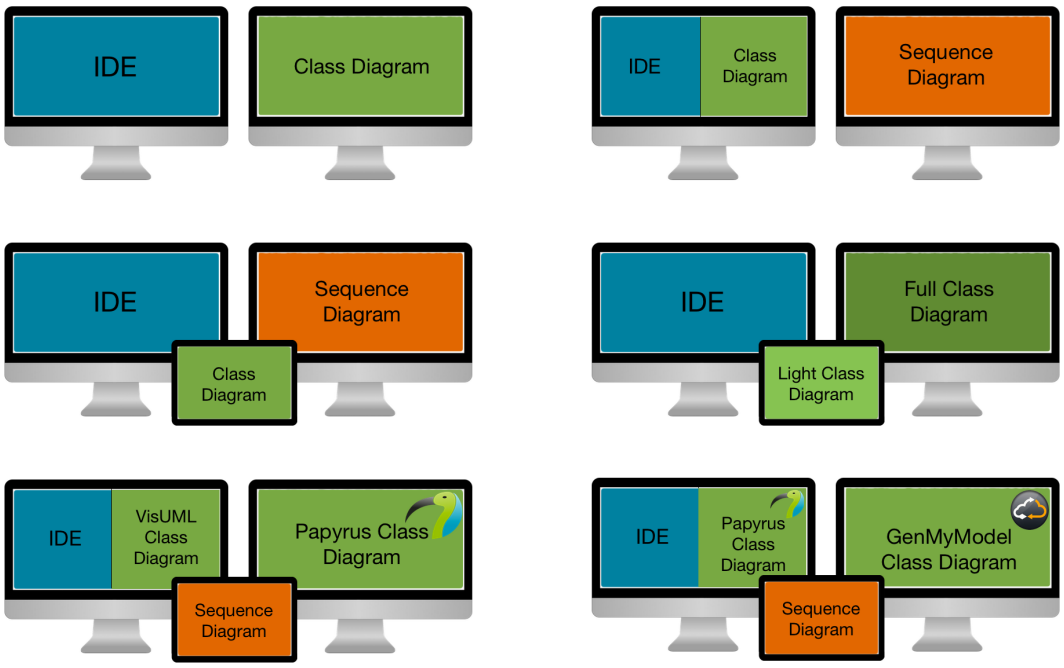


Figure 3.6 – Exemples de contextes de travail avec VisUML

Le plugin Papyrus permet d'obtenir un diagramme de classe effectuant la majorité des actions implémentées sur les pages web, mais directement sur le modeleur UML. Certaines fonctionnalités comme les filtres ou les tags, présentées ultérieurement, ne sont pas disponibles sur Papyrus, mais à l'inverse ce dernier permet une validation formelle de modèle généré, ainsi que son export sous différents formats. La figure 3.7 page suivante montre un diagramme de classe généré automatiquement dans Papyrus.

De même, l'implémentation partielle de VisUML sur GenMyModel nous

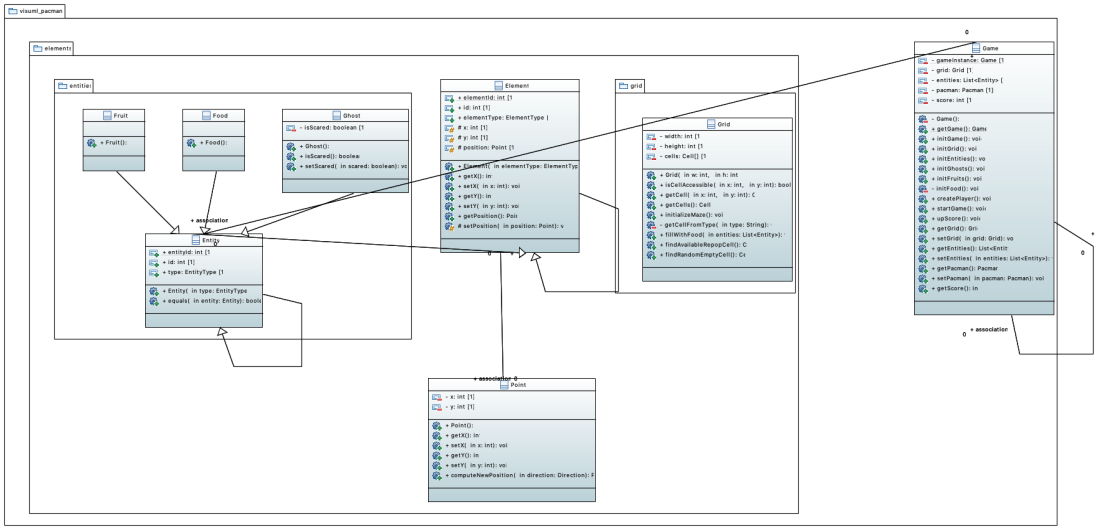


FIGURE 3.7 – VisUML dans Papyrus

permet de bénéficier des fonctions de cet outil. Les données envoyées par VisUML sur GenMyModel permettent de créer un modèle des éléments ouverts et associés, ainsi qu'un diagramme de classe par défaut correspondant à la vue web de VisUML. Les éléments du modèle sont mis à jour en temps réel et peuvent être utilisés pour produire tout type de diagramme proposé par GenMyModel.

3.2.3 Bus de communication

Pour faire communiquer les différents outils, VisUML utilise un bus de données nommé WSE, présenté ci-après. Ce bus permet de séparer les différentes parties de VisUML en fonction de leurs rôles ; ainsi l'analyse de code est faite sur l'EDI tandis que les représentations graphiques peuvent être portées sur d'autres plateformes. Comme vu précédemment, il existe des représentations graphiques sur le web et via un plugin Papyrus. Ces deux supports utilisent WSE pour récupérer les données des entités (*Entity*) et envoyer des messages suite aux interactions de l'utilisateur. Ce bus utilise un mécanisme de sessions, ou *room*, afin de séparer les différents utilisateurs. De plus, les plugins VisUML se connectent toujours sur deux sessions : l'une utilisée pour les interactions utilisateur, et l'autre pour les données du projet. Ce système permet la mise en place d'une certaine forme de collaboration : si plusieurs utilisateurs utilisent VisUML avec un projet identique, leurs données seront partagées, mais chacun pourra interagir avec sa représentation graphique, sans conflit avec les autres. Le travail collaboratif étant un domaine de recherche complexe et à la marge de

nos objectifs initiaux, cet aspect de VisUML n'a pas été développé [11]. Le nom de la session d'interaction est généré aléatoirement et de façon unique, afin d'éviter que les interactions d'une personne soient reçues par une autre. En revanche, la session contenant les données du projet a pour identifiant le nom du projet, celui-ci est récupéré grâce aux données fournies par l'EDI. La figure 3.8 montre deux diagrammes de classe générés par VisUML, le projet est commun aux deux, mais chaque utilisateur interagit avec ce dernier comme il le souhaite (déplacement, filtres,...).

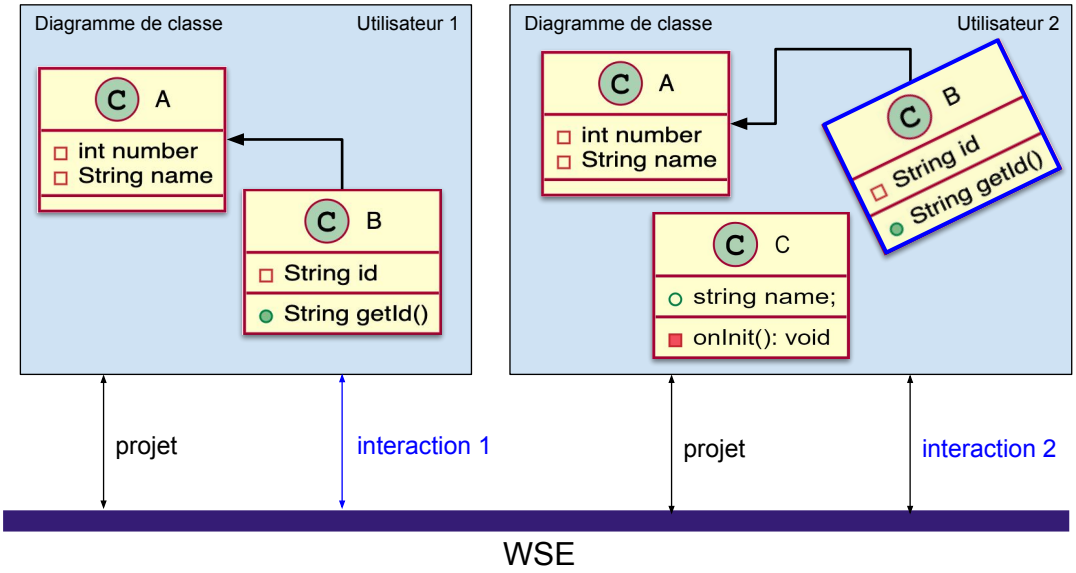


FIGURE 3.8 – VisUML sur un projet pour deux utilisateurs

Les messages utilisés dans VisUML sont présentés dans la section 3.4 page 50.

WSE [12] est un bus de données qui a été développé initialement par LE PALLEC et al. pour faire communiquer différentes applications, tel que présenté dans « A support to multi-devices web application » [30]. Il permet d'envoyer des messages au format JSON aux différentes applications connectées, et ce quel que soit le système d'exploitation.

Ce bus utilisait initialement la méthode *long pulling* [13] pour gérer les messages, introduisant ainsi une certaine latence dans leur traitement. Nos proto-

11. Nous avons cependant développé un prototype de connecteur à GenMyModel afin de bénéficier de ses fonctions collaboratives, sans avoir à en gérer tous les aspects.
12. `http://www.lifl.fr/miny/index.php?page=index`
13. `https://en.wikipedia.org/wiki/Push_technology#Long_polling`

types nécessitant un temps de réponse très court, nous avons décidé d'implémenter une version de WSE en utilisant Node.js [14] et socket.io [15].

Node.js est un « environnement d'exécution JavaScript construit sur le moteur JavaScript V8 de Chrome. Node.js utilise un modèle basé sur l'événementiel et des entrées/sorties non bloquantes, ce qui le rend léger et efficace. » [16].

Socket.io est un framework JavaScript, qui permet la communication bidirectionnelle en temps réel entre un client et un serveur. Grâce à un mécanisme de « *room* » *(« salles »)*, il permet un « broadcast » sélectif aux utilisateurs d'une même « salle ». Enfin, il utilise le protocole *WebSocket* [17] par défaut. Il est également compatible avec d'autres protocoles (HTTP Long Polling par exemple), et s'adapte automatiquement aux clients.

3.3 Structure des entités VisUML

Les entités utilisées dans VisUML servent à représenter des éléments très variés, du plus petit comme une variable, un attribut, une méthode voire une ligne de code jusqu'à des éléments plus complexes comme une classe, une interface ou des relations entre classes et interfaces. Ces entités servent également à créer les représentations UML et contiennent donc les données requises pour un affichage correct, pour les différents types de diagramme (pour VisUML, ceux de classe et de séquence). Cependant, les entités sont facilement extensibles et il existe déjà un objet spécifique qui contient des informations telles que le nombre de lignes (dans le cas d'une classe par exemple), la date de dernière modification, l'auteur, la JavaDoc, etc. Ces informations, actuellement non visibles sur les diagrammes, peuvent servir à filtrer les éléments, à les trier ou peuvent simplement être affichées.

Comme la figure 3.9 page suivante le montre sous forme de diagramme de classe, la classe qui contient tous les autres objets est *Entity* (en bas à gauche). Cet objet stocke les informations de base d'un élément de type classe, énumération ou interface (ce qui équivaut au type *Classifier* d'UML) :

— Nom
— Package
— Fully Qualified Name (FQN)
— Type (classe, énum, interface, ..),
— *Flags* (static, final, abstract)
— Visibilité (public, private, protected)

14. `https://nodejs.org/`
15. `https://socket.io/`
16. Citation tirée de `https://nodejs.org/fr/`
17. `https://en.wikipedia.org/wiki/WebSocket`

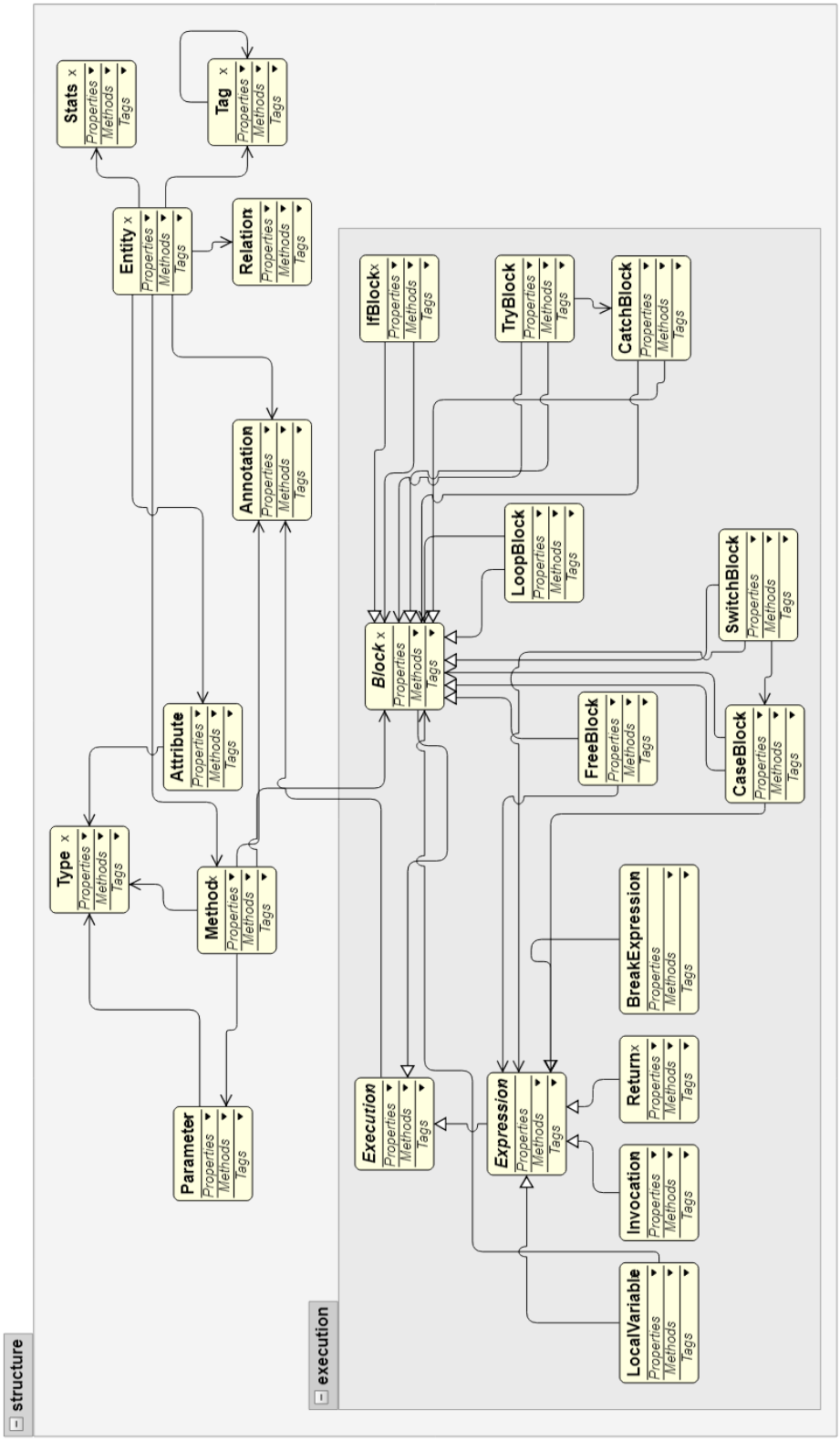


FIGURE 3.9 – Diagramme de classe des entités de VisUML

En plus de ces informations, une *Entity* possède plusieurs listes d'objets : une liste d'attributs (type *Attribute*), de méthodes (type *Method*), d'annotations (*Annotation*) et une liste de relations (*Relation*). Ces différents types sont présentés ci-après.

En complément de ces types, il en existe un autre, non utilisé par *Entity*, la classe *Type*, qui sert à représenter un type (au sens objet, par exemple : *Integer*, *String*...) et ne contient que des informations simples sur celui-ci (comparé à *Entity*) : son nom et son FQN. Ces informations permettent de faire des liens rapides avec d'éventuelles *Entity* ou d'autres *Type*, sans stocker toutes leurs informations. Le code 3.1 met en évidence l'utilisation de cette classe *Type*.

```
{ // Attribute -> private Integer monAttribut;
    "name": "monAttribut",
    "type": { // Type
        "name": "Integer",
        "fqn": "java.lang.Integer"
     },
     // ...
}

{ // Attribute -> private Element monElement;
    "name": "monElement",
    "type": { // Type
        "name": "Element",
        "fqn": "fr.exemple.structures.Element"
     },
     // ...
}
```

Code 3.1 – Structure d'un attribut

Les objets de type *Attribute* possèdent un nom, un type (*Type*) ainsi que des informations sur leurs visibilités et leurs *flags* (*static*, *synchronized*, *volatile*...). De plus, la valeur initiale de l'attribut est stockée si celle-ci est définie dans le code. Enfin les commentaires associés aux attributs sont récupérés et stockés sous forme de chaînes de caractères, et il en est de même dans le cas où une JavaDoc est présente.

Les objets de type *Method* contiennent également ces informations, mais possèdent aussi un booléen *isConstructor* permettant d'indiquer si cette méthode est un constructeur ou non. De même, ils possèdent une liste de paramètres (de type *Parameter*, qui eux-mêmes ont le nom et le type (*Type*) d'un paramètre). Enfin,

un objet *Method* enregistre aussi une liste d'objets de type *Block*, permettant de représenter l'exécution de celle-ci. Ces blocs sont présentés plus loin dans cette partie.

Les objets *Annotation* contiennent le nom de l'annotation, ainsi qu'une liste de couples <nom, valeur> pour les valeurs de cette annotation. Ces objets *Annotation* sont toujours liés à un *Attribute*, une *Method* ou une *Entity*.

Enfin, le dernier type d'objets utilisé par *Entity* est *Relation*. Ce dernier représente n'importe quel type de lien entre deux *Entity*, ou entre une *Entity* et un *Type*. Il contient le nom et FQN de l'élément de départ, ainsi que ceux de celui de destination (que ce soit une *Entity* ou un *Type*, ces deux informations étant stockées par les deux objets). Il possède un type, sous forme de chaîne de caractères, par exemple *généralisation* ou *association*, ainsi que des cardinalités. Cette représentation d'une relation reste très basique comparé aux relations dans UML, mais elle est suffisante pour l'utilisation actuelle de VisUML. Les relations *association* et *composition* sont fusionnées dans notre outil car il n'est pas possible d'inférer cette informations à partir du code. Rien n'empêche une évolution future de ce concept.

En complément de ces classes, un ensemble de concepts permet de représenter le contenu des méthodes du code, et est utilisé pour créer le diagramme de séquence d'une méthode. Les relations entre ces différentes classes sont visibles sur la figure 3.9 page 45, partie droite. La classe mère principale, dont héritent toutes les autres, est *Execution*. Elle ne contient que des informations générales :

- Type, défini par chaque sous-classe avec son propre nom (ex : *invocation*, *localVariable*, *alt*, *loop*...)
- Numéro de ligne, où se situe ce morceau de code
- Liste d'annotations (type *Annotation*)
- Liste de commentaires (chaînes de caractères)
- Chaîne de caractères représentant cette exécution (le code réel)

Le reste des éléments est ensuite divisé en deux sous-ensembles : les blocs (*Block*) et les expressions (*Expression*). Un bloc représente un groupe d'exécution (par exemple, ensemble d'instructions d'une boucle), tandis que *Expression* représente un appel unique dans le code, par exemple une déclaration de variable.

Blocs Il existe donc un type héritant de *Block* pour chaque structure de contrôle, ici en Java :

- **IfBlock** contenant une condition et deux listes de *Block*, une possédant les blocs exécutés si la condition est remplie, l'autre contenant ceux exécutés sinon.
- **LoopBlock** possèdant un *loopType* (égal à *for*, *foreach*, *do* ou *while*), une condition d'arrêt et une liste de *Block*.
- **TryBlock** contient une liste de *Block* exécutés dans la partie « try », une liste de *CatchBlock* ainsi qu'une liste de *Block* exécutés dans la partie « finally ».
- **CatchBlock** possède une liste des exceptions gérées, ainsi qu'une liste de *Block* exécutés si l'une des exceptions est interceptée.
- **SwitchBlock** possède un sélecteur, ainsi qu'un ensemble de *CaseBlock*.
- **CaseBlock** représente un *case* d'un switch. Il contient la condition et un ensemble de *Block*
- **FreeBlock** représente une liste d'*Expression*, c'est un ensemble sans condition particulière.

Expression *Expression* représente les éléments basiques du code :

- **Invocation** représente un appel de méthode. Il contient le nom de la méthode appelée, le type de retour de celle-ci, ainsi que la liste des paramètres (nom, type et valeur). Cet objet possède également des informations sur la variable ou l'objet faisant l'appel, son nom et son type, ou uniquement son type dans le cas d'un appel statique.
- **LocalVariable** représente une déclaration de nouvelle variable : il possède des informations sur son type, son nom et sa valeur, ou une *Invocation* appelée lors de sa création.
- **Return** représente un *return* dans le code. Les informations stockées sont le type de retour et la valeur ou l'*Invocation* utilisée.

Grâce à ces éléments et leurs relations, VisUML peut générer un diagramme de séquence de n'importe quelle méthode du code analysé, tout en connaissant la ligne associée, les commentaires, les éléments parents et suivants, etc.

En complément de ces informations, une partie spécifique à Android a également été implémentée. Celle-ci stocke des informations relatives au développement Android dans le cas où l'*Entity* en possède. Par exemple, dans le cas où une classe hérite de *Activity*, l'analyseur de code va chercher le fichier Extensible Markup Language (XML) de « Layout » associé, les widgets définis dans le code, ou encore les *Intent* et autres moyens de navigation. Ces informations sont ajoutées à l'entité associée et envoyées sur le bus. Bien que pour le moment, aucune représentation graphique n'utilise ces données, elles pourraient permettre de faire un diagramme des différents écrans Android et de leurs relations. Cette fonctionnalité a d'ailleurs été implémentée récemment dans Android Studio

au travers d'un *Navigation Graph* représentant les liens entre les écrans d'une l'application. Ce graphique est visible sur la figure 3.10 [18].

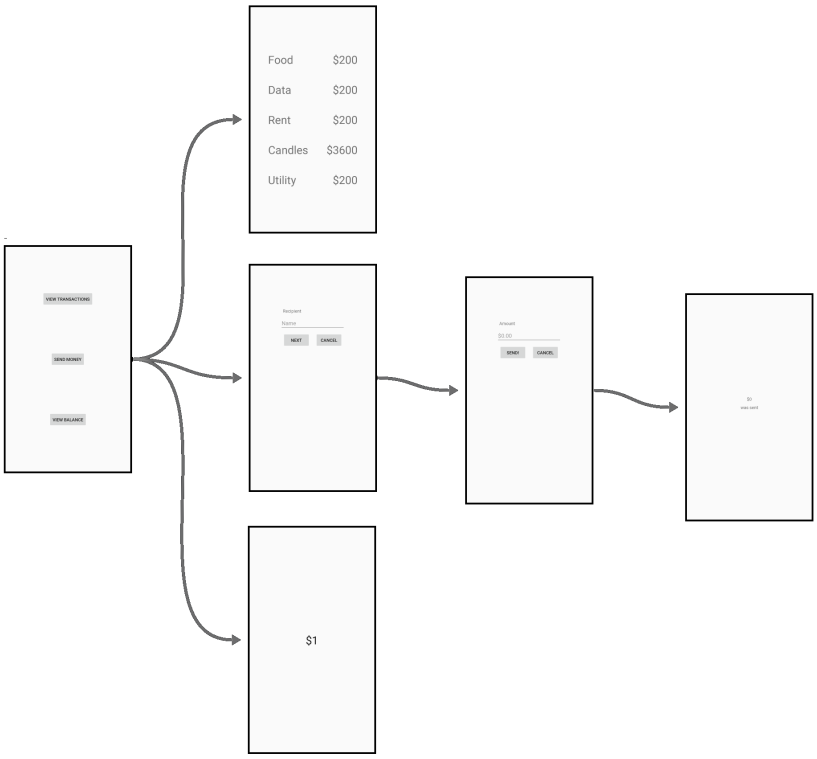


Figure 3.10 – Android Studio : Diagramme de navigation d'une application

Enfin, un support partiel de Sonar [19] a également été développé. Sonar est un outil permettant d'analyser un projet afin d'obtenir des informations sur la qualité du code. Une fois l'analyse terminée, les données sont accessibles via une Application Programming Interface (API). Le plugin VisUML ne récupère pas directement les données, mais fournit les informations nécessaires aux représentations graphiques pour effectuer des requêtes sur l'API. Sonar utilise un fichier *.properties* contenant ces informations, VisUML analyse donc ce fichier, s'il est présent, et ajoute un attribut Sonar aux *Entity*. Sonar est un exemple de données externes que l'on peut associer grâce à VisUML.

Pour résumer, rien dans l'architecture n'empêche d'ajouter autant d'informations que souhaité. Cependant, les représentations graphiques actuelles ne

18. Android Studio - diagramme de navigation : `https://developer.android.com/topic/libraries/architecture/navigation/navigation-implementing`
19. Sonar : `https://www.sonarqube.org/`

gèrent qu'un sous-ensemble de ces données. De nombreuses améliorations sont possibles pour utiliser ces informations (voir chapitre 6.6 page 143).

Tous ces objets sont par la suite transformés au format JavaScript Object Notation (JSON). Cependant, les seuls objets réellement envoyés sur le bus sont ceux de type *Entity*, puisqu'ils contiennent eux-mêmes tous les autres concepts. Ces messages JSON de VisUML sont détaillés dans la section 3.4.

3.4 Structure des messages

Les différentes parties de VisUML communiquent grâce à des messages au format JSON qui passent par le bus de données présenté dans la section 3.2.3 page 42. Ces messages possèdent tous une structure commune. Le morceau de code 3.2 est un exemple de cette base.

```
1 {
2     "application": "VisUML",
3     "applicationPage": "classDiagram",
4     "uniqueID": "AA-BB-CC-DD-EE-FF",
5     "action": "exampleAction",
6     "version": "0.3.8"
7 }
```

Code 3.2 – Base d'un message de VisUML

Cette structure de base permet de rapidement filtrer les messages sur des informations simples et identifiables : l'application (ainsi que la page dans le cas des représentations graphiques) émettant le message, l'identifiant unique de l'émetteur ainsi que l'action à effectuer. Le numéro de version peut aussi servir à gérer plusieurs versions de l'outil, dans le cas où il y a eu un changement important sur le plugin et qu'il n'est pas encore à jour partout. L'*action* est également un identifiant unique correspondant à une tâche à effectuer sur l'EDI ou la représentation graphique (la ou les applications qui reçoivent le message font l'action). Il existe quatre catégories et treize types de messages dans VisUML.

Messages envoyés par l'EDI

- **me** : Informations (identifiant unique, version et nom du projet) concernant l'EDI envoyées aux applications
- **createOrUpdateUML** : Indique à la représentation graphique la création ou la mise à jour d'une *Entity*
- **highlightClass** : Demande à la représentation graphique de mettre en avant une *Entity*

— **removeClass** : Demande à la visualisation de supprimer une *Entity*

Message envoyé depuis les représentations graphiques

— **started** : Informations (type de représentation et version) concernant la représentation graphique

Messages envoyés depuis le diagramme de classe

— **switchToClass** : Demande à l'EDI de mettre en avant l'*Entity*
— **higlightAttribute** : Demande à l'EDI de montrer et sélectionner l'*Attribute*
— **highlightMethod** : Idem pour une *Method*. Le diagramme de séquence se met à jour avec la *Method* envoyée
— **closeElement** : Demande à l'EDI de fermer cette *Entity*

Messages envoyés depuis le diagramme de séquence

— **switchToLifeline** : Demande à l'EDI de montrer et sélectionner la variable sélectionnée.
— **switchToMethod** : Idem pour une *Invocation*.
— **switchToGroup** : Idem pour un *Block* (if/else, boucle, ..)
— **switchToLine** : Idem pour une ligne spécifique

3.4.1 Objets JSON des entités VisUML

Chaque entité de VisUML possède une structure JSON contenant la plupart de ses informations. La plus grande est *Entity*, qui contient elle-même un ou plusieurs objets JSON correspondant aux autres entités. Le code 3.3 est un exemple d'une entité en JSON. Cet objet contient toutes les informations nécessaires pour recréer cet élément sur la représentation graphique. Il est inclus dans plusieurs messages. Le détail de la formation des messages est décrit dans la section 3.4.2 page 59.

Les sous-sections suivantes présentent les objets JSON correspondant aux entités présentées précédemment. Certains champs finissent par un *O*, qui correspond à *Objet*, indiquant que ces champs contiennent un objet et non une valeur.

3.4.1.1 Structure d'une classe

Dans le code 3.3 page suivante, les rubriques « superClass » et « superClassO », ainsi que « interfaces » et « interfacesO », peuvent sembler redondantes. L'explication est que « superClass » et « interfaces » ont été créés initialement et

ont été conservés pour des raisons de compatibilité avec nos prototypes précédents. Les réalisations récentes et futures doivent utiliser les versions « O », qui fournissent plus d'informations.

```
{
    // Heritage et implementation
    "superClass":"MaClasseMere",
    "superClassO":{  // Objet Type, cf code  3.6 page 55
        "name":"MaClasseMere",
        "location":"x.y.z.MaClasseMere"
    },
    "interfaces":[
        // Liste des interfaces implementees sous forme de
            haine de caracteres
    ],
    "interfacesO":[
        // Liste des interfaces implementees
        // Objets Type, cf code  3.6 page 55
    ],

    // Partie flags
    "isStatic":false / true,
    "isAbstract":false / true,
    "isFinal":false / true,
    "visibility":"public", // "private", "protected", "
      package"

    // Informations de base
    "type":"class", // "interface", "enumeration"
    "project":"nom_du_projet",
    "packageName":"nom_du_package",
    "name":"nom_de_la_classe",
    "location":"nom_du_package.nom_de_la_classe", // FQN

    // Autres concepts inclus dans l'entite
    "attributes":[
        // Objets Attribute, cf code  3.4 page ci-contre
    ],
    "methods":[
        // Objets Method, cf code  3.5 page 54
    ],
```

```
    "annotations":[
        // Objets Annotation, cf code  3.9 page 56
    ],
    "relations":[
        // Objets Relation, cf code  3.7 page 56
    ],
    "android":{
        // Objet Android, cf code  3.10 page 57
    },

    // Informations complementaires
    "author":"Mickael",
    "javadoc":""
}
```

Code 3.3 – Structure d'une Entity de VisUML

3.4.1.2 Structure d'un attribut

Dans le code 3.4, les rubriques « type » et « typeO » contiennent des informations semblables. « type » est la version initiale, conservée pour des raisons de compatibilité avec nos prototypes précédents. Les réalisations récentes et futures doivent utiliser « typeO », qui fournissent plus d'informations.

```
{
        // Partie flags
           "isStatic":false / true,
           "isFinal":false / true,
           "visibility":"public", // "private", "protected
              ", "package"


        // Informations de base
        "type":"monTypeAttribut",
        "typeO":{
                // Objet Type, cf code  3.6 page 55

        }
        "name":"",
        "value":"",

        // Informations complementaires
```

```
18          "javadoc": "",
19          "comments":[
20                  // Chaines de caracteres
21          ],
22  }
```

Code 3.4 – Structure d'un Attribut de VisUML

3.4.1.3 Structure d'une méthode

Dans le code 3.5, les rubriques « type » et « typeO » contiennent des informations semblables. « type » est la version initiale, conservée pour des raisons de compatibilité avec nos prototypes précédents. Les réalisations récentes et futures doivent utiliser « typeO », qui fournissent plus d'informations.

La partie « execution » contient le corps de la méthode, sous la forme d'un ensemble de *Block*, comme présenté dans le code 3.11 page 58.

L'information « line » correspond au numéro de la première ligne de la méthode dans le code source.

```
1   {
2           // Partie flags
3           "visibility":"public", // "private", "protected",
              "package"
4           "isStatic":false / true,
5           "isAbstract":false / true,
6           "isFinal":false / true,
7           "isSynchronized":false / true,
8           "isConstructor":false / true,
9
10          // Informations de base
11          "type":"monTypeDeRetour",
12          "typeO":{
13                  // Objet Type, cf code  3.6 page suivante
14          },
15          "name":"createMethod",
16          "parameters":[
17  // Objets Parameter, cf code  3.8 page 56
18          ]
19
20          // Execution - Sequence de la methode
21          "execution":[
```

```
22                  // Objets Block, cf code  3.11 page 58
23          ],
24
25          // Informations complementaires
26          "javadoc":"",
27          "comments":[
28                  // Chaines de caracteres
29          ],
30          "annotations":[
31                  // Objets Annotation, cf code  3.9 page
                      suivante
32          ],
33          "line":100
34  }
```

Code 3.5 – Structure d'une Method de VisUML

3.4.1.4 Structures supplémentaires

Le code 3.6 peut représenter un type de retour d'une méthode, un type d'un attribut ou d'un paramètre d'une méthode.

```
1   {
2       // Informations de base (Pour un objet de type :
            MaListe<MonType>)
3       "simpleName": "MaListe",
4       "qualifiedName": "x.y.z.MaListe", // FQN
5
6       // Informations complementaires
7       "isArray":false / true,
8       "isList":true / false,
9
10      // Si liste : type contenu par la liste
11      "actualTypesSimpleNames":[
12          "MonType"
13      ],
14      "actualTypesQualifiedNames":[
15          "x.y.z.MonType"
16      ]
17  }
```

Code 3.6 – Structure d'un Type de VisUML

Un objet *Relation* (code 3.7) représente un lien entre deux entités (*from* et *to*). Cette relation est caractérisée par un *type* pouvant être *association*, *generalisation* (pour un héritage) ou *implementation* (pour une implémentation d'interface).

```
{
        "type":"association", // "generalisation", "
           implementation"
        "from":{
                "name":"monObjetSource",
                "location":"x.y.z.monObjetSource"
        },
        "to":{
                "name":"monObjetDestination",
                "location":"x.y.z.monObjetDestination"
        }
}
```

Code 3.7 – Structure d'une Relation de VisUML

Dans le code 3.8, les rubriques « type » et « typeO » contiennent des informations semblables. « type » est la version initiale, conservée pour des raisons de compatibilité avec nos prototypes précédents. Les réalisations récentes et futures doivent utiliser « typeO », qui fournit plus d'informations.

```
{
    "name":"monParametre",
    "type":"monType",
    "typeO":{
        // Objet Type, cf code  3.6 page précédente
     }
 }
```

Code 3.8 – Structure d'un Parameter de VisUML

Le code 3.9 représente la structure d'une annotation dans VisUML. Cette annotation possède un nom et un ensemble de valeurs. Une annotation peut être présente sur une classe, un attribut ou une méthode.

```
{
    "name":"monAnnotation",
    "values":{
        "name": "maValeur"
```

```
5       }
6   }
```

Code 3.9 – Structure d'une Annotation de VisUML

Le code 3.10 représente les informations spécifiques aux projets Android que nous récupérons depuis le code source. Ces informations ne sont pas encore affichées sur une représentation graphique.

```
1   {
2           // ID du layout
3           "xmlLayout":"R.id.layout.monLayout",
4           "intents":{
5                   // Liste des Intent de cette Activity
6           },
7           "fragments":[
8                   // Liste des Fragments utilises dans l'
                       Activity
9           ],
10          "widgets":[
11                  // Liste des vues de l'Activity
12          ],
13          // Informations sur l'eventuelle base de donnees
14          "database":{
15                  "dataBaseOpenHandler":false / true,
16                  "columns":[
17
18                  ],
19                  "name":""
20          }
21  }
```

Code 3.10 – Structure de la partie Android de VisUML

3.4.1.5 Structures des exécutions

Comme présenté dans la section 3.3 page 44, toutes les entités relatives à l'exécution d'une méthode possèdent une base commune, celle-ci est représentée en JSON par plusieurs attributs, illustrés par le code 3.11.

La propriété *line* du code 3.11 page suivante correspond au numéro de la première ligne du bloc ou de l'instruction dans le code source.

```
{
        "type":"free", // loop, alt, try, catch, ...
        "comments":[
                // Chaines de caracteres
        ],
        "annotations":[
                // Objets Annotation, cf code  3.9 page 56
        ],
        "line":89
}
```

Code 3.11 – Structure de base des entités d'exécution de VisUML

Chaque bloc contient ensuite un ou plusieurs attributs contenant un ou plusieurs blocs, chaque type de bloc étant défini à l'identique. Un exemple est donné code 3.12, montrant un objet JSON correspondant à un if/else.

```
{
    // Informations communes
    "type":"alt",
    "comments":[
        "Mon commentaire"
    ],
    "annotations":[

    ],
    "line":137,

    // Condition du groupe
    "condition":"maCondition",

    // Blocs executes si condition vraie
    "if":{
        "blocks":[
            {

                "comments":[

                ],
                "line":138,
                "annotations":[

```

```
26                  ],
27                  "type":"free",
28                  "expressions":[
29                      {
30                          "returnedTypeName":"boolean",
31                          "returnedExpression":"true",
32
33                          "comments":[
34
35                          ],
36                          "line":138,
37                          "annotations":[
38
39                          ],
40                          "returnedTypeFQN":"boolean",
41                          "type":"return"
42                      }
43                  ]
44              }
45          ]
46      },
47
48      // Blocs executes sinon
49      "else":{
50          "blocks":[
51
52          ]
53      }
54  }
```

Code 3.12 – Exemple de *Block* de type "alt"

3.4.2 Formation des messages

Les messages envoyés sont formés à partir des éléments présentés précédemment. Chaque type de message (défini par son action) a besoin d'un ou plusieurs éléments. Cette section détaille, pour chaque type, les informations requises.

— **createOrUpdateUML** : Un objet *Entity*, avec pour clé *classInfo* (voir ligne 7 du code 3.13 page suivante)
— **highlightClass** : Idem

— **highlightAttribute** : Idem, ainsi qu'un objet *Attribute* avec pour clé *selectedAttribute*
— **highlightMethod** : Idem, mais pour une *Method* avec *selectedMethod* (voir lignes 7 et 15 du code 3.14)
— **switchToXXX** : Un objet *Entity* avec pour clé *classInfo*, la ligne à mettre en valeur (clé *line*)
— **close/removeElement** : Une chaîne de caractères contenant le FQN de l'*Entity* à fermer, ayant pour clé *entity*.

```
1  {
2      "application": "IntelliJ",
3      "applicationPage": "",
4      "uniqueID": "AA-BB-CC-DD-EE-FF",
5      "action": "createOrUpdateUML",
6      "version": "0.3.8",
7      "classInfo": {
8          // Objet Entity
9          "location": "",
10         "name": "",
11         "attributes": [ ],
12         "methods": [ ]
13        // ...
14     }
15 }
```

Code 3.13 – Exemple de message *createOrUpdateUML*

```
1  {
2      "application": "VisUML",
3      "applicationPage": "classDiagram",
4      "uniqueID": "AA-BB-CC-DD-EE-FF",
5      "action": "highlightMethod",
6      "version": "0.3.8",
7      "classInfo": {
8          // Objet Entity
9          "location": "",
10         "name": "",
11         "attributes": [ ],
12         "methods": [ ]
13        // ...
14     },
15     "selectedMethod": {
```

```
16          // Objet Method
17          "name": "",
18          "type": "",
19          "parameters": [ ],
20          // ...
21      }
22  }
```

Code 3.14 – Exemple de message *highlightMethod*

3.5 Conclusion

Ce chapitre a présenté l'architecture de VisUML. Cet outil est composé de deux parties, un plugin pour EDI et des visualisations.

Le plugin est également architecturé de sorte à être facilement extensible à d'autres langages et d'autres EDI. Il est actuellement fonctionnel sur Eclipse et IntelliJ (et Android Studio), et gère le langage Java, ainsi qu'une partie de PHP.

L'analyse du langage de programmation a tout d'abord été déléguée à une bibliothèque spécifique pour Java (Spoon). Durant la dernière partie de la thèse, nous avons déporté cette analyse sur une abstraction uniforme des éléments fournies par IntelliJ pour ses outils. L'analyse du code devient alors plus simple et supporte un ensemble plus grand de langages.

Les visualisations graphiques quant à elles correspondent à des diagrammes UML légèrement modifiés. UML étant le langage de modélisation le plus utilisé par les développeurs (bien que peu présent en entreprise), a été choisi comme standard de modélisation. Deux types de diagrammes sont actuellement implémentés et fonctionnels : le diagramme de classe et le diagramme de séquence. Le premier permet d'afficher des informations concernant les éléments présents dans un projet ainsi que sur leurs relations, tandis que le second affiche un niveau de détail plus fin sur une méthode particulière et ses relations avec les autres classes.

Un troisième type de diagramme a été implémenté en tant que preuve de concept, le diagramme d'activité. Ce dernier reprend les mêmes informations que le diagramme de séquence, mais les affiche sous une forme différente. Bien que ce diagramme soit fonctionnel, la façon dont les informations sont affichées mérite un travail supplémentaire avant d'être réellement exploitable.

Les visualisations peuvent être affichées sur des pages web dédiées, développées en Javascript, ou sur des outils de modélisation. Nous avons implémenté deux plugins pour outils de modélisation, l'un sur Papyrus et l'autre sur

GenMyModel. Ces deux plugins sont encore en cours de développement mais permettent de synchroniser un EDI avec les modèles gérés par ces outils.

Ces deux parties (le plugin et les visualisations) sont liées grâce à un bus de données développé par l'équipe Carbon. Ce bus permet le transfert d'informations, sous la forme de messages, entre les applications connectées. Le plugin envoie aux visualisations les données du projet, les classes, méthodes, attributs, relations... et écoute les messages d'interaction et de navigation envoyés par les visualisations.

Lors d'une modification du code, le plugin analyse automatiquement le fichier modifié, et envoie une mise à jour aux diagrammes connectés. Ces derniers mettent alors à jour leurs visualisations afin de répliquer les changements sur la vue graphique. Il en va de même lors de la création ou de la suppression d'un fichier ou d'un package.

Chaque action possède un message spécifique, permettant aux visualisations ou au plugin de répondre correctement à une interaction. Ces messages sont stockés dans une base de données (comme présenté dans la figure 3.1 page 35, dans la partie Traces) ce qui nous a permis d'effectuer des statistiques sur l'utilisation de VisUML. C'est également grâce à cette base de données que nous avons mesuré les interactions lors de l'évaluation de l'outil.

Le chapitre suivant présente VisUML au travers de ses fonctionnalités et des choix que nous avons faits pour faciliter le quotidien des développeurs, afin de leur fournir un ensemble d'informations utiles aux bons moments.

Chapitre 4

VisUML : Un outil centré humain

Sommaire

Ce chapitre présente la vision de VisUML en termes d'interactions et de visualisations. Il présente et explique les choix faits par rapport aux différentes représentations, au contenu affiché, aux filtres ainsi que les fonctionnalités présentes dans VisUML. En revanche, ce chapitre ne présente pas les fonctionnalités de navigation, qui seront détaillées dans le chapitre suivant.

4.1 Des informations utiles pour le développeur

4.1.1 Réduire la charge cognitive en limitant les informations

La plupart des outils de rétro-ingénierie actuels permettent la création de diagrammes à partir d'un projet, en utilisant toutes les entités de ce projet. Cette fonctionnalité permet de passer du code à une représentation graphique plus abstraite, comme le diagramme de classe UML. Ce type de diagramme est particulièrement utile pour visualiser les liens entre les différents éléments du code. Il permet également d'obtenir des informations sur les dépendances et relations de ces éléments, leur appartenance à un package, etc.

Cependant, les outils actuels présentent un défaut majeur : ils utilisent l'ensemble des fichiers du projet comme source, ce qui forme un diagramme de classe pouvant contenir plusieurs dizaines voire centaines de classes, ce qui rend le diagramme illisible et donc inutilisable. Une solution partielle à ce problème implémentée dans certains outils est de forcer l'utilisateur à sélectionner les éléments qu'il veut voir. Cette solution permet d'obtenir des diagrammes de classe plus légers et plus utiles, mais elle requiert un grand travail de la part du développeur avant d'obtenir un rendu, ce dernier ayant dû au préalable réfléchir à quels éléments sélectionner, à leurs relations, etc.

Dans VisUML, nous avons voulu réduire le nombre d'interactions au minimum. Pour cela, le diagramme de classe généré par VisUML se base sur les onglets actuellement ouverts sur l'EDI du développeur. Ce choix a deux avantages majeurs : d'une part, réduire la complexité du diagramme, celui-ci contenant moins d'éléments que dans des rétro-ingénieries classiques, et d'autre part de limiter la charge cognitive de l'utilisateur, déjà lourde du fait du nombre d'interactions et d'informations affichées lors des phases de codage [33]. L'effort mental lors du passage du code au diagramme (et inversement) est donc réduit puisque les mêmes éléments sont représentés dans ces deux points de vue. La figure 4.1 représente le lien entre l'EDI et le diagramme de classe.

En plus des éléments ouverts, notre diagramme de classe affiche, de façon plus discrète, tous les éléments possédant au moins une relation (dépendance, association, héritage…) avec l'un des éléments ouverts. Cette fonctionnalité ajoute un certain nombre de classes qui pourraient gêner la lecture, mais qui sont essentielles pour appréhender la complexité d'un élément, ainsi que les liens qu'il possède avec les autres composants d'un projet. En effet, un diagramme de classe utilise les liens pour représenter l'héritage et l'implémentation. Or, dans le code, ces informations sont affichées directement dans l'élément. Pour combler ce manque d'informations, en complément de l'élément ouvert, le diagramme de classe affiche tous les éléments liés à celui-ci. Cela permet, à tout instant, d'avoir une vue globale sur un élément, son contenu ainsi que sa classe

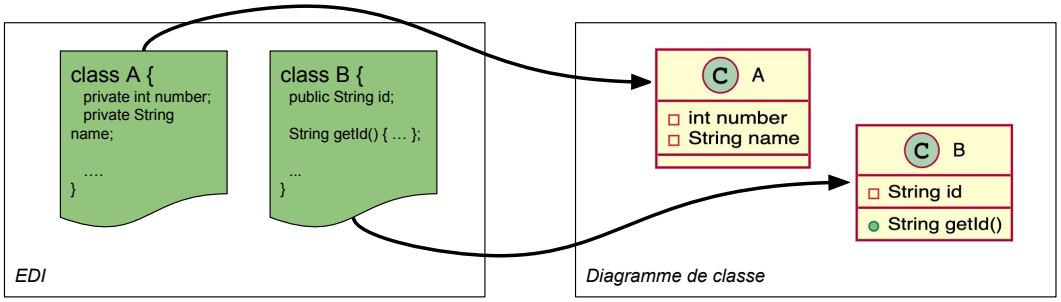
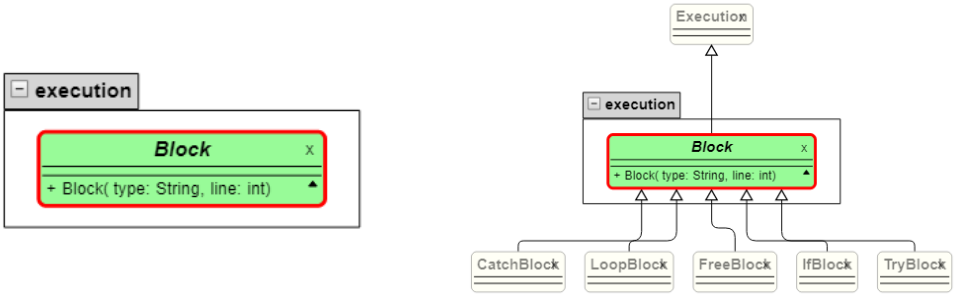


FIGURE 4.1 – Représentation schématique des liens entre EDI et diagramme de classe

mère, ses interfaces, etc. La figure 4.2 montre l'ajout d'éléments grâce à cette fonctionnalité.



(a) Uniquement l'onglet ouvert (b) Onglet ouvert et ses relations

FIGURE 4.2 – Diagrammes de classe avec et sans les éléments reliés non ouverts

Il est possible de passer d'une vue à l'autre (avec ou sans élément relié non ouvert) grâce à un filtre spécifique, présenté dans la section 4.3.2 page 75.

4.1.2 Augmentation de la sémantique des classes

Afin d'améliorer la sémantique des éléments affichés sur le diagramme, nous avons souhaité utiliser un système de mots-clés (*tags*) définis pour chaque élément d'un diagramme. Ces mots-clés doivent ajouter un sens à un élément, pour permettre une meilleure différenciation dans un ensemble. Il y a deux parties à ce travail, effectué avec une étudiante de Master 1 Informatique dans le cadre de son projet personnel.

La première partie consiste en la production des mots-clés via une analyse des fichiers sources d'un projet. Cette analyse est effectuée par le plugin VisUML, en tâche de fond et en parallèle de l'analyse du code par Spoon. A la différence de ce dernier, qui forme un Abstract Syntax Tree (AST) à partir des fichiers, cette analyse utilise tous leurs contenus comme du simple texte, avec le code, les commentaires, la javadoc, les annotations.... Afin d'obtenir des mots-clés pertinents, la première étape de l'analyse consiste à créer, pour chaque classe, un fichier temporaire dans lequel les mots les plus courants du langage Java (et de la langue française) ont été supprimés. Par exemple, les modificateurs *private*, *public*, *static*, les mots-clés Java *class*, *interface* ou certains déterminants *la*, *le*, *les*..., sont supprimés de ces fichiers car ils n'ajoutent pas de sens au fichier. De plus, la plupart des modificateurs apparaissent déjà dans les diagrammes, sous forme de signe par exemple (*private* : #, *public* : +, ...).

Une fois ces fichiers créés, il faut en extraire les mots intéressants. Nous avons testé plusieurs algorithmes pour déterminer les mots ou ensemble de mots les plus intéressants. Une approche basée sur le TF-IDF (*term frequency-inverse document frequency*, fréquence du terme - fréquence inverse du document) semble être la plus pertinente puisqu'elle permet de différencier un document au sein d'un ensemble. Les mots-clés ainsi générés sont censés mettre en avant les éléments plus présents dans un document par rapport à son ensemble.

Ce choix d'algorithme et d'analyse des fichiers fait l'objet d'un projet étudiant cette année, afin d'améliorer au maximum le sens sémantique associé.

Enfin, chaque mot-clé est associé à un sens (via l'API de Cortical [1]) pour limiter les incompréhensions et également pouvoir créer (dans le futur) des liens entre différents mots-clés ayant sensiblement le même sens. Par exemple, le mot « Avocat » peut avoir plusieurs sens, et sans lien (ou synonyme, contexte) associés, il serait impossible d'en déterminer le sens exact. Nous associons donc à chaque mot-clé un ensemble de synonymes ou termes proches. Pour « Avocat », selon le sens, ces synonymes peuvent être « fruit, nourriture, ... » ou « justice, loi, ... ».

Cette analyse est, pour le moment, effectuée uniquement sur les classes dans leur ensemble. Elle pourrait également être faite sur chaque attribut et chaque méthode, afin d'ajouter de la sémantique sur chacun des éléments. Ce travail fait partie des perspectives envisagées, voir chapitre 6.6 page 143.

Une fois cette analyse terminée, les informations des mots-clés sont envoyées aux représentations graphiques.

L'autre partie porte sur l'affichage des mots-clés sur les représentations graphiques. Dans le cadre du projet, seul le diagramme de classe implémente une visualisation de ces mots-clés, cependant ils sont accessibles par les autres repré-

1. API Cortical : `http://api.cortical.io`

sentations connectées et pourront être utilisés dans le futur par le diagramme de séquence par exemple.

Le diagramme de classe VisUML utilise le troisième bloc des classes (*Classifier UML*) pour afficher les mots-clés associés à une classe. La figure 4.3 montre un diagramme de classe avec des mots-clés.

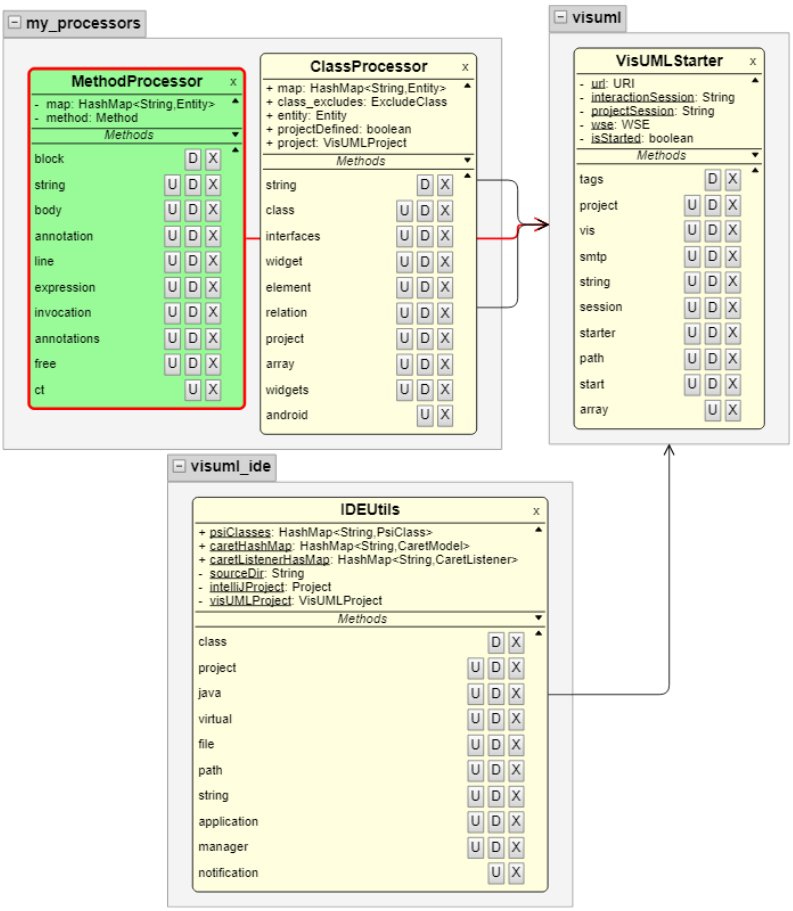


Figure 4.3 – Diagramme de classe de VisUML avec mots-clés

Sur cette figure, on peut voir la liste de tags associés à chaque classe. Il est possible de cacher un tag et de le déplacer dans la liste. Cette information est sauvegardée sur le navigateur de l'utilisateur, mais n'a pas encore d'impact sur la sémantique de l'élément. Il est également possible pour l'utilisateur de cacher une liste complète en cliquant sur la flèche ou de cacher tous les tags du projet via un filtre. Ces fonctionnalités sont présentées dans la section 4.3.3 page 76. Il est également possible d'effectuer un filtre complexe à base de requêtes via ces mots-clés, grâce à la fonctionnalité présentée dans la section 4.3.6 page 80

Les synonymes (ou termes proches) liés à chaque *tag* sont également visibles lorsque l'utilisateur laisse sa souris sur un mot-clé. Les figures 4.4 et 4.5 montrent deux exemples de mots-clés avec synonymes.

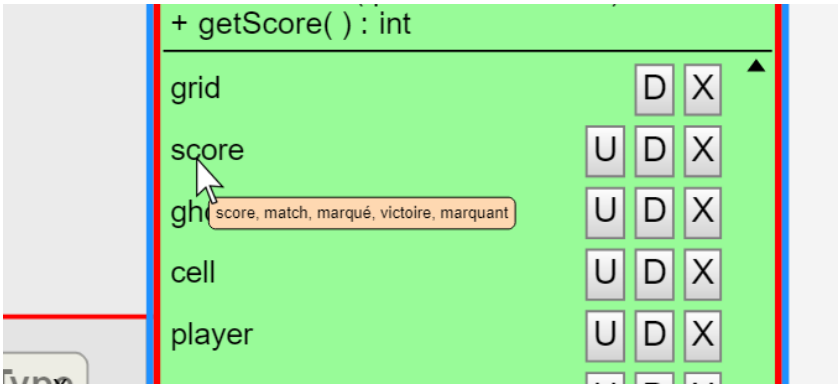


FIGURE 4.4 – Mots liés à un *tag* du diagramme de classe

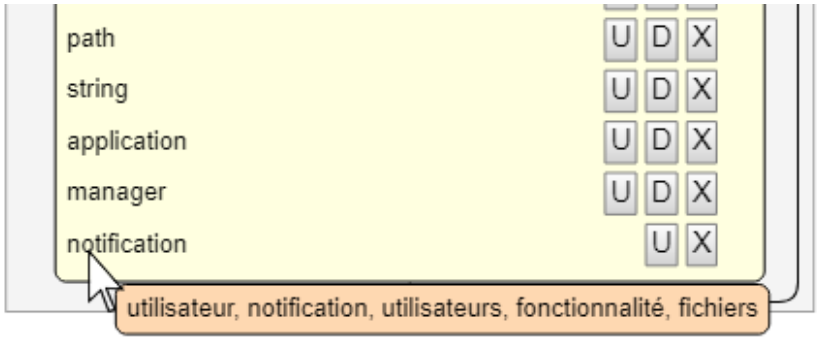


FIGURE 4.5 – Mots liés à un *tag* du diagramme de classe (2)

Ces mots liés ne peuvent pas encore être utilisés dans la représentation graphique, mais les tags et les fonctionnalités associées font partie des perspectives à court et moyen termes, présentées dans le chapitre 6.6 page 143.

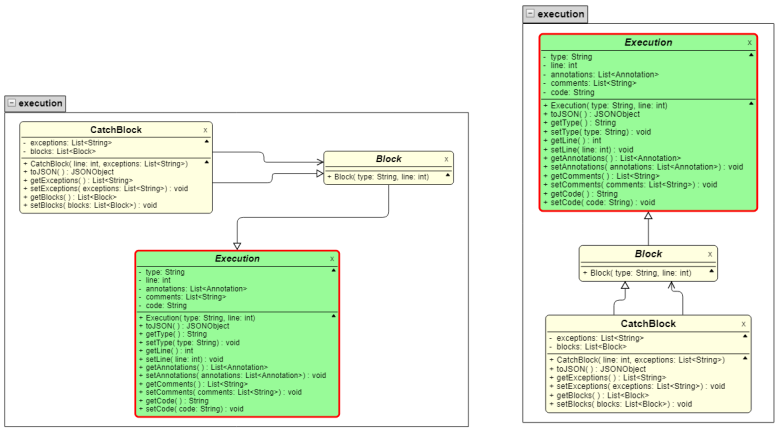
4.2 Aide à la compréhension grâce aux variables visuelles

Limiter les éléments affichés n'est pas toujours suffisant pour réduire la complexité de la représentation graphique. Dans VisUML, nous avons utilisé plusieurs variables visuelles afin de simplifier la compréhension des éléments et

de leurs relations. Cette utilisation des variables visuelles reste basique et pourrait être améliorée par de futures évolutions. Les travaux de Yossr El Ahmar [17, 1] se concentrent sur ces aspects et ont permis d'utiliser les variables visuelles correctement.

4.2.1 Positionnement automatique des éléments

L'un des premiers moyens visuels d'appréhender des relations est la position des éléments les uns par rapport aux autres. Dans la représentation graphique du diagramme de classe, un *layout* en arbre est automatiquement appliqué à l'ensemble du modèle. Par défaut, toutes les classes sont positionnées les unes à côté des autres, de façon horizontale. Cependant, lorsqu'une classe est en relation avec une autre, ce positionnement devient alors vertical, un élément héritant d'un autre se trouvera toujours en dessous de son parent. Toutes les relations n'entraînent pas de changement de position, seuls les héritages et les implémentations impliquent, dans ce type de diagramme, qu'un élément se situe à un niveau *inférieur* par rapport à un autre.



(a) Positionnement aléatoire

(b) Positionnement intelligent

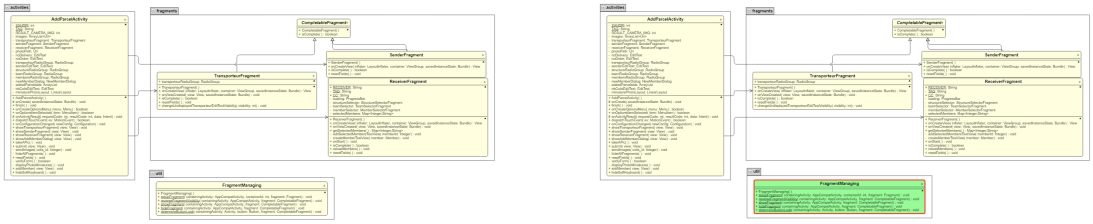
FIGURE 4.6 – Diagramme de classe avec et sans positionnement automatique

Ainsi, comme le montre la figure 4.6, visuellement et avec un minimum de réflexion, l'utilisateur peut avoir une compréhension rapide des relations d'héritage et d'implémentation et donc de la hiérarchie des classes du projet. Il peut alors facilement identifier une classe importante car héritée par beaucoup d'autres classes.

Cette position automatique n'empêche pas l'utilisateur de déplacer les éléments, rien n'est fixé. Il peut donc déplacer un ou plusieurs éléments en les sélectionnant. La nouvelle position est alors sauvegardée en local sur le navigateur. Premièrement, cela permet à l'utilisateur de retrouver le diagramme dans la dernière position sauvegardée, même après avoir fermé la page. Deuxièmement, l'utilisateur peut ainsi disposer spatialement les éléments selon sa façon de penser et les retrouver rapidement par la suite. Plusieurs études sur le positionnement de documents dans un espace 2D ou 3D[8, 41] montrent en effet qu'un utilisateur mémorisera de façon plus efficace un ensemble de documents ou d'objets s'il a la possibilité de les placer où il le souhaite.

4.2.2 Identification rapide de l'élément actif

Une autre variable visuelle essentielle utilisée par VisUML est la couleur. Elle permet de mettre en évidence, sur les diagrammes, l'élément actif dans l'EDI de l'utilisateur.



(a) Sans mise en évidence de l'élément actif (b) Élément actif mis en évidence (en vert)

FIGURE 4.7 – Mise en évidence de l'élément actif sur le diagramme de classe

Ainsi, lors d'un changement de fichier, la nouvelle classe active est aussitôt mise en évidence sur le diagramme de classe par la couleur verte, ce qui facilite la lecture. Grâce à cet ajout, et comme le montre la figure 4.7, il est très simple de repérer l'élément sur lequel le développeur est en train de travailler sur l'EDI. Cette information est visible par quiconque ayant accès à ce diagramme puisque cette représentation est sur une page web publique, accessible aux personnes possédant son URL.

En plus de cette couleur, VisUML propose aussi une interaction 3D utilisant le LeapMotion [2]. Cette interaction a été ajoutée dans le cadre d'une démonstration de l'outil, afin de montrer les possibilités et extensions possibles. Elle utilise le LeapMotion pour détecter la position de la main par rapport à l'écran. Plus

2. LeapMotion : dispositif de reconnaissance de mouvement des mains. (https://www.leapmotion.com/

l'utilisateur approche sa main, plus l'élément actif s'agrandit. Cela permet donc de l'identifier rapidement, mais surtout d'y voir plus de détails. En effet, si le diagramme contient quelques éléments et que l'utilisateur a réduit le niveau de zoom pour voir la globalité de ce dernier, il est peut être compliqué de lire le détail d'un élément précis. Avec cette interaction gestuelle, l'utilisateur peut augmenter la lisibilité de l'élément qui l'intéresse tout en conservant la disposition de son diagramme une fois la tâche terminée.

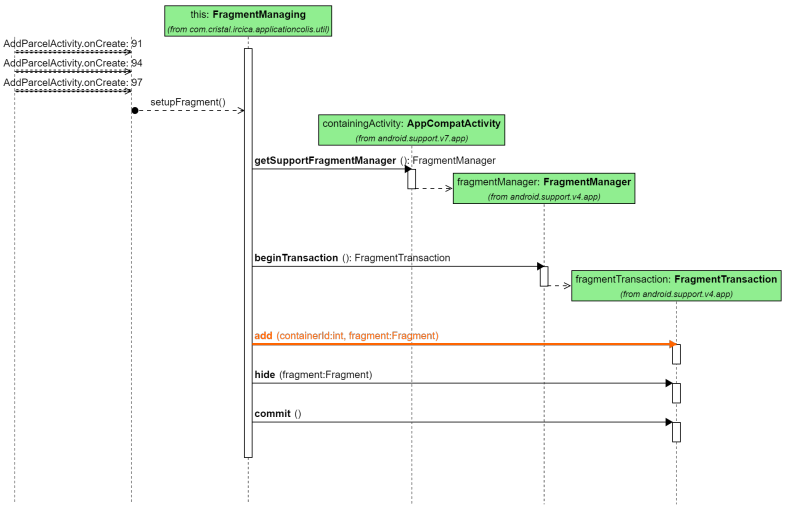


FIGURE 4.8 – Mise en évidence (en orange) d'un message sur le diagramme de séquence

Dans le diagramme de séquence, la couleur est également utilisée pour mettre en évidence le message ou fragment actuellement actif (position du curseur) dans l'EDI. La figure 4.8 montre un diagramme de séquence avec un message actif. De la même façon que pour le diagramme de classe, cette mise en évidence permet à l'utilisateur un passage facile du code vers le diagramme ainsi qu'une lecture plus rapide.

4.2.3 Identification des éléments reliés

Comme présenté précédemment, en plus des éléments ouverts dans l'EDI, le diagramme de classe contient des éléments possédant au moins une relation avec ceux-ci. Afin de ne pas les confondre et informer davantage l'utilisateur, ces éléments possèdent une opacité plus faible que les éléments ouverts. La figure 4.9 montre la différence entre ces deux types d'éléments : l'opacité et le nombre d'informations affichées.

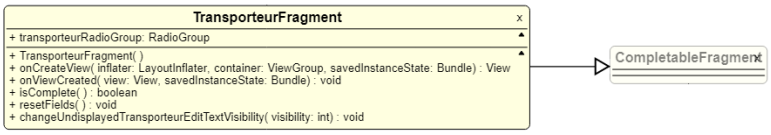
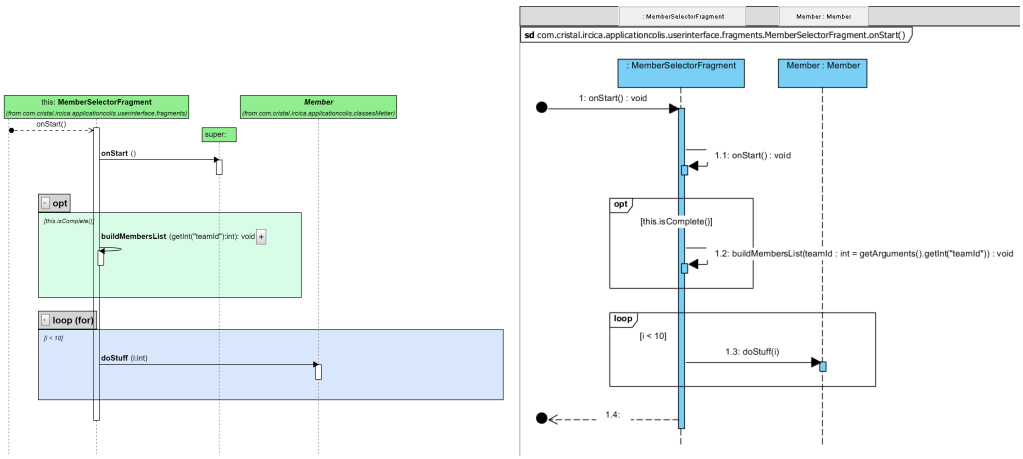


FIGURE 4.9 – Différence entre élément ouvert et non ouvert

4.2.4 Identification des types de fragments

La couleur est aussi utilisée dans le diagramme de séquence pour mettre en valeur les différents types de « Fragment ». En effet, dans UML, un fragment peut correspondre à une condition (*opt* ou *alt*), à une boucle (*loop*), et en plus dans VisUML à un *try*, un *catch* ou un *switch*. A chaque type est associée une couleur, définie arbitrairement. Suites aux perspectives, voir chapitre 6.6 page 143, les futures versions de VisUML intégreront les travaux de Yossr [17] pour améliorer les choix de couleurs, par exemple. Dans VisUML, un bloc *condition* sera représenté en vert, une boucle en bleu, un *try* en orange et un *catch* en rouge. La figure 4.10 montre (a) différents types de blocs dans un diagramme de séquence, et (b) le même contenu dans Visual Paradigm. On constate immédiatement la facilité de lecture avec VisUML pour différencier les types de blocs.



(a) VisUML (b) Visual Paradigm

FIGURE 4.10 – Différents types de fragments dans un diagramme de séquence

De plus, VisUML utilise des couleurs avec une opacité faible. Cet aspect permet d'identifier rapidement les éléments les plus imbriqués puisque cette imbrication augmente l'opacité des blocs en fonction de leur niveau d'imbrication, tel que montré sur la figure 4.11 page ci-contre (deux blocs verts imbriqués).

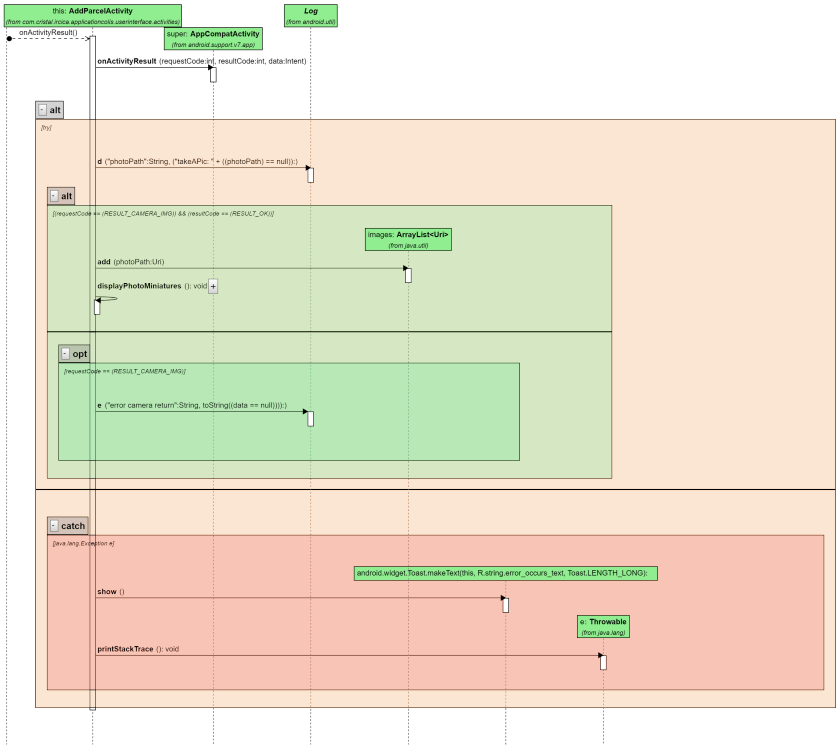


FIGURE 4.11 – Fragments imbriqués dans un diagramme de séquence

4.3 Filtrer les informations affichées

VisUML limite les informations de base sur le diagramme, ce qui permet déjà de le simplifier grandement. De plus, le diagramme devient plus proche de la tâche active du développeur, puisque n'étant composé que des éléments ouverts dans son EDI. Cependant, malgré ces réductions, il peut arriver que le diagramme possède trop d'éléments, ou trop de détails sur ces éléments, pour être correctement lisible. Pour palier ces problèmes, VisUML propose six solutions de filtrage. Ces filtres sont visibles sur la partie gauche de la figure 4.12.

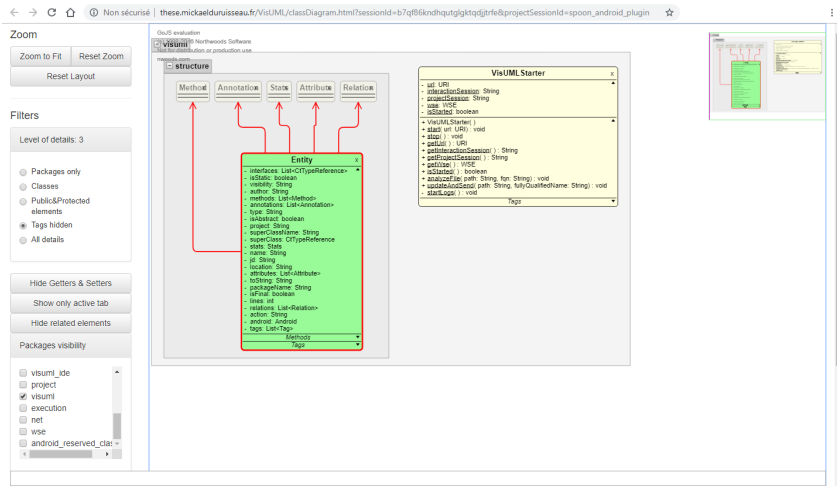


FIGURE 4.12 – Interface complète du diagramme de classe (menu de filtrage à gauche)

4.3.1 Masquer le détail d'un élément

Dans le diagramme de classe, chaque élément est représenté avec son nom, suivi d'un bloc contenant les attributs, puis d'un autre contenant les méthodes. Afin de limiter les informations parfois superflues d'un élément en particulier, ces blocs peuvent s'ouvrir et se fermer, ce qui affiche ou cache leur contenu. L'utilisateur peut donc afficher une vue partielle d'un ou plusieurs éléments, tout en gardant le détail des autres. La figure 4.13 montre un exemple de cette fonctionnalité, la classe *AddParcelActivity* (en vert) possède des attributs et méthodes mais ceux-ci sont cachés.

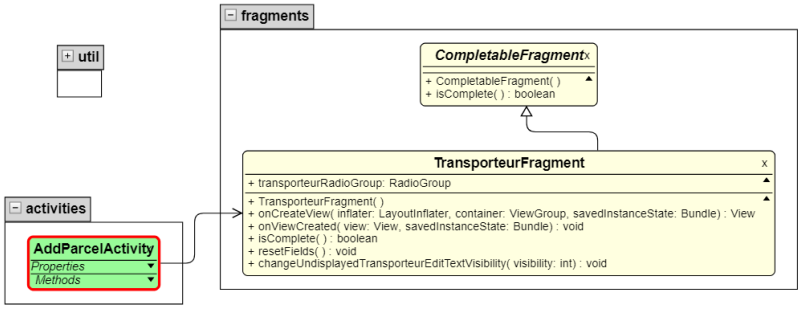
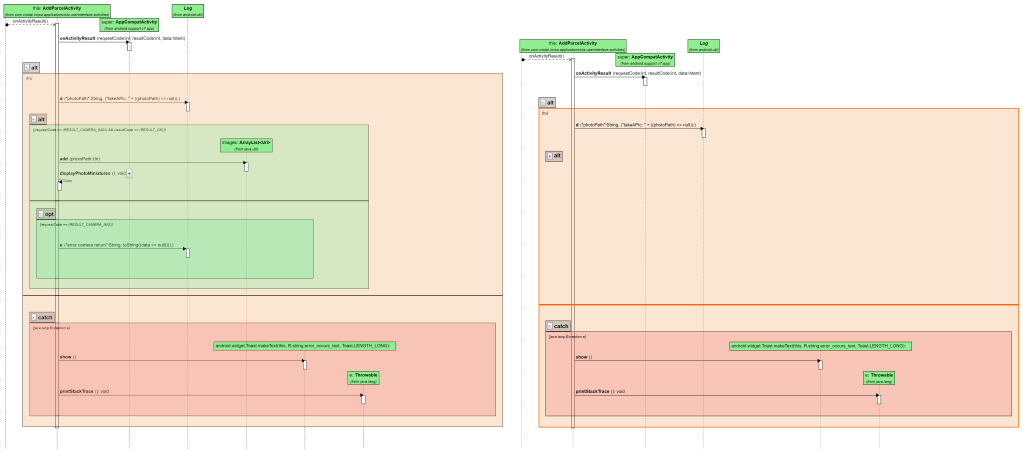


FIGURE 4.13 – Mise en évidence de l'élément actif sur le diagramme de classe

Il est également possible de « fermer » un package, ce qui fait disparaître son contenu. La figure 4.13 montre également la différence entre un package ouvert et un fermé. Ici, le package *util* est fermé et n'affiche pas ses éléments, tandis que le package *fragments* est ouvert et affiche deux éléments.

Dans le diagramme de séquence, les fragments se comportent de la même manière que les packages, ils peuvent afficher ou cacher leur contenu. En revanche, à l'inverse du diagramme de classe, la place occupée par le fragment ne change pas, ce qui donne une indication à l'utilisateur sur la taille des informations cachées. La figure 4.14 présente deux fragments, dont un *fermé*.



(a) Fragment *alt* ouvert (b) Fragment *alt* fermé

FIGURE 4.14 – Fragment *alt* fermé mais conservant sa hauteur

4.3.2 Cacher les éléments reliés non ouverts

Les éléments non ouverts, mais possédant une relation avec un élément ouvert, sont affichés par défaut, avec une opacité plus faible que les autres. Ces

informations sont utiles pour appréhender un élément et les relations qu'ils possèdent, mais peuvent ne pas servir à l'utilisateur selon sa tâche. Il existe donc un bouton permettant de cacher (ou afficher) ces éléments simplement.

Cette fonctionnalité, uniquement sur le diagramme de classe, peut être utile si trop d'éléments sont ouverts, ou si l'utilisateur souhaite avoir une vue restreinte sur ses onglets ouverts, dans le cas d'une création de documentation par exemple. La figure 4.2 page 65 présentée précédemment montre l'intérêt des éléments reliés non ouverts.

4.3.3 Filtrer les types d'informations affichés

Le diagramme de classe peut servir lors de plusieurs étapes d'un projet ainsi qu'à plusieurs types de personnes, que ce soit un architecte logiciel, un développeur, un chef de projet ou même un client. Chaque type d'utilisateur a sa vision du diagramme et a besoin d'informations particulières, ou d'une vue plus ou moins détaillée des éléments. Dans VisUML, il existe cinq vues, paramétrées par rapport à l'utilisation actuelle de l'outil. Ces vues, bien que définies dans le code, peuvent être paramétrables par l'utilisateur, ce qui pourrait lui permettre de gérer des profils de diagramme, certains affichant toutes les informations, d'autres un sous-ensemble, ou encore d'autres n'affichant que les packages. Cette partie n'a pas encore été développée dans l'outil, mais c'est une évolution envisagée par la suite.

Les cinq vues définies actuellement sont les suivantes :

1. Vue complète (avec tags)
2. Vue complète (sans tags)
3. Éléments publics et protégés
4. Classes sans détail
5. Packages uniquement

La vue 1 contient tous les éléments et correspond à l'usage par défaut du diagramme de classe, ainsi que les tags associés à chaque élément (voir section 4.1.2 page 65).

La vue 2 contient les mêmes éléments que la vue 1, mais cache les mots-clés associés.

La vue 3 permet de limiter les informations affichées pour ne retenir que les attributs et méthodes publiques et protégées. Cette vue est particulièrement utile lorsque l'on utilise des librairies, dont l'utilisation repose sur ces informations. De la même façon, un développeur peut générer rapidement un diagramme de classe de sa propre librairie, afin de fournir une documentation à ses utilisateurs.

La vue 4 cache, pour tous les éléments, le détail des attributs et des méthodes. Il en résulte un diagramme de classe simplifié, permettant une lecture rapide des noms des classes ainsi que des relations existantes entre celles-ci. Cette vue permet d'afficher plus de classes, celles-ci étant nettement plus compactes sans détail. Il est toujours possible d'afficher ou de cacher les éléments reliés mais non ouverts via le bouton dédié, présenté dans la sous-section précédente. De même, il est possible d'afficher ou de cacher un bloc (attributs ou méthodes) sur n'importe quel élément affiché sur le diagramme. Ceci permet d'avoir une sorte de raccourci permettant d'ouvrir ou de fermer tous les blocs de tous les éléments.

Enfin, la vue 5 agit comme un diagramme de packages, n'affichant que les packages et leurs relations sans information complémentaire. Cette vue nécessite encore du travail, notamment pour son apparence, afin d'être vraiment efficace. Cependant, c'est un exemple du type de vue qu'il est possible de définir dans VisUML.

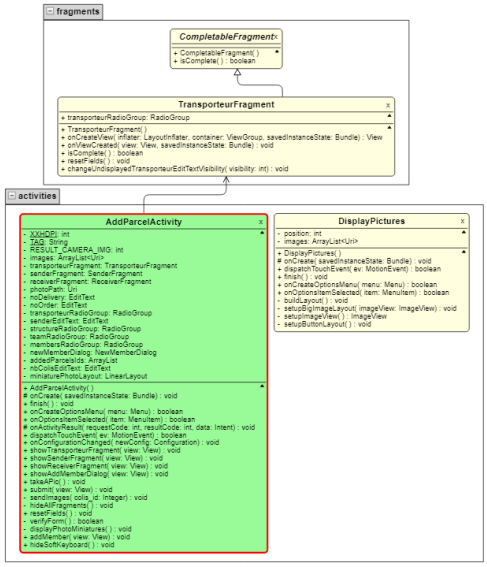
La figure 4.15 page suivante contient le même diagramme avec quatre des cinq vues présentées.

4.3.4 Filtrer les éléments et packages

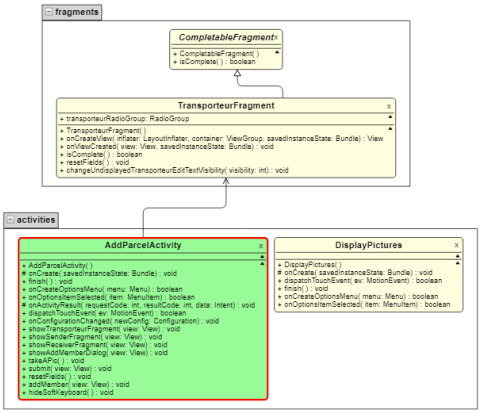
Enfin, l'un des filtres les plus basiques mis en place est la possibilité de cacher ou d'afficher un unique élément. Chaque élément, une fois reçu par le diagramme, va apparaître dans un menu, permettant à l'utilisateur de l'afficher ou de le cacher en cliquant la case à cocher correspondante. Ceci fonctionne à la fois pour les éléments ouverts et pour les éléments reliés non ouverts. De la même façon, il existe un filtre similaire mais agissant sur les packages.

4.3.5 Niveau de profondeur du diagramme de séquence

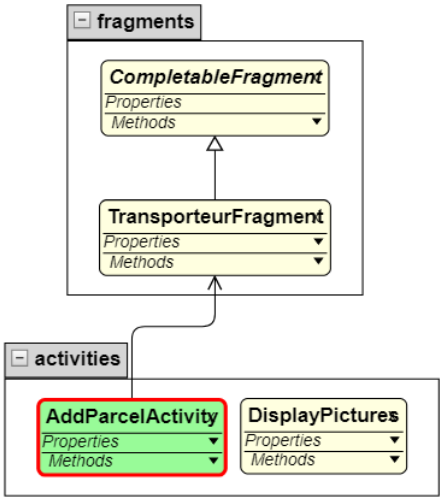
Lors de l'affichage d'un diagramme de séquence, les invocations de la méthode sélectionnée peuvent être sur des éléments « basiques », comme *String* ou *Int*, par exemple l'invocation *X.toString()*, va correspondre à un message *toString()* vers la ligne de vie *X*. L'implémentation de cette méthode *toString()* n'est pas connue par VisUML. Cependant, une invocation peut être aussi effectuée sur un objet d'un type connu, ayant déjà été analysé par VisUML. Dans ces conditions, il est possible d'afficher sur le diagramme de séquence cette implémentation à la suite du message d'invocation. La figure 4.16 page 79 présente trois niveaux de profondeur dans le diagramme de séquence.



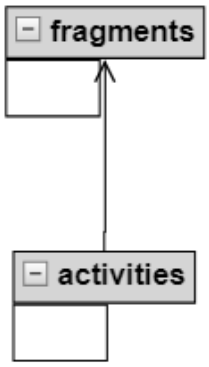
(a) Vue 2 : par défaut (sans tags)



(b) Vue 3 : éléments publics et protégés uniquement

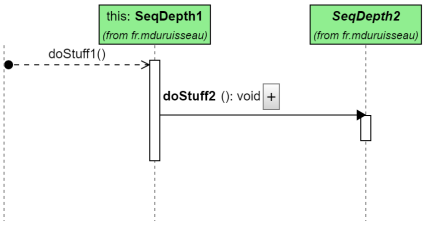


(c) Vue 4 : classes sans détails

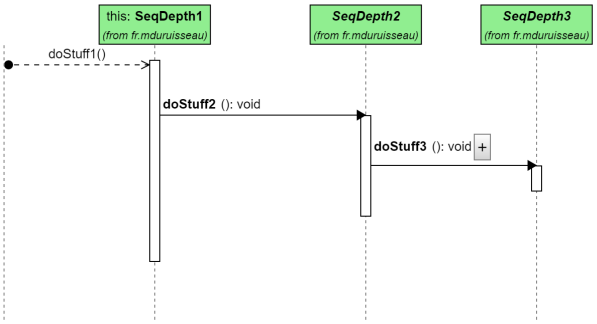


(d) Vue 5 : packages seulement

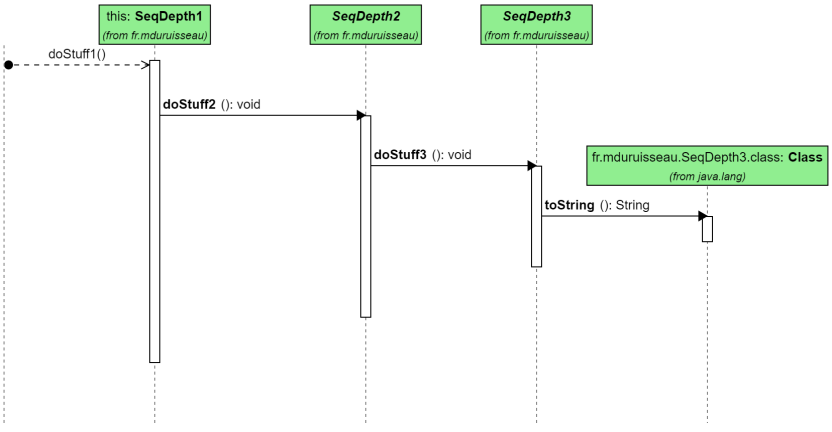
FIGURE 4.15 – Profils de vues sur le diagramme de classe



(a) Affichage avec le niveau par défaut (niveau 1)



(b) Affichage au niveau 2



(c) Affichage au niveau 3

FIGURE 4.16 – Changement du niveau de profondeur du diagramme de séquence

Il en va de même pour d'éventuelles invocations dans l'implémentation nouvellement affichée, et ainsi de suite. On peut donc définir un niveau de profondeur au diagramme de séquence, correspondant à la limite d'imbrication d'une invocation pour laquelle on ne détaillera pas son implémentation. Cette limite de niveau est potentiellement infinie, mais pour des raisons techniques, de chargement et de complexité du diagramme, elle est actuellement bloquée à cinq dans VisUML.

Ce niveau permet de définir un paramètre global à toutes les invocations de méthodes. Cependant, l'utilisateur peut souhaiter ne voir que le détail d'une invocation précise. Pour se faire, VisUML ajoute un bouton « + » à côté du message d'invocation, lorsque celle-ci peut être affichée en détail. Il suffit alors à l'utilisateur de cliquer dessus pour faire apparaître l'invocation. La figure 4.17 présente cette fonctionnalité.

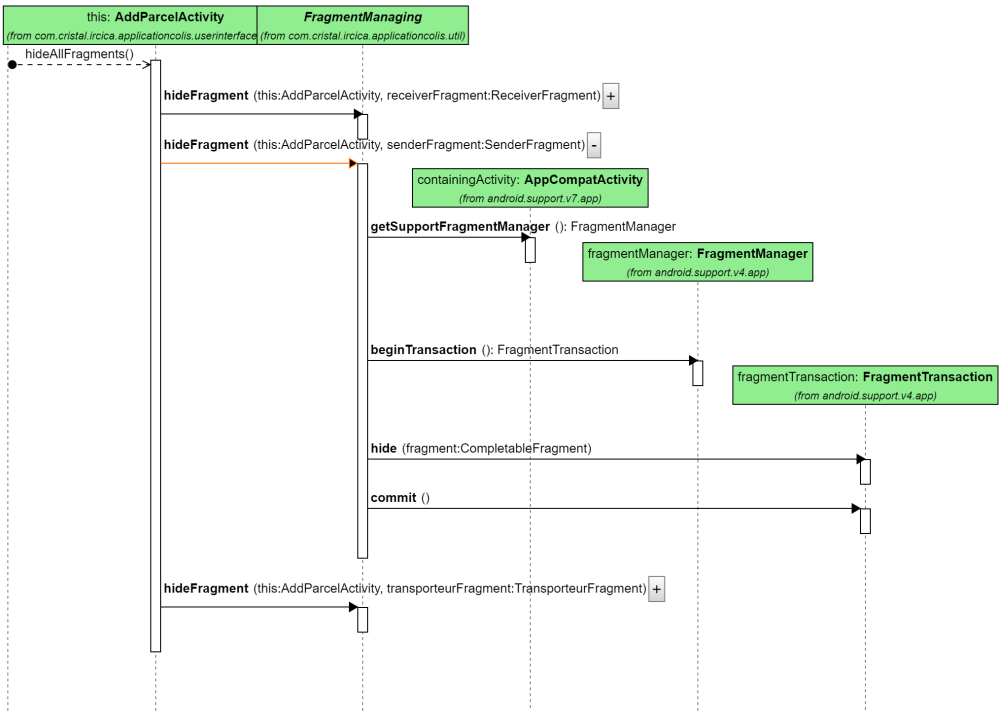


FIGURE 4.17 – Diagramme de séquence avec détail d'une invocation sur trois niveaux

4.3.6 Filtres complexes et mise en valeur d'éléments

La dernière solution de filtre est l'implémentation d'une grammaire spécifiquement définie pour filtrer des éléments sur un diagramme de classe. Cette

grammaire, définie grâce à PEG.js [3], permet à l'utilisateur de mettre en évidence un ou plusieurs éléments sur son diagramme en utilisant une commande qui se rapproche d'une phrase simplifiée.

Une commande de filtre se forme toujours sous cette forme : [*action*] [*type*] [*filtres*] ; où **action** représente le changement à effectuer sur les éléments, **type** correspond à un identifiant d'élément et **filtres** représente un ou plusieurs filtres selon les valeurs des éléments.

Les différents types d'actions sont :

— *show* : affiche un élément
— *hide* : cache un élément
— *color [color]* : change la couleur de fond des éléments filtrés
— *border [color]* : change la couleur de la bordure des éléments filtrés
— *borderStyle [normal|dash|point]* : définit le type de bordure
— *borderWidth [0-9]+* : définit la largeur de la bordure
— *rotate [0-360]* : change l'orientation des éléments
— *scale [0-9.,]+* : change la taille des l'éléments

Ces actions ont été définies en utilisant les variables visuelles dont nous avons parlé précédemment (section 1.4 page 11). Ces dernières permettent la mise en évidence d'un élément par modification de sa représentation graphique, éventuellement en cumulant plusieurs filtres (par exemple, bordure seule ou couleur + bordure + rotation).

Les types d'éléments sont : classe, interface et package. Ce type permet d'effectuer un premier filtre général sur le type d'éléments à mettre en évidence. Il devient alors simple de cacher toutes les interfaces (*hide interface ;*) ou à l'inverse de ne mettre en évidence que les classes, par exemple : *scale 2 class ;*.

Les filtres se composent de plusieurs parties selon l'élément concerné. La première partie sera toujours *with* ou *without*, indiquant si ce filtre doit ajouter ou retirer l'élément des résultats. La seconde partie indique sur quelle composante le filtre va être effectué, par exemple *name*, *attribute*, *method* ou *visibility*. La suite dépend de la composante choisie. Il est possible de cumuler plusieurs filtres en les ajoutant les uns à la suite des autres. Par exemple : *show class with attribute nb >= 2 with method nb <= 3 ;* permettra d'afficher les classes avec au moins deux attributs et au maximum trois méthodes.

Il est à noter que ce type de filtre est actuellement impossible à réaliser dans un EDI classique, tel que IntelliJ ou Eclipse.

La grammaire complète ainsi que les différentes options pour ces composantes sont présentées en annexe B page 172.

La figure 4.18 sert de point de départ avant l'application de chacun des filtres présentés ci-dessous.

3. PEG.js : `https://pegjs.org/`

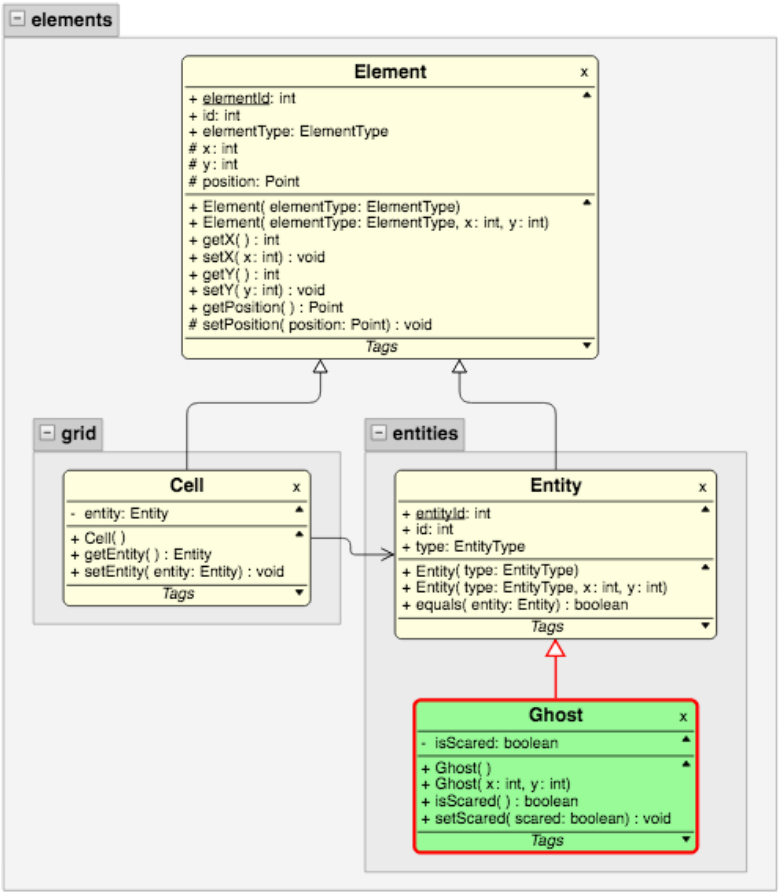


FIGURE 4.18 – Diagramme de classe sans filtre

La figure 4.19 montre l'utilisation du filtre : *hide class with name starts by E ;,* permettant de cacher les classes dont le nom commence par E.

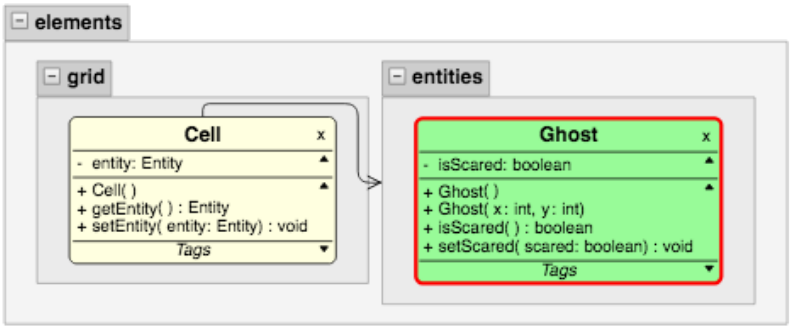


FIGURE 4.19 – Diagramme de classe avec le filtre « *hide class with name starts by E ;* »

La figure 4.20 montre l'utilisation du filtre : *hide class with attribute nb > 3 ;*, permettant de cacher les classes dont le nombre d'attributs dépasse trois.

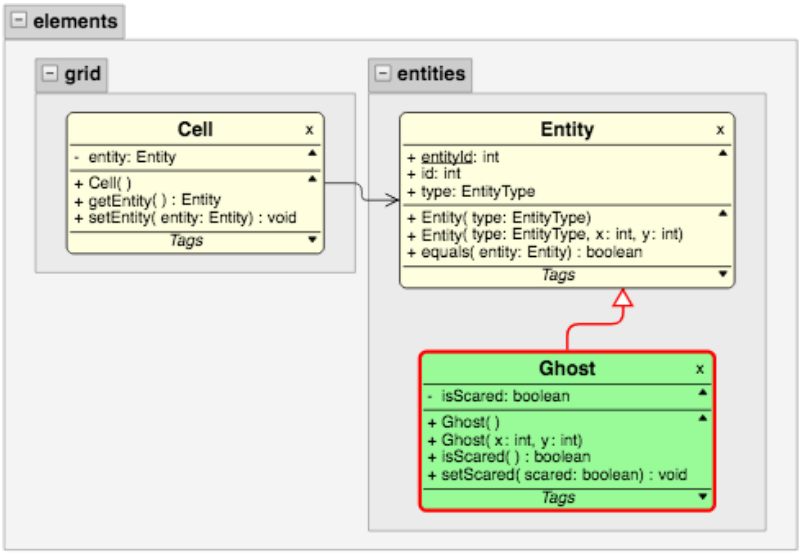


FIGURE 4.20 – Diagramme de classe avec le filtre « *hide class with attribute nb > 3 ;* »

La figure 4.21 montre l'utilisation du filtre : *color lightblue class with method name starts by get ;*, permettant de colorier en bleu clair les classes qui possèdent au moins une méthode dont le nom commence par get. Autrement dit, ce filtre met en évidence les classes avec au moins un *getter*.

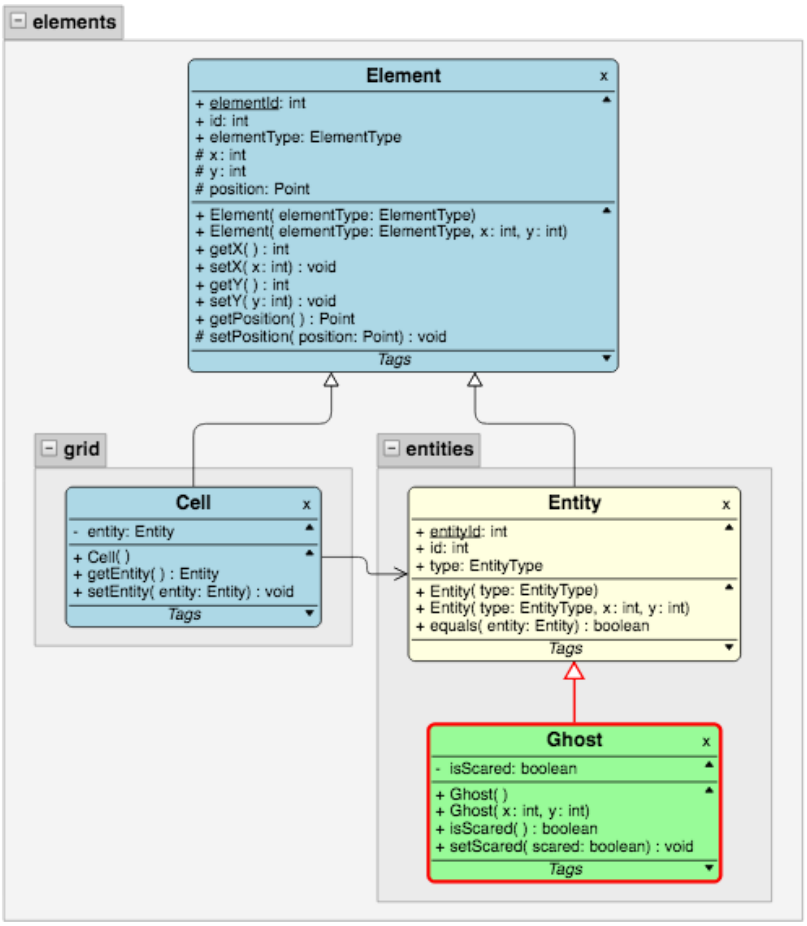


FIGURE 4.21 – Diagramme de classe avec le filtre « *color lightblue class with method name starts by get ;* »

La figure 4.22 ajoute des éléments au diagramme de classe présenté dans la figure 4.18 et sert de nouvelle base pour les filtres suivants.

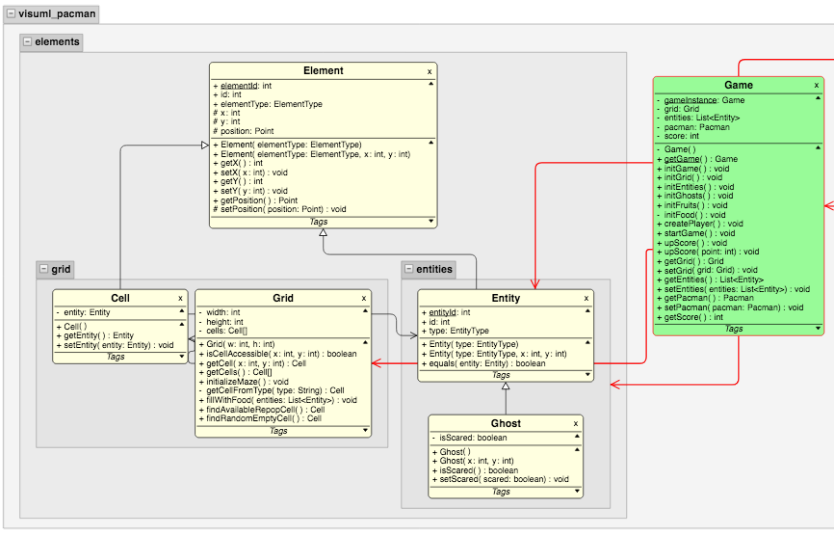


FIGURE 4.22 – Nouveau diagramme de classe sans filtre

La figure 4.23 montre l'utilisation du filtre : *borderwidth 10 class with method type Entity;*, permettant d'augmenter la taille des bordures des classes possédant au moins une méthode retournant un objet de type Entity.

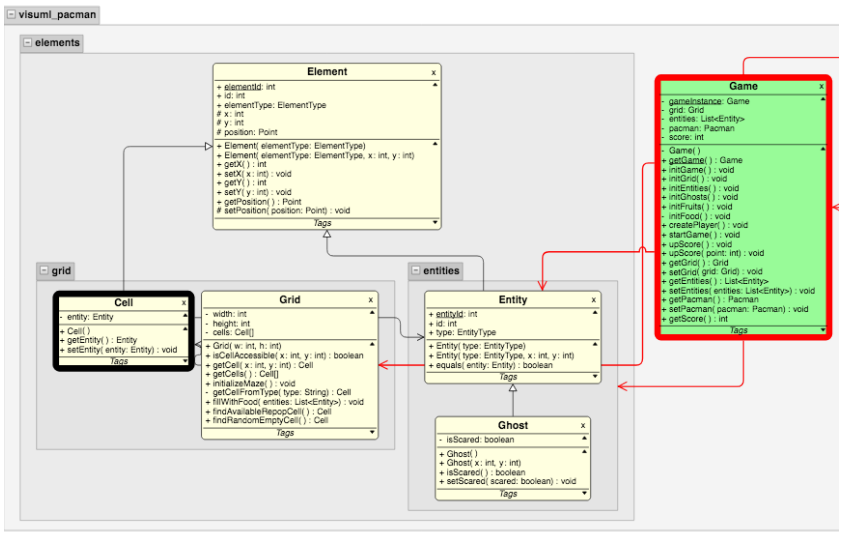


FIGURE 4.23 – Diagramme de classe avec le filtre « textitborderwidth 10 class with method type Entity; »

La figure 4.24 montre l'utilisation du filtre : *rotate 45 class with name starts by E with method nb > 4;*, permettant d'effectuer une rotation sur les classes dont le

nom commence par E et dont le nombre de méthodes dépasse 4. Cette phrase combine plusieurs filtres qui sont liés par des ET logiques.

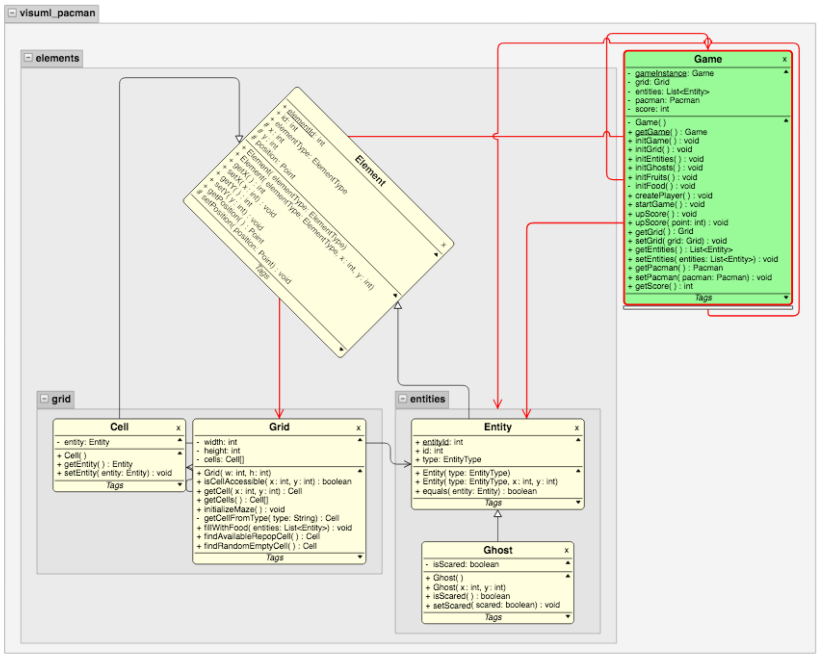


FIGURE 4.24 – Diagramme de classe avec le filtre « *rotate 45 class with name starts by E with method nb > 4 ;* »

Grâce à ces filtres, l'utilisateur peut avoir une vue rapide sur des éléments spécifiques à ses besoins. Il peut par exemple demander à n'afficher que les classes qui héritent d'une autre, ou voir uniquement celles qui ont un attribut d'un certain type, etc. La version actuelle de cette grammaire offre un nombre de possibilités intéressantes quant au filtrage. Néanmoins il est possible d'étendre cette grammaire en ajoutant de nouveaux filtres et paramètres. Pour cela, il est nécessaire de modifier la grammaire (par exemple, ajouter un type tel que *activity*) et de répercuter ces modifications sur VisUML pour prendre en compte ces nouveaux filtres.

Une autre évolution possible serait d'utiliser ce principe de filtrage, non pas sur le diagramme, mais directement dans l'EDI, voir chapitre 6.6 page 143.

Cette partie étant arrivée tardivement dans l'outil, elle n'a pas fait l'objet d'une évaluation à ce jour.

4.4 Conclusion

Ce chapitre a présenté les différentes méthodes utilisées par VisUML pour mettre en évidence des informations (tags et filtres) à l'utilisateur, tout en réduisant leur volume (affichage des onglets ouverts uniquement).

La première méthode est de limiter le nombre d'éléments affichés à l'utilisateur. Cela évite une surcharge du nombre d'informations et permet donc une meilleure compréhension globale du diagramme. Cela évite l'affichage de diagrammes illisibles ou nécessitant de nombreuses actions de navigation (zoom, défilement…) au sein de ces derniers.

La seconde méthode est la mise en évidence d'éléments importants. Il existe plusieurs façons de donner du sens à un objet, dont la première est le positionnement. Il est très courant par exemple de représenter une hiérarchie sous la forme d'un arbre (*TreeLayout*), l'élément le plus important étant au sommet et ses enfants situés en dessous de façon récursive. Pour les diagrammes de classe de VisUML, cette notion d'arbre utilise les liens de généralisation et d'implémentation. Ainsi, pour un élément A héritant d'un élément B, l'élément A va apparaître en-dessous de B. Cela permet à l'utilisateur de repérer rapidement la hiérarchie entre ses classes.

Les packages regroupent quant à eux plusieurs éléments au sein d'un même groupe, représenté par un conteneur gris sur VisUML. Ils peuvent eux-mêmes appartenir à un autre package, et cette appartenance est représentée par un package plus petit au sein du parent. L'utilisation d'une couleur transparente permet également d'avoir une vue rapide sur l'appartenance ou non à un package.

De la même façon que les classes, les packages sont également organisés grâce à un positionnement en arbre.

Enfin, la dernière méthode utilisée est le filtrage. Dans VisUML, nous avons implémenté plusieurs niveaux de filtres, du plus basique permettant de sélectionner soi-même les éléments ou packages affichés ou non, au plus complexe permettant de choisir quels types d'informations peuvent apparaître.

Les filtres présentés dans la section 4.3.6 permettent à l'utilisateur de mettre en évidence des éléments selon un ou plusieurs critères complexes. Ces filtres peuvent être comparés à des requêtes SQL qui ont permis d'élaborer la grammaire. Cette dernière pourrait encore être étendue afin de supporter plus de critères ou d'éléments.

Un utilisateur de VisUML peut utiliser les variables visuelles, telles que la couleur ou la bordure, pour obtenir un diagramme affichant des informations supplémentaires sans pour autant augmenter le nombre d'éléments. Les figures présentées dans la section précédente sont des exemples de tels filtres.

Ce chapitre a donc présenté un ensemble de choix faits afin de ne pas sur-

charger d'informations l'utilisateur, tout en lui permettant de mettre lui-même en évidence les données qu'il souhaite voir. Ceci constitue un avantage majeur de VisUML sur les outils comparés dans le chapitre 2 page 16, puisque ces derniers affichent généralement des diagrammes de classe complets et nécessitent beaucoup d'interactions pour limiter les informations.

Le chapitre suivant se concentre sur les fonctionnalités offertes à l'utilisateur en termes de navigations et d'interactions au sein des diagrammes et dans l'EDI.

Chapitre 5

VisUML : Navigation et interactions

Sommaire

Ce chapitre présente les interactions implémentées sur les diagrammes et les EDI supportés, permettant une navigation facile entre ces artefacts. Il existe trois types de navigation : du code aux diagrammes, d'un diagramme vers le code et enfin entre diagrammes. Ce chapitre contient trois parties, chacune présentant l'un de ces types. Les interactions correspondent à l'envoi ou la réception de l'un des messages présentés section 3.4 page 50.

5.1 De l'EDI aux diagrammes

Les interactions provenant de l'EDI servent à maintenir le diagramme à jour mais aussi à faciliter la lecture et la navigation entre le code et les représentations graphiques. En effet, elles permettent à l'utilisateur d'avoir un indicateur visuel sur l'élément actif ou actuellement modifié.

5.1.1 Synchroniser les diagrammes avec le code

L'une des fonctionnalités principales de VisUML est de garder les diagrammes toujours à jour par rapport au code. La première « interaction » ne nécessite pas d'action spécifique de la part de l'utilisateur et est déclenchée par la modification du code, tâche habituelle d'un développeur. Lors d'une modification du code ou d'une sauvegarde (ces dernières étant automatique sur IntelliJ), le plugin connecté à l'EDI va aussitôt envoyer un message *createOrUpdateUML* contenant toutes les informations de l'élément (classe, interface, énumération...) qui vient d'être modifié. Ces informations sont reçues par les diagrammes de classes et de séquences en temps réel et ceux-ci se mettent à jour automatiquement. Il est ainsi possible d'obtenir un diagramme de séquence de la méthode actuellement développée. Il en va de même pour les classes, dont les attributs et méthodes sont rafraîchis à chaque mise à jour.

5.1.2 Réduire la charge mentale du développeur

Le développeur va souvent passer d'une visualisation (code, diagrammes, images, autres ressources) à l'autre lors des différentes phases de travail (compréhension du projet, recherche d'informations, développement...). Grâce à VisUML, le développeur a accès à des diagrammes toujours à jour. Cette vue est disponible et accessible au même moment que le code. La navigation entre deux représentations (textuelle pour le code, graphique pour les diagrammes) peut prendre du temps et requiert une charge mentale relativement importante. VisUML a pour objectif de réduire cette charge mentale au maximum, en facilitant la lecture des diagrammes. Ces fonctionnalités sont détaillées dans la partie 4.1.1 page 64. Cette sous-section se concentre sur la partie interaction de ces fonctionnalités.

5.1.2.1 Maintenir à jour la liste des onglets ouverts

Comme présenté chapitre 4 page 63, le diagramme de classe ne contient que les onglets actuellement ouverts sur l'EDI, ainsi que leurs relations. Afin de

n'afficher que ces éléments, il est nécessaire d'écouter les interactions d'ouverture et de fermeture d'onglet.

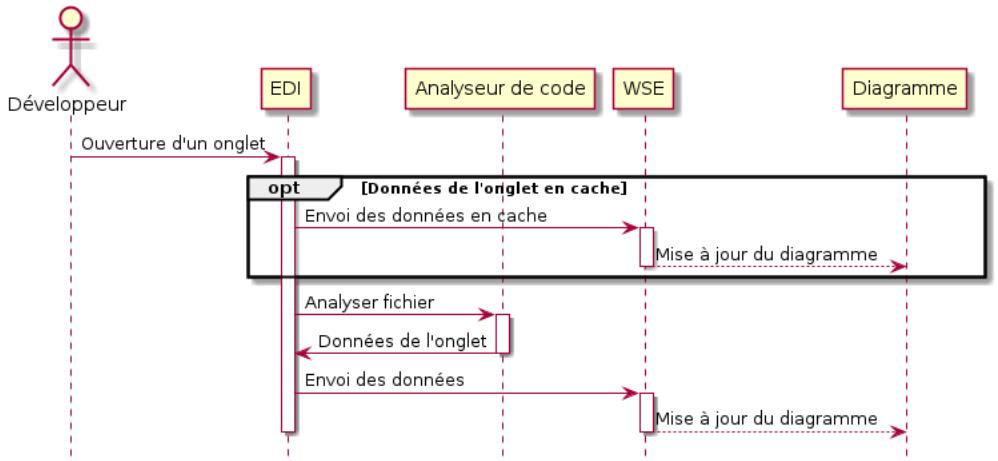


FIGURE 5.1 – Diagramme de séquence d'ouverture d'un onglet

Comme présenté dans la figure 5.1, les ouvertures d'onglet déclenchent une analyse du fichier associé, puis l'envoi, via WSE, du message contenant les données de l'élément. Ce message, une fois reçu par un diagramme, va déclencher la mise à jour de celui-ci. Nous avons également ajouté un mécanisme de cache, qui permet l'envoi des informations de la classe souhaité dès le clic, avant de faire une mise à jour complète de cet objet. Dans les prochaines versions de VisUML (voir le chapitre 6.6 page 143), nous aimerions intégrer un calcul de différence afin de n'envoyer que le *delta* entre deux versions.

Les fermetures d'onglet ne déclenchent que l'envoi d'un message aux diagrammes, permettant de supprimer ou cacher la représentation graphique associée.

Grâce à ces deux interactions, et toujours sans action de la part du développeur, le diagramme de classe reflète exactement l'EDI.

5.1.2.2 Mise à jour de l'onglet actif

Cette fonctionnalité, présentée section 4.2.2 page 70, permet de mettre en évidence l'onglet ouvert. Elle nécessite l'écoute d'un autre type d'interaction : le changement d'onglet. Cet événement est aussi déclenché après l'ouverture d'un nouvel onglet.

Une fois cet événement détecté, l'EDI envoie un message sur WSE, indiquant quel élément est actuellement actif. Les diagrammes reçoivent ce message et mettent à jour leur affichage.

5.1.2.3 Mise à jour de la séquence

Afin de contextualiser encore plus les diagrammes, le diagramme de séquence affichera toujours la méthode actuellement parcourue par le développeur.

Lorsque l'utilisateur déplace son curseur dans le corps d'une méthode, un message est envoyé sur WSE afin d'en avertir les diagrammes. Les informations envoyées par l'EDI contiennent le nom de la classe, celui de la méthode ainsi que la ligne où se trouve le curseur. Ces informations permettent au diagramme de se mettre à jour afin d'afficher la séquence correspondante à cette méthode. Le détail des actions effectuées lors du déplacement du curseur est représenté dans la figure 5.2.

La ligne du curseur étant envoyée, il est possible de mettre en évidence le message ou le groupe correspondant sur le diagramme. Cette fonctionnalité est décrite section 4.2.2 page 70.

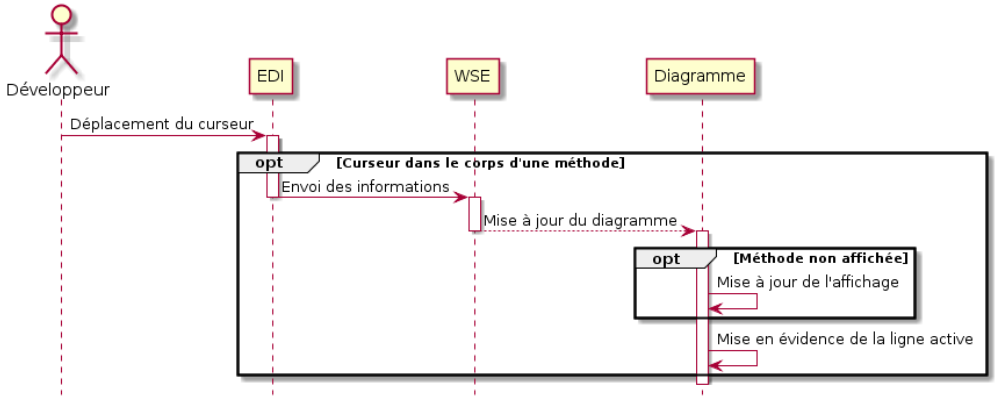


FIGURE 5.2 – Diagramme de séquence des actions effectuées lors d'un mouvement du curseur

5.2 Des diagrammes à l'EDI

Fournir une représentation graphique mise à jour en temps réel permet à l'utilisateur d'avoir un second point de vue sur son code, lui facilitant la détection des relations ou des patterns de son projet. Cependant, une telle représentation n'est utile que si elle permet à l'utilisateur d'interagir avec. Cette possibilité existe d'ailleurs dans plusieurs outils de développements. L'intérêt de VisUML est d'ajouter plusieurs interactions permettant une navigation depuis les diagrammes vers le code. Ces interactions fonctionnent à la fois sur les vues web et sur le plugin Papyrus (partie diagramme de classe), et correspondent

principalement à des clics sur des éléments. Dans le cas des vues web, les événements tactiles (type « *touch* ») sont également pris en charge et interprétés comme des clics.

5.2.1 Diagramme de classe

5.2.1.1 Navigation depuis un élément

Le moyen le plus simple pour naviguer d'un diagramme de classe vers le code est de cliquer sur un élément (classe, interface, énumération) du diagramme. Ce clic déclenche l'envoi d'un message « *switchToClass* », présenté section 3.4 page 50, qui, une fois reçu par l'EDI, met à jour l'onglet actif sur le fichier contenant l'élément. Ce changement d'onglet déclenche par ailleurs la mise à jour du visuel associé, cette fonctionnalité est présentée dans la section précédente.

Il est également possible de cliquer sur un élément relié non ouvert du diagramme de classe. Ces éléments sont présentés dans la section 4.1.1 page 64 et correspondent à des éléments reliés d'une façon ou d'une autre à un élément ouvert (un onglet).

Lorsque l'utilisateur clique sur l'un de ces éléments, le plugin EDI va chercher l'onglet associé dans la liste des onglets ouverts afin de le mettre au premier plan. S'il ne s'y trouve pas, l'EDI ouvrira le fichier associé dans un nouvel onglet, puis le mettra en avant.

Ces deux actions ont pour effet d'envoyer aux diagrammes une représentation complète de l'élément (dont seul son nom et son FQN avaient été envoyés jusqu'ici). Les diagrammes vont alors se mettre à jour. Un message indiquant quel élément est actuellement actif est également envoyé par l'EDI, ce qui permet aux diagrammes de le mettre en évidence visuellement, comme présenté précédemment.

Grâce à cette interaction de navigation, il est possible pour le développeur d'ouvrir des fichiers sur son EDI sans avoir à les chercher dans une liste ou un menu. Si une classe affichée sur le diagramme l'intéresse, il lui suffit de cliquer dessus pour l'ouvrir sur l'EDI. Cette capacité d'exploration du code à partir des diagrammes UML répond aux deux aspects énoncés par la dimension cognitive « Visibilité »[2, 21, 20], à savoir la capacité d'accéder facilement à un élément et de rendre l'exploration plus efficace.

Etant donné que l'ouverture de cette nouvelle classe va détailler le contenu de sa représentation graphique et potentiellement ajouter de nouveaux éléments reliés non ouverts, il est possible de naviguer plusieurs fois successivement de classe en classe. Cette navigation permet par exemple de parcourir les relations d'héritage et d'implémentation d'une classe. Un exemple est présenté figure 5.4 page 95.

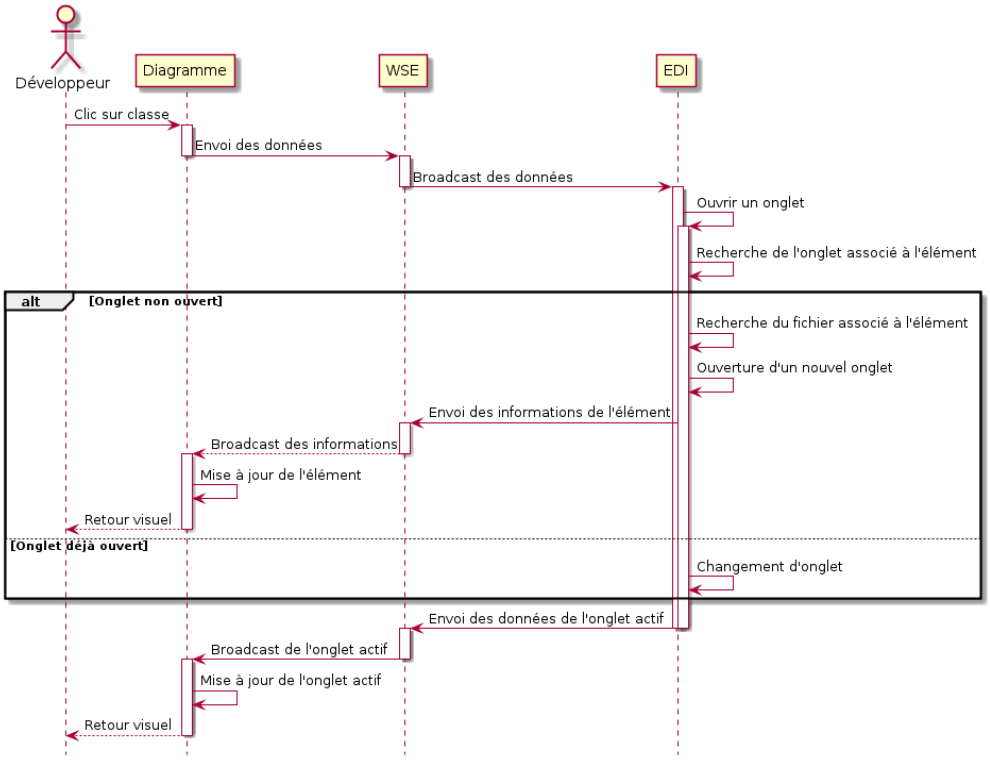


FIGURE 5.3 – Diagramme de séquence des actions effectuées lors d'un clic sur un classe

De même, puisque les relations peuvent aussi être des classes *filles*, il est possible de naviguer de *haut en bas* (classes qui héritent), pour trouver une classe *sœur*, qui hérite de la même super-classe. La figure 5.5 page 96 montre cette navigation.

5.2.1.2 Navigation depuis un attribut ou une méthode

Afin de permettre une navigation plus précise du diagramme vers le code, VisUML implémente également des interactions (clic ou touch) sur les attributs et les méthodes des éléments affichés. Ces interactions déclenchent, si nécessaire, un changement d'onglet afin de mettre en avant le fichier associé à l'élément cliqué. Une fois celui-ci au premier plan, l'EDI va faire défiler le contenu afin de centrer l'attribut ou la méthode dans l'éditeur, puis va le sélectionner. La figure 5.6 page 97 détaille la façon dont l'interaction est interprétée. Ces interactions permettent une navigation précise vers n'importe quel type d'élément présent dans un diagramme de classe.

Enfin, il est possible de fermer un élément en cliquant sur une croix (X) située

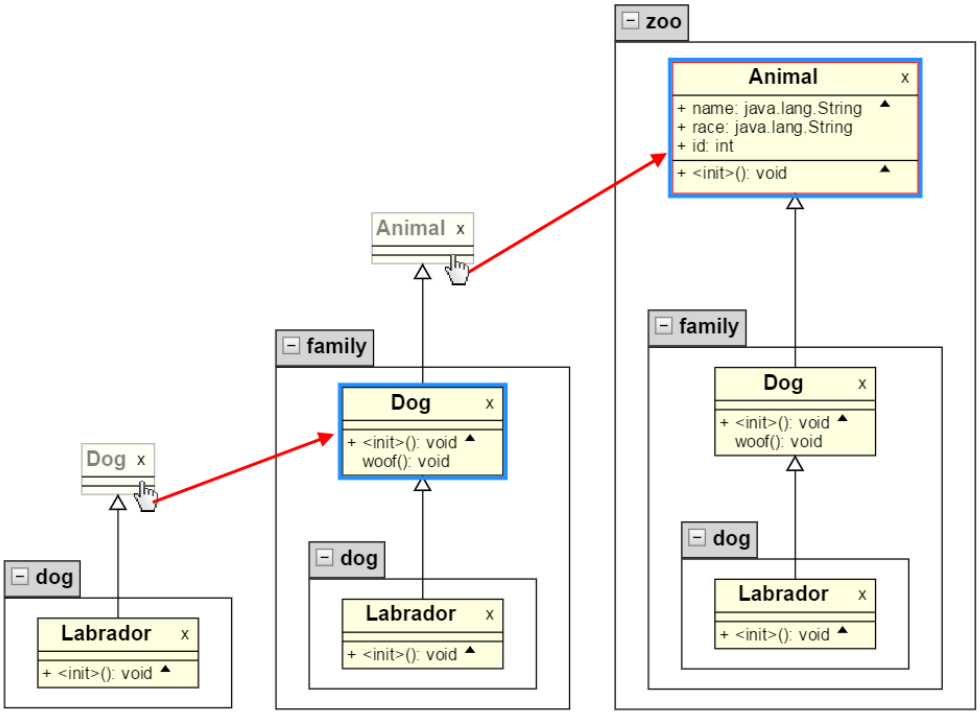


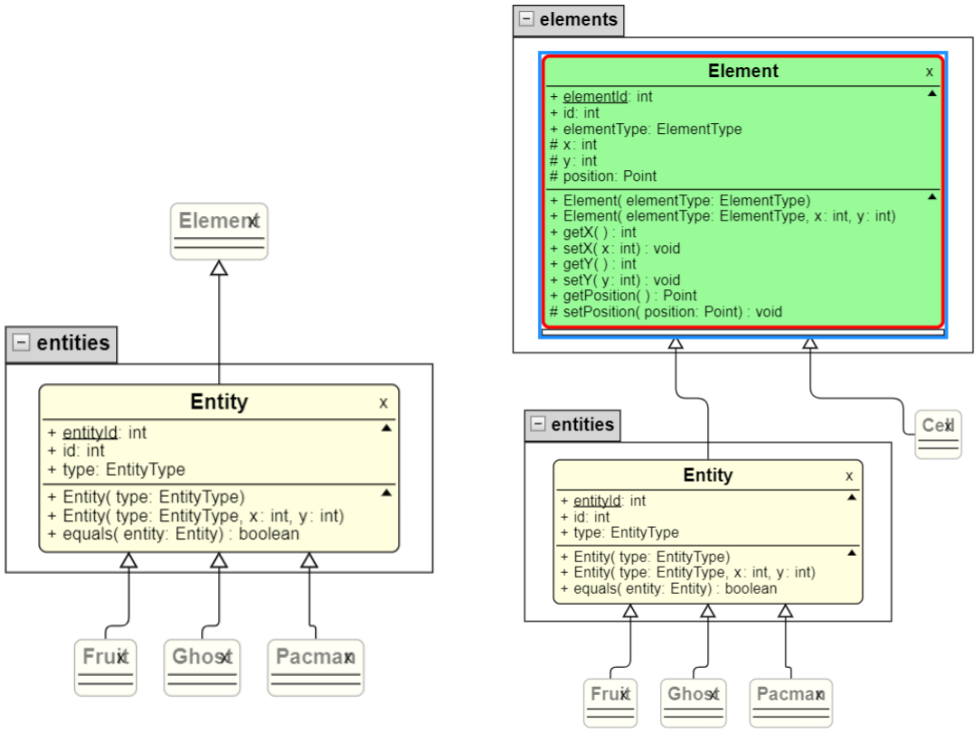
FIGURE 5.4 – Exemple de navigation via les liens d'héritage dans un diagramme de classe

dans le coin supérieur droit de la représentation graphique. Cette action entraîne également la fermeture de l'onglet associé (sauf dans le cas d'un élément relié non ouvert) sur l'EDI, voir figure 5.7 page 98.

5.2.2 Diagramme de séquence

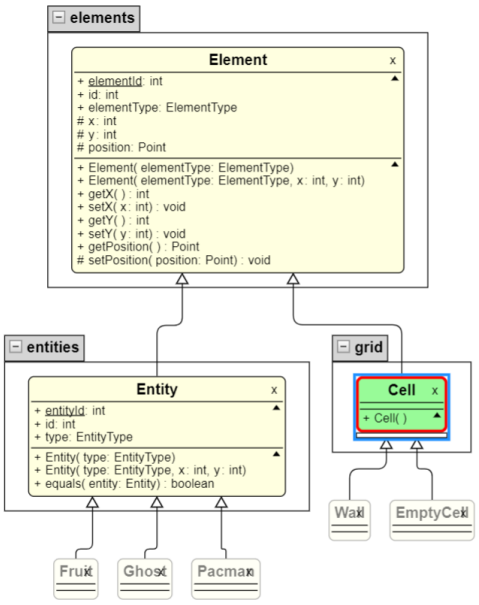
De la même façon, le diagramme de séquence est lui aussi interactif. Un clic sur un message (lien entre deux lignes de vie) va changer l'onglet actif (si nécessaire), défiler le code jusqu'à l'élément associé au message, puis le mettre en valeur. Ces actions sont identiques à celles effectuées lors d'un clic sur un attribut ou une méthode, mais à un niveau plus précis (ligne spécifique d'une méthode). Il est également possible de cliquer sur un fragment (groupe contenant un ou plusieurs messages), représentant un if/else, une boucle ou un try/catch. Cette interaction aura pour effet de sélectionner l'élément entièrement, ce qui correspond à plusieurs lignes dans l'EDI (le changement d'onglet et le défilement sont également effectués si nécessaire).

Enfin, il est possible de cliquer sur une ligne de vie, ce qui met en avant sur



(a) Diagramme de base

(b) Apparition de *Cell* après un clic sur *Element*



(c) Le diagramme après un clic sur *Cell*

FIGURE 5.5 – Navigation dans les deux sens via les liens d'héritage dans un diagramme de classe

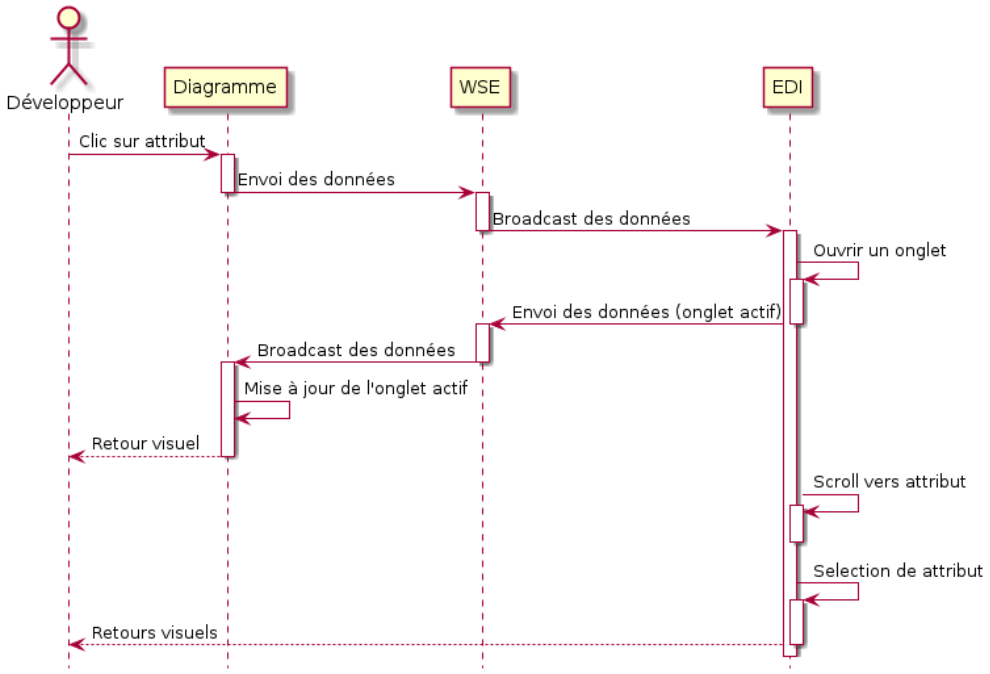


FIGURE 5.6 – Diagramme de séquence des actions effectuées lors d'un clic sur un attribut

l'EDI la classe associée à la variable (ou classe statique) et déclenche donc un changement d'onglet. Cette mise en valeur pourrait être remplacée par une mise en avant de la variable en elle-même, ce qui implique une sélection partielle de la ligne.

Ces interactions permettent à l'utilisateur de naviguer dans le code à un niveau très précis, c'est-à-dire au niveau d'une ligne de code. Cela permet de maintenir un lien entre ce que l'utilisateur voit dans le diagramme et la (ou les) ligne(s) associée(s) dans le code. Il peut alors cliquer sur le message ou fragment qui l'intéresse pour le sélectionner dans le code, et étudier plus en détail le code associé.

5.3 Navigations entre et dans les diagrammes

En supplément des interactions bidirectionnelles code - diagrammes (modèle), il est également possible de naviguer entre le diagramme de classe et le diagramme de séquence, mais aussi entre différents diagrammes de séquence facilement.

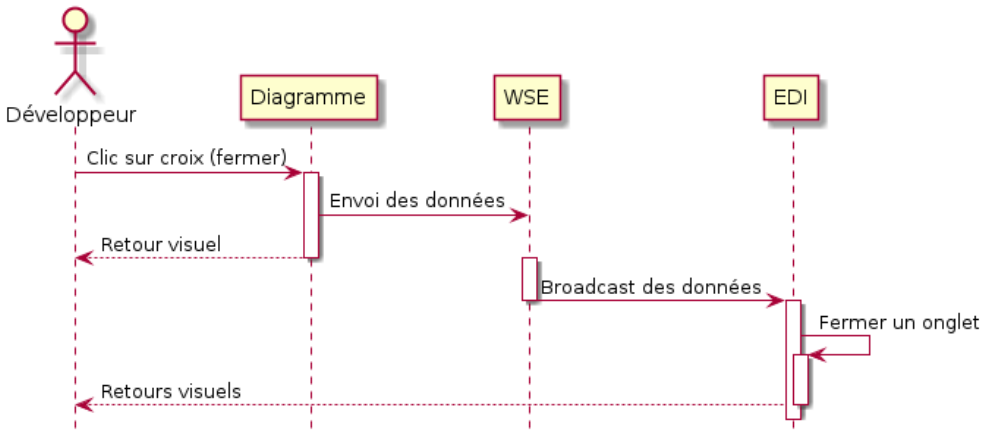


FIGURE 5.7 – Diagramme de séquence des actions effectuées lors de la fermeture d'une représentation graphique

5.3.1 Mise à jour automatique du diagramme de séquence

La seule interaction actuelle permettant de naviguer entre les deux types de diagramme est le clic sur une méthode depuis un diagramme de classe. Ce clic (ou *touch*) déclenche la mise à jour de l'EDI, comme présentée section 5.2.1.2 page 94, mais également la mise à jour du (ou des) diagramme(s) de séquence ouvert(s), associé(s) au projet.

Cette interaction permet aux utilisateurs de passer d'un type de diagramme à l'autre, sans avoir à agir sur le code ou l'EDI, tout en gardant ce dernier synchronisé (onglet actif correspondant à l'élément actif dans les diagrammes). Il est donc facile, en imaginant deux ou trois écrans, d'avoir une page diagramme de séquence ouverte, dans laquelle on peut voir les méthodes voulues, uniquement en interagissant avec le diagramme de classe.

En revanche, il n'est pas possible depuis le diagramme de classe de mettre en évidence une ligne particulière d'un diagramme de séquence, les éléments affichés dans ce dernier étant beaucoup plus précis.

5.3.2 Naviguer entre les séquences

Le diagramme de séquence d'une méthode peut contenir une (ou plusieurs) invocation associée à une méthode connue par VisUML. Comme présenté partie 4.3.5 page 77, il est possible de régler le niveau de profondeur des appels dans le diagramme de séquence. Il est modifiable de façon globale, ou via l'utilisation de bouton « + » situé sur le message associé. En complément de ces possibilités, l'utilisateur peut effectuer un clic avec la touche *alt* enfoncée (alt+clic) sur l'un

de ces messages. Cette interaction va mettre à jour le diagramme afin d'afficher le détail de l'implémentation de la méthode sélectionnée, et ceci sans déplacer le curseur dans le code source et par conséquent sans modifier l'affichage du diagramme de classe. Par exemple, pour une classe possédant deux méthodes A et B, si la méthode A fait un appel à la méthode B, celle-ci sera affichée dans le diagramme de séquence sous la forme d'un message (lien). L'utilisateur peut alors faire un alt+clic sur ce message pour obtenir le diagramme de séquence de la méthode B. Cette interaction ajoute donc une navigation rapide depuis et vers un diagramme de séquence, sans action sur d'autres artefacts (code ou diagramme de classe). Elle est utile pour chercher un appel précis dans une grande méthode, ou simplement pour mieux comprendre comment fonctionne une méthode en particulier.

5.3.3 Appels parents de la méthode courante

Il arrive qu'en voyant le diagramme de séquence d'une méthode, l'utilisateur ait besoin de trouver tous les endroits où cette méthode est utilisée. Certains EDI fournissent des fonctions de recherches textuelles pour obtenir ce genre d'informations. Cependant, les interactions nécessaires pour lancer cette recherche sont souvent longues et non intuitives. L'affichage des résultats, sous forme de texte, est également inadapté pour cette tâche. La figure 5.8 présente une liste de ce type et contient toutes les utilisations de la classe « Entity » dans le projet. Les lignes encadrées de vert correspondent à un héritage. Cette information est noyée au milieu d'autres utilisations de la même classe, sans possibilité de filtrer ou de regrouper par types d'utilisations.

Dans VisUML, le diagramme de séquence peut afficher ces informations sous la forme de messages partant de la gauche (sans être sur une ligne de vie particulière) et allant sur la ligne de vie principale, avant l'exécution de la méthode. La figure 5.9 montre des messages de ce type. Dans cet exemple, la méthode *setPosition()* est appelée dans la méthode *setX* à la ligne 41 ainsi que dans la méthode *setY*, ligne 50.

Ces messages se comportent comme n'importe quel autre message du diagramme, et possèdent les mêmes interactions. Un clic va donc mettre à jour l'EDI sur la ligne associée, tandis qu'un alt+clic va afficher le détail de la méthode *parente*. Ces messages fournissent donc un moyen de naviguer entre les différents appels d'une méthode particulière et affichent des informations importantes, sans action requise par l'utilisateur. Ces informations ne perturbent pas la lecture de la méthode et sont adaptées au diagramme, puisqu'elles utilisent la même syntaxe graphique que les autres éléments.

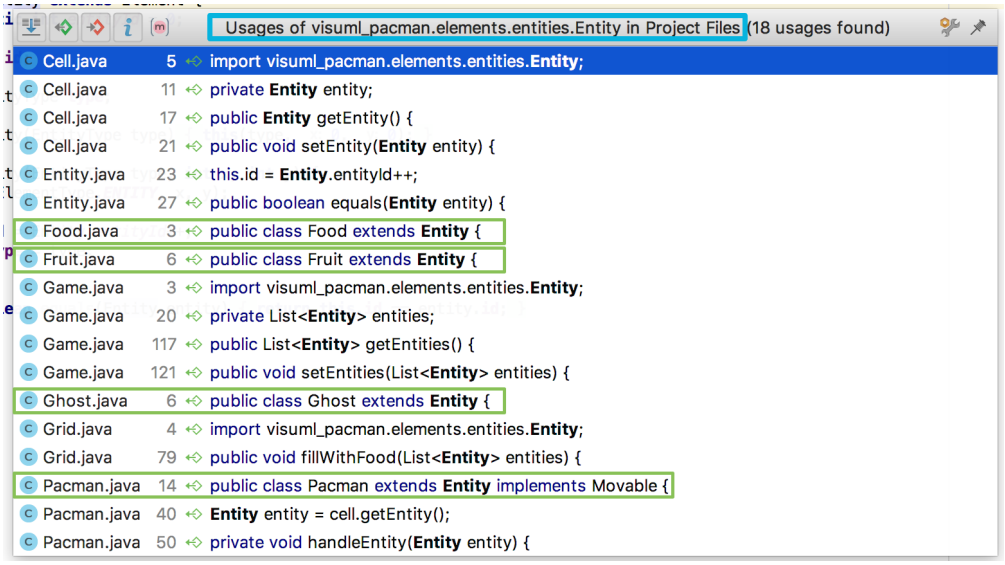


FIGURE 5.8 – Fonction "Find Usage" de IntelliJ, en vert les lignes correspondant à un héritage

5.3.4 Vue combinant les deux diagrammes

Les diagrammes possèdent chacun leur propre page, il est donc possible de voir simultanément plusieurs diagrammes, dans le cas où l'utilisateur possède plusieurs écrans. Cependant certains utilisateurs n'utilisent qu'un écran, ou ne souhaitent pas changer de fenêtre pour interagir avec l'un ou l'autre des diagrammes.

Pour palier ce problème, nous avons imaginé une vue contenant les deux types de diagrammes. Cette vue se découpe en trois parties, présentées figure 5.10.

La partie 1 de la figure contient deux aperçus en temps réel, le premier correspond au diagramme de classe, le second à celui de séquence. Ces *overviews* contiennent une vue miniature de leur diagramme, qui est interactive et permet une navigation rapide à n'importe quel endroit du diagramme.

La partie 2 contient le diagramme actif, soit le diagramme de classe, soit un diagramme de séquence. Ils agissent exactement comme les diagrammes présentés précédemment, et contiennent les mêmes informations. Il est possible de passer du diagramme de classe au diagramme de séquence simplement en passant la souris sur l'aperçu associé, dans la partie 1. Le changement de diagramme est instantané et prend tout l'espace de la partie 2.

La partie 3 contient un historique sous forme de miniatures des dix derniers diagrammes de séquence qui ont été générés par l'utilisateur. Ce dernier peut

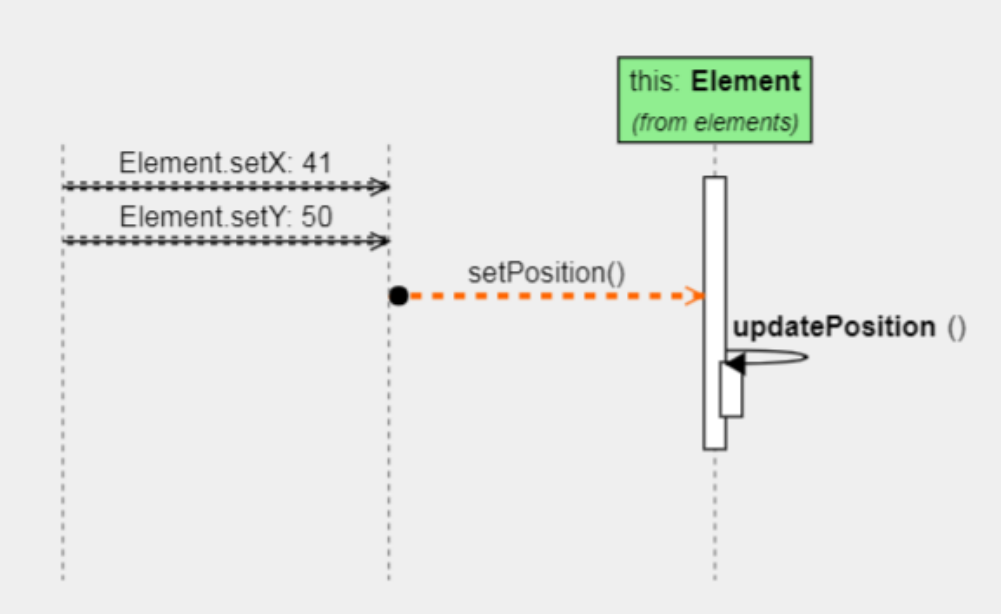


FIGURE 5.9 – Diagramme de séquence possédant deux appels parents

afficher le diagramme de séquence de la méthode associée en passant simplement sa souris sur la miniature, comme pour les aperçus de la partie 1. Nous avons choisi de gérer un historique de diagramme de séquence plutôt que de diagramme de classe car notre approche se base sur la découverte de code, et la navigation au sein des méthodes du projet. Un utilisateur va souvent parcourir plusieurs méthodes à la suite pour comprendre un point particulier ou à l'inverse pour découvrir les endroits où une méthode est appelée. Cet historique de séquence permet à l'utilisateur de « retourner en arrière » par rapport à sa navigation.

Grâce à ces éléments et interactions, les deux diagrammes sont contenus dans une seule page, et il est possible de passer de l'un à l'autre très rapidement et sans perdre le contexte (changement de fenêtres et temps d'adaptation). L'historique des diagrammes de séquences permet à l'utilisateur d'utiliser les mécanismes de navigation sans crainte de devoir passer du temps à retourner en arrière pour retrouver une information. Une fenêtre d'information contenant le nom de la classe et de la méthode associée apparaît sur la case d'historique lorsque l'utilisateur navigue dessus. Cette popup est montrée dans la figure 5.11

Cette représentation graphique ne constitue en soit qu'une première étape dans le but que nous cherchons à atteindre ici. Dans la version actuelle, cette vue correspond à une activité d'exploration du code dans un but de compréhension du code ouvert dans l'EDI ; ceci peut être associé à une sous-activité de l'activité

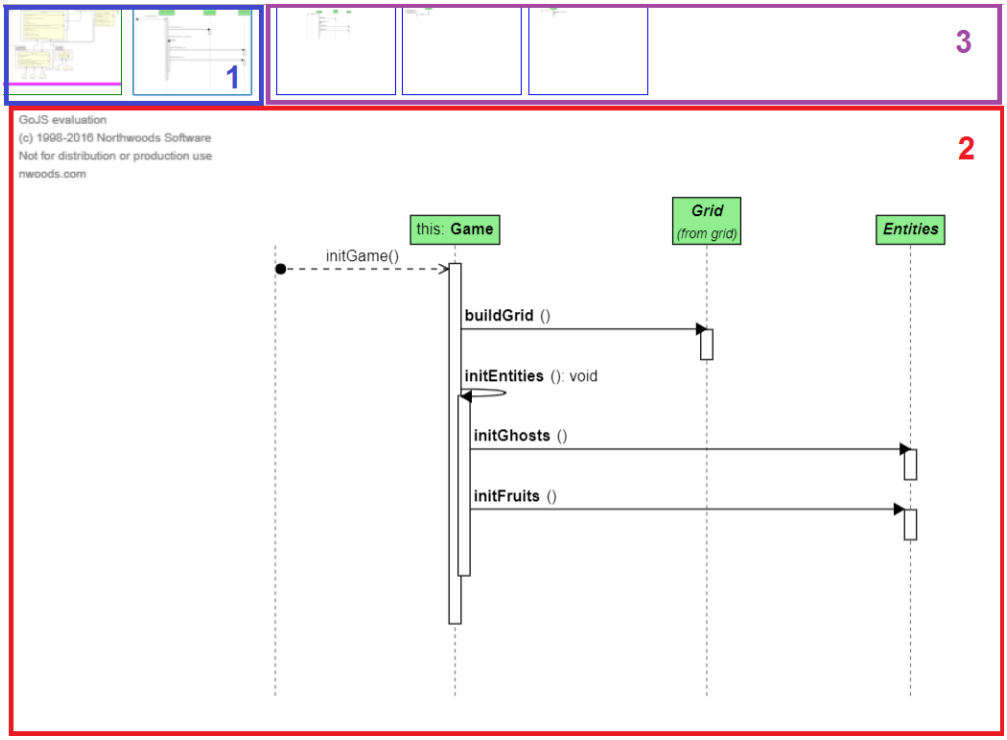


FIGURE 5.10 – Page web permettant une navigation simple entre les deux types de diagrammes

globale du développeur qui est constituée de beaucoup d'autres sous-activités (recherche de bug, *refactoring*, *merge*...). L'objectif à terme est de faire en sorte que VisUML permette de supporter l'activité du développeur dans sa totalité (voir chapitre 6.6 page 143).

5.4 Conclusion

Ce chapitre rassemble les interactions et navigations fournies dans VisUML. Il est découpé en trois parties, la première regroupant les interactions ayant pour source l'EDI, la seconde celles ayant pour source les diagrammes, et la troisième est composée des navigations internes aux diagrammes, sans interaction sur l'EDI.

Toutes les interactions ou navigations présentées dans ce chapitre ont pour but de simplifier l'utilisation des diagrammes générés, que ce soit par une synchronisation automatique des éléments à afficher ou via des interactions simples et rapides, permettant d'afficher des informations supplémentaires ou de naviguer vers un élément lié.

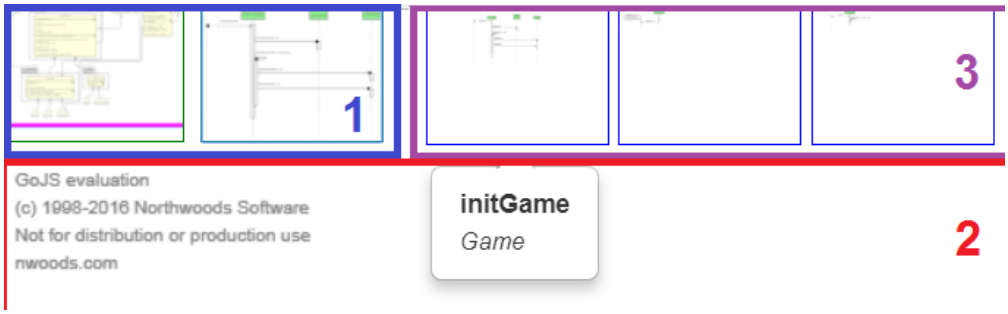


FIGURE 5.11 – Info-bulle d'information sur les aperçus des diagrammes de séquences

La première section du chapitre présente le mécanisme de synchronisation que nous avons utilisé pour VisUML. Les données du modèle caché de VisUML proviennent directement de l'EDI, et ne représente qu'un sous-ensemble des données du projet : les onglets ouverts et leurs relations.

En complément de cette synchronisation des données, VisUML maintient aussi à jour l'onglet actuellement actif sur l'EDI, et met en évidence ce dernier sur le diagramme de classe, en changeant sa couleur et la couleur des liens qui lui sont associés. Il en va de même pour le diagramme de séquence, qui va automatiquement afficher la méthode actuellement parcourue par l'utilisateur. Plus précisément, la ligne active (où est positionné le curseur) est également mise en valeur (couleur et épaisseur de la ligne représentant le message), et mise à jour en temps-réel.

Ces affichages automatiques permettent de réduire le temps nécessaire à la compréhension du diagramme, en mettant en avant les éléments actuellement actifs sur le code. Lors d'un passage du code au diagramme (et vice-versa), le développeur retrouve très rapidement les éléments mis en évidence et peut donc facilement continuer sa tâche sur l'une ou l'autre des représentations.

La deuxième section de ce chapitre regroupe les navigations implémentées au sein des diagrammes. De manière générale, il est possible de cliquer sur tous les éléments du diagramme (que ce soit celui de classe ou de séquence), afin de mettre en avant sur l'EDI le morceau de code lié. Cette interaction simple permet de passer encore plus rapidement du diagramme à l'EDI, sans avoir à interagir avec ce dernier. Il suffit au développeur de cliquer sur l'élément qui l'intéresse et ce dernier sera mis au premier plan. Le curseur du développeur sera également placé directement sur cet élément lorsque c'est possible, dans le cas d'un attribut ou d'une méthode pour le diagramme de classe, et dans tous les cas (ligne de vie, message, fragment) pour le diagramme de séquence.

Enfin, la dernière section présente les navigations possibles entre les dia-

grammes. Bien que ces derniers utilisent le code comme source de données, ils peuvent être utilisés sans avoir à naviguer dans l'EDI. Ainsi, le diagramme de séquence se met à jour automatiquement lors d'un clic sur une méthode dans le diagramme de classe. Le développeur peut, tout en ayant une visualisation globale de ses onglets ouverts, parcourir une méthode précise, simplement en cliquant sur cette dernière dans le diagramme de classe.

Il en va de même pour naviguer d'une méthode à l'autre dans le diagramme de séquence, une interaction simple (alt+clic) permet d'afficher le détail de la méthode souhaitée. Il est à noter que cette interaction ne met pas à jour l'EDI, mais qu'interagir avec un élément de cette nouvelle méthode, par exemple cliquer sur un message, va afficher le code correspondant dans l'EDI.

Afin de faciliter la navigation entre méthodes, nous avons également ajouté un nouveau type de message, représentant les appels parents à la méthode visualisée. Le développeur peut donc voir (sans aucune interaction) les différents emplacements où cette méthode est appelée, et afficher dans l'EDI le code associé s'il le souhaite (via un clic).

Enfin, l'un des derniers prototypes implémentés est une vue regroupant les deux diagrammes (classe et séquence). Cette vue contient un aperçu des deux diagrammes, ainsi qu'une interaction simple (passage de la souris sur un aperçu) pour naviguer de l'un à l'autre. De plus, cette vue conserve un historique des dix derniers diagrammes de séquences générés, qui peuvent ensuite être mis en avant via cette même interaction. Cette vue permet de limiter le nombre de fenêtres nécessaires à la visualisation de l'ensemble des parties d'un projet (macro et micro).

Le chapitre suivant présente l'évaluation que nous avons effectuée pour valider le travail sur VisUML.

Chapitre 6

VisUML : Evaluation

Sommaire

Nous avons élaboré VisUML en nous basant entre autres sur nos connaissances issues de nombreuses rencontres avec des professionnels lors de nos activités d'enseignement et de recherche. VisUML étant à présent parvenu à un stade suffisamment abouti pour être utilisable en condition professionnelle, nous avons voulu savoir s'il répondait bien aux attentes, aux besoins et aux contraintes des développeurs. Ce chapitre présente l'évaluation que nous avons conduite dans ce but avec des professionnels. Afin de ne pas surcharger les personnes évaluant l'outil et pour évaluer correctement les fonctionnalités, nous avons été contraints de réduire le nombre de fonctions présentées durant cette évaluation. Ainsi, dans ce chapitre, l'évaluation de VisUML se concentre sur les pages web des diagrammes de classe et de séquence, ainsi que l'analyseur de code. Des évaluations futures (voir 6.6 page 143) porteront sur les modules pour outils UML (tels que Papyrus et GenMyModel), les filtres et les tags.

6.1 Protocole expérimental

Les participants à cette évaluation sont des développeurs qui connaissent Java et UML. Etant donné que leurs connaissances et pratiques sont potentiellement très variées, elles peuvent introduire des biais. Pour éviter ces biais, nous avons fait le choix de nous reposer sur des niveaux très basiques pour Java et UML Concernant Java, seule une connaissance de base (par exemple une formation universitaire de type Licence) suffit car le projet utilisé durant l'évaluation ne contient que peu de lignes de code et d'un niveau de complexité faible. Pour l'aspect UML, nous nous sommes reposés également sur des connaissances de

base de type universitaire, et uniquement sur les diagrammes de classe et de séquence.

L'élaboration du protocole nous a également obligé à restreindre les tâches à réaliser durant l'évaluation. Il aurait été difficile d'évaluer VisUML si les tâches demandées étaient trop en relation avec le niveau d'expertise des développeurs. Par exemple, trouver ou corriger un bug fait appel à des connaissances qui dépendent fortement du développeur, et donc pas seulement de notre outil. Aussi, nous nous sommes concentrés sur la tâche la plus courante pour un développeur : la découverte d'informations (problématique que VisUML tente de résoudre). Malheureusement cette tâche est encore très dépendante du niveau du développeur et de ses expériences passées. Il est donc compliqué de mesurer quantitativement des actions de ce type.

Suite à ces constats, nous avons opté pour une évaluation qualitative de notre outil, tout en récupérant par ailleurs des donnés quantitatives (datation des actions, capture vidéo des écrans, des déplacements de la souris et des interactions). L'objectif des participants était de trouver des informations sur le projet test, de la manière dont il le souhaitait, sachant que VisUML fonctionnait durant toute l'expérience et était donc accessible à tout moment.

Le projet Java utilisé par les participants est un jeu (Pacman) dont les méthodes n'ont pas toutes été développées. Il contient quatre *packages*, dix classes, deux énumérations et une interface. Le code du projet est disponible en annexe F page 208.

Nous étions présents durant toute l'expérience pour répondre aux questions des participants. Dans le même temps, nous prenions des notes sur les réponses et la façon dont les participants allaient chercher les informations. Chaque participant était également chronométré et les écrans enregistrés afin d'étudier plus précisément les interactions à la suite de l'expérimentation. Pour la saisie de commentaires par les participants à la fin de la session, nous laissions ces derniers seuls afin de ne pas fausser les réponses.

Le protocole se découpe en plusieurs étapes (détaillées ci-après) :

1. Questionnaire de niveau sur le développeur
2. Vidéo de présentation de VisUML
3. Réponses aux éventuelles questions du participant
4. Présentation de l'environnement de travail
5. Evaluation, participant libre et questions
6. Questionnaire de fin

6.2 Questionnaire de niveau

Afin de moduler les résultats des participants et de définir un profil, nous avons fourni un questionnaire (voir annexe E page 195) contenant des questions simples sur le profil général du participant, son poste de travail, la création de schémas et de diagrammes UML. Les numéros des sous-sections suivantes font référence aux numéros des questions du formulaire trouvé en annexe.

Au total, vingt personnes (toutes sont des développeurs professionnels) ont passé l'évaluation. Dix-sept se sont déplacées à l'Université, et pour les trois autres nous nous sommes rendus dans les locaux de l'entreprise. La configuration était la même dans tous les cas.

6.2.1 Q1.2 - Métier du participant

La première question du questionnaire porte sur le métier actuel du participant. Ce dernier a le choix parmi une liste des métiers les plus communs en informatique, avec la possibilité de répondre librement. L'objectif de cette question est de déterminer si un participant a plus souvent besoin de passer du temps sur de l'analyse ou de la documentation par exemple.

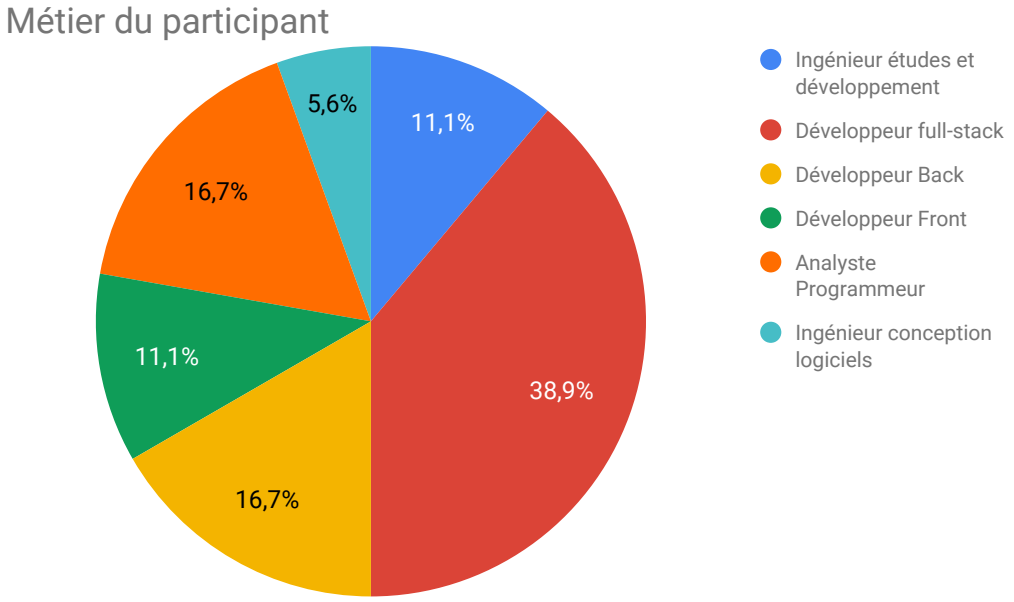


FIGURE 6.1 – VisUML Q1.2 - Métier du participant

La figure 6.1 montre la répartition des métiers parmi les participants de

l'évaluation. On remarque que 88,9% (somme de full-stack, back, ingénieur études et développement, ingénieur conception logiciels et analyse programmeur) des participants ont un poste orienté vers le *back-end*, tandis que les 11,1% restants sont des développeurs *front-stack*. Il est à noter également que 16,7% se désignent comme « Analyste Programmeur », avec donc une partie de leur temps de travail attribué à de l'analyse.

6.2.2 Q1.3 - Activités principales

La question suivante porte sur les activités principales du participant. Ce dernier doit estimer ses compétences sur un niveau de 1 (niveau faible) à 5 (niveau élevé), pour chacune des quatre activités suivantes :

— Analyse
— Développement de nouvelles fonctionnalités
— Tests
— Correction de bugs

L'objectif de cette question est de mettre en avant la compétence pour laquelle un participant s'estime le plus expert, afin de moduler les résultats de l'évaluation.

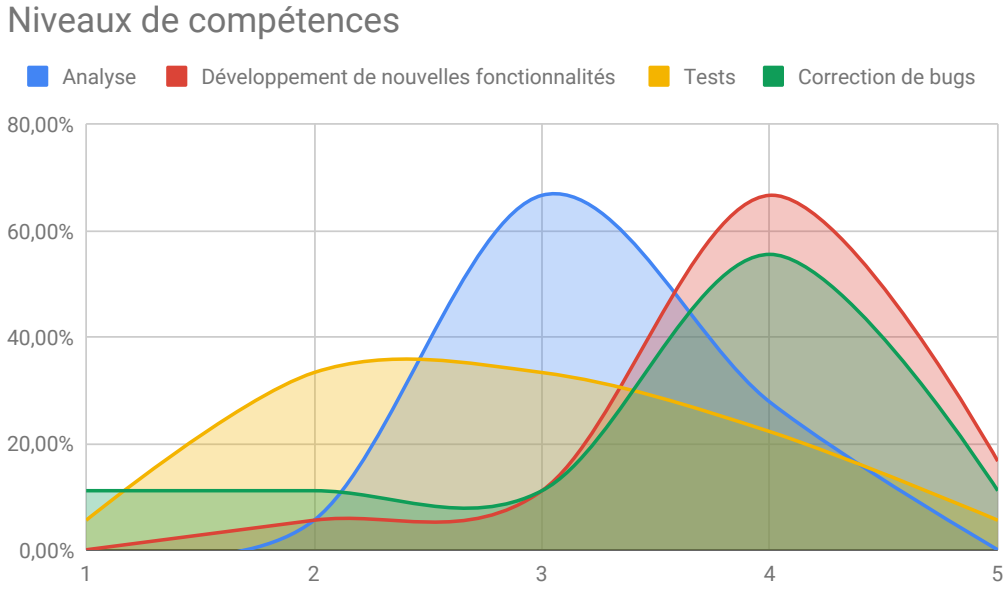


FIGURE 6.2 – VisUML Q1.3 - Niveaux de compétences des activités principales

La figure 6.2 représente les courbes de niveaux des activités principales

d'un développeur. On remarque que les participants s'estiment plus compétents sur le développement de nouvelles fonctionnalités (66,67% au niveau 4) et la correction de bugs (55,56% au niveau 4). L'analyse vient ensuite, avec 66,67% des participants situés au niveau 3. Les tests sont quant à eux moins bien estimés avec 33,33% aux niveaux 2 et 3 (soit les deux tiers des participants).

6.2.3 Q1.4 - Poste de travail

La question suivante porte sur l'installation du poste de travail du participant. Ce dernier peut choisir parmi un liste de choix ou indiquer librement sa réponse. L'objectif de cette question est de déterminer si la majorité des développeurs travaillent avec plusieurs écrans, et de distinguer les grands écrans des écrans de portable par exemple. Dans le cadre de l'évaluation, les participants utilisaient deux grands écrans et devaient naviguer entre ces derniers.

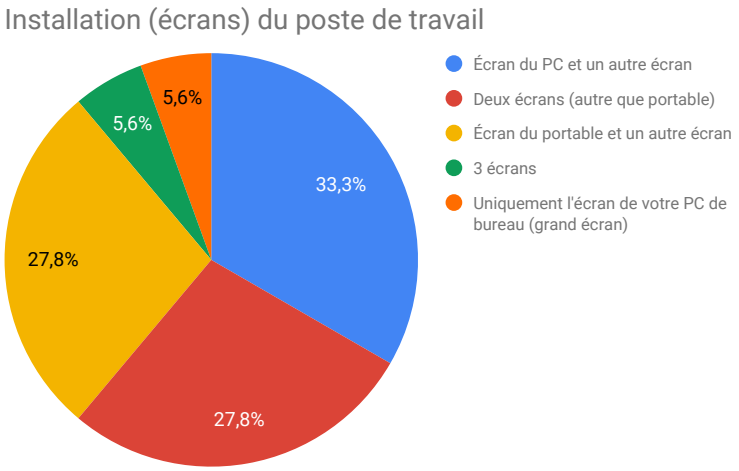


FIGURE 6.3 – VisUML Q1.4 - Installation du poste de travail du participant

La figure 6.3 présente les résultats de cette question. On remarque que les deux tiers (66,7%) des participants utilisent deux grands écrans ou plus. 278% utilisent un grand écran en plus de l'écran de leur portable, tandis que les 56% restant ne possèdent qu'un seul grand écran. Ce résultat est positif pour VisUML, dont les visualisations sont plus efficaces lorsqu'elles sont affichées dans un grand conteneur (il est plus facile de parcourir les éléments et de naviguer au sein des diagrammes).

6.2.4 Q1.5 - Onglets ouverts

La question suivante demande aux participants d'indiquer le nombre (parmi des intervalles) d'onglets ouverts en moyenne dans leur EDI. L'objectif de cette question est d'estimer le nombre moyen d'onglets ouverts par les participants, et donc, dans le cadre de VisUML, le nombre de classes qui apparaîtraient sur un diagramme.

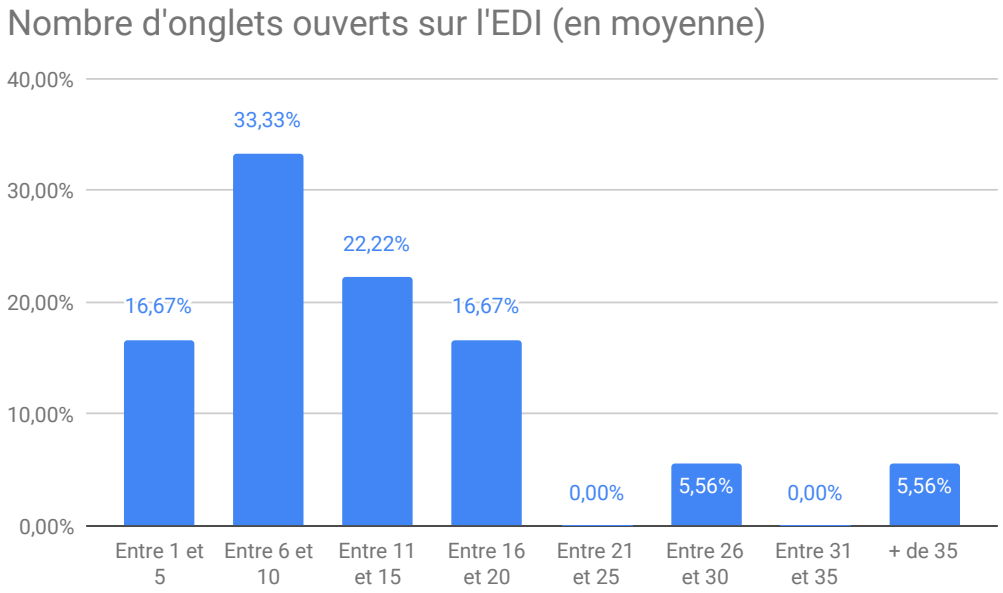


FIGURE 6.4 – VisUML Q1.5 - Nombres d'onglets ouvert en moyenne dans l'EDI

La figure 6.4 présente les résultats de cette question. On remarque que 50% des participants n'ouvrent pas plus de dix onglets en même temps (cinq et moins pour 16,67%). 38,89% en ouvrent entre onze et vingt au maximum. Les 11,12% restant dépassent les vingt-six onglets ouverts en moyenne. Bien que 50% se contentent de dix onglets, l'autre majorité en utilise jusqu'à trente-cinq et plus. Un diagramme de classe contenant dix éléments est déjà relativement difficile à lire et le placement automatique des éléments génère parfois des diagrammes très grands et donc difficilement navigables. L'utilisation de VisUML (ou d'un autre outil de visualisation de code) sur autant d'éléments risque de produire des résultats moins efficaces. Il est également intéressant de comparer le nombre moyen d'onglets ouverts avec les nouvelles fonctionnalités des EDI, ces derniers limitant automatiquement le nombre d'onglets ouverts. Par exemple, IntelliJ, sans configuration particulière, limite par défaut à dix le nombre d'onglets ouverts simultanément, en fermant automatiquement l'onglet le moins utilisé.

6.2.5 Q1.6 - Création et utilisation de schémas

La question suivante, et dernière de la première étape, demande aux participants s'ils créent des schémas dans leur métier.

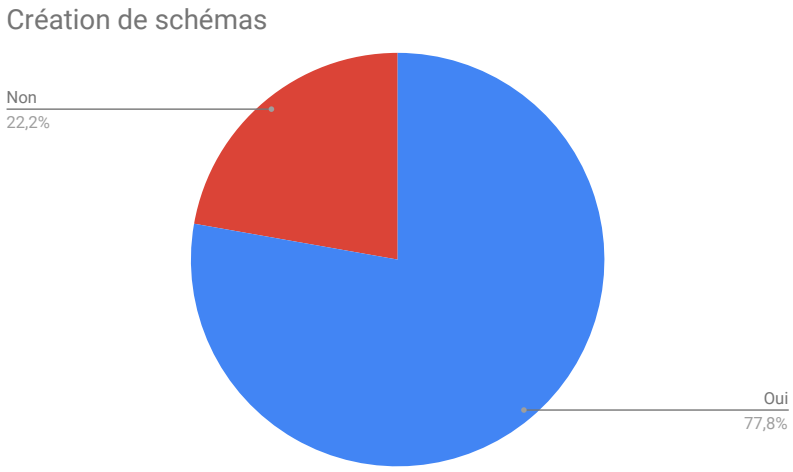


FIGURE 6.5 – VisUML Q1.6 - Utilisations des schémas

La figure 6.5 montre que 77,8% des participants créent et utilisent des schémas pendant leur travail.

6.2.6 Q1.7 - Types de schéma

La première question de la partie « Schémas » demande aux participants les types de schémas qu'ils utilisent dans leur travail. Ils ont le choix parmi une liste des types les plus communs, et peuvent également ajouter librement une réponse. La liste fournie est la suivante :

— Maquettes écran
— Base de données
— Architecture (type client-serveur)
— Diagramme de classe
— Diagramme de séquence
— Diagramme de cas d'utilisation

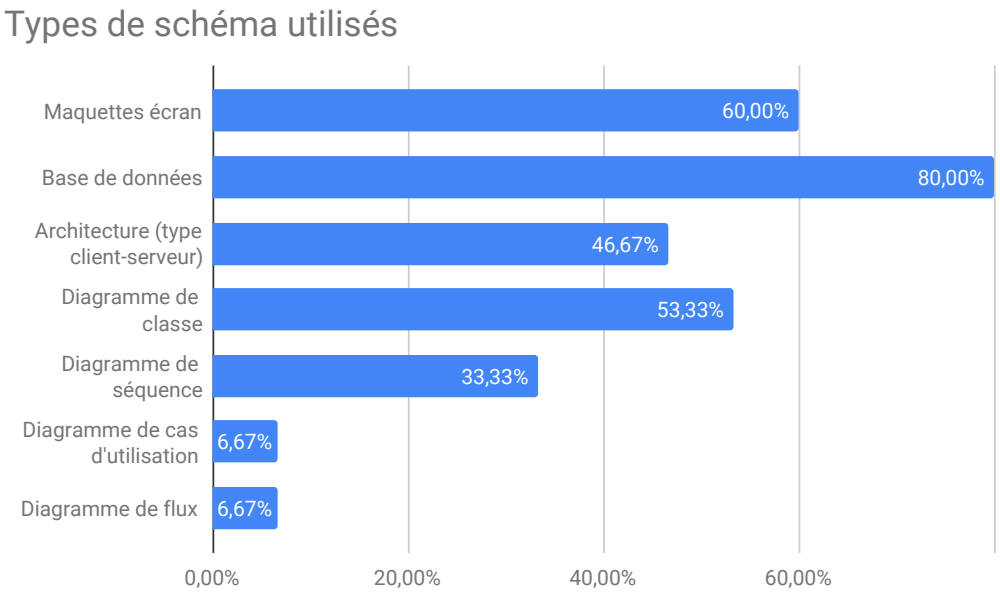


FIGURE 6.6 – VisUML Q1.7 - Types de schéma utilisés

La figure 6.6 montre que 80% des participants utilisent des diagrammes de base de données, 60% des maquettes écrans et 46,67% des diagrammes de type client-serveur. Pour cette évaluation, les deux diagrammes qui nous intéressent le plus sont le diagramme de classe, utilisé à 53,33% par les participants et le diagramme de séquence (33,33%). Ces nombres sont en phase avec les données recueillies par les différents sondages et études de la littérature.

6.2.7 Q1.8 à 14 - Supports d'utilisation des schémas

Les questions 8 à 14 du questionnaire portent sur les supports utilisés lors de la création des différents types de schémas. Pour chaque type qu'ils ont précédemment sélectionné, les participants peuvent indiquer s'ils utilisent le papier, un tableau, un outil informatique ou autre chose (champ libre).

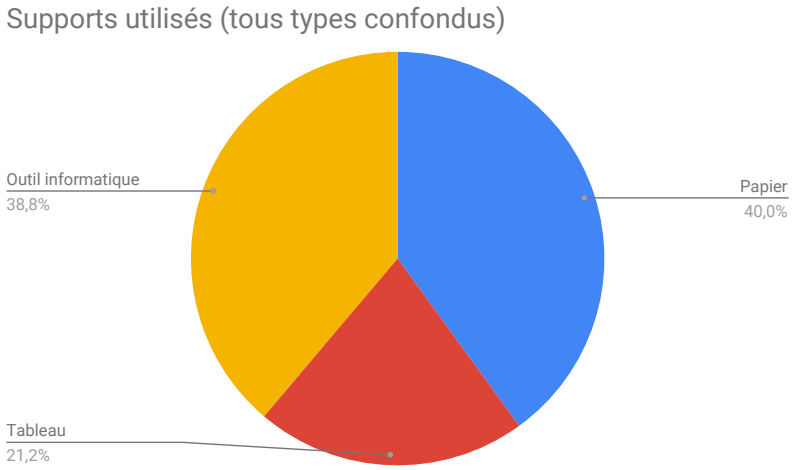


FIGURE 6.7 – VisUML Q1.8 à 14 - Supports d'utilisation des schémas

La figure 6.7 nous montre que le papier et les outils informatiques sont, en moyenne, utilisés à (sensiblement) la même proportion (respectivement 40% et 38,8%) pour créer des schémas (tous types confondus). Le tableau est également utilisé, mais uniquement pour 20% des participants.

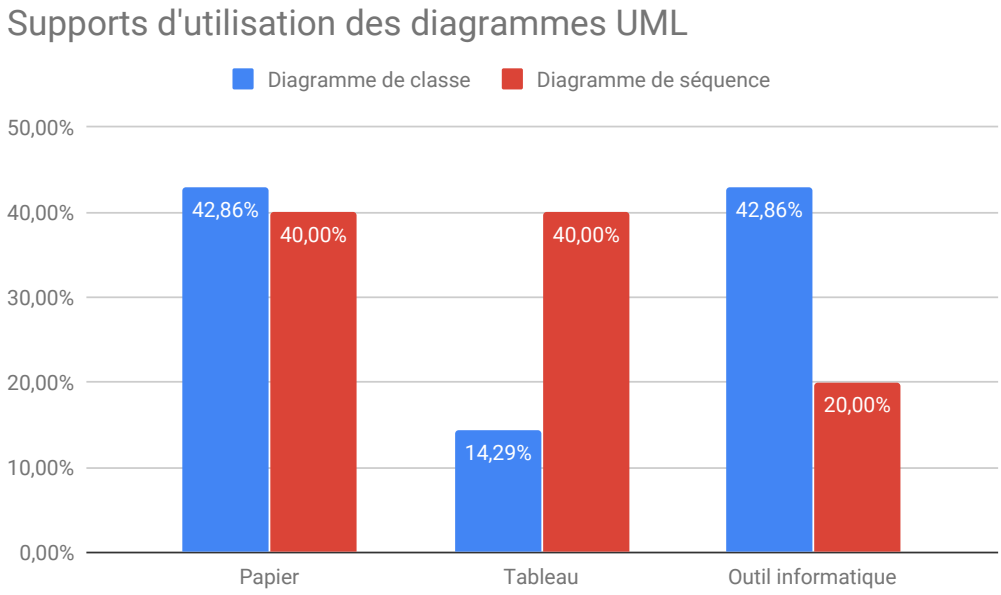


FIGURE 6.8 – VisUML Q1.11 et 12 - Supports d'utilisation des diagrammes de classe et de séquence

La figure 6.8 page précédente quant à elle, se concentre sur les diagrammes UML utilisés dans VisUML. On constate que, pour le diagramme de classe, papier et outil informatique sont utilisés de manière similaire, tandis que le tableau est en retrait. A l'inverse, le diagramme de séquence est plutôt créé sur papier ou tableau, tandis que l'utilisation d'un outil informatique est plus faible. Cette valeur peut être expliquée par le niveau de difficulté de création d'un tel diagramme sur les outils actuels.

6.2.8 Q1.15 à 17 - Réutilisation des schémas produits

Les questions 15 à 17 du questionnaire portent sur la réutilisation des schémas terminés. La Q1.15 demande aux participants s'ils réutilisent leurs schémas, avec modification ou non. Les questions 16 et 17 sont des réponses libres permettant aux participants d'indiquer comment ils les utilisent, ou, le cas échéant, comment ils aimeraient les utiliser.

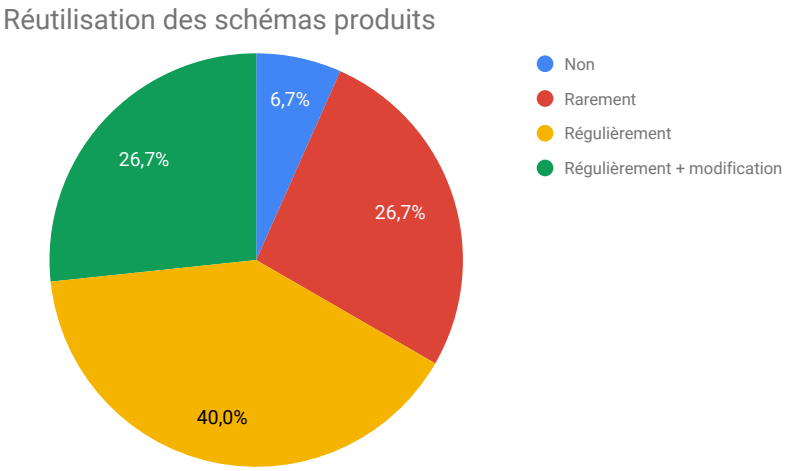


Figure 6.9 – VisUML Q1.15 - Réutilisation des schémas produits

La figure 6.9 montre que 93,3% réutilisent, même si ce n'est que rarement, les schémas produits, indiquant qu'ils n'ont pas été produits pour rien. 40% les réutilisent régulièrement dans leur tâche de travail, et 26,7% vont même les modifier afin d'avoir un diagramme toujours à jour.

La majorité des participants (66,7%) utilise ces schémas créés comme base pour l'implémentation du code, avec des modifications si nécessaire pour 10% d'entre eux. Les schémas sont également utilisés comme documentation et support de réflexion pour 33,3%. Enfin, 33,3% des participants indiquent également

les utiliser comme support de communication ou lors de transmission de connaissances.

Les participants ne réutilisant pas les schémas souhaiteraient pouvoir les importer automatiquement dans un logiciel de modélisation (50%) ou les annexer et permettre une recherche par mots-clés (50%).

6.2.9 Q1.18 - Utilisation d'UML

La question 18 demande aux participants s'ils utilisent le langage UML dans le cadre de leur activité professionnelle. Nous avons voulu insister sur le fait que même une très faible utilisation de ce langage était suffisant pour répondre aux questions.

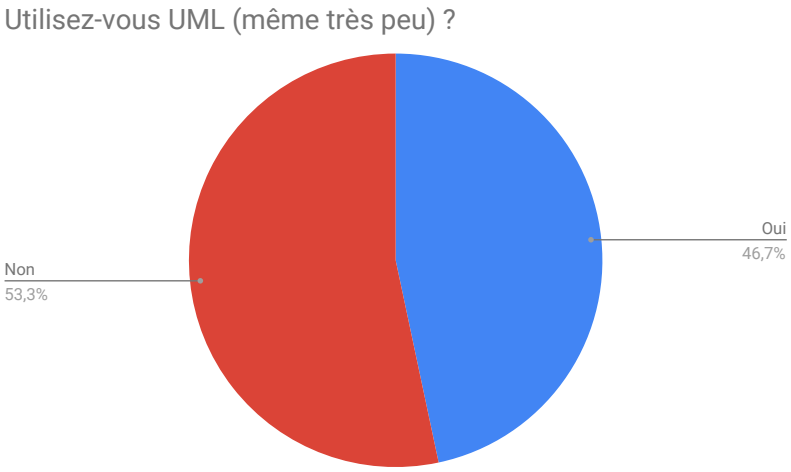


Figure 6.10 – VisUML Q1.18 - Utilisation d'UML

La figure 6.10 représente les pourcentages d'utilisation de UML parmi les participants. On constate que moins de la moitié utilise ce langage pendant leur travail. Ce nombre est également en accord avec les études de la littérature sur la modélisation au sein des entreprises. Cette méthodologie n'est pas encore présente dans toutes les équipes, et peut paraître trop coûteuse en temps pour être adoptée.

6.2.10 Q1.19 - Niveau de connaissance d'UML

La question 19 vise à estimer le niveau de connaissance moyen d'UML des participants. Ces derniers doivent indiquer leur niveau sur une échelle de 1 (débutant) à 8 (expert).

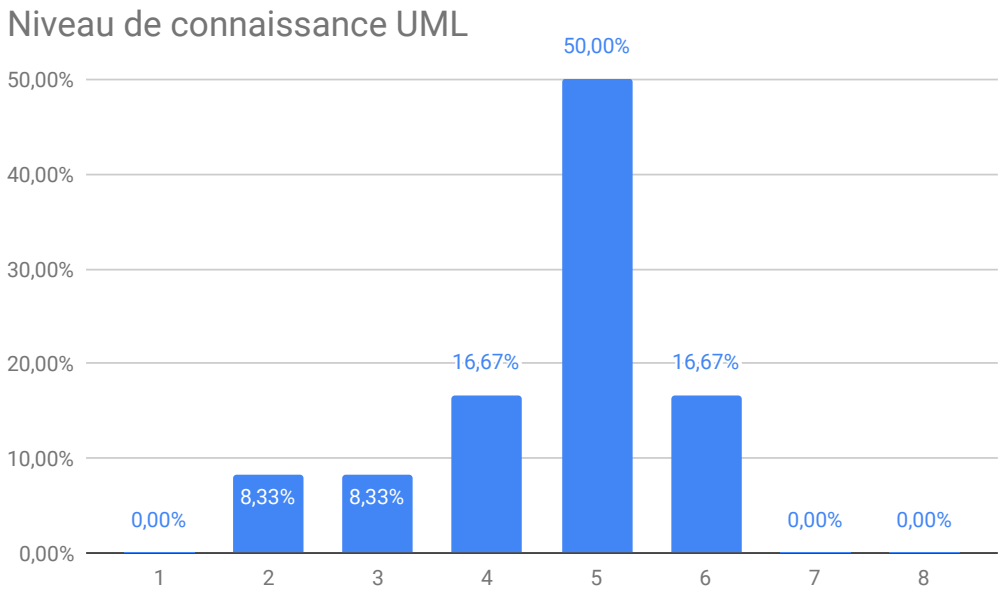


FIGURE 6.11 – VisUML Q1.19 - Niveau de connaissance d'UML

La figure 6.11 montre que la majorité (50%) des participants estime leur niveau de connaissance à 6/8, ce qui est un bon niveau. Un tiers (33,3%) estime son niveau entre 2 et 4, et les 16,7% restants au niveau 6. La moyenne du niveau se situe donc à 4,58 sur 8.

6.2.11 Q1.20 - Fréquence d'utilisation d'UML

La question 20 vise à estimer la fréquence d'utilisation d'UML. Comme pour la question précédente, les participants doivent indiquer leur fréquence d'utilisation sur une échelle de 1 (jamais) à 8 (tous les jours).

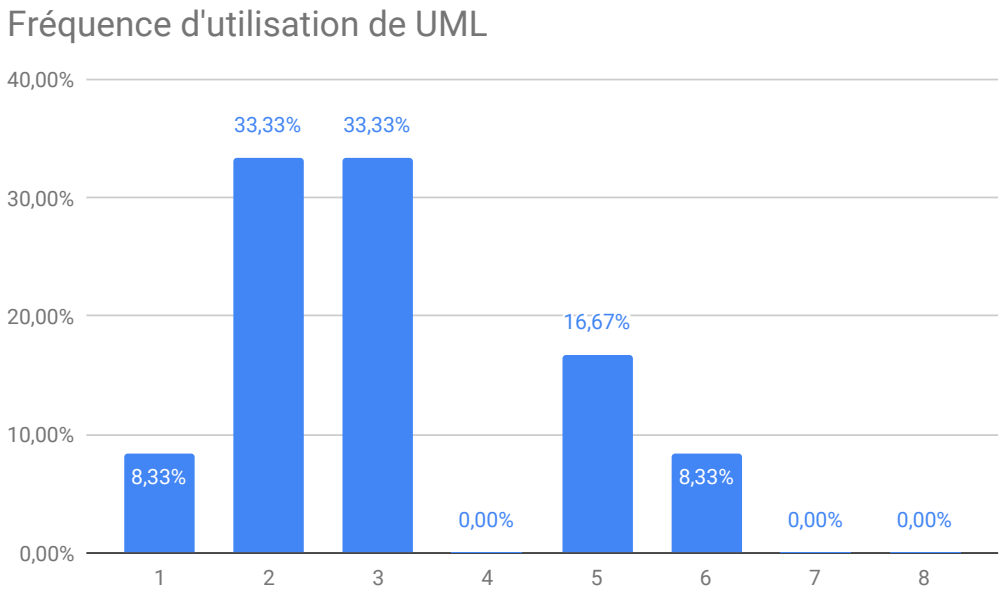


FIGURE 6.12 – VisUML Q1.20 - Fréquence d'utilisation d'UML

La figure 6.12 montre que, bien que les participants estiment leur connaissance à un niveau élevé, la fréquence à laquelle ils utilisent UML reste très faible. 75% des participants utilisent UML à une fréquence inférieure ou égale à 3 (sur 8), tandis que les 25% restants l'utilisent à une fréquence maximum de 6 sur 8. La principale raison à cette faible fréquence, d'après les retours des participants, est le manque de temps pour créer ces diagrammes.

6.2.12 Q1.21 - Type d'utilisation d'UML

La question 21 vise à estimer la façon dont les participants utilisent UML. Cette utilisation peut être informelle, à des fins de communication ou de documentation, ou très formelle, afin de valider une analyse et/ou de générer une partie du code. Comme pour les questions précédentes, les participants doivent indiquer leur type d'utilisation sur une échelle de 1 (informelle) à 8 (formelle).

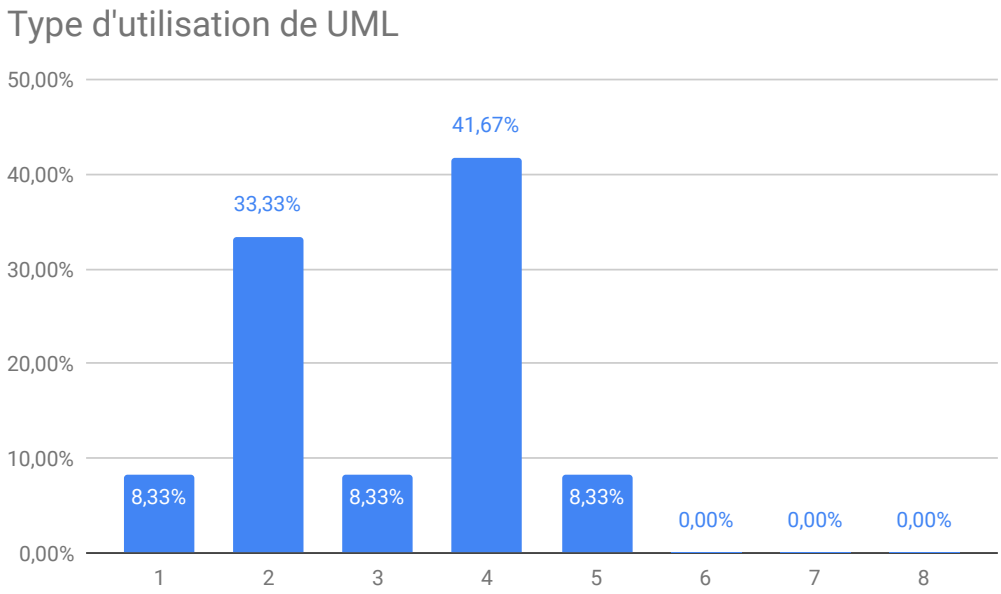


FIGURE 6.13 – VisUML Q1.21 - Type d'utilisation d'UML

La figure 6.13 nous montre que tous les participants utilisent UML de manière informelle, la majorité d'entre eux à un niveau 4/8, suivi de près par le niveau 2. Cette donnée peut encore une fois être liée aux études de la littérature sur l'utilisation faite d'UML, de manière informelle et pour des objectifs de documentation ou de communication.

6.2.13 Q1.22 à 25 - Diagrammes UML

Les questions 22 à 24 demandent aux participants d'indiquer jusqu'à trois diagrammes UML qu'ils utilisent le plus souvent. Pour chaque question, ils peuvent cocher un des quatorze diagrammes UML, ainsi que le choix « Aucun » s'il n'en utilise qu'un, par exemple. L'objectif est de nous permettre de vérifier nos choix de diagrammes, et de confronter les réponses des participants aux résultats que l'on peut trouver dans la littérature.

La question 25 leur permet par la suite d'indiquer les autres diagrammes qu'ils utilisent pendant leur activité.

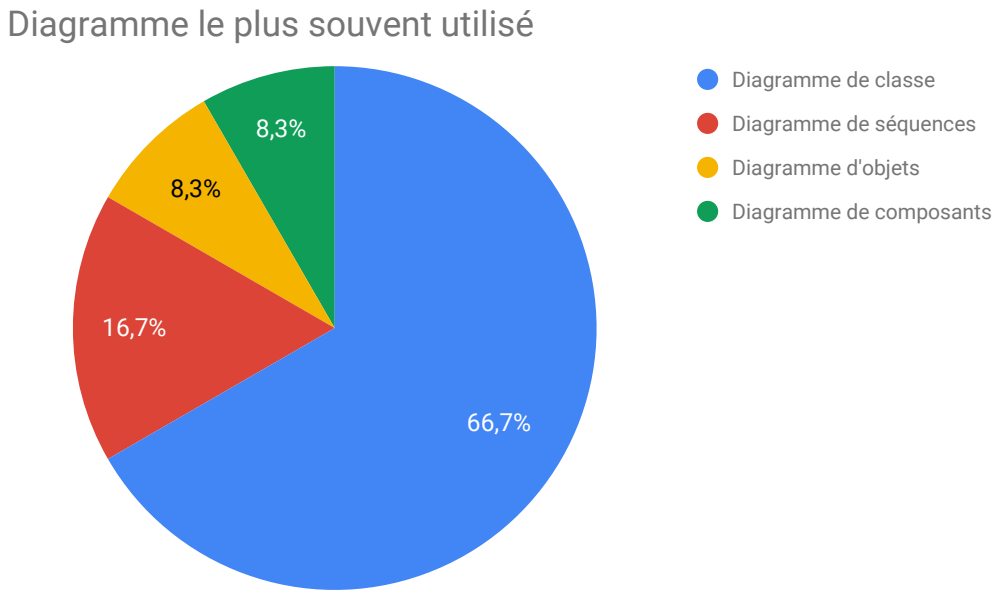


FIGURE 6.14 – VisUML Q1.22 - Diagramme UML le plus souvent utilisé

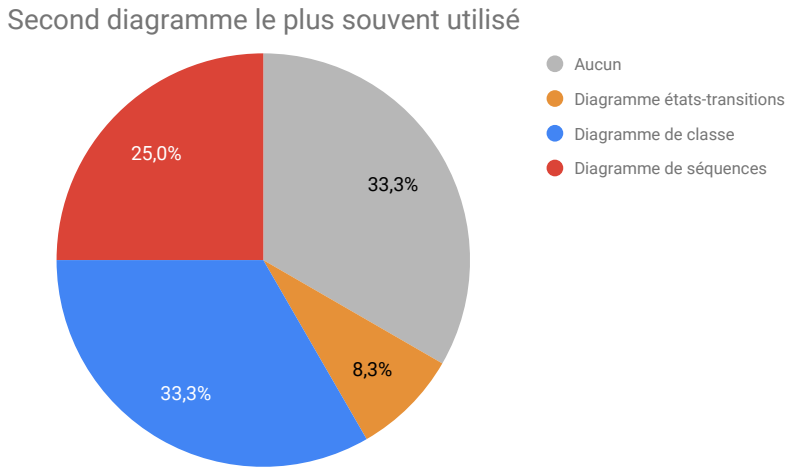


FIGURE 6.15 – VisUML Q1.23 - Second diagramme UML le plus souvent utilisé

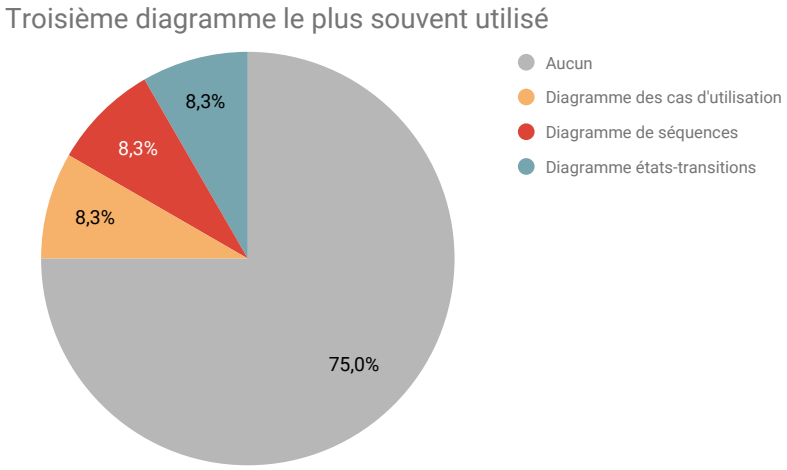


Figure 6.16 – VisUML Q1.24 - Troisième diagramme UML le plus souvent utilisé

Les figures 6.14 page précédente, 6.15 page ci-contre et 6.16 nous montrent que le diagramme de classe est très majoritairement utilisé le plus souvent. Deux tiers des participants le placent en première position, tandis qu'un tiers le place en deuxième. Le diagramme de séquence arrive ensuite, avec 16,7% en tant que plus utilisé, puis 25% en tant que deuxième plus utilisé.

On remarque également que 33.3% des participants n'utilisent qu'un diagramme, et que seuls 25% en utilisent jusqu'à trois.

Seuls trois participants ont indiqué utiliser d'autres diagrammes (Q25), parmis lesquels le diagramme de séquence (en quatrième position), d'activité et de package.

6.2.14 Q1.26 - Taille des projets

La question 26 vise à estimer la taille moyenne des projets sur lesquels travaillent les participants. Ces derniers doivent estimer la taille de leurs projets sur une échelle de 1 (petit projet) à 8 (très grand projet).

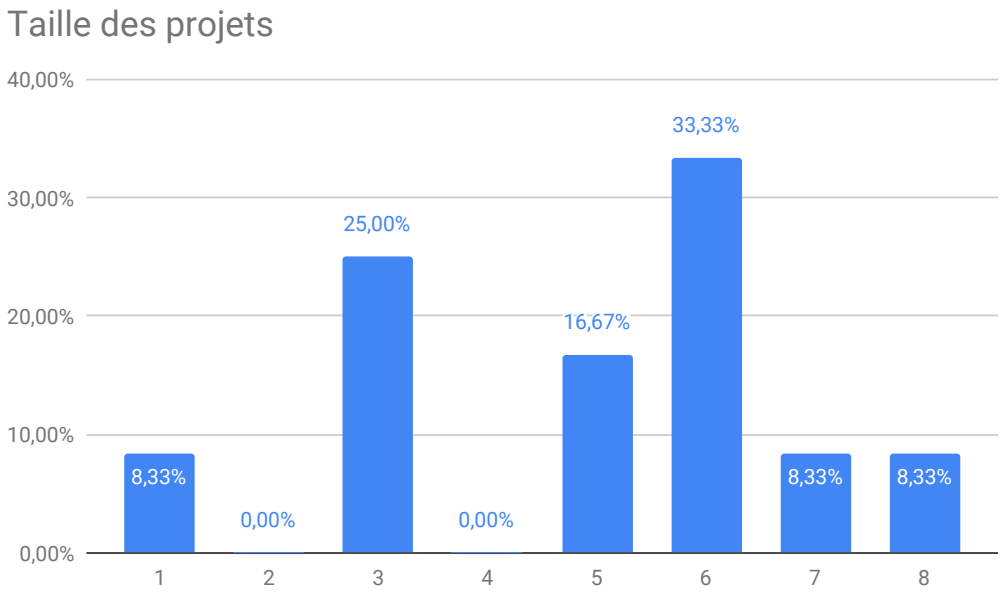


FIGURE 6.17 – VisUML Q1.26 - Taille des projets

La figure 6.17 montre que deux tiers des participants travaillent sur des projets de moyenne à très grande taille. De tels projets impliquent plusieurs collaborateurs et une communication essentielle au bon déroulement du projet. Le tiers restant travaille sur des projets de taille moyenne, voir très petite.

6.2.15 Q1.27 - Modélisation collaborative

La question 27 demande aux participants à combien ils modélisent leurs projets, de façon collaborative. Chaque participant avait le choix entre les valeurs 1, 2 et 3+. La question est à choix multiple, et certains participants ont donc coché plusieurs valeurs.

Un mode collaboratif pour VisUML a été imaginé (et présenté précédemment), mais non développé entièrement. Cette question nous permet de savoir si une telle fonctionnalité serait bénéfique pour les utilisateurs.

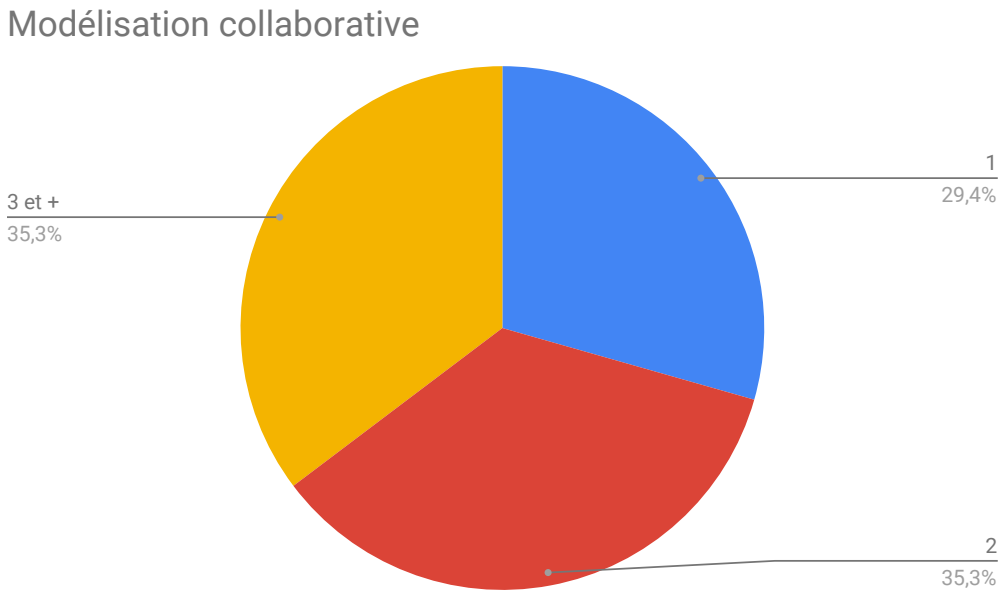


FIGURE 6.18 – VisUML Q1.27 - Modélisation collaborative

La figure 6.18 représente la répartition de la façon dont les participants collaborent lors d'une phase de modélisation. Bien que les nombres soient très proches, on remarque que seul un tiers des participants modélise tout ; ceci peut s'expliquer par l'usage habituel d'UML pour la documentation et la communication dont plutôt en groupe.

6.2.16 Q1.28 à 30 - Supports de modélisation

Les questions 28 à 30 visent à définir les supports et outils les plus utilisés par les participants pour modéliser. Les objectifs de ces questions sont de voir si les outils informatiques sont utilisés ou si les participants préfèrent modéliser sur un support physique (papier ou tableau). Elles permettent également de trier les outils de modélisation par ordre d'utilisation.

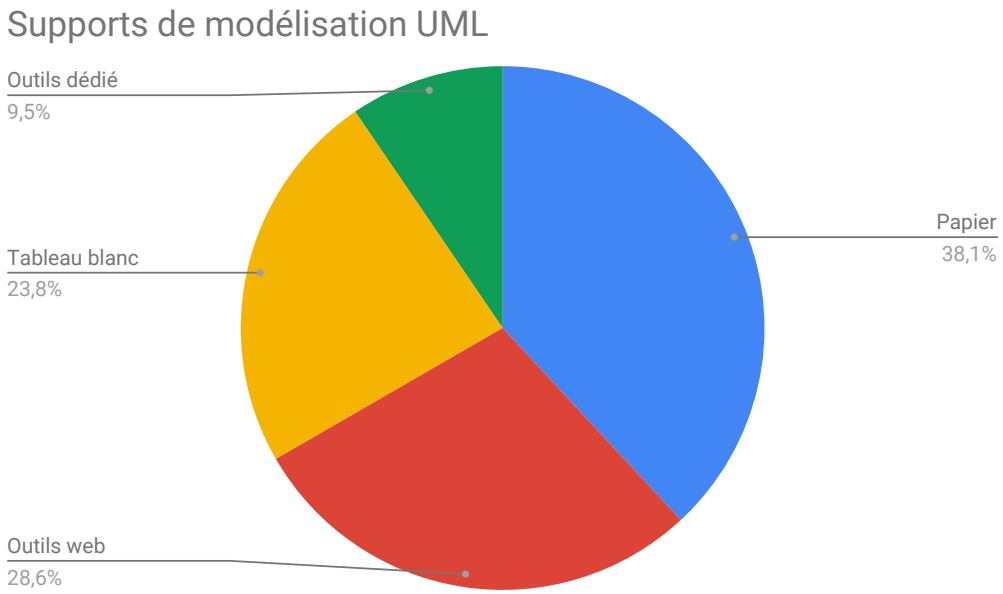


FIGURE 6.19 – VisUML Q1.28 - Supports de modélisation

La figure 6.19 représente la répartition des supports de modélisation. On remarque que l'utilisation du papier est majoritaire avec 38,1%. Les outils web viennent ensuite avec 28,6%, puis l'utilisation du tableau blanc avec 23,8%. En dernière position se trouvent les outils dédiés, avec 9,5%. Cumulés, les supports physiques (papier et tableau blanc) représentent 61,9% des supports utilisés. Cela montre que les outils informatiques sont encore trop peu utilisés, que ce soit parce qu'ils sont trop complexes ou simplement trop peu efficaces en terme d'interactions. Enfin, les outils dédiés à la modélisation sont en dernière position, ce qui est surprenant étant donné que la création et modification de diagrammes sont leurs fonctionnalités principales. Il serait donc intéressant de définir exactement les raisons qui poussent les utilisateurs à préférer le papier aux outils.

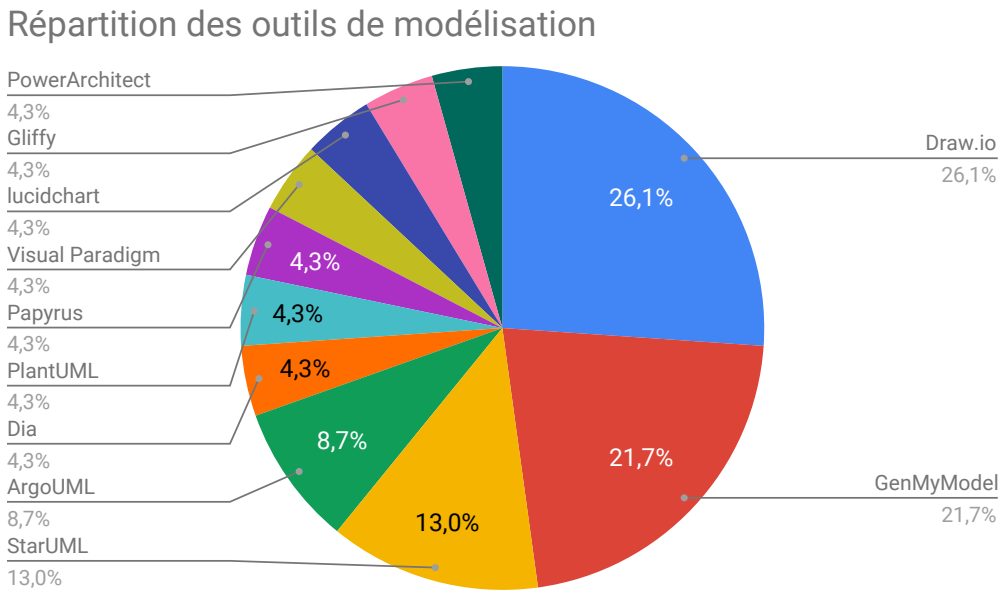


FIGURE 6.20 – VisUML Q1.29-30 - Outils de modélisation

La figure 6.20 représente la répartition des outils de modélisation. On remarque que Draw.io, GenMyModel et StarUML forment la majorité des outils utilisés (60,8%) et sont des outils web (ce qui est conforme aux résultats précédents). Il est intéressant de noter que GenMyModel, qui est un outil de modélisation mais sur le web, arrive en seconde position, ce qui peut montrer qu'un outil de modélisation possédant un support et des interactions simples peut être apprécié des utilisateurs. Il est également à noter que GenMyModel propose une fonction de collaboration poussée, ce qui peut jouer en sa faveur. Les logiciels dédiés arrivent en fin de classement, avec très peu d'utilisateurs (4,3%).

6.3 Vidéo d'introduction à VisUML

Suite à ce questionnaire, le participant visionnait une courte vidéo introductive à VisUML [1]. Cette vidéo regroupe une présentation des fonctionnalités de l'outil. Les deux diagrammes implémentés sont présentés en détails et les interactions disponibles sur ceux-ci sont égalements expliquées.

Le participant pouvait également poser des questions à n'importe quel moment de l'expérimentation. Cependant, la vidéo semble suffisante puisqu'il n'y a pas eu de questions importantes lors de cette phase.

1. Vidéo VisUML : `https://www.youtube.com/watch?v=buyGojmbUpQ`

Enfin, chaque participant découvre le projet de la même façon, l'EDI étant ouvert avec deux classes (c'est-à-dire deux onglets). Ces deux classes ont été choisies en rapport avec leurs rôles dans le projet. La première est la classe *Game* qui est le point d'entrée du projet et la seconde est *Entity* qui est la classe abstraite représentant n'importe quel objet du projet.

Le plugin VisUML étant affiché, nous lançons nous-mêmes le plugin puis les deux visualisations (diagrammes de classe et de séquence) dans un navigateur web. Une fois cette installation effectuée, le participant a un contrôle total sur l'EDI et les visualisations.

6.4 Évaluation

L'expérimentation n'a pas de contrainte de temps. Le participant peut explorer le projet comme bon lui semble et utiliser l'EDI, le diagramme de classe et le diagramme de séquence de la façon dont il le souhaite.

Pendant la durée de l'expérimentation, nous posons huit questions aux participants. Ces questions sont simples, mais nécessite de naviguer dans le projet et d'analyser le code source. Nous indiquons explicitement aux participants que ce ne sont pas les réponses à ces questions qui seront évaluées, mais la façon dont elles vont être obtenues (navigation, outils utilisés,...).

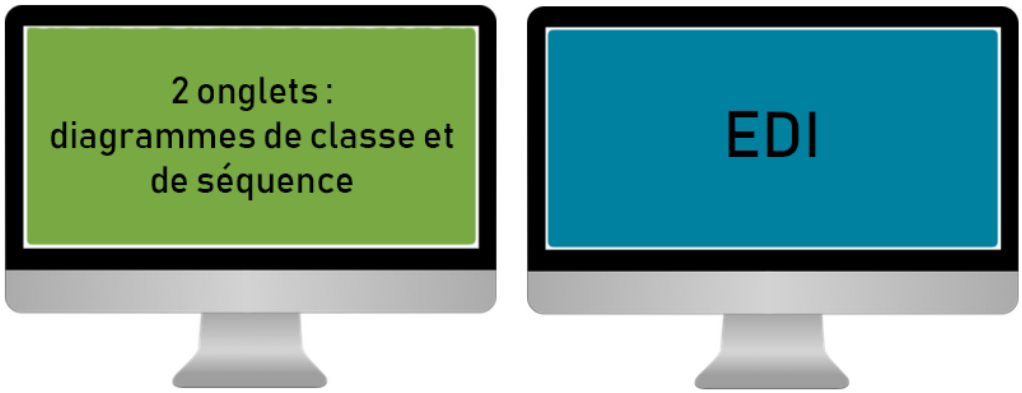


FIGURE 6.21 – Contexte de travail pendant l'expérimentation de VisUML

Les questions sont toujours posées dans le même ordre :

1. Combien de classes filles possède la classe *Entity* ?
2. Combien de classes sœurs [2] possède la classe *Entity* ?

2. On définit deux classes comme *sœurs* si elles héritent d'une même classe.

3. Combien de classes filles possède la classe *Cell* ?
4. Quel est la complexité de la méthode *initializeMaze* de la classe *Grid* ?
5. Où est utilisée la méthode *initializeMaze* de la classe *Grid* ?
6. Quel est la complexité de la méthode *initGame* de la classe *Game* ?
7. Dans la méthode *initEntities* de la classe *Game*, y a-t-il du code redondant ?
8. Dans la méthode *findRandomEmptyCell* de la classe *Grid*, combien y a-t-il de boucles / tests imbriqués ?

En plus de ces questions, nous avons noté la façon dont les participants passent d'un élément à l'autre ou trouvent une information (avec un diagramme, en parcourant le code, etc.) La figure 6.22 représente les répartitions des visualisations par question et transition entre les questions. Le détail de chacune de ces répartitions est présenté dans les sous-sections suivantes.

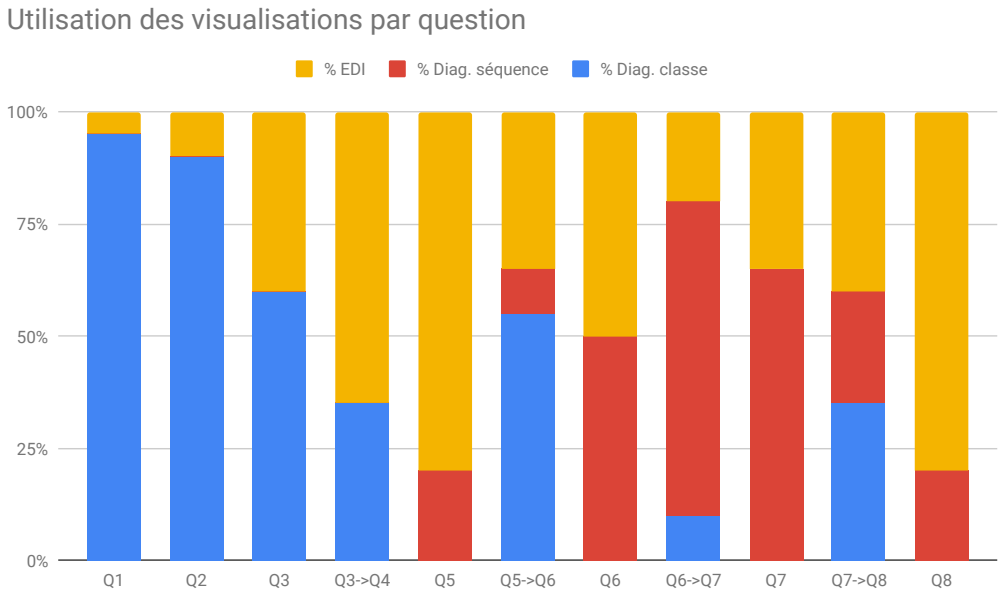


FIGURE 6.22 – Utilisation des visualisations par question

6.4.1 Q1 - Découverte des classes filles de *Entity*

La première question concerne la classe *Entity*, ouverte au début de l'expérimentation. L'objectif de la question est d'identifier les classes filles de *Entity*.

Cette information est visible à la fois sur l'EDI et sur le diagramme de classe. Pour obtenir les classes filles sur l'EDI, une partie (5%) a utilisé des

raccourcis clavier et la recherche textuelle de l'outil, avec « extends Entity » pour n'afficher que les résultats intéressants. En revanche, 95% des participants ont utilisé le diagramme de classe pour obtenir cette information, qui apparaît automatiquement lorsqu'une classe est ouverte dans l'EDI.

On peut expliquer cette différence par le fait qu'avant de poser la première question, la plupart des participants ont étudié rapidement le diagramme de classe qui s'est généré, et ont pu repérer la classe en question. Il est également important de comparer le nombre nécessaire d'actions pour obtenir l'information sur l'EDI (environ trois : ouverture de la recherche, écriture des mots-clés, puis clic pour afficher les résultats), au nombre nécessaire sur le diagramme de classe (dans ce cas : aucune, et une action si la classe n'avait pas été ouverte).

6.4.2 Q2 - Découverte des classes sœurs de *Entity*

La seconde question demande aux participants d'identifier les classes « sœurs » de la même classe que précédemment. La définition d'une classe sœurs est rappelée au moment de la question pour que tous les participants aient la même vision.

Pour cette question, il était nécessaire d'effectuer cette recherche en deux étapes : identifier la classe mère, puis ses autres *enfants*. Comme pour la question précédente, une grande majorité (90%) des participants a utilisé le diagramme de classe pour identifier les éléments. Une fois le parent identifié, il suffit d'un clic pour faire apparaître toutes les classes filles. Il est cependant à noter qu'une partie (30% du total) a répondu que la classe *Entity* ne possédait pas de sœur, car elles n'apparaissaient pas (sans action) sur le diagramme. Les 10% ayant utilisé l'EDI ont utilisé la recherche textuelle, via plusieurs méthodes : via les menus, commande + clic sur la classe mère, raccourci clavier.

Une fois la première recherche effectuée, une partie des participants (20% du total) ont « vérifié » leurs résultats en allant voir l'EDI.

6.4.3 Q3 - Découverte des classes filles de *Cell*

La troisième question porte sur une nouvelle classe : *Cell*. Cette classe est la sœur de la classe *Entity*, et devait être identifiée dans la question précédente.

L'objectif est de donner le nombre de classes héritant de *Cell*. Comme pour les deux questions précédentes, l'utilisation du diagramme de classe est en tête avec 60%, contre 40% pour l'EDI. On remarque (sur la figure 6.22 page précédente) cependant que de plus en plus de participants ont utilisé ce dernier pour découvrir des éléments, via une recherche textuelle (« extends Cell ») ou la fonctionnalité *Find usages*. Cette dernière est cependant peu efficace car elle affiche beaucoup d'informations sans être filtrable (voir figure 5.8 page 100).

6.4.4 Q4 - Analyse de la méthode *initializeMaze*

Pour la question suivante, nous demandons aux participants d'aller étudier la méthode *initializeMaze* de la classe *Grid*.

Pour accéder à cette méthode, 65% des participants utilisent l'EDI, avec la fonction de recherche ; ils doivent donc réécrire le nom exact de la méthode. Les 35% restants ont utilisé le diagramme de classe afin d'accéder directement au fichier et à la première ligne de la méthode, en cliquant sur le nom de cette dernière dans le diagramme.

6.4.5 Q5 - Appels parents de *initializeMaze*

Cette question porte sur la même méthode que la précédente, *initializeMaze*. L'objectif est d'identifier les emplacements du projet où cette méthode est appelée.

80% des participants ont utilisé l'EDI comme support, via la fonctionnalité *Find Usages* ou l'utilisation du raccourci cmd (ctrl) + clic sur la signature de la méthode. Le reste (20%) a utilisé le diagramme de séquence et les appels parents pour identifier les méthodes appelantes. Cette fonctionnalité ne faisant pas partie du standard UML, il n'est pas étonnant que les participants aient privilégié une approche plus classique avec l'EDI.

6.4.6 Q6 - Analyse de la méthode *initGame*

La question 6 porte sur une nouvelle méthode : *initGame* de la classe *Game*. Le premier objectif est de naviguer jusqu'à cette méthode, puis d'en estimer la complexité de manière approximative.

Pour accéder à cette question, 55% des participants utilisent le diagramme de classe, avec un clic sur la méthode. Une faible partie (10%) a utilisé le diagramme de séquence puis le diagramme de classe. Cela s'explique par le fait que le diagramme actif au moment du changement de question était le diagramme de séquence, les participants concernés ont donc parcouru ce diagramme, puis utilisé le diagramme de classe une fois la méthode *initGame* identifiée.

La complexité est, quant à elle, étudiée à 50% sur l'EDI et sur le diagramme de séquence. Cette activité est laborieuse sur l'EDI puisqu'il faut parcourir plusieurs méthodes et sous-méthodes pour estimer correctement la complexité. Sur le diagramme de séquence en revanche, il est possible d'afficher rapidement le contenu des sous-méthodes (et de leurs sous-méthodes) grâce à l'utilisation d'un bouton (+). Sur les 50% ayant utilisé le diagramme de séquence, 70% ont utilisé cette fonctionnalité, leur permettant d'étendre le diagramme sans parcourir le code. Enfin, il est intéressant de noter que les participants ont préféré cette

fonctionnalité plutôt que le niveau général de profondeur du diagramme, qui étend automatiquement toutes les sous-méthodes jusqu'au niveau désiré. Cette modification implique de gros changements dans le diagramme affiché, ce qui peut perturber la lecture.

6.4.7 Q7 - Analyse de la méthode *initEntities*

La question 7 porte sur la méthode *initEntities* de la classe *Game*. Cette méthode est appelée par la méthode *initGame* de la question précédente, et apparaissait normalement sur les visualisations du participant. L'objectif de cette question est d'identifier du code redondant au sein de cette méthode (et de ses sous-méthodes).

Pour atteindre la méthode, 70% des participants ont utilisé le diagramme de séquence. Ce chiffre peut s'expliquer d'une part avec les 50% des participants qui ont utilisé ce diagramme pour la question précédente. D'autre part, 20% des participants ont, suite à la question précédente, utilisé le diagramme de séquence pour comparer leurs résultats obtenus avec l'EDI. Ces derniers se trouvaient donc également sur le diagramme de séquence lors de la transition vers cette question. 20% ont utilisé l'EDI (via les fonctions de recherche), et 10% ont utilisé le diagramme de classe et un clic sur la méthode.

Afin d'identifier le code redondant, 65% des participants ont utilisé le diagramme de séquence et les fragments colorés pour détecter les messages similaires. Les 35% restants ont utilisé l'EDI et ont parcouru l'ensemble du code des méthodes concernées pour détecter d'éventuelles duplications de code.

Il est intéressant de noter que, comme pour la question précédente, le niveau de profondeur du diagramme de séquence joue un rôle important. Pour cette question, nous avons conseillé aux participants utilisant le diagramme de séquence de placer le niveau général sur 2 au minimum, afin d'obtenir une vue plus globale de la méthode concernée.

6.4.8 Q8 - Analyse de la méthode *findRandomEmptyCell*

Enfin, la dernière question porte sur la méthode *findRandomEmptyCell* de la classe *Grid*. L'objectif de cette question était simplement de noter sur quelle visualisation le participant allait chercher les informations, à savoir le nombre de conditions et boucles imbriquées.

La navigation vers la méthode a été effectué à 40% sur l'EDI, 35% sur le diagramme de classe et 25% sur le diagramme de séquence. Il est intéressant de noter que ces nombres sont proches du tiers, indiquant que les participants étaient plus à l'aise avec les diagrammes en fin d'évaluation.

La détection du nombre de conditions/boucles imbriquées s'est cependant faite à 80% via l'EDI, contre 20% pour le diagramme de séquence.

6.5 Questionnaire de fin

Suite à cette évaluation, un deuxième questionnaire est proposé aux participants. Ce questionnaire vise à obtenir un avis général de l'outil, une estimation de leur utilisation des différentes représentations, ainsi que leurs idées et remarques. A la différence des questions posées pendant l'évaluation, nous laissons les participants seuls pendant qu'ils remplissent le formulaire. Les questions 1 et 2 sont utilisées pour identifier le participant (il devait indiquer son numéro).

6.5.1 Q2.3 - Avis sur VisUML

La question 3 propose aux participants de noter VisUML. Comme pour les questions à échelle du premier formulaire, les participants doivent indiquer leur avis sur une échelle de 1 à 8.

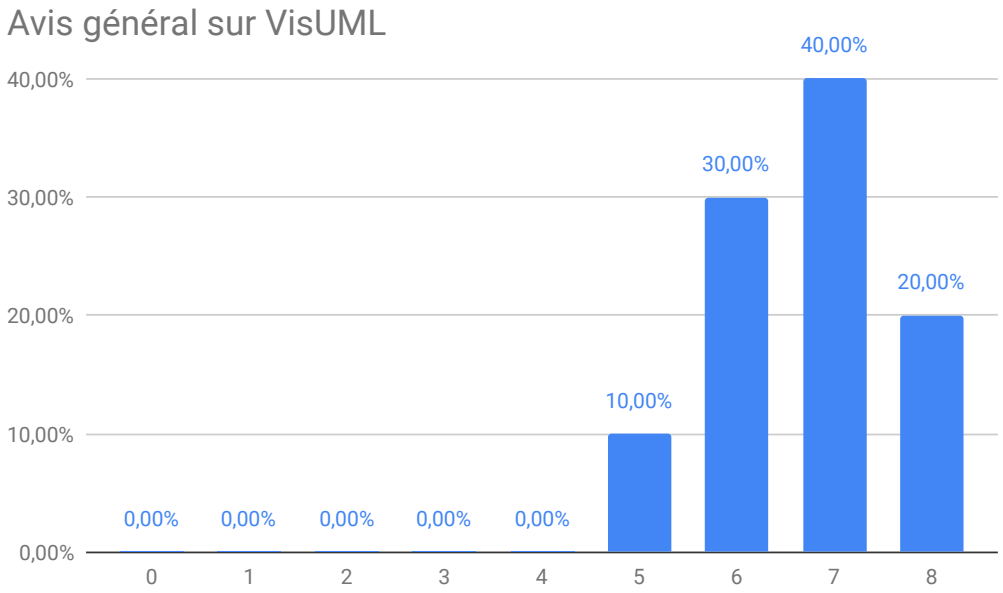


Figure 6.23 – VisUML Q2.3 - Avis sur VisUML

La figure 6.23 représente la répartition des avis laissés par les participants. On remarque que tous les avis sont supérieurs à la moyenne, avec une majorité

(40%) à 7 sur 8. La note moyenne pour VisUML est égale à 6.7 sur 8, ce qui est très positif comme retour.

6.5.2 Q2.4 à 6 - Utilisations des représentations

Les questions 4, 5 et 6 demandent aux participants d'estimer le taux d'utilisation de chacune des représentations de l'évaluation : l'EDI, le diagramme de classe et le diagramme de séquence. Le participant doit indiquer un nombre entre 0 et 100 pour chaque question, la somme de ces trois nombres devant être égale à 100 une fois ces nombres additionnés.

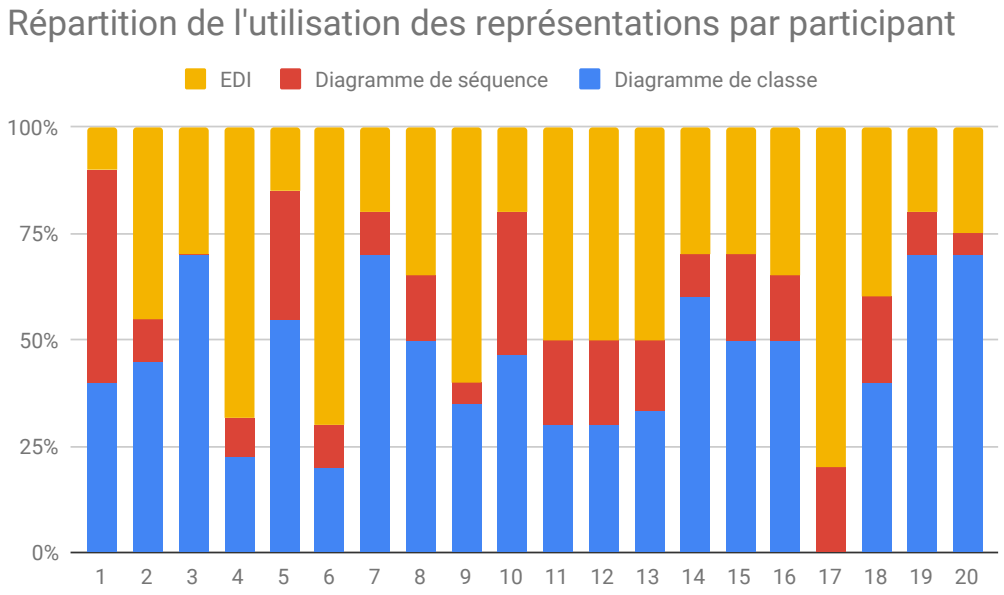


FIGURE 6.24 – VisUML Q2.4 - Utilisations des représentations

La figure 6.24 représente la répartition de l'utilisation des représentations. Le graphique affiche, par participant, les pourcentages d'utilisation de l'EDI (en jaune), le diagramme de classe (en bleu) et le diagramme de séquence (en rouge). On remarque que le diagramme de séquence est le moins utilisé de manière générale, en dehors du premier participant, qui l'a utilisé le plus souvent. La figure 6.25 page suivante montre la moyenne d'utilisation des représentations.

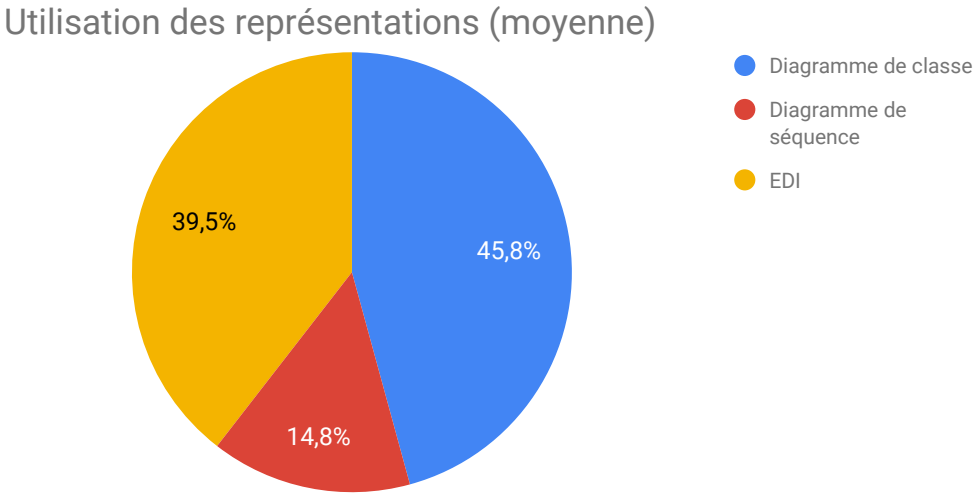


Figure 6.25 – VisUML Q2.4.2 - Moyenne d'utilisation des représentations

On remarque plus clairement sur cette figure que le diagramme de classe est en tête, avec 45,8%, suivi par l'EDI avec 39,5% puis par le diagramme de séquence avec 14,8%. Cette répartition est également un retour positif pour notre outil, puisque les participants n'ont pas utilisé que l'EDI.

6.5.3 Q2.7 - Interactions simples et efficaces

La question 7 demande aux participants d'indiquer les interactions qu'ils ont trouvées les plus simples et efficaces parmi la liste suivante :

- Clic sur un attribut => Navigation dans le code
- Clic sur une méthode => Navigation dans le code
- Clic sur une méthode => Mise à jour du diagramme de séquence
- Déplacement du curseur (depuis l'IDE) => Mise à jour du diagramme de séquence
- Clic sur un message (ou lien du diagramme de séquence) => Navigation dans le code

En plus de ces choix, le participant a la possibilité d'en ajouter une via un texte libre.

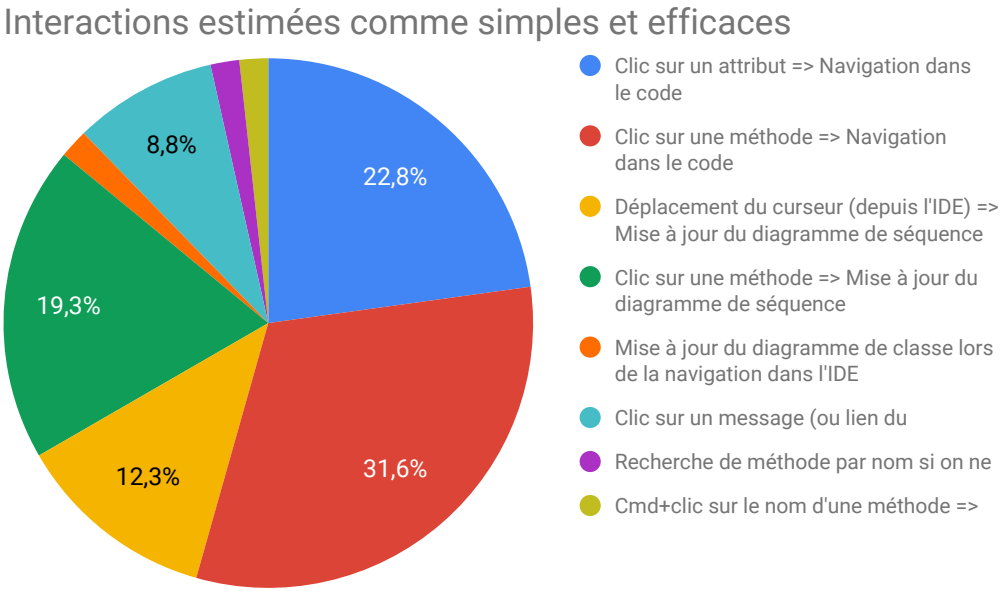


FIGURE 6.26 – VisUML Q2.7 - Interactions les plus simples et efficaces

La figure 6.26 représente la répartition des interactions jugées comme simples et efficaces par les participants. On remarque que le clic sur une méthode pour naviguer dans le code est en tête avec 31,6%, ce qui peut s'expliquer par le fait que cette interaction a beaucoup été sollicitée pendant l'évaluation. La seconde interaction (22,8%) est le clic sur un attribut, similaire à un clic sur une méthode, qui permet de naviguer dans le code. Vient ensuite le clic sur une méthode, pour mettre à jour le diagramme de séquence (19,3%), puis le déplacement du curseur sur l'EDI pour mettre à jour le diagramme de séquence (12,3%). Ces interactions sont les principales utilisées pendant l'évaluation et peut expliquer une partie des résultats.

6.5.4 Q2.8 - Informations manquantes

La question 8 demande aux participants s'ils ont identifié des informations manquantes ou qu'ils aimeraient voir apparaître sur les diagrammes. Nous proposons de base trois options : « Informations de débug (en temps réel) », « Informations d'erreurs de syntaxe » et « Informations sur les annotations ». Les participants ont également la possibilité d'en ajouter autant qu'ils le souhaitent.

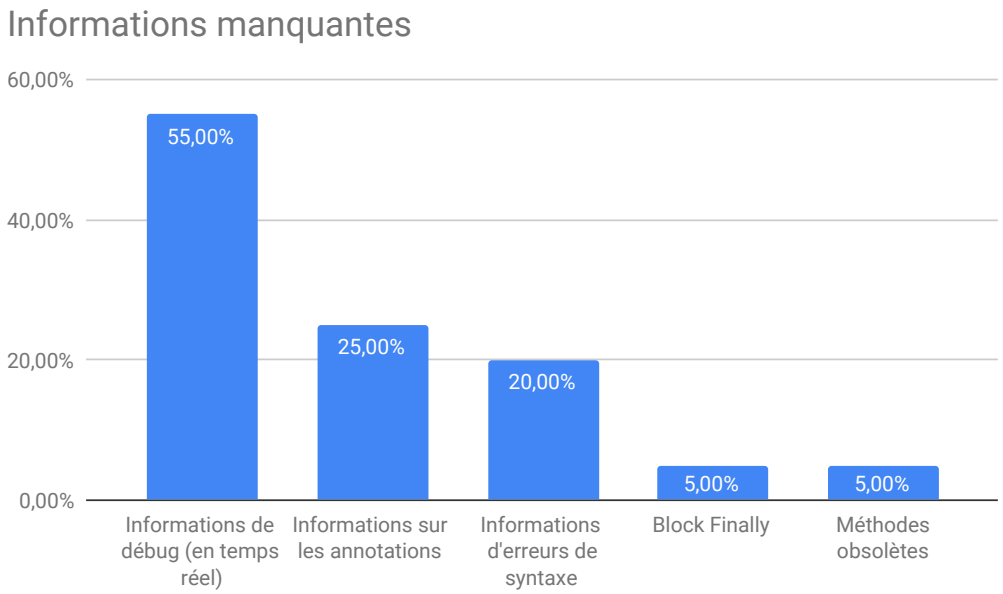


FIGURE 6.27 – VisUML Q2.8 - Informations manquantes sur les diagrammes

La figure 6.27 montre que 55% des participants aimeraient obtenir des informations de debug en temps réel sur les diagrammes. Les annotations sont également une information que les participants souhaiteraient avoir à 25%. Pour rappel, les données des annotations sont déjà analysées par VisUML, mais n'ont pas encore de représentation sur les diagrammes. 20% des participants souhaiteraient afficher des informations sur les erreurs de syntaxe (que l'EDI fournit). 5% des participants voudraient également mettre en évidence les méthodes obsolètes. Enfin, 5% aimeraient que le block *finally* des *try/catch* soit présent. Cette dernière fonctionnalité n'était pas encore disponible lors de l'évaluation, mais a depuis été développée.

6.5.5 Q2.9 - Modification du diagramme

La question 9 demande aux participants s'ils auraient aimé pouvoir modifier le diagramme pour mettre à jour le code. Cette fonctionnalité, *Round-Trip Engineering*, fait partie des perspectives envisagées pour VisUML, et l'objectif de cette question est d'estimer son taux d'utilisation.

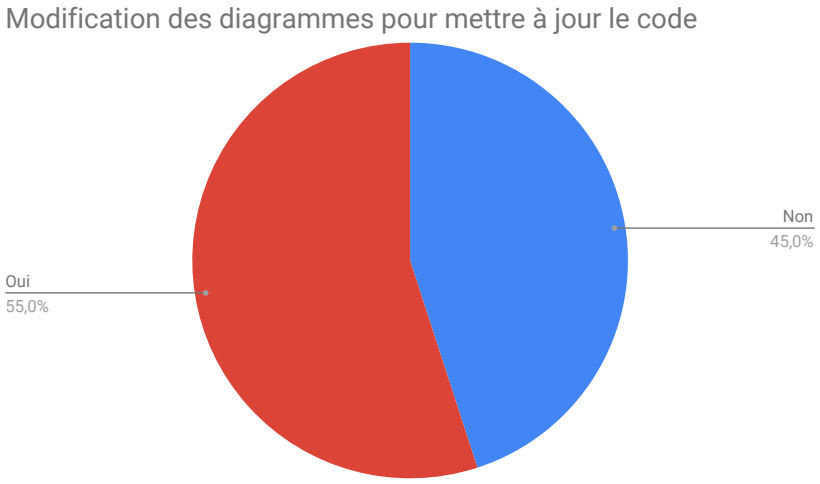


Figure 6.28 – VisUML Q2.9 - Round-Trip Engineering

La figure 6.28 montre que 55% des participants aimeraient pouvoir modifier les diagrammes pour mettre à jour le code, tandis que 45% ne pensent pas en avoir besoin. Bien que l'évaluation ne portait pas sur de la modification de code, on remarque que la majorité pense que cette fonctionnalité leur serait utile. Il serait intéressant de refaire une évaluation portant sur la modification, pour comparer les résultats obtenus.

6.5.6 Q2.10 et 11 - Fréquence d'utilisation de VisUML

La question 10 demande aux participants d'estimer la fréquence à laquelle ils utiliseraient VisUML dans leur travail. Le participant doit répondre un chiffre entre 1 (jamais) et 8 (tous les jours). La question 11 est un champ libre permettant aux participants d'expliquer pourquoi ils utiliseraient fréquemment ou non l'outil.

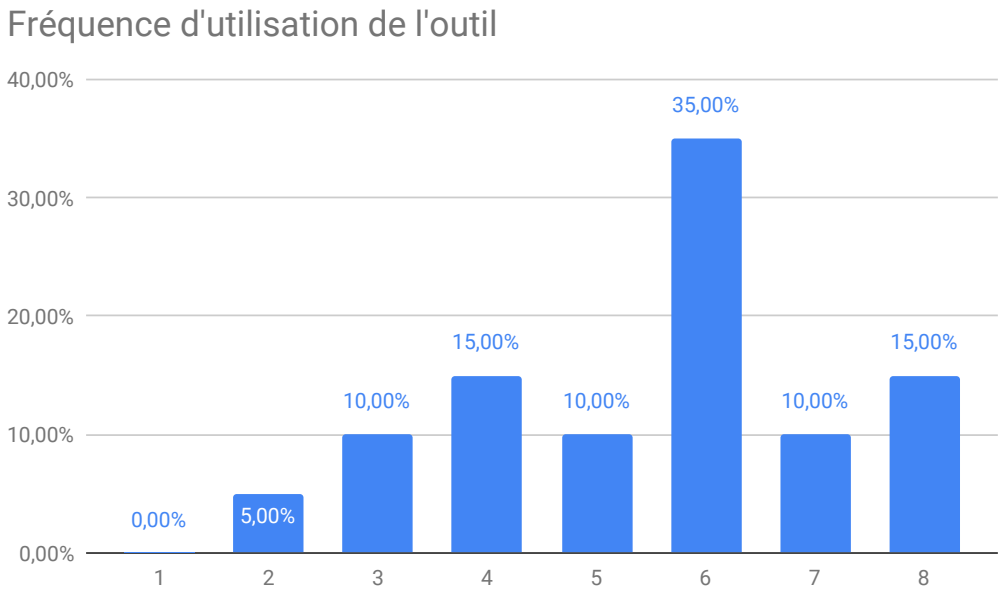


FIGURE 6.29 – VisUML Q2.10 - Fréquence d'utilisation de VisUML

La figure 6.29 montre que 60% des participants utiliseraient l'outil à une fréquence de 6 et + sur 8 lors de leur travail. Ce résultat est encourageant pour VisUML, puisqu'il montre une potentielle utilisation hebdomadaire. 15% des participants utiliseraient l'outil à une fréquence de 3 et moins, indiquant une utilisation assez rare.

La question 11, à réponse libre, regroupe les explications des participants. On constate notamment que l'aspect visuel est apprécié (« *avoir une vue d'ensemble sur beaucoup d'informations, meilleure visu globale, cela me pousserait à utiliser et visualiser plus souvent mon projet avec des diagrammes, dans le cas où je reprendrais le développement d'un module cela pourrait m'être utile pour avoir une vue d'ensemble de celui-ci* »). De même, le gain de temps en navigation et recherche d'information est mis en avant ((« *surtout pour les gros projets, un gain de temps pour la navigation dans le code, faciliterait grandement les tâches de recherche de code pour les modifications, surtout sur les gros projets* »). L'utilisation de VisUML sur un nouveau projet, un projet d'une autre personne ou non utilisé depuis longtemps est également un point positif selon les participants : (« *surtout quand c'est un projet que je ne connais pas ou peu, cela est utile quand on veut refactorer du code, quand on reprend un projet, très pratique lorsque l'on reprend un projet développé il y a longtemps, reprise de code, TMA, arrivé tardive dans un projet, demande d'aide (pair programing), revue de code, très pratique pour faire des revues de code* ».

Enfin, la génération de diagramme peut également servir pour de la docu-

mentation ou comme support de communication : (« *présenter des diagrammes lors de meeting technique, pour avoir une documentation à jour* ».

6.5.7 Q2.12 et 13 - Bénéfices de VisUML

La question 11 demande aux participants s'ils estiment être allés plus vite pendant leur tâche grâce à VisUML.

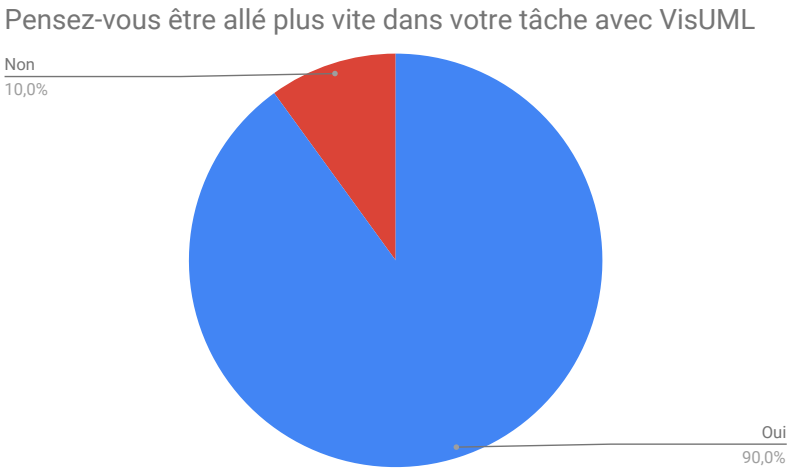


FIGURE 6.30 – VisUML Q2.12 - Bénéfices de VisUML (rapidité à la tâche)

La figure 6.30 montre que 90% des participants estiment avoir été plus rapide grâce à VisUML. Ce résultat est très positif pour notre outil, et montre qu'il peut avoir un bénéfice en terme de gain de temps lors de certaines tâches de travail. Les 10% ayant répondu non expliquent ce choix par leur manque de connaissance de l'outil, et le fait qu'ils ont des habitudes et des réflexes sur leurs outils actuels.

La question 12 est un champ libre permettant aux participants d'expliquer leur choix :

Les participants ayant répondu oui :

— *Aspect visuel,* ***facilité d'accès***
— *Une* ***recherche plus rapide*** *des fonctionnalités et des classes*
— ***Bonne visu d'ensemble du projet*** *et agencement des classes ce qui permet un gain de temps quant à la navigation.*
— ***Les relations entre les classes sont bien plus concrètes visuellement que dans le code*** *où il faut dérouler algorithmiquement l'ordre des étapes ou rechercher laborieusement de quelle classe hérite quelle autre classe.*

— ***On trouve plus rapidement les classes*** *que l'on cherche. Par exemple, si on modifie une méthode qui en appelle un autre,* ***un simple clic sur l'appel de cette méthode sur le diagramme de séquence pourra nous ouvrir la méthode en question****.*
— *l'****interaction entre le diagramme et le code en temps réel***
— *D'avoir une* ***visualisation directe du diagramme****, de pouvoir aller directement dans la méthode souhaitée (plutôt que de scroller éternellement)*
— *Grâce au clic sur les méthodes/attributs permettant d'****accéder rapidement au code***
— ***Bien plus rapide de retrouver les classes/méthodes via un diagramme de classes*** *que via l'arborescence.* ***Le diagramme de séquences est très utile aussi*** *pour visualiser l'algo (complexité, notamment)*
— *La possibilité de rapidement sauter d'une classe à l'autre en ayant une* ***navigation plus logique et visuelle que celle d'un IDE***
— *Diagramme de séquence (debug, optimisation)*
— *Dans certains cas cités au dessus / ou si quelqu'un me pose des questions sur le code*
— *Clic sur une méthode/attribut => Navigation dans le code*
— *L'interaction avec l'IDE*
— ***Meilleur visualisation de la complexité***
— *Diagramme de séquence*
— *Navigation proposée par le diagramme de classes, hiérarchie des appels de fonctions internes du diagramme de séquences*
— ***Simplicité du diagramme de classes et du peu d'informations affichées***

Pour les participants ayant répondu non :

— *Je ne pourrai aller sur aucun outil plus vite que sur IntelliJ alors que je viens de le découvrir. J'utilise intelliJ depuis deux ans et j'ai l'habitude avec beaucoup de raccourcis claviers et de fonctionnalités qui permettent une utilisation optimisée. Pour résumer, j'utilise intelliJ de façon optimisée alors que ce n'est pas le cas pour visUML*
— *Ne connaissant pas assez l'outil*

6.6 Conclusion de l'évaluation

Ce chapitre a présenté la première évaluation de VisUML. Pour cette évaluation, nous avons décidé de tester des fonctionnalités simples de l'outil (navigation et interactions), afin de déterminer s'il apporte une nouvelle façon de naviguer à l'utilisateur.

La première étape de ce travail a été de définir quelles parties de VisUML nous allions mettre à disposition des participants pour qu'ils puissent juger de

l'apport ou non de l'outil. Pour cette expérimentation, l'objectif était d'observer la façon dont des utilisateurs recherchent des informations au sein d'un projet qu'ils ne connaissent pas.

Nous avons ensuite défini un projet Java contenant un ensemble de classe plus ou moins complexes. Une partie des méthodes étaient également implémentées, de sorte à pouvoir poser des questions sur celles-ci.

Nous avons également défini deux questionnaires à destination des participants, l'un pour obtenir une estimation générale de leur niveau et utilisation des diagrammes (UML et autres), et le second pour leur demander leurs retours à la suite de l'expérimentation.

Au total, l'expérimentation a été effectuée avec vingt participants, pour la plupart d'entre eux dans les locaux de l'Université, mais nous avons eu la chance de pouvoir effectuer cette expérimentation au sein d'une entreprise proche, pour trois des participants.

L'ensemble des retours que nous avons eus sur l'outil sont positifs et encourageants, et mettent en avant la simplicité de l'outil. La génération automatique, surtout pour le diagramme de séquence, est une fonctionnalité que les participants ont beaucoup appréciée. Les interactions simples et la synchronisation entre les visualisations sont également deux fonctionnalités qui, d'après les participants, permettent une meilleure compréhension d'un nouveau projet.

L'analyse des données récoltées pendant l'expérimentation, ainsi que les retours des participants, nous ont permis de constater que l'utilisation de VisUML améliore la compréhension d'un projet, d'une méthode en particulier et également les liens existants entre plusieurs méthodes ou plusieurs classes.

Ces données nous permettent également de comparer les utilisations estimées des visualisations avec des valeurs calculées. Les figures 6.31 page suivante, 6.32 page ci-contre et 6.33 page 142 montrent ces utilisations.

On remarque que, dans l'ensemble, les courbes sont proches et correspondent donc aux estimations des participants, notamment pour le diagramme de classe. En revanche, il est intéressant de constater, sur la figure 6.33 page 142, que les participants ont tendance à sous-estimer l'utilisation qu'ils ont faite du diagramme de séquence.

Suite à l'expérimentation, plusieurs participants ont exprimé leur intérêt pour l'outil et pour une utilisation dans le cadre de leur travail.

Cette première évaluation de VisUML nous a donc permis de tester les bases de l'outil. Nous aimerions continuer ces évaluations par la suite, en testant des fonctionnalités précises de l'outil, comme les filtres, l'utilisation d'un outil de modélisation, les tags.... Ces perspectives seront présentées dans le chapitre 6.6 page 143

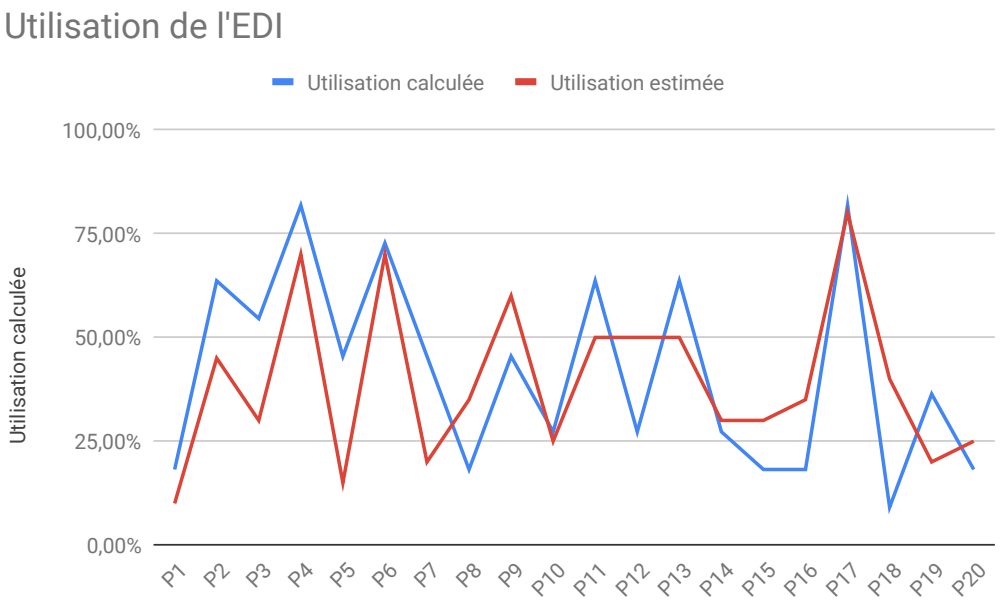


FIGURE 6.31 – Utilisations estimées et calculées de l'EDI

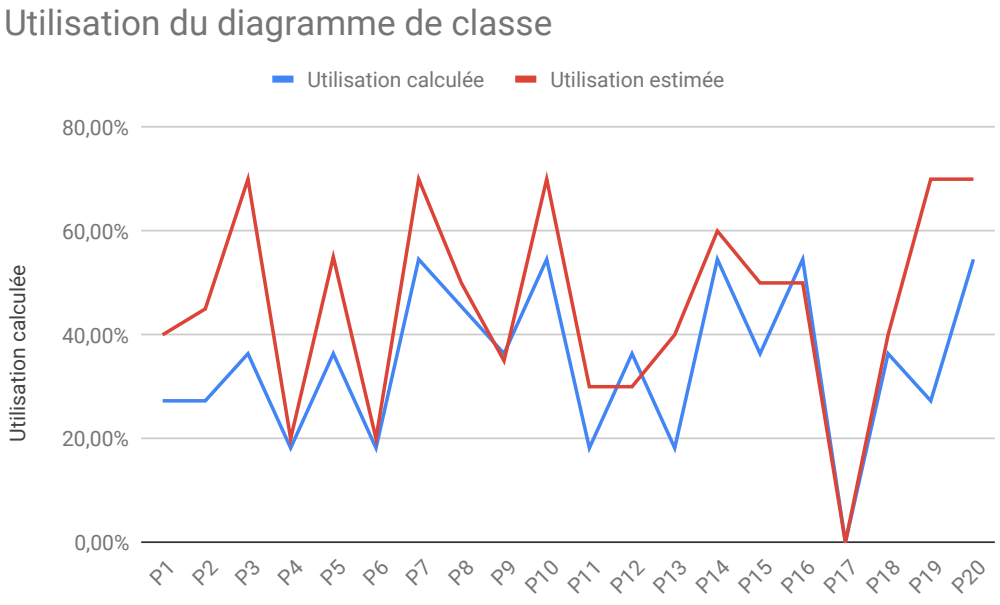


FIGURE 6.32 – Utilisations estimées et calculées du diagramme de classe

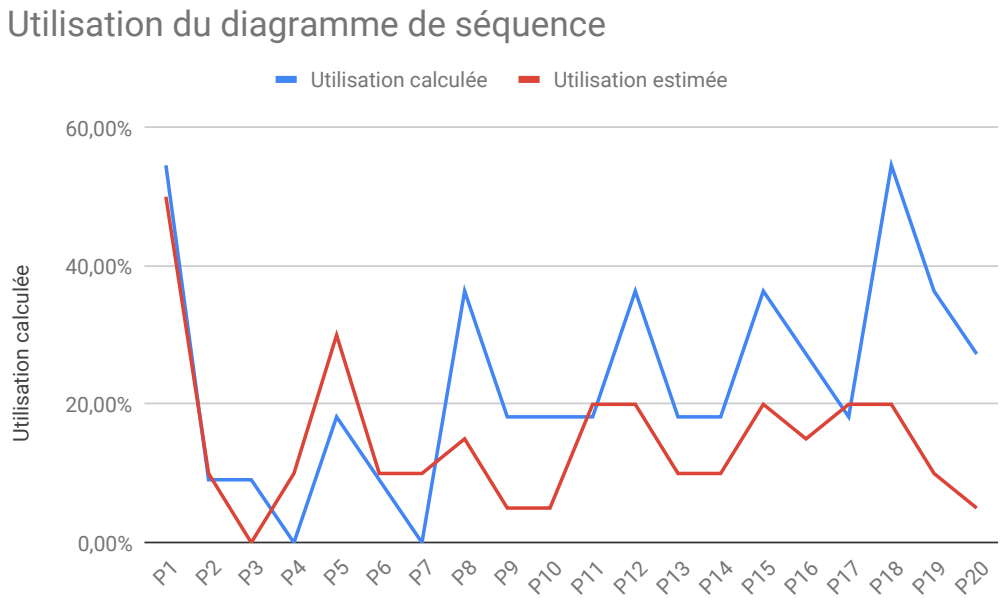


FIGURE 6.33 – Utilisations estimées et calculées du diagramme de séquence

Conclusion et perspectives

Conclusion

Cette thèse avait pour objectif d'améliorer le quotidien des développeurs en leur fournissant plusieurs types de visualisations synchronisées entre elles, plus particulièrement en proposant deux diagrammes UML connectés au code source d'un projet. Dans cette thèse, nous souhaitions répondre à la problématique : « **Comment améliorer la compréhension d'un programme à l'aide de diagrammes** ». Pour ce faire, nous avons développé **VisUML**, un outil de visualisation dynamique et interactif permettant aux développeurs d'obtenir un ou plusieurs diagrammes UML correspondant à leur activité en cours.

Dans ce manuscrit, nous avons tout d'abord détaillé le contexte dans lequel cette thèse s'est déroulée, c'est-à-dire les métiers du développement informatique qui ont subi plusieurs évolutions au cours des dernières années, que ce soit par l'augmentation de la taille des projets, donc du nombre de fichiers, ou par l'augmentation du nombre moyen de participants à un projet (encore plus important pour des projets open source). Ces deux changements impliquent une façon de travailler différente, afin de mieux répartir les tâches, tout en gardant une cohérence au sein du projet. Heureusement, plusieurs organisations de travail ont également vu le jour (que ce soit des patrons de conception ou des frameworks par exemple), et définissent des ensembles de règles et de bonnes pratiques à appliquer par tous pour obtenir un ensemble uni.

Dans le même temps, les EDI ont, eux aussi, évolué ces dernières années. Ils permettent désormais d'afficher de plus en plus d'informations à l'écran, de colorier syntaxiquement des langages de programmation, mais aussi de faciliter les échanges de code et la communication au sein d'une même équipe par exemple. Cependant, leur façon d'afficher des informations n'a pas évolué : le texte est toujours le support le plus utilisé.

L'une des solutions à ce problème est la modélisation et les fonctionnalités de « *reverse-engineering* », qui permettent d'obtenir des représentations graphiques d'un projet. Malheureusement, les outils implémentant ces solutions manquent encore de contexte et ne se contentent que d'afficher le plus d'informations

possible. Le langage principal pour la modélisation logicielle est UML. Plusieurs études montrent cependant que, même s'il est appris lors des études, il reste très peu utilisé en entreprise.

D'autres études dans le domaine de la psychologie du développeur montrent que des représentations graphiques, comme le diagramme de classe ou le diagramme de séquence, sont plus compréhensibles pour un développeur que du texte brut, mais qu'ils sont souvent trop chargés et nécessitent donc un temps de lecture et de compréhension élevé.

Pour avoir une vision claire des outils permettant d'afficher des représentations graphiques d'un projet, nous avons comparé un ensemble de ces outils sur plusieurs critères spécifiques au « *reverse-engineering* ». Ces critères ont été regroupés en trois groupes : les données en entrée, les données et visualisations en sortie et les fonctionnalités des outils. Nous avons montré dans le chapitre 2 page 16 que VisUML tente de répondre aux problèmes des outils actuels. Ses visualisations ne contiennent qu'un sous-ensemble du projet, dont les éléments correspondent à la tâche active du développeur. Elles sont également synchronisées et donc mises à jour en temps réel à chaque modification du code. Cette synchronisation permet aux développeurs de passer d'une représentation à l'autre en limitant le travail mental à effectuer. Les visualisations sont également interactives et permettent une navigation rapide entre elles (code et diagrammes), ce qui réduit également la charge cognitive associée à un changement de contexte.

Lors du développement de VisUML, nous avons tout de suite voulu faire un outil modulable, où chaque composant peut se connecter et se déconnecter à tout moment. L'architecture de VisUML utilise ainsi un bus de données afin de permettre cette fonctionnalité.

Pour fonctionner, VisUML repose sur un plugin associé à des représentations graphiques. Nous avons ainsi développé un plugin pour EDI (un pour Intellij/Android Studio et un pour Eclipse) qui analyse les fichiers du projet pour créer des représentations des entités (classe, interface…). Il permet également d'obtenir des informations sur le contexte de travail du développeur : les onglets ouverts, la classe actuellement parcourue, la position du curseur… qui sont aussi envoyées sur le bus de données. Ce plugin écoute également les messages provenant des représentations graphiques, afin d'interagir avec ces dernières.

Afin de représenter les données issues du projet de façon graphique, nous avons implémenté deux diagrammes UML. Ces visualisations sont disponibles sur page web mais également, via des plugins, pour des outils de modélisation comme Papyrus ou GenMyModel.

L'objectif principal de VisUML étant de simplifier l'activité d'un développeur, nous avons implémenté plusieurs fonctionnalités pour réduire sa charge cognitive. La première est de limiter les informations affichées. A l'inverse d'un

outil de « *reverse-engineering* » classique, VisUML n'affiche que les éléments ouverts dans l'EDI, généralement dix ou moins, ainsi que les éléments possédant au moins une relation avec un élément ouvert. Ce choix permet d'obtenir un diagramme de classe associé à l'activité actuelle du développeur qui peut ainsi passer du code au diagramme sans perdre son contexte.

La seconde fonctionnalité est de mettre en avant graphiquement plusieurs informations. L'onglet actif dans l'EDI va apparaître d'une autre couleur, permettant une lecture rapide du diagramme. Lorsqu'un élément est sélectionné sur le diagramme de classe, toutes les relations dont il fait partie sont également mises en avant, via une couleur, pour permettre aux développeurs de naviguer entre ces relations rapidement. De même, nous avons choisi d'associer une couleur par type de fragment sur le diagramme de séquence, permettant une lecture très rapide d'une méthode contenant plusieurs structures de contrôle.

Enfin, pour faciliter encore plus la lecture, nous avons implémenté plusieurs systèmes de filtres, des plus basiques permettant de cacher ou afficher un ensemble commun d'éléments (packages, classes, éléments non ouverts...), aux plus complexes en utilisant une grammaire créée spécialement pour VisUML, et qui permet d'effectuer des requêtes complexes (par exemple mettre en couleur toutes les classes qui possèdent une méthode dont le type de retour est X).

Les visualisations de VisUML sont également interactives. L'objectif de ces interactions est de limiter la charge mentale lors d'une navigation de l'une à l'autre des représentations, et de simplifier l'utilisation des diagrammes générés. La synchronisation automatique des éléments ouverts et de leurs contenus permet de conserver le contexte de travail et de réduire le temps nécessaire à la compréhension des diagrammes. Chaque élément généré sur les diagrammes est interactif ; il est possible de cliquer dessus pour naviguer vers le code associé dans l'EDI (en ouvrant un fichier si nécessaire). Ces interactions permettent aux développeurs d'obtenir rapidement des informations sur un élément, mais également de parcourir le code de façon simple. De même, il est possible de naviguer entre plusieurs diagrammes de séquence sans avoir à retourner sur le code. Une interaction simple (alt+clic) permet en effet d'afficher la séquence associée à une méthode. Nous avons également augmenté le diagramme de séquence pour y afficher un nouveau type de message, représentant les appels parents d'une méthode, ce qui permet une navigation « de bas en haut » rapide.

Afin de valider les choix faits pour VisUML, nous avons effectué une expérimentation de ses fonctionnalités (à l'exception des filtres, tags et des modules pour outils de modélisation). Nous avons fait passer cette expérimentation sur vingt participants. Les retours sont tous positifs, et mettent en avant la simplicité d'utilisation de VisUML. Ils indiquent également, pour la plupart, être allés plus vite dans la compréhension du projet grâce à l'outil et aux diagrammes

connectés.

Nous souhaitons continuer ces expérimentations, en nous concentrant sur chaque fonctionnalité fournie par VisUML, les filtres, les tags et également les modules complémentaires.

Perspectives

Il reste de très nombreuses évolutions possibles à apporter à VisUML. Elles concernent à la fois des améliorations des fonctionnalités existantes, et de nouvelles fonctionnalités. Nous pouvons les répartir sur trois échéances : court, moyen et long terme.

Perspectives à court terme

Gestion multi-projets

Actuellement, l'utilisateur doit fermer son EDI pour ouvrir un nouveau projet avec VisUML, sans quoi les informations (sessions web, analyse de projet, etc.) sont conservées depuis le projet précédent et ne se mettent pas à jour pour le nouveau projet. Etant donné l'environnement de travail des développeurs aujourd'hui, il est absolument nécessaire de permettre à VisUML de pouvoir gérer plusieurs projets en simultané ou même plus simplement à la suite, sans devoir quitter VisUML à chaque fois. Cette évolution, bien que simple a priori, nécessite un travail conséquent, car même si WSE gère de façon simple les sessions, il n'en va pas de même pour faire remonter les informations à l'EDI. En effet, si IntelliJ est ouvert avec trois projets différents, on doit être capable de savoir vers quelle instance d'IntelliJ on doit faire remonter les messages WSE. Nous n'avons pas encore exploré cette piste, mais nous pensons que ces instances sont identifiées de façon unique soit dans IntelliJ (ou tout autre EDI) soit dans l'environnement de travail (Windows, Linux...).

Intégration de la sémiologie graphique

Notre façon de représenter UML repose d'une part sur la librairie GoJS et ses templates (que nous avons modifiés en partie) et d'autre part sur des modifications graphiques que nous avons choisies d'apporter à UML (par exemple en colorant les blocs). Cependant, nous avons fait des choix arbitraires quant au reste (par exemple le bleu pour les éléments sélectionnés (séquence, méthode, attributs), ou encore le vert pour la classe active dans le diagramme de classe. Même si durant l'évaluation personne n'a contesté ces choix, nous comptons intégrer les résultats de la sémiologie graphique en commençant par les résultats

issus de la thèse de Yossr El Ahmar [17, 1]. Ceci devrait permettre d'autres expérimentations pour voir quelles seraient les meilleures couleurs (et autres aspects graphiques) pour nos diagrammes, mais également pour voir si ces choix graphiques sont généraux ou fortement dépendants des projets modélisés, des utilisateurs en fonction de leurs niveaux de compétences, leurs métiers, leurs contextes de travail, etc. Une telle étude pourrait faire l'objet à elle seule d'une thèse en ergonomie.

Navigation multi-schémas sur une seule page

Comme évoqué dans ce mémoire (cf. section 5.3.4 page 100), nous avons réalisé une preuve de concept qui permet de visualiser sur une seule et même page web le diagramme de classe et les dix derniers diagrammes de séquence. Les personnes qui ont testé cette page nous ont dit qu'elles préféraient ce moyen à celui actuel (diagramme de classe sur une page web et diagramme de séquence sur une autre page web). Il serait donc judicieux de développer totalement ce nouveau mode de navigation et de le tester. Sachant que nous avons également réalisé une preuve de concept quant au diagramme d'activité (mais que ce dernier est à réécrire pour être totalement intégrable à VisUML), il serait intéressant de d'ajouter la navigation dans les dix derniers diagrammes d'activité en plus des dix diagrammes de séquence, voire de passer de l'un à l'autre.

Visualisation multi-couches

Le diagramme de classe actuel n'affiche que les classes, leurs contenus en terme de méthodes et attributs, leurs relations avec d'autres classes, leur package, et les tags. Or, nous récupérons beaucoup d'autres informations lors de l'analyse du projet par VisUML. Nous connaissons par exemple l'auteur, les dates de création, de modification, le nombre de lignes, les annotations, les noms des layouts (pour les projets Android), les noms des bases de données et des tables [3], les requêtes sur les bases de données [4], etc. Toutes ces informations pourraient être affichées sur le diagramme de classe sous la forme par exemple de calques transparents affichables à la demande. Ainsi, un développeur pourrait voir par exemple que :

— la classe sur laquelle il travaille est reliée à une autre classe par une injection de dépendance,
— sa classe est associée à un layout sur lequel le bouton Valider appelle la méthode validateClient,

3. informations récupérées grâce à la programmation orienté aspect
4. informations récupérées grâce à la programmation orienté aspect

— la classe reliée à sa classe est connectée à une base de données de type Room [5]
— sa classe a été modifiée hier par Marc qui a modifié la méthode getClient,
— etc.

Pour le diagramme de séquence, nous avons débuté l'implémentation d'une couche visuelle représentant le code source d'un message. Cette approche se base sur les CodeBubbles [3, 4, 39] et OctoBubble [25] et ajoute, pour chaque message, une *bulle* contenant la ligne exacte de code associé à ce message. La figure 6.34 montre un exemple de *CodeBubbles* dans VisUML.

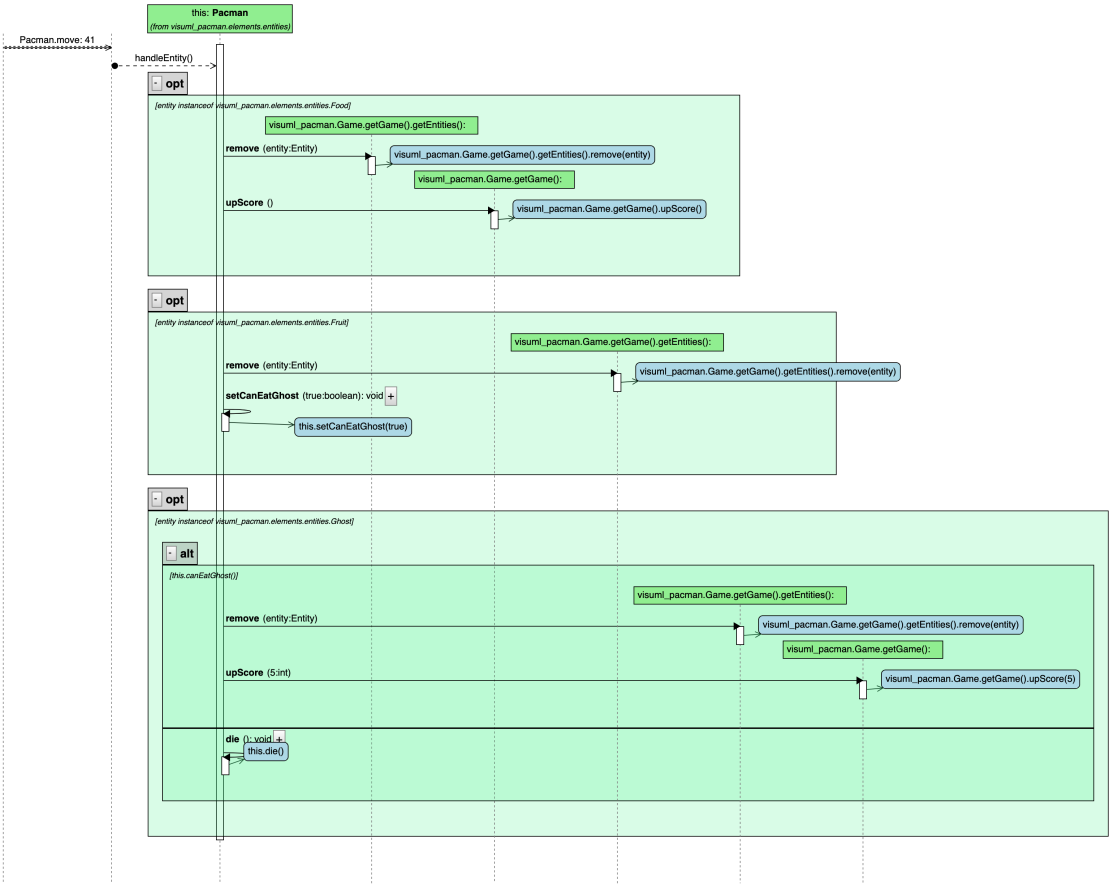


FIGURE 6.34 – *CodeBubbles* dans VisUML

L'ajout de ces informations permet d'ajouter un lien supplémentaire entre le code et la représentation graphique, ce qui réduit le travail que le développeur doit effectuer lorsqu'il passe du code au diagramme, ou du diagramme au code.

5. Room est une bibliothèque fournie par Google permettant d'abstraire l'utilisation de SQLite : `https://developer.android.com/topic/libraries/architecture/room`

Ces *bulles* sont également interactives et agissent de la même façon que le message auquel elles sont associées : un clic sur une *bulle* va mettre en évidence le code associé sur l'EDI

Amélioration des requêtes de filtrage

Le langage que nous avons développé pour filtrer les données affichées (par exemple *show class with ...*) reste à améliorer. Certes, il est déjà plus puissant en terme de filtrage que les simples fonctions de recherche proposées dans les EDI, mais il y a encore beaucoup de fonctions à ajouter. Tout d'abord, il faudrait l'étendre quant aux paramètres supportés. Ensuite, il faudrait proposer des requêtes plus puissantes (par exemple avec des ET et des OU, et avec des niveaux d'imbrication élevés). Enfin, pour optimiser le temps de saisie, il faudrait des complétions automatiques et la possibilité d'enregistrer des commandes type (un peu comme des macros). Cela pourrait donner lieu à une étude pour voir comment les développeurs s'approprient notre langage, l'utilisent, le paramètrent, et l'optimisent.

Amélioration de l'analyse du code source

La version actuelle de VisUML repose sur la librairie Spoon de l'INRIA. Cette librairie se charge de l'analyse du projet pour en extraire toutes les informations et permettre une navigation dans celles-ci. L'avantage d'utiliser Spoon est double. Premièrement, il permet l'analyse, mais également la modification du code source analysé, mais cette dernière fonctionnalité ne nous est pas utile. Deuxièmement, il permet à VisUML de fonctionner sur IntelliJ, Android Studio et Eclipse. Nous avons découvert que IntelliJ (et Android Studio) propose un système d'analyse interne équivalent à Spoon [6]. L'avantage d'utiliser ce système interne est qu'il fonctionne en tâche de fond, donc plus besoin de gérer nous-mêmes l'analyse de projet, et qu'il est plus rapide que Spoon. L'inconvénient est qu'il nous prive de l'usage de l'EDI Eclipse. Nous pouvons donc tout à fait imaginer que VisUML fonctionne avec, au choix, Spoon (pour Eclipse) et le système interne pour IntelliJ/Android Studio. Dernièrement, nous avons creusé la piste du système interne. Pour le moment, et uniquement pour Java, nous avons pu retirer Spoon (donc uniquement pour des projets Java) pour le remplacer par le système interne proposé par IntelliJ qui fournit exactement les mêmes résultats quant au diagramme de classe. Il resterait donc à poursuivre ce travail pour le diagramme de séquence, voire pour les autres aspects évoqués ici (par exemple récupérer les annotations, les layouts Android, etc., ce que fait déjà parfaitement IntelliJ !). Par ailleurs, nous avons découvert que, dans IntelliJ,

6. PSIElements pour Java

chaque langage possède ses propres éléments pour l'analyse du code source. Malheureusement, même si les PSIElements gèrent les projets Java, il est très difficile à ce jour de connaître le nom des éléments pour les autres langages (impossible de connaître ceux pour Kotlin par exemple). Cependant, nous avons vu récemment que IntelliJ possèdent des UElements (associés à UAST [7]) qui se comporteraient un peu comme des méta-PSIElements et permettraient donc de gérer tous les langages supportés par IntelliJ (PHP pour lequel nous avions déjà fait une preuve de concept d'analyse de code, C++, C#, Kotlin, Javascript, etc.).

Améliorer la création des tags

Nous générons actuellement les tags en utilisant d'une part l'API Cortical [8] et des algorithmes élémentaires tels que TF-IDF. Le résultat à ce jour est intéressant (par exemple nous obtenons bien des mots-clés associés aux bases de données dans les classes qui parlent de base de données, idem pour les activités Android, etc.) mais il est encore très loin de ce que nous espérions au départ (beaucoup de 'bruit' dans les mots-clés produits). Par ailleurs, la langue est compliquée à gérer (par exemple du code source en anglais, mais avec des commentaires en français). Enfin, les tags ne sont générés que pour les classes alors que nous voulions les générer à différents niveaux (méthode, classe, package, projet). Aussi, dans un premier temps, et outre le problème de la langue qu'il est essentiel de gérer proprement (langue du code source -> tags du code source, langue des commentaires -> tags des commentaires), il faudra améliorer la qualité des mots-clés produits. Pour cela, il faudra étudier d'autres algorithmes, mais aussi se reporter à la littérature concernant l'extraction de mots-clés dans des textes, ou bien encore la compréhension et la synthèse de programme (program comprehension, program summarization[6, 11, 23]). Ensuite, Il sera nécessaire de permettre une meilleure manipulation de ces tags (ajout, modification, suppression, classement…) de façon interactive par exemple sur le diagramme de classe. Enfin, il faudra étudier cette production de mots-clés aux différents niveaux voulus (méthode, classe, package, projet).

Connexion à d'autres outils

Durant la thèse, nous avons montré qu'il était possible de connecter VisUML à d'autres outils, pour autant que ceux-ci soient capables d'échanger des informations avec l'extérieur. C'est ainsi que nous avons connecté Papyrus, GenMyModel ou bien encore Sonar (pour la qualité de code), sans oublier une télécommande (pour modifier les diagrammes UML). Dernièrement, nous avons associé Git

7. UAST : https://github.com/JetBrains/intellij-community/tree/master/uast
8. Cortical : http://api.cortical.io/

aux tags. Un projet étudiant est en cours sur ce point. Pour le moment, nous produisons les tags au moment des *commits* et nous les stockons dans un fichier JSON. L'idée à la fin du projet est que chaque *commit* produise les tags du commentaire du commit, ainsi que les tags des classes ouvertes et les tags des classes modifiées. Lors du *push*, ces informations (les tags et les éléments auxquels ils sont reliés) ainsi que des informations associées au diagramme de classe ouvert (par exemple la position des classes à l'écran) seront stockées sur le dépôt Git. Pour prendre en compte les contraintes et habitudes de travail des développeurs actuels, il serait bon d'associer d'autres outils tels que :

— JIRA pour le ticketing : toutes les entreprises utilisent Git ou des équivalents, ainsi que des outils de ticketing (JIRA ou autre). Quand un développeur réceptionne un ticket et procède à la correction de celui-ci, il dépose le n° de ticket dans le texte du commit. Ainsi, on sait toujours qui a fait quoi (grâce au commit), et pourquoi (grâce au n° du ticket qui est en fait un hyperlien permettant de remonter au ticket réel). VisUML devrait pouvoir se connecter à ce type de système, et permettre de tagguer les informations associées et naviguer dans celles-ci.

— la connexion avec Sonar a été faite de façon simple pour montrer la faisabilité, mais la mise à jour du code ne produisait pas la mise à jour de l'analyse Sonar. Avec les outils actuels d'intégration continue, cela serait beaucoup plus simple. VisUML pourrait donc afficher tout un nouvel ensemble d'informations sur le diagramme de classe par exemple (est-ce que les tests sont passés ? quels sont les résultats ? qui a fait les tests ? est-ce que cette classe ralentit le serveur ? etc.).

— l'un des reproches faits à VisUML lors de l'évaluation est qu'il n'est pas possible de voir l'exécution en temps réel. Il serait donc intéressant d'ajouter cette fonctionnalité par exemple en connectant le débogueur de l'EDI (point d'arrêt, stack trace…) mais aussi l'analyseur de code quant aux erreurs de syntaxe dans le code source, ou aux avertissements (telle méthode n'est jamais appelée, etc.).

Filtrer depuis l'EDI

Nous avons montré qu'il est simple de filtrer/modifier les informations sur le diagramme de classe à partir de la ligne de commande située en dessous et du langage de filtrage associé. Cette ligne de commande pourrait tout à fait être utilisée également dans l'EDI. Ainsi, le développeur pourrait demander par exemple à ouvrir/fermer :

— toutes les classes du projet (donc pas les Interfaces ou autres fichiers)

— toutes les classes dont le nom contient Model

— toutes les classes dont le nom contient Model et qui ont au moins 3

méthodes renvoyant un booléen
— toutes les classes dont le nom contient Model et qui ont au moins 3 méthodes avec au plus 2 paramètres et renvoyant un booléen
— etc.

Le développeur n'aurait alors plus besoin de parcourir sa liste de fichiers. A notre connaissance, aucun outil actuel ne propose cette fonctionnalité.

Perspectives à moyen terme

Gérer les différences entre versions de fichiers

Dans sa version actuelle, VisUML analyse chaque fichier dans sa globalité afin de générer une entité représentant la classe (ou interface, énumération...). Cette entité contient l'ensemble des paramètres de l'élément (nom, FQN, commentaires, *flags*...), des attributs et des méthodes (dont leurs contenus). Lors de la modification d'un fichier, quelle que soit cette modification, VisUML va relancer une analyse complète du fichier pour re-créer l'entité et envoyer la nouvelle version de cette dernière aux représentations graphiques. Cette approche a un coût, en temps et en ressource, à la fois du côté plugin (analyse du fichier) et du côté des représentations graphiques (envoi du message sur le bus, réception et transformation en élément graphique), qui pourrait être réduit.

L'une des solutions envisagées est l'utilisation de *patch* représentant la ou les modifications effectuées par l'utilisateur entre la version actuellement affichée sur les représentations graphiques et la version venant d'être sauvegardée sur l'EDI. Un *patch* de la sorte contient beaucoup moins d'informations que l'analyse complète d'un fichier et réduit donc le coût associé aux traitements côté plugin et visualisations. Ce travail de représentation d'une modification a été effectué par Michel Dirix [12] dans sa thèse au sein de l'équipe Carbon et pourra servir de base pour une évolution de VisUML.

Intégrer des notions de points de vue

VisUML ne propose qu'une seule vue par projet, celle que le développeur voit quand il ouvre son projet. Certes, si deux personnes ouvrent le même projet, chacune peut interagir indépendamment sur le diagramme de classe et le diagramme de séquence pour naviguer dans son code, mais les deux personnes voient le même diagramme de classe. Cependant, VisUML propose d'ores et déjà à chacun de modifier son affichage pour l'adapter à ses besoins (grâce aux différents filtres et interactions sur les diagrammes). Il est donc tout à fait envisageable d'ajouter la notion de point de vue à VisUML. Par exemple :

— un développeur A pourrait voir le projet comme il le veut,

— un développeur B ferait de même avec sa propre vision,
— un chef de projet pourrait avoir une version simplifiée (vision type classe réduite mais avec surcharge de type structuration du projet, tags, etc.),
— un graphiste ne verrait que les layouts et les interactions associées
— un architecte ne verrait que la structure du projet
— un testeur ne verrait que la qualité du code, les résultats des tests, etc.

Intégrer la collaboration entre personnes sur un même projet

Dès le début de cette thèse, nous avons choisi de ne pas aborder la collaboration entre développeurs dans VisUML. Même si nous avons ajouté quelques fonctionnalités simples allant dans ce sens (connexion à Git, interactions séparées pour des personnes connectées sur le même projet), tout reste à faire pour une vraie collaboration efficace. Dans notre équipe Carbon, Michel Dirix[13, 12] a montré qu'il était possible d'ajouter cet aspect sur un système qui ne le prenait pas en compte initialement (GenMyModel). Ce travail est tout à fait reproductible sur VisUML, mais demandera toutefois un grand travail en terme de développement. Cependant, grâce à notre connexion à Git, certaines tâches devraient être plus faciles à coder, par exemple reconstruire l'historique des modifications du projet. De plus, notre prototype de télécommande a montré que nous étions capable de prendre en compte les ordres de refactoring au sein de VisUML et d'IntelliJ. Ainsi, chaque développeur pourrait modifier son code (directement ou par VisUML), et utiliser Git pour fusionner son code avec celui des autres développeurs.

Utiliser les diagrammes UML pour modifier le code

Comme évoqué ci-dessus, notre prototype de télécommande permet de modifier très facilement un diagramme de classe en proposant des interactions contextualisées en fonction de la sélection actuelle. Par exemple, sans aucune sélection, la télécommande propose de créer une classe, une interface, un package… Avec plusieurs classes sélectionnées, il suffit d'un clic pour créer une classe mère commune à toutes les classes sélectionnées, ou encore un seul clic pour les déplacer dans un package, etc. VisUML envoie alors des ordres de refactoring à l'EDI qui, si l'utilisateur les valide, met à jour le code provoquant ainsi la modification du diagramme de classe, Par manque de temps, nous nous sommes cependant arrêtés là quant à cette fonctionnalité. Il serait judicieux de l'étendre au minimum sur le diagramme de séquence (par exemple en déplaçant des séquences ou des blocs), voire au diagramme d'activité. Nous avions déjà cependant déjà intégré les fonctions 'delete' et 'safe delete' proposées par les EDI. Il serait bon de lister toutes les fonctionnalités de refactoring proposés par

ces EDI et voir lesquelles seraient utiles et intégrables à notre télécommande.

Perspectives à long terme

Augmenter la sémantique des tags

Dans la version actuelle des tags, l'API de Cortical connectée à VisUML propose tous les termes associés à chaque mot-clé (par exemple voiture est associée à des mots évidents comme roues, carrosserie, grand prix, course, ferrari, mais aussi à des mots moins évidents comme prison, blessé, appartement, train...). Récemment, Cortical a modifié son API et propose maintenant la notion de « *contexte* ». Ainsi, pour voiture, les contextes proposés sont roues, course, prison, berline, trains, carrosserie. On remarquera que ces contextes ne sont pas encore très évolués, mais des ontologies disponibles sur le web (telles que BabelNet [9]...) proposent déjà de meilleurs résultats (transport, pollution, garage, usine...). Quoi qu'il en soit, pour le moment il n'est pas possible de sélectionner le ou les contextes (ou les domaines) en lien avec les tags. Il serait donc intéressant de pouvoir associer aux tags des domaines proposés par les ontologies. Ainsi, si un tag est « voiture » et que la classe est « VoitureAutonome », le développeur devra pouvoir cocher par exemple le domaine Transport proposé par l'ontologie. De fait, ce choix sera propagé dans toute la classe et dans tout le projet, et pourra par exemple faire remonter des tags dans leur classement, ou encore mieux induire des changements profonds (proposés au développeur ou gérés automatiquement) dans le projet (choix ergonomiques pour l'IHM, structuration du projet, etc.). Associer de l'Intelligence Artificielle (IA) à ce type d'amélioration pourrait se révéler une piste intéressante. Enfin, nous aimerions étendre la production de ces tags aux documents reliés au code source, par exemple des photos prises en réunion de travail, des comptes-rendus de réunions, des enregistrements vocaux, etc. Ainsi, un projet pourrait afficher des tags issus des documents annexes indiquant qu'il s'agit d'un projet concernant la mode vestimentaire en Europe, alors que les tags du code source montreraient des aspects structurels (MVC et base de données SQLite par exemple).

Augmenter le niveau sémantique des actions sur le code source

Trouver et mettre en évidence des informations tel que le propose notre langage de filtrage est déjà en soi une fonctionnalité qui n'existe pas ailleurs, mais ceci manque encore de puissance. Nous aimerions tout d'abord augmenter le niveau sémantique de notre langage par exemple en permettant des requêtes telles que « `show pattern MCV', `show class with pattern Itera-`

9. BabelNet : `https://babelnet.org/`

tor ». Ceci peut se faire dans un premier temps en se basant sur le nom des classes (par exemple Client*Model*, Client*Controller*, Client*View*) ou autres méthodes et packages. Cependant, pour une réelle compréhension des patterns par exemple, il sera nécessaire de procéder à une analyse plus poussée du code source. L'IA pourrait être une piste intéressante ici. De même, en utilisant de l'IA conjointement sur les tags et le code source, nous pensons qu'il est possible d'aller encore plus loin avec des commandes qui permettraient par exemple de mettre en évidence les classes qui devraient utiliser un pattern MVC, et qui proposeraient ensuite de le faire automatiquement en demandant à l'EDI un refactoring du code.

Navigation dans les « nuages » de tags de milliers de projets

Actuellement, nous produisons des tags pour chaque classe, du moment qu'elle a été ouverte au moins une fois, et si le projet est connecté à un dépôt Git, alors chaque *commit*, sauvegarde les tags des classes ouvertes. Cette année, un projet étudiant va plus loin en faisant en sorte que chaque commentaire de *commit* produise ses propres tags, et que chaque *push* dépose les fichiers habituels d'un *push* en y associant en plus les tags de ces fichiers et les tags des *commits*, ainsi que des informations associées au diagramme de classe ouvert au moment des commits (par exemple la liste et la position des classes/interfaces, packages. . . , mais aussi les filtres actifs, etc.). Aussi, nous devrions être capables de retrouver de l'information plus facilement grâce à ces tags. Un projet étudiant est actuellement en train d'être réalisé et a pour but de produire une première version de cet outil de recherche. Par la suite, nous devrions être capables de trouver par exemple les commits pour des tags donnés, puis de 'fouiller' (textuellement et/ou graphiquement) parmi ce résultat de recherche (approfondissement de la recherche, visualisation des commits, des fichiers et des modifications associées, etc.). Ce système peut tout à fait être étendu à une entreprise et donc à tous les projets qu'elle gère. Ainsi, des entreprises ayant des centaines de développeurs et des milliers de projets pourraient exploiter ce type de « fouille » pour améliorer leur production de code (faciliter la mise en place de bonnes pratiques, éviter de refaire des choses déjà faites, permettre aux nouveaux développeurs de se familiariser plus rapidement avec les projets et les habitudes de l'entreprise, etc.). Enfin, on peut imaginer ce système à l'échelle mondiale. Si des développeurs du monde entier utilisent VisUML, alors leurs dépôts Git seront 'tagués' avec notre système et il est alors possible d'imaginer un outil de fouille de dépôts Git, non plus au niveau local (une entreprise par exemple), mais au niveau mondial. Ainsi, un développeur pourrait facilement trouver des projets proches du sien. En effet, la recherche sur Git au niveau mondial n'est qu'une recherche « *plain-text* » qui se base uniquement sur le nom

des projets et leur description, voir image 6.35. Aussi, si on cherche « chat web node » parmi tous les projets Git mondiaux, on obtient des centaines de résultats (voir image 6.36).

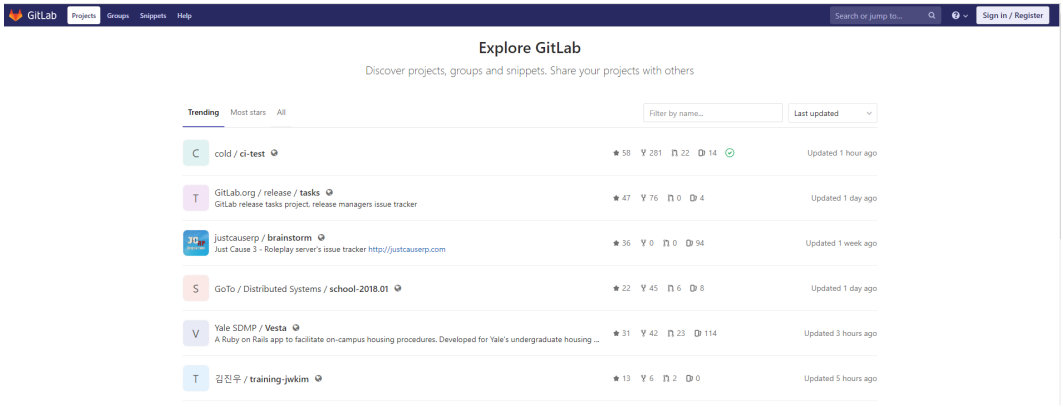


FIGURE 6.35 – Recherche de projet sur Gitlab

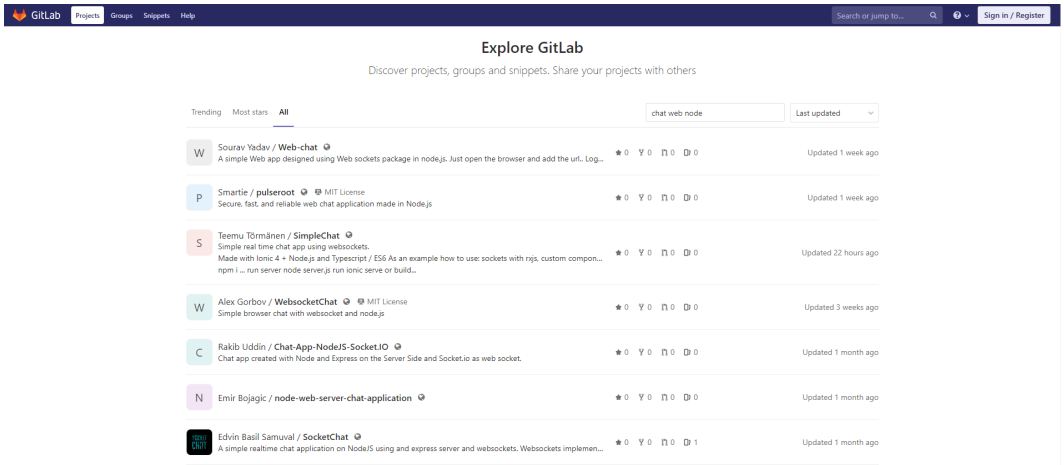


FIGURE 6.36 – Recherche de projet contenant « chat web node » sur Gitlab

Git, Gitlab et GitHub, ainsi que des projets disponibles sur ces mêmes plateformes, proposent des outils de recherche un peu plus évolués, mais tous ces outils restent toujours au niveau du code, et du texte des *commits*, des descriptions et des *readme*. Certains vont un peu plus loin en proposant des filtrages sur le nom des *packages*, les dates de dépôts, les auteurs, etc., mais tout ceci reste à un niveau d'abstraction proche de zéro. Ainsi, :

— la recherche de base de Gitlab (`https://docs.gitlab.com/ee/user/search/`) ne cherche que dans les « *issues* » et « *merge request* » pour les

propres projets de la personne qui cherche, ou alors que dans le nom et la description des projets autres que les siens.
— la recherche de Gitlab avec syntaxe avancée (`https://docs.gitlab.com/ee/user/search/advanced_search_syntax.html`) nécessite ElasticSearch [10] et reste encore très basique car elle ne fonctionne que sur le texte des fichiers et des commits. Par ex « *bug display | sound* » permet de trouver toutes les *issues* qui contiennent les mots (quel que soit l'ordre) « *bug* » et « *display* », ou « *bug* » et « *sound* ».
— la recherche avancée globale de Gitlab (`https://docs.gitlab.com/ee/user/search/advanced_global_search.html`) pouvait rechercher dans les applications GitLab, les projects, les *repositories*, les *commits*, les *issues*, les *merge requests*, les *milestones*, les notes (*comments*), les *snippets*, et les wiki, mais cette fonctionnalité a été retirée de Gitlab pour le moment (URL visitée le 13/03/2019) [11]
— la recherche de base de Github (`https://github.com/search`) ne recherche que dans le nom, la description des projets, ou les fichiers *Readme*, ou au mieux dans les *commits* (`https://help.github.com/en/articles/searching-commits`) la recherche avancée de Github (`https://github.com/search/advanced`) propose des options supplémentaires mais toujours très orientées Github (nom de l'auteur, date de dépôt...)

VisUML, avec les tags, compléterait très bien tous ces outils, ou d'autres tels que `https://sourcegraph.com` qui permet d'explorer ses propres projets de façon plus évoluée que Git, GitHub ou Gitlab le proposent. Il apporterait une montée en abstraction grâce aux nuages de mots qui sont associés aux tags ainsi qu'aux domaines issus des ontologies comme indiqué précédemment.

Adapter VisUML aux habitudes du développeur

Les interactions proposées par VisUML sont simples et consistent principalement en des clics sur différents éléments. Cependant, l'évaluation a montré que, pour certaines personnes, l'utilisation de raccourcis clavier est importante. En outre, la personnalisation de ces raccourcis est également un plus non négligeable pour ces développeurs.

Pendant la thèse, nous avons développé un prototype permettant à un utilisateur de choisir, pour chaque action disponible dans l'application, l'interaction qu'il préfère, parmi une liste fournie. Dans ce prototype, chaque application est représentée par un bloc, contenant une entrée par action disponible, et chaque appareil (ou application considérée comme tel) est représenté par un bloc conte-

10. ElasticSearch : `https://www.elastic.co/fr/products/elasticsearch`
11. Note Advanced Global Search (powered by Elasticsearch) is not yet available on GitLab.com. We are working on adding it. Follow this epic for the latest updates.

nant une sortie par interaction proposée. La figure 6.37 montre schématiquement le principe de ce prototype.

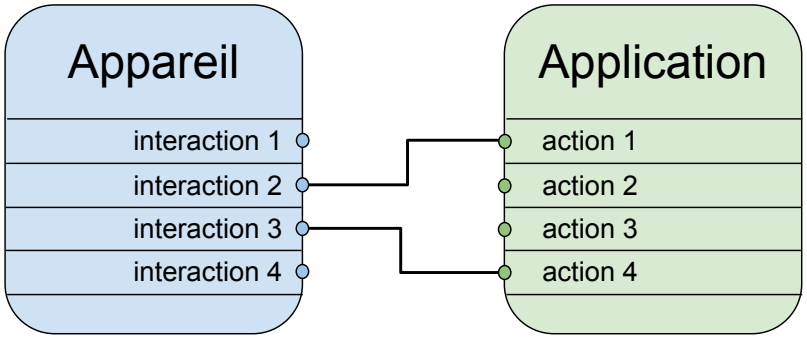
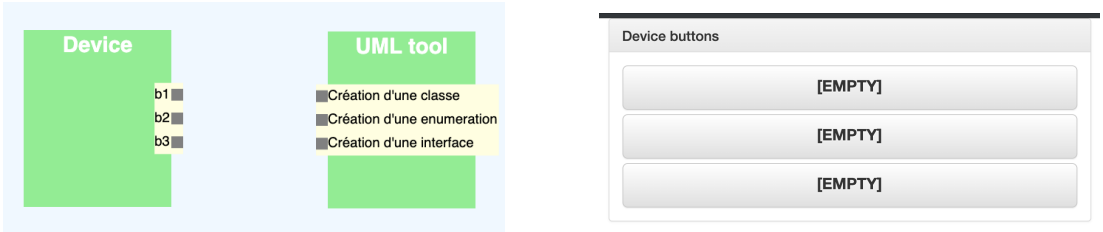


FIGURE 6.37 – Prototype de personnalisation du lien entre interaction et action

De la même façon, un second prototype permet de créer une palette d'action personnalisable par l'utilisateur. Cette première version propose un *appareil* comportant trois boutons, et une application proposant trois actions (voir figure 6.38).



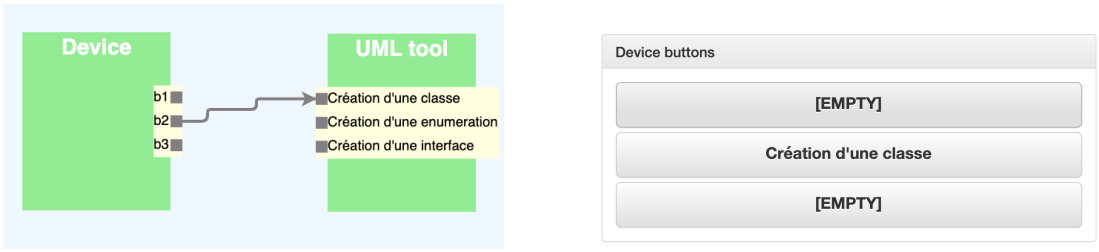
(a) Interface de gestion des liens

(b) Interface de l'*appareil*

FIGURE 6.38 – Prototype de palette d'action personnalisable

L'utilisateur peut tirer des liens entre une interaction et une action (voir figure 6.39 page suivante), ce qui va mettre à jour la vue *appareil* et, lors d'un clic, déclencher l'action associée sur l'application.

Les liens sont actuellement direct entre un appareil et une application. Dans le futur, l'objectif est de fournir plusieurs *portes* logiques, comme un ET ou un OU logique, permettant à un utilisateur de combiner plusieurs interactions simples afin de déclencher une action.



(a) Interface de gestion des liens

(b) Interface de l'*appareil*

FIGURE 6.39 – Prototype de palette d'action personnalisable avec un lien

Supporter l'activité du développeur dans sa globalité

La dernière fonctionnalité, et non des moindres, serait de prendre en compte l'activité globale du développeur. Cette fonctionnalité demande le recueil d'informations très diverses dont l'analyse est loin d'être évidente. Par exemple, en connaissant le profil de l'utilisateur (développeur, ergonome, chef de projet...), son activité (que fait-il, qu'a-t-il fait, avec qui, quand, sur quelles données....), son contexte de travail (matériel et configuration utilisée, date et lieu, mais aussi pourquoi pas des informations biométriques comme son niveau de stress, etc.), et en croisant tout ceci avec son activité actuelle, nous pourrions lui proposer des représentations parfaitement adaptées à son travail courant ainsi que des actions pour lui faciliter la tâche (refactoring, commit...). Nous sommes déjà capables d'apporter beaucoup de ces informations, ou de les apporter assez facilement, mais leur analyse de façon intelligente reste encore une lourde tâche à réaliser.

Bibliographie

[1] Yosser El Ahmar et al. « Visual Variables in UML : a First Empirical Assessment To cite this version : HAL Id : hal-01693537 Visual Variables in UML : a First Empirical Assessment ». In : (2018).

[2] Alan Blackwell et al. « Cognitive dimensions of notations : Design tools for cognitive technology ». In : *Cognitive Technology* 2001.Lnai 2117 (2001), p. 325–341. issn : 16113349 03029743. doi : 10.1007/3-540-44617-6_31. url : https://www.cl.cam.ac.uk/%7B~%7Dafb21/publications/CT2001.pdf.

[3] Andrew Bragdon et al. « Code Bubbles : A Working Set-based Interface for Code Understanding and Maintenance ». In : *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '10. Atlanta, Georgia, USA : ACM, 2010, p. 2503–2512. isbn : 978-1-60558-929-9. doi : 10.1145/1753326.1753706. url : http://doi.acm.org/10.1145/1753326.1753706.

[4] A. Bragdon et al. « Code bubbles : rethinking the user interface paradigm of integrated development environments ». In : *2010 ACM/IEEE 32nd International Conference on Software Engineering*. T. 1. Mai 2010, p. 455–464. doi : 10.1145/1806799.1806866.

[5] Michel R.V. Chaudron, Werner Heijstek et Ariadi Nugroho. « How effective is UML modeling ? : An empirical perspective on costs and benefits ». In : *Software and Systems Modeling* 11.4 (2012), p. 571–580. issn : 16191366. doi : 10.1007/s10270-012-0278-4. url : http://link.springer.com/10.1007/s10270-012-0278-4.

[6] Tse Hsun Chen, Stephen W. Thomas et Ahmed E. Hassan. *A survey on the use of topic models when mining software repositories*. T. 21. 5. Empirical Software Engineering, 2016, p. 1843–1919. isbn : 1066401594028. doi : 10.1007/s10664-015-9402-8. url : http://dx.doi.org/10.1007/s10664-015-9402-8.

[7] L Church et M Marasoiu. « A fox not a hedgehog : What does PPIG know ? » In : *PPIG 2016 - 27th Annual Workshop*. 2016. Chap. Psychology, p. 17–31.

[8] Andy Cockburn et Bruce McKenzie. « Evaluating the effectiveness of spatial memory in 2D and 3D physical and virtual environments ». In : 4 (2003), p. 203. doi : 10.1145/503411.503413.

[9] Tiago Silva Da Silva et al. « User-centered design and agile methods : A systematic review ». In : *Proceedings - 2011 Agile Conference, Agile 2011* (2011), p. 77–86. doi : 10.1109/AGILE.2011.24. url : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6005488.

[10] S P Davies. « Externalising Information During Coding Activities : Effects of Expertise, Environment and Task ». In : *Empirical Studies of Programmers : Fifth Workshop*. 1993. Chap. Empirical, p. 42–61.

[11] Andrea De Lucia et al. « Labeling source code with information retrieval methods : An empirical study ». In : *Empirical Software Engineering* 19.5 (2014), p. 1383–1420. issn : 15737616. doi : 10.1007/s10664-013-9285-5.

[12] Michel Dirix. « Pour une Collaboration Efficace dans les Outils de Modélisation Logicielle ». Thèse de doct. University Lille, 2016.

[13] Michel Dirix, Xavier Le Pallec et Alexis Muller. « Software Support Requirements for Awareness in Collaborative Modeling ». In : *Confederated International Conferences : CoopIS, and ODBASE 2014* (2014), p. 382–399. issn : 16113349. doi : 10.1007/978-3-662-45563-0_22.

[14] Brian Dobing et Jeffrey Parsons. « Current Practices in the Use of UML ». In : *Perspectives in Conceptual Modeling, ER'05* (2005), p. 2–11.

[15] Brian Dobing et Jeffrey Parsons. « How UML is used ». In : *Communications of the ACM* 49.5 (2006), p. 109–113. issn : 00010782. doi : 10.1145/1125944.1125949. url : http://dl.acm.org/citation.cfm?doid=1125944.1125949.

[16] Wojciech James Dzidek, Erik Arisholm et Lionel C. Briand. « A realistic empirical evaluation of the costs and benefits of UML in software maintenance ». In : *IEEE Transactions on Software Engineering* 34.3 (2008), p. 407–432. issn : 00985589. doi : 10.1109/TSE.2008.15.

[17] Yosser El Ahmar et al. « Enhancing the communication value of UML models with graphical layers ». In : *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems, MODELS 2015 - Proceedings*. Sept. 2015, p. 64–69. isbn : 9781467369084. doi : 10.1109/MODELS.2015.7338236.

[18] Matthieu Foucault et al. « Impact of developer turnover on quality in open-source software ». In : *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015* (2015), p. 829–841. doi : 10.1145/2786805.2786870. url : http://dl.acm.org/citation.cfm?doid=2786805.2786870.

[19] Daniel Graziotin et al. « On the Unhappiness of Software Developers ». In : *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. EASE'17. Karlskrona, Sweden : ACM, 2017, p. 324–333. isbn : 978-1-4503-4804-1. doi : 10.1145/3084226.3084242. url : http://doi.acm.org/10.1145/3084226.3084242.

[20] T. R. G. Green et a. Blackwell. « Cognitive Dimensions of Information Artefacts : a tutorial ». In : *Applied Psychology* October (1998), p. 75.

[21] T.R.G. Thomas R. G. Green et Marian Petre. « Usability Analysis of Visual Programming Environments : A 'Cognitive Dimensions' Framework ». In : *Journal of Visual Languages and Computing* 7.2 (1996), p. 131–174. issn : 1045-926X. doi : 10.1006/jvlc.1996.0009. arXiv : 1008.1900. url : http://www.sciencedirect.com/science/article/pii/S1045926X96900099%20http://linkinghub.elsevier.com/retrieve/pii/S1045926X96900099.

[22] Martin Grossman, Jay E. Aronson et Richard V. McCarthy. « Does UML make the grade? Insights from the software development community ». In : *Information and Software Technology* 47.6 (2005), p. 383–397. issn : 09505849. doi : 10.1016/j.infsof.2004.09.005.

[23] Sonia Haiduc, Jairo Aponte et Andrian Marcus. « Supporting program comprehension with source code summarization ». In : (2010), p. 223. doi : 10.1145/1810295.1810335.

[24] Morten Hertzum et Annelise Mark Pejtersen. « Information-seeking practices of engineers : Searching for documents as well as for people ». In : *Information Processing and Management* 36.5 (2000), p. 761–778. issn : 03064573. doi : 10.1016/S0306-4573(00)00011-X.

[25] Rodi Jolak et al. « OctoBubbles : A Multi-view interactive environment for concurrent visualization and synchronization of UML models and code ». In : *25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings* 2018-March (2018), p. 482–486. doi : 10.1109/SANER.2018.8330244.

[26] Ehsan Kouroshfar et al. « A study on the role of software architecture in the evolution and quality of software ». In : *IEEE International Working Conference on Mining Software Repositories* 2015-Augus (2015), p. 246–257. ISSN : 21601860. DOI : 10.1109/MSR.2015.30.

[27] Christian F J Lange, Michel R V Chaudron et Johan Muskens. « In practice : UML software architecture and design description ». In : *IEEE Software* 23.2 (2006), p. 40–46. ISSN : 07407459. DOI : 10.1109/MS.2006.50.

[28] Jill Larkin et Herbert Alexander Simon. « Why a Diagram is (Sometimes) Worth Ten Thousand Words ». In : *Cognitive Science* 11.1 (1987), p. 65–99. ISSN : 03640213. DOI : 10.1016/S0364-0213(87)80026-5. URL : http://mechanism.ucsd.edu/teaching/f12/cs200/readings/larkin.whyadiagramissometimesworth.1987.pdf.

[29] Thomas D LaToza, Gina Venolia et Robert DeLine. « Maintaining mental models : a study of developer work habits ». In : *ICSE '06 Proceedings of the 28th international conference on Software engineering* (2006), p. 492–501. ISSN : 09574174. DOI : 10.1145/1134285.1134355. URL : http://portal.acm.org/citation.cfm?id=1134285.1134355.

[30] Xavier Le Pallec et al. « A support to multi-devices web application ». In : *Adjunct proceedings of the 23nd annual ACM symposium on User interface software and technology - UIST '10*. UIST '10 (2010), p. 391. DOI : 10.1145/1866218.1866235. URL : http://portal.acm.org/citation.cfm?doid=1866218.1866235.

[31] Timothy C Lethbridge et King Edward Ave. « Perceptions of Software Modeling : A Survey of Software Practitioners Table of Contents ». In : *5th workshop from code centric to model centric : evaluating the effectiveness of MDD (C2M : EEMDD)*. March. 2008, p. 1–102. URL : https://www.site.uottawa.ca/eng/school/publications/techrep/2008/TR-2008-07-Survey-On-Software-Modeling-Forward-Lethbridge.pdf.

[32] Rodi Jolak Michel R.V. Chaudron. « A Vision on a New Generation of Software Design Environments (Empirical Theory) ». In : *HuFaMo@ MODELS*. HuFaMo (2015), p. 11–16. URL : http://ceur-ws.org/Vol-1522/Chaudron2015HuFaMo.pdf.

[33] Roberto Minelli et al. « Visualizing developer interactions ». In : *Proceedings - 2nd IEEE Working Conference on Software Visualization, VISSOFT 2014* (2014), p. 147–156. DOI : 10.1109/VISSOFT.2014.31.

[34] Audris Mockus. « Succession : Measuring transfer of code and developer productivity ». In : *Proceedings - International Conference on Software Engineering* (2009), p. 67–77. ISSN : 02705257. DOI : 10.1109/ICSE.2009.5070509.

[35] Daniel L Moody. « The physics of notations : Toward a scientific basis for constructing visual notations in software engineering ». In : *IEEE Transactions on Software Engineering* 35.6 (2009), p. 756–779. ISSN : 00985589. DOI : 10.1109/TSE.2009.67.

[36] Gary M. Olson, Sylvia. Sheppard et Elliot. Soloway. « Empirical studies of programmers : second workshop ». In : *Empirical studies of programmers : second workshop*. Sous la dir. de Gary M Olson, Sylvia Sheppard et Elliot Soloway. Norwood, NJ, USA : Ablex Publishing Corp., 1987. Chap. Parsing an, p. 263. ISBN : 0893914614. DOI : 10.1037/030774. URL : http://dl.acm.org/citation.cfm?id=54984.

[37] Renaud Pawlak et al. « SPOON : A library for implementing analyses and transformations of Java source code ». In : *Software - Practice and Experience* 46.9 (2016), p. 1155–1179. ISSN : 1097024X. DOI : 10.1002/spe.2346. arXiv : 1008.1900. URL : https://hal.archives-ouvertes.fr/hal-01078532/document.

[38] Marian Petre. « UML in practice ». In : *Proceedings - International Conference on Software Engineering* (2013), p. 722–731. ISSN : 02705257. DOI : 10.1109/ICSE.2013.6606618.

[39] S. P. Reiss. « The Challenge of Helping the Programmer during Debugging ». In : *2014 Second IEEE Working Conference on Software Visualization*. Sept. 2014, p. 112–116. DOI : 10.1109/VISSOFT.2014.27.

[40] Diomidis Spinellis. « A repository with 44 years of Unix evolution ». In : *IEEE International Working Conference on Mining Software Repositories* 2015-Augus (2015), p. 462–465. ISSN : 21601860. DOI : 10.1109/MSR.2015.64.

[41] Henry Tucker. « Data mountain : Using Spatial Memory for Document Management ». In : *Itnow* 55.3 (2013), p. 18–19. ISSN : 17465702. DOI : 10.1093/itnow/bwt040.

Annexe A

Modèles UML de VisUML

Sommaire

A.1 Diagramme de classes global

A.2 Diagramme de classes des entités

FIGURE A.1 – Diagramme de classe de VisUML

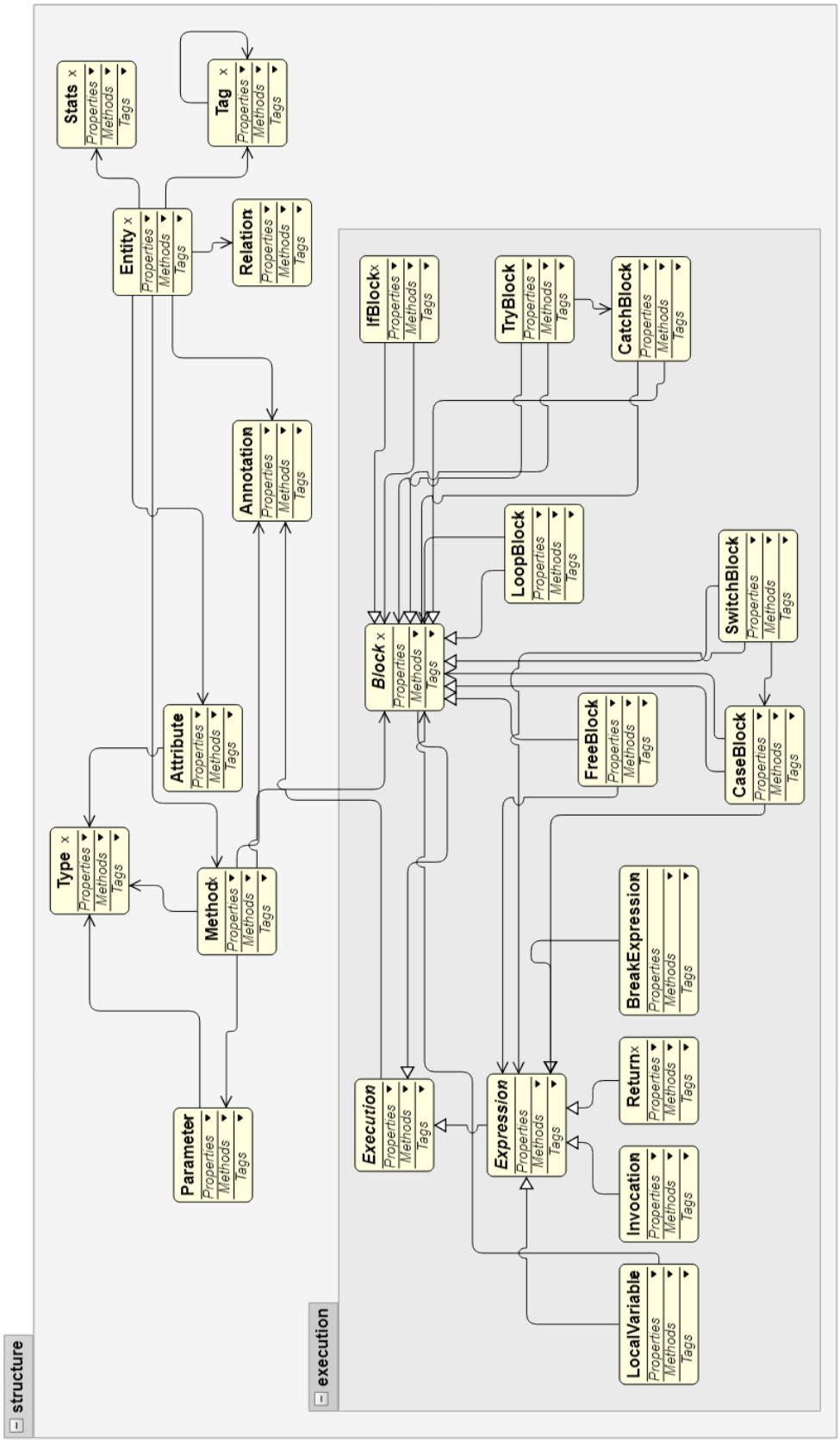


FIGURE A.2 – Diagramme de classes des entités de VisUML

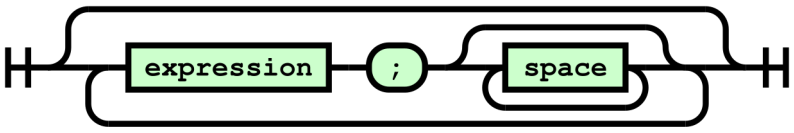
Annexe B

Grammaire des filtres VisUML

Sommaire

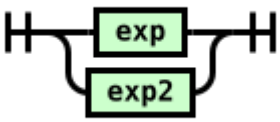
Cette annexe présente la grammaire utilisée pour les filtres sur VisUML.

start



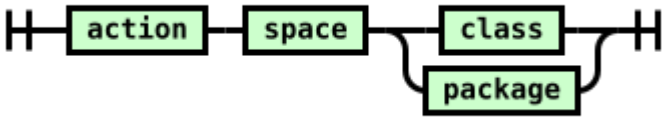
References : expression, space

expression



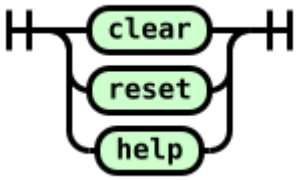
Used by : start
References : exp, exp2

exp



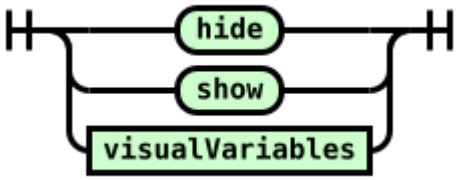
Used by : expression
References : action, space, class, package

exp2



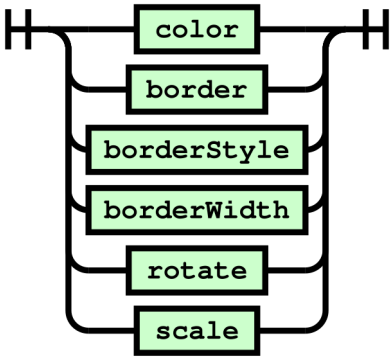
Used by : expression

action



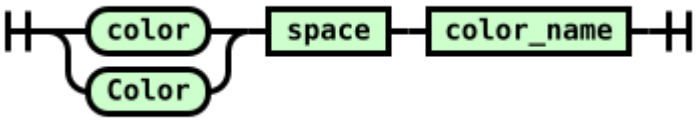
Used by : exp
References : visualVariables

visualVariables



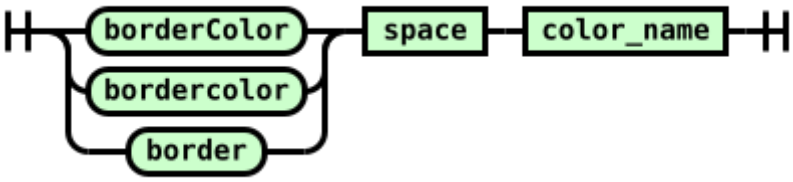
Used by : action
References : color, border, borderStyle, borderWidth, rotate, scale

color



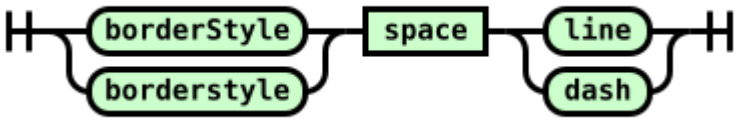
Used by : visualVariables
References : space, color_name

border



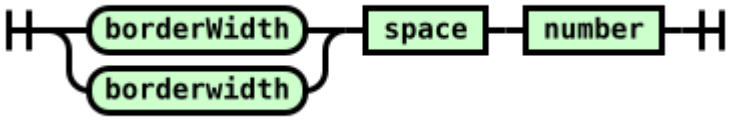
Used by : visualVariables
References : space, color_name

borderStyle



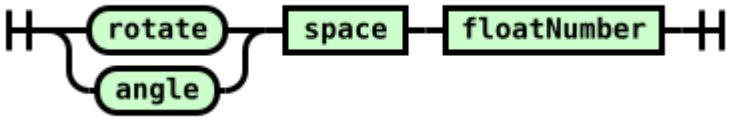
Used by : visualVariables
References : space

borderWidth



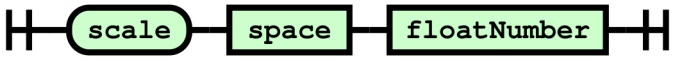
Used by : visualVariables
References : space, number

rotate



Used by : visualVariables
References : space, floatNumber

scale

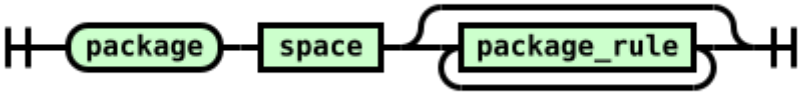


Used by : visualVariables
References : space, floatNumber

space

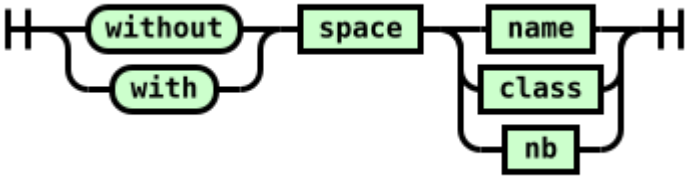
Used by : start, exp, color, border, borderStyle, borderWidth, rotate, scale, package, package_rule, class, class_rule_1, class_rule_2, name, name_filter, starts_ends, method, param, attribute, visibility, tag, nb, line, contains, type

package



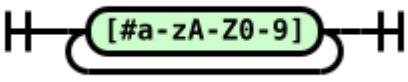
Used by : exp
References : space, package_rule

package_rule



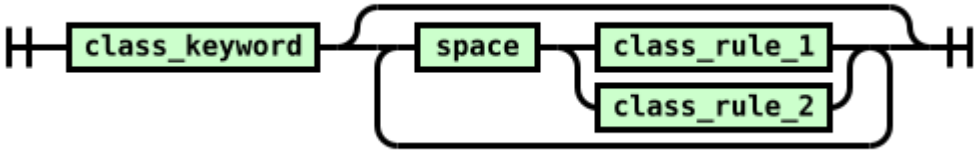
Used by : package
References : space, name, class, nb

color_name



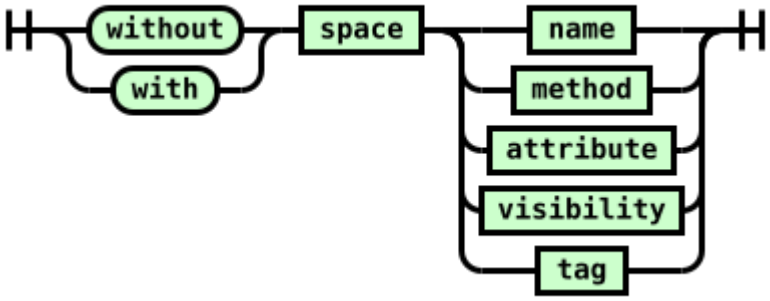
Used by : color, border

class



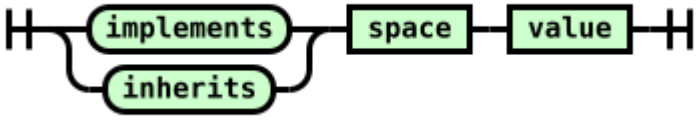
Used by : exp, package_rule
References : class_keyword, space, class_rule_1, class_rule_2

class_rule_1



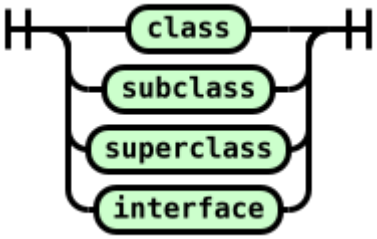
Used by : class
References : space, name, method, attribute, visibility, tag

class_rule_2



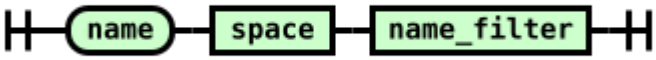
Used by : class
References : space, value

class_keyword



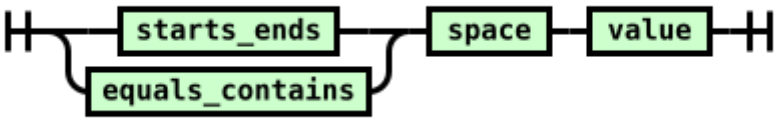
Used by : class

name



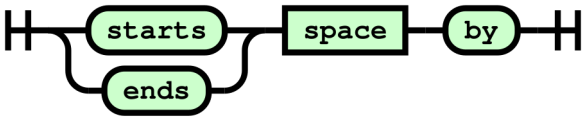
Used by : package_rule, class_rule_1, method, param, attribute References : space, name_filter

name_filter



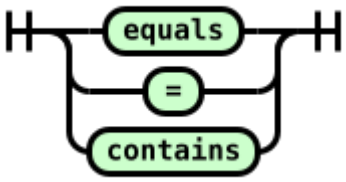
Used by : name
References : starts_ends, equals_contains, space, value

starts_ends



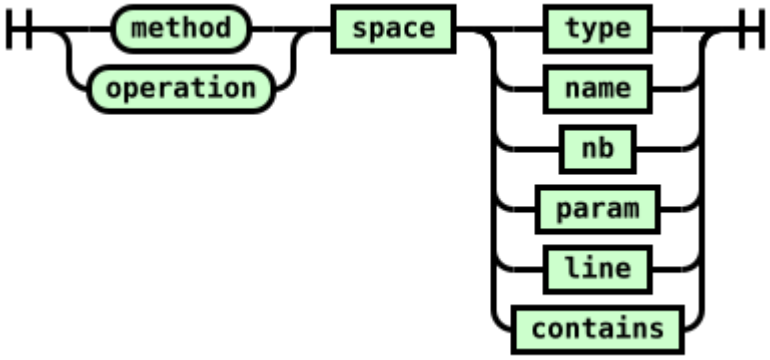
Used by : name_filter
References : space

equals_contains



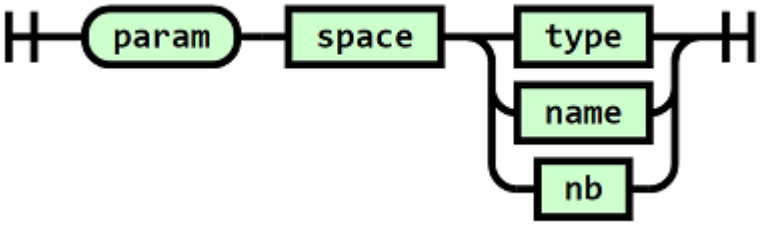
Used by : name_filter

method



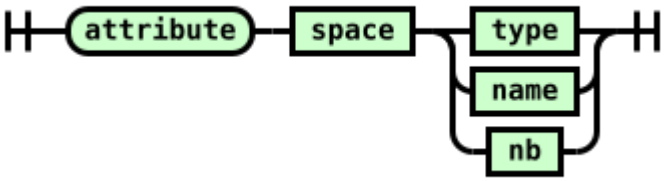
Used by : class_rule_1
References : space, type, name, nb, param, line, contains

param



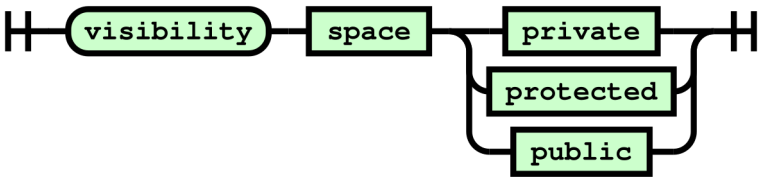
Used by : method
References : space, type, name, nb

attribute



Used by : class_rule_1
References : space, type, name, nb

visibility

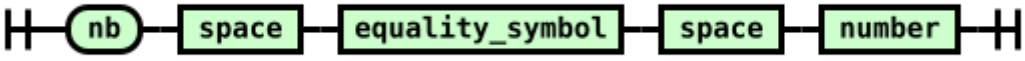


Used by : class_rule_1
References : space, private, protected, public

tag

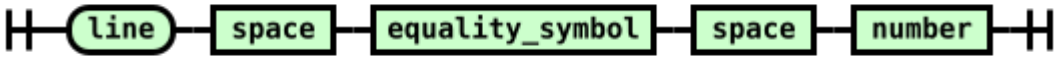
Used by : class_rule_1
References : space, value

nb



Used by : package_rule, method, param, attribute References : space, equality_symbol, number

line



Used by : method

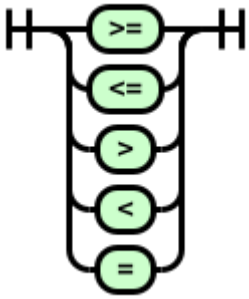
References : space, equality_symbol, number

contains

Used by : method

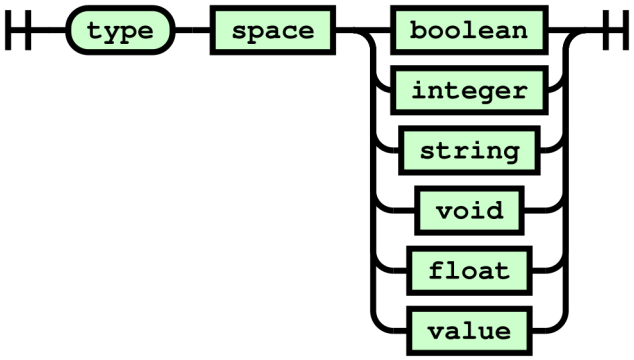
References : space, value

equality_symbol



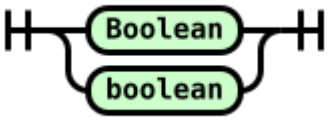
Used by : nb, line

type



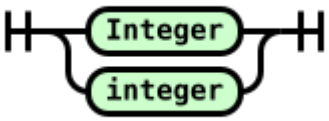
Used by : method, param, attribute
References : space, boolean, integer, string, void, float, value

boolean



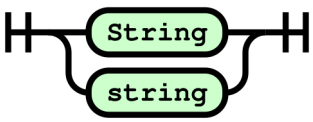
Used by : type

integer



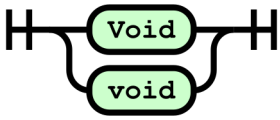
Used by : type

string



Used by : type

void

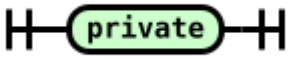


Used by : type

float

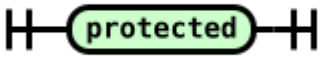
Used by : type

private



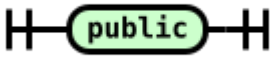
Used by : visibility

protected



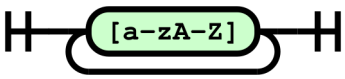
Used by : visibility

public



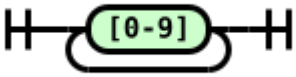
Used by : visibility

value



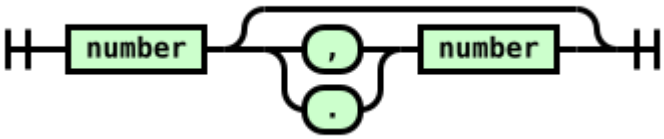
Used by : class_rule_2, name_filter, tag, contains, type

number



Used by : borderWidth, nb, line, floatNumber

floatNumber



Used by : rotate, scale References : number

Annexe C

Diagramme d'activité de VisUML

Cette annexe présente le prototype de diagramme d'activité développé au sein de VisUML.

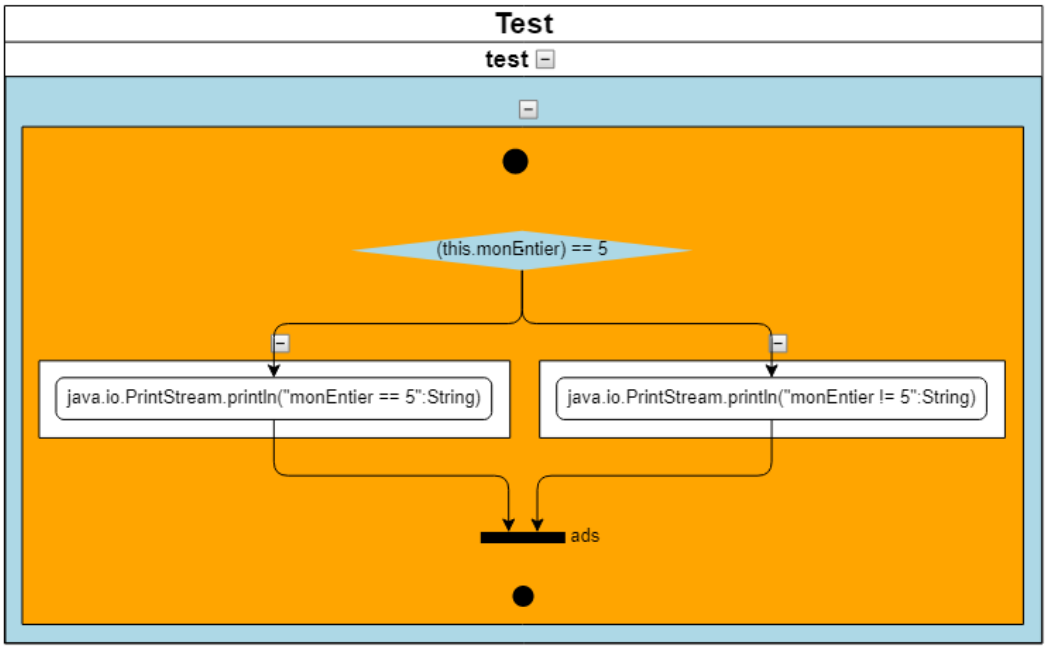


Figure C.1 – Diagramme d'activité d'une méthode simple

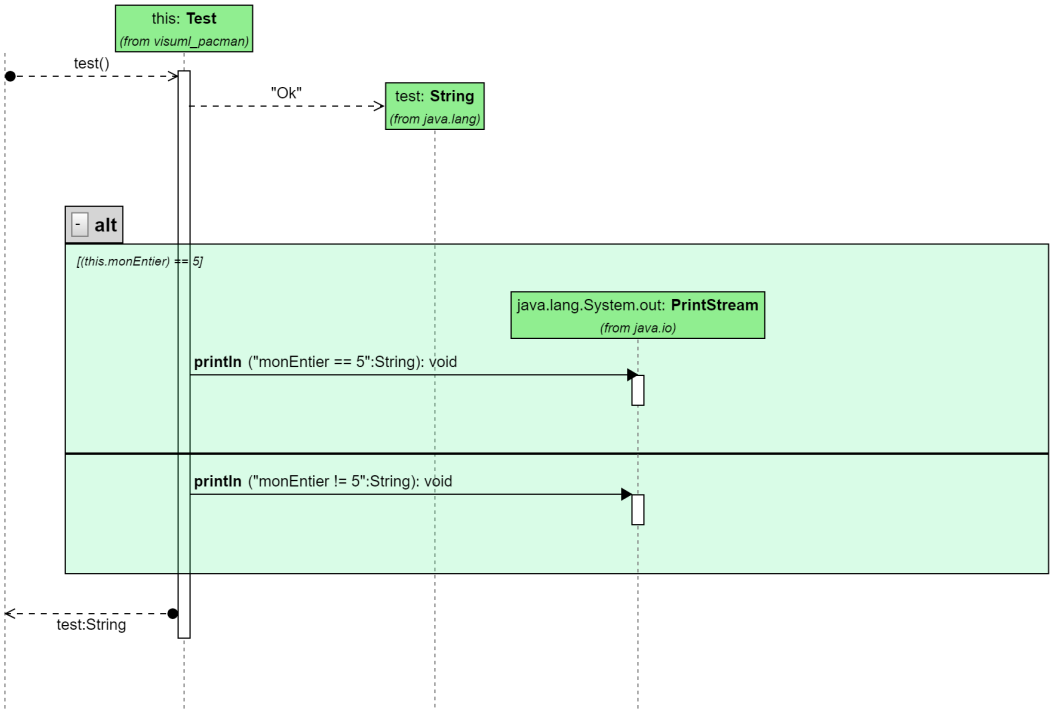


FIGURE C.2 – Diagramme de séquence comparatif (figure C.1)

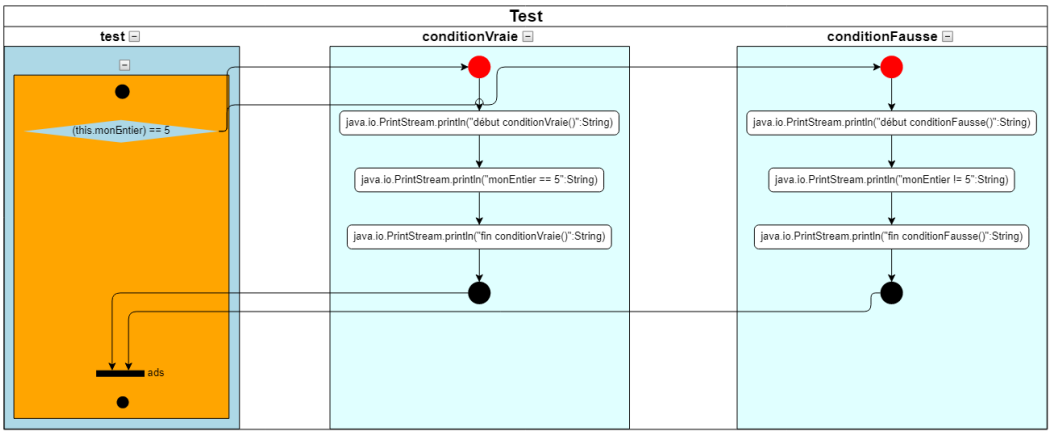


FIGURE C.3 – Diagramme d'activité combinant plusieurs méthodes

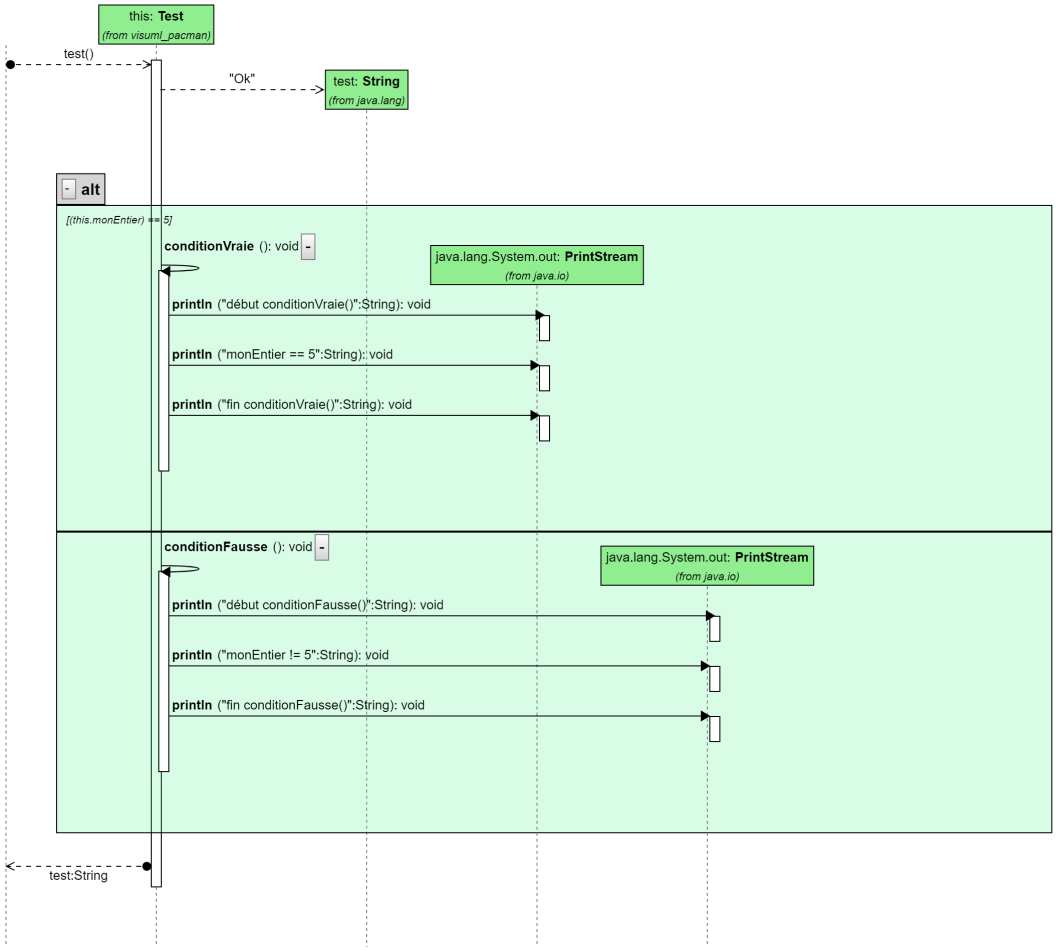


Figure C.4 – Diagramme de séquence comparatif (figure C.3)

Annexe D

Palette de commandes contextuelles

Cette annexe présente le prototype de palette de commandes contextuelles liées à VisUML. Ces commandes sont basées sur la notion de contexte associé aux éléments sélectionnés sur le diagramme de classe.

Il existe actuellement 13 contextes :

- Sélection vide (empty_selection)
- Un élément sélectionné parmi classe, interface ou énumération (element_1)
- Deux éléments sélectionnés parmi classe, interface ou énumération (element_2)
- N éléments sélectionnés parmi classe, interface ou énumération (element_N)
- Un package sélectionné (package_1)
- Deux packages sélectionnés (package_2)
- N packages sélectionnés (package_N)
- Un lien sélectionné (link_1)
- Deux liens sélectionnés (link_2)
- N liens sélectionnés (link_N)
- Un objet sélectionné (lien, package, classe, interface, énumération) (selection_1)
- Deux objets sélectionnés et de types différents (lien, package, classe, interface, énumération) (selection_2)
- N objets sélectionnés et de types différents (lien, package, classe, interface, énumération) (selection_N)

Nous avons également défini plusieurs actions (8 pour le moment), allant de la simple création d'élément à une action complexe pouvant regrouper plusieurs sous-actions :

— Créer une classe
— Créer une énumération
— Créer une interface
— Créer un package
— Déplacer un ou plusieurs éléments (voir figure D.1)
— Demander la suppression d'un ou plusieurs éléments
— Supprimer un ou plusieurs éléments
— Créer une classe mère pour un ou plusieurs éléments

L'action « Créer une classe mère pour un ou plusieurs éléments » est un exemple d'action complexe, permettant de créer un fichier de classe Java, puis de modifier tous les éléments sélectionnés par l'utilisateur pour les faire hériter de cette classe. Une action de ce genre nécessite normalement plusieurs clics et navigation au sein de l'EDI pour trouver puis modifier chaque élément. Avec cette palette contextuelle, l'utilisateur n'a qu'à sélectionner ses éléments puis cliquer une fois sur l'action (sur GenMyModel, une action similaire nécessite clics). Suite à cela, l'EDI propose de réaliser l'action en demandant la validation de l'utilisateur par l'intermédiaire d'une boîte de dialogue qui permet de vérifier les paramètres de l'action avant de l'exécuter (voir figure D.1 pour un exemple de déplacement de fichiers dans un paquetage)

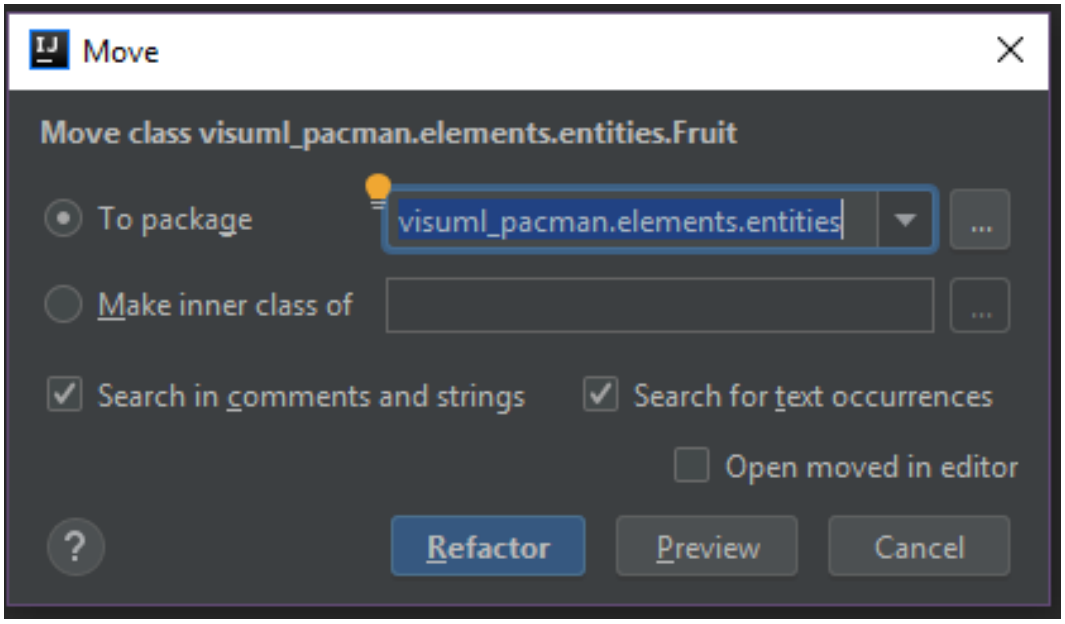


FIGURE D.1 – Palette contextuelle - Demande de *refactoring* sur l'EDI

Une action est définie en JSON avec le format suivant :

```
createClass: {
    id: "createClass",
    name: "Create a new class",
    simpleName: "New class",
    type: "create",
    subtype: "class",
    params: {},
    contexts: [ CONTEXT.EMPTY_SELECTION, CONTEXT.PACKAGE_1
        ],
    icons: "create",
    color: "blue",
    position: 1
}
```

Code D.1 – Palette contextuelle - Exemple d'action

Figure D.2 – Palette contextuelle - Contexte « empty_selection »

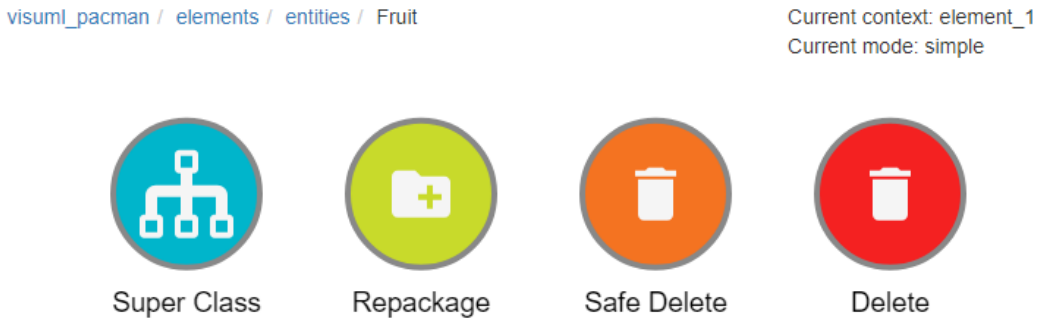
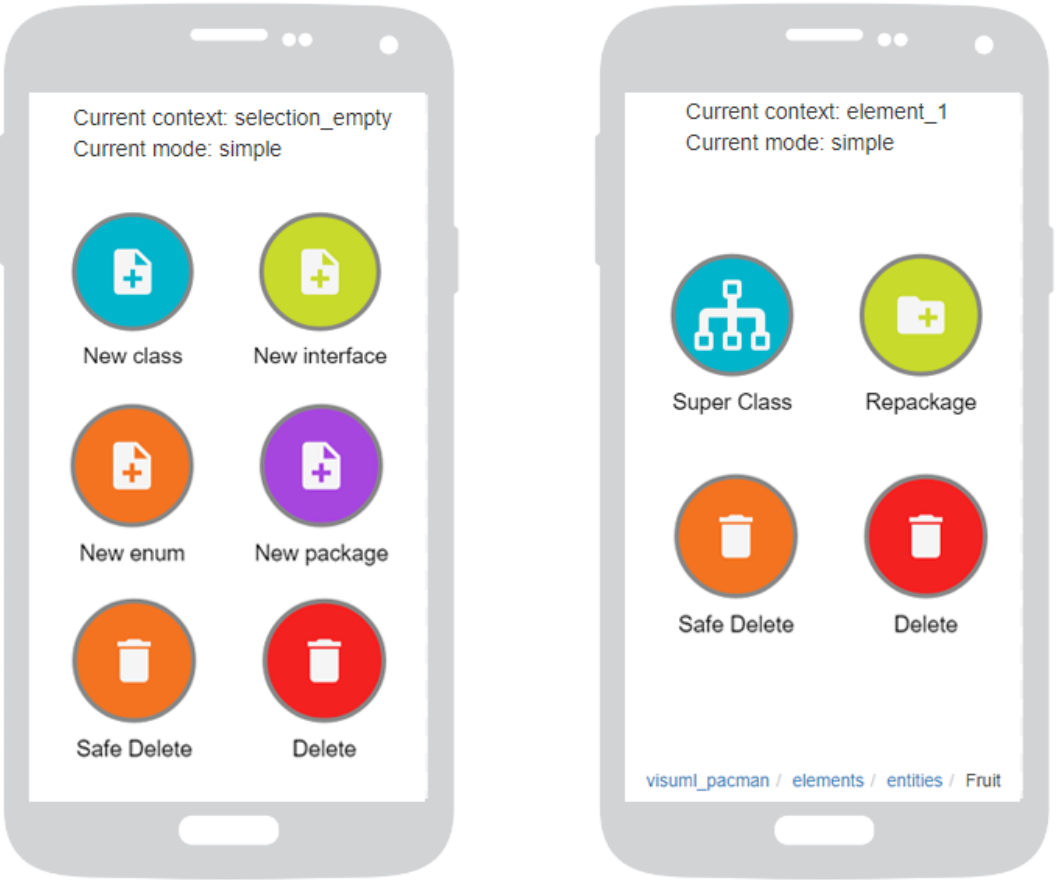


Figure D.3 – Palette contextuelle - Contexte « element_1 »



(a) Contexte « empty_selection » (b) Contexte « element_1 »

FIGURE D.4 – Palette contextuelle sur mobile

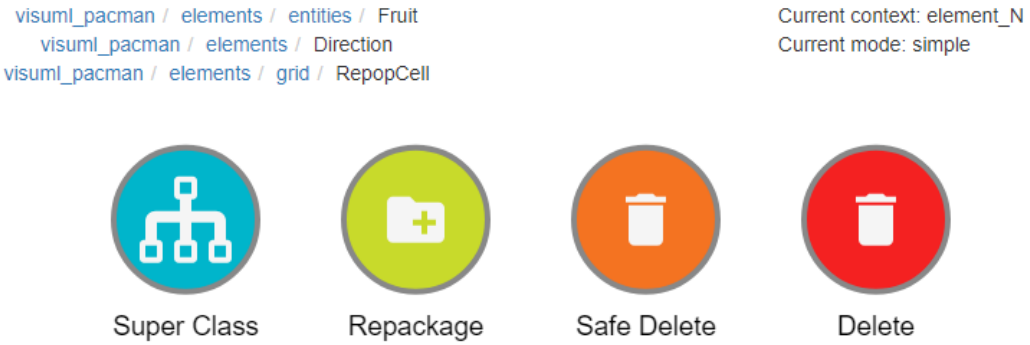


FIGURE D.5 – Palette contextuelle - Contexte « element_n »

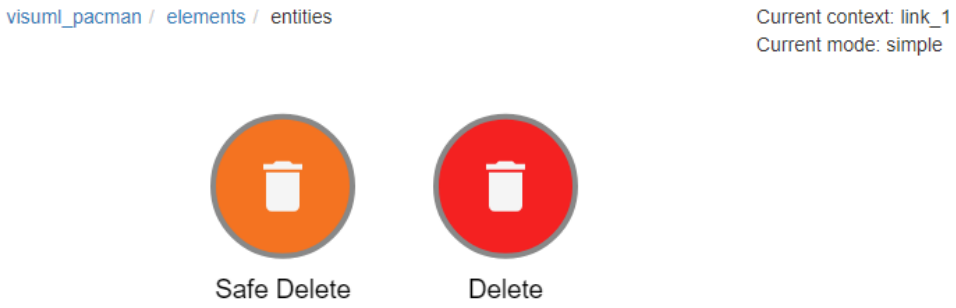


FIGURE D.6 – Palette contextuelle - Contexte « link_1 »

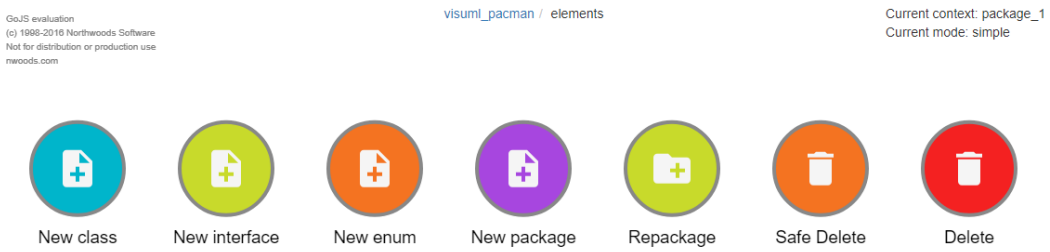


FIGURE D.7 – Palette contextuelle - Contexte « package_1 »

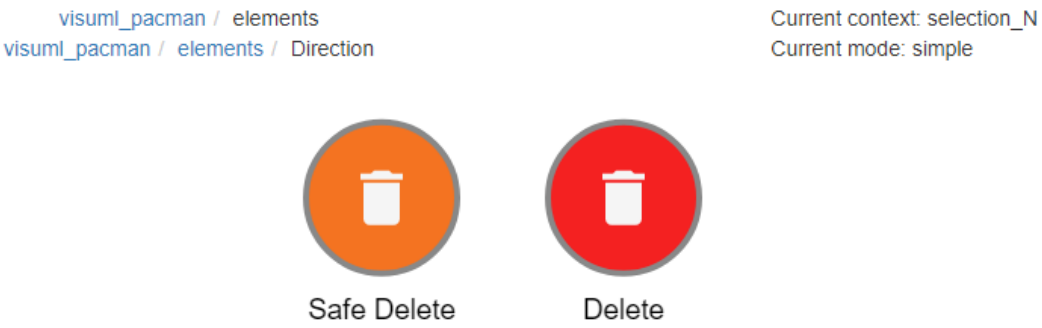


FIGURE D.8 – Palette contextuelle - Contexte « selection_n »

Annexe E

Questionnaires d'évaluations de VisUML

Sommaire

E.1 Questionnaire de début d'évaluation

Évaluation VisUML

*Obligatoire

1. **Numéro participant** *

Utilisation de schémas dans votre travail

2. **Intitulé du métier** *

Exemple: Ingénieur conception logiciel, Développeur, ...
Une seule réponse possible.

- ◯ Développeur full-stack
- ◯ Développeur Front
- ◯ Développeur Back
- ◯ Analyste Programmeur
- ◯ Ingénieur conception logiciels
- ◯ Ingénieur de recherche
- ◯ Autre : _____

3. **Activités principales** *

Donner pour chaque activité une estimation de vos compétences (1 = basse, 5 = élevée)
Une seule réponse possible par ligne.

	1	2	3	4	5
Analyse	◯	◯	◯	◯	◯
Développement de nouvelles fonctionnalités	◯	◯	◯	◯	◯
Tests	◯	◯	◯	◯	◯
Correction de bugs	◯	◯	◯	◯	◯

4. **La majorité du temps, vous travaillez avec :** *

Une seule réponse possible.

- ◯ Uniquement l'écran de votre portable
- ◯ Uniquement l'écran de votre PC de bureau (grand écran)
- ◯ Écran du portable et un autre écran
- ◯ Écran du PC et un autre écran
- ◯ Deux écrans (autre que portable)
- ◯ Autre : _____

5. **Nombre d'onglets ouverts sur l'IDE (en moyenne)**
Fichiers Java uniquement
Une seule réponse possible.

- ◯ Entre 1 et 5
- ◯ Entre 6 et 10
- ◯ Entre 11 et 15
- ◯ Entre 16 et 20
- ◯ Entre 21 et 25
- ◯ Entre 26 et 30
- ◯ Entre 31 et 35
- ◯ + de 35

6. **Faites vous des schémas ? (Base de données, architecture, maquette, ...)** *
Une seule réponse possible.

- ◯ Oui
- ◯ Non *Arrêtez de remplir ce formulaire.*

Schémas

7. **Type de schéma** *
Plusieurs réponses possibles.

- ☐ Maquettes écran
- ☐ Base de données
- ☐ Architecture (type client-serveur)
- ☐ Diagramme de classe
- ☐ Diagramme de séquence
- ☐ Diagramme de cas d'utilisation
- ☐ Autre : _____

8. **Sur quel(s) support(s) : maquettes écran**
Plusieurs réponses possibles.

- ☐ Papier
- ☐ Tableau
- ☐ Outil informatique
- ☐ Autre : _____

9. **Sur quel(s) support(s) : base de données**
   Plusieurs réponses possibles.

   - [ ] Papier
   - [ ] Tableau
   - [ ] Outil informatique
   - [ ] Autre : _____

10. **Sur quel(s) support(s) : architecture (type client serveur)**
    Plusieurs réponses possibles.

    - [ ] Papier
    - [ ] Tableau
    - [ ] Outil informatique
    - [ ] Autre : _____

11. **Sur quel(s) support(s) : diagramme de classe**
    Plusieurs réponses possibles.

    - [ ] Papier
    - [ ] Tableau
    - [ ] Outil informatique
    - [ ] Autre : _____

12. **Sur quel(s) support(s) : diagramme de séquence**
    Plusieurs réponses possibles.

    - [ ] Papier
    - [ ] Tableau
    - [ ] Outil informatique
    - [ ] Autre : _____

13. **Sur quel(s) support(s) : diagramme de cas d'utilisation**
    Plusieurs réponses possibles.

    - [ ] Papier
    - [ ] Tableau
    - [ ] Outil informatique
    - [ ] Autre : _____

14. **Sur quel(s) support(s) : autres**

_____

_____

_____

_____

_____

15. **Une fois vos schémas terminés, est-ce une ressource que vous utilisez ?** *
Une seule réponse possible.

- ◯ Non
- ◯ Rarement
- ◯ Régulièrement
- ◯ Régulièrement + modification
- ◯ Autre : _____

16. **Si oui, comment utilisez-vous ces ressources ?**

_____

_____

_____

_____

_____

17. **Si non, aimeriez-vous les utiliser et comment ?**

_____

_____

_____

_____

_____

18. **Utilisez-vous UML (même très peu) ?** *
Une seule réponse possible.

- ◯ Oui
- ◯ Non *Arrêtez de remplir ce formulaire.*

UML

Cette partie est destiné à évaluer votre connaissance et utilisation d'UML

19. **Niveau de connaissance UML** *
Une seule réponse possible.

	1	2	3	4	5	6	7	8	
Débutant	◯	◯	◯	◯	◯	◯	◯	◯	Expert

20. **Fréquence d'utilisation d'UML** *
Une seule réponse possible.

	1	2	3	4	5	6	7	8	
Jamais	◯	◯	◯	◯	◯	◯	◯	◯	Tous les jours

21. **Type d'utilisation d'UML** *

Il est connu que la plupart des gens utilisent UML de manière informelle, c'est à dire qu'ils utilisent par exemple le diagramme de classe mais sans respecter la totalité des conventions (ex: nombres de cases d'un classifier, cardinalités des liens, ...).
Une seule réponse possible.

	1	2	3	4	5	6	7	8	
Informelle	◯	◯	◯	◯	◯	◯	◯	◯	Formelle

22. **Diagramme le plus souvent utilisé** *

Une seule réponse possible.

- ◯ Diagramme de profils
- ◯ Diagramme de packages
- ◯ Diagramme de séquences
- ◯ Diagramme des cas d'utilisation
- ◯ Aucun
- ◯ Diagramme de classe
- ◯ Diagramme de communication
- ◯ Diagramme d'activités
- ◯ Diagramme de structure composite
- ◯ Diagramme de déploiement
- ◯ Diagramme de temps
- ◯ Diagramme de composants
- ◯ Diagramme états-transitions
- ◯ Diagramme d'objets
- ◯ Diagramme global d'interaction

23. **Second diagramme le plus souvent utilisé** *

Une seule réponse possible.

- ◯ Diagramme de séquences
- ◯ Diagramme global d'interaction
- ◯ Diagramme des cas d'utilisation
- ◯ Diagramme de communication
- ◯ Diagramme d'activités
- ◯ Diagramme de classe
- ◯ Diagramme d'objets
- ◯ Diagramme de temps
- ◯ Diagramme de déploiement
- ◯ Aucun
- ◯ Diagramme de structure composite
- ◯ Diagramme de packages
- ◯ Diagramme états-transitions
- ◯ Diagramme de profils
- ◯ Diagramme de composants

24. **Troisième diagramme le plus souvent utilisé** *

Une seule réponse possible.

- ◯ Diagramme de séquences
- ◯ Diagramme états-transitions
- ◯ Diagramme d'activités
- ◯ Diagramme des cas d'utilisation
- ◯ Diagramme d'objets
- ◯ Diagramme de classe
- ◯ Diagramme de composants
- ◯ Diagramme de communication
- ◯ Diagramme de déploiement
- ◯ Diagramme de structure composite
- ◯ Aucun
- ◯ Diagramme de profils
- ◯ Diagramme de temps
- ◯ Diagramme global d'interaction
- ◯ Diagramme de packages

25. **Autres diagrammes utilisés**

Plusieurs réponses possibles.

- ☐ Diagramme de profils
- ☐ Diagramme global d'interaction
- ☐ Diagramme états-transitions
- ☐ Diagramme de composants
- ☐ Diagramme de séquences
- ☐ Diagramme de classe
- ☐ Diagramme de structure composite
- ☐ Diagramme des cas d'utilisation
- ☐ Diagramme d'objets
- ☐ Diagramme d'activités
- ☐ Diagramme de déploiement
- ☐ Diagramme de communication
- ☐ Diagramme de temps
- ☐ Diagramme de packages

26. **Taille des projets** *

Une seule réponse possible.

	1	2	3	4	5	6	7	8	
Petit projet	◯	◯	◯	◯	◯	◯	◯	◯	Très gros projet

27. **Modélisation collaborative** *

A combien êtes-vous lorsque vous modélisez ?
Plusieurs réponses possibles.

- [ ] 1
- [ ] 2
- [ ] 3 et +

28. **Utilisation des outils de modélisation** *

Modélisez-vous sur papier, tableau ou sur un outil dédié ?
Plusieurs réponses possibles.

- [ ] Papier
- [ ] Tableau blanc
- [ ] Tableau interactif
- [ ] Outils web (GenMyModel, yuml, ...)
- [ ] Outils dédié (Papyrus, Visual Paradigm, ...)
- [ ] Autre : _____

29. **Utilisation des outils de modélisation (suite)**

Si vous utilisez un ou des outils de modélisation, lesquels ?
Plusieurs réponses possibles.

- [ ] Papyrus
- [ ] VisualParadigm
- [ ] Together
- [ ] StarUML
- [ ] ArgoUML
- [ ] Rational Rose
- [ ] BOUML
- [ ] Autre : _____

30. **Utilisation des outils web (suite)**

Si vous utilisez un ou des outils web, lesquels ? (A la fois outils de modélisation (GenMyModel, ..) et de dessin (Draw.io, ..))
Plusieurs réponses possibles.

- [ ] GenMyModel
- [ ] yUML
- [ ] Gliffy
- [ ] Draw.io
- [ ] umlet
- [ ] lucidchart
- [ ] Autre : _____

E.2 Questionnaire de fin d'évaluation

Évaluation VisUML - Questionnaire 2

Ce formulaire nous permet d'avoir votre avis sur VisUML et son utilisation

*Obligatoire

1. **Nom et prénom** *

2. **Adresse email**

Utilisation de VisUML

3. **Avis sur VisUML** *
   Une seule réponse possible.

	1	2	3	4	5	6	7	8	
Inutile	◯	◯	◯	◯	◯	◯	◯	◯	Utile

4. **Utilisation du diagramme de classe** *
   En tenant compte du temps global passé sur l'outils, selon vous, combien de temps (en %) avez vous passé sur le diagramme de classe

5. **Utilisation de l'IDE** *

6. **Utilisation du diagramme de séquence** *
   Même question, pour le diagramme de séquence

7. **Quelles interactions (sur les diagrammes) vous ont été les plus simples et efficaces ?** *
   Plusieurs réponses possibles.

   - ☐ Clic sur un attribut => Navigation dans le code
   - ☐ Clic sur une méthode => Navigation dans le code
   - ☐ Clic sur une méthode => Mise à jour du diagramme de séquence
   - ☐ Déplacement du curseur (depuis l'IDE) => Mise à jour du diagramme de séquence
   - ☐ Clic sur un message (ou lien du diagramme de séquence) => Navigation dans le code
   - ☐ Autre :

8. **Quelles informations manquaient sur les diagrammes ?**
Plusieurs réponses possibles.

- [ ] Informations de débug (en temps réel)
- [ ] Informations d'erreurs de syntaxe
- [ ] Informations sur les annotations
- [ ] Autre : _____

9. **Auriez-vous souhaité pouvoir modifier le diagramme pour mettre à jour le code ?** *
Une seule réponse possible.

- ◯ Oui
- ◯ Non
- ◯ Autre : _____

10. **A quelle fréquence utiliseriez-vous cet outil ?** *
Une seule réponse possible.

	1	2	3	4	5	6	7	8	
Jamais	◯	◯	◯	◯	◯	◯	◯	◯	Tous les jours

11. **Expliquez la réponse ci-dessus** *

_____

_____

_____

_____

_____

12. **Pensez-vous être allé plus vite dans votre tâche en utilisant VisUML ?** *
Une seule réponse possible.

- ◯ Oui
- ◯ Non

13. **Grâce/A cause de quoi ?** *

_____

_____

_____

_____

_____

14. **Avez vous des idées d'améliorations pour l'outil ?**

Fourni par
Google Forms

Annexe F

Code du projet évaluation de VisUML

Sommaire

F.1 Classe Game

```
package visuml_pacman;

import visuml_pacman.elements.entities.Entity;
import visuml_pacman.elements.entities.Ghost;
import visuml_pacman.elements.entities.Pacman;
import visuml_pacman.elements.grid.Cell;
import visuml_pacman.elements.grid.Grid;

import java.util.List;

public class Game {
    private static Game gameInstance = null;

    private Grid grid;

    private List<Entity> entities;
    private Pacman pacman;

    private int score = 0;

    private Game() {
    }

    public static Game getGame() {
        if (Game.gameInstance == null) {
            Game.gameInstance = new Game();
        }

        return Game.gameInstance;
    }

    public void initGame() {
        this.initGrid();
        this.initEntities();
        this.createPlayer();
    }

    public void initGrid() {
        Game game = Game.getGame();

        Grid grid = new Grid(Configuration.GRID_WIDTH, Configuration.
            GRID_HEIGHT);
        grid.initializeMaze();

        game.setGrid(grid);
    }
```

```
public void initEntities() {
    this.initGhosts();
    this.initFruits();
    this.initFood();
}


public void initGhosts() {
    for (int i = 0; i < Configuration.GHOST_LIMIT; i++) {
        Cell cell = this.getGrid().findAvailableRepopCell();
        if (cell != null) {
            Ghost ghost = new Ghost(cell.getX(), cell.getY());
            cell.setEntity(ghost);
            entities.add(ghost);
        } else {
            // There is no more free space
            break;
        }
    }
}

public void initFruits() {

}

private void initFood() {
    Grid grid = this.getGrid();
    grid.fillWithFood(this.getEntities());
}

public void createPlayer() {

}

public void startGame() {

}

public void upScore() {
    ++this.score;
}

public void upScore(int point) {
    this.score += point;
}

public Grid getGrid() {
    return grid;
```

```
    }

    public void setGrid(Grid grid) {
        this.grid = grid;
    }

    public List<Entity> getEntities() {
        return entities;
    }

    public void setEntities(List<Entity> entities) {
        this.entities = entities;
    }

    public Pacman getPacman() {
        return pacman;
    }

    public void setPacman(Pacman pacman) {
        this.pacman = pacman;
    }

    public int getScore() {
        return score;
    }
}
```

Code F.1 – Classe Game

F.2 Classe Configuration

```
package visuml_pacman;

public class Configuration {
    public static final int GRID_WIDTH = 20;
    public static final int GRID_HEIGHT = 20;

    public static final int GHOST_LIMIT = 4;

    public static final String[] GRID_MAZE =
            {
                    "WWWWWWWWWWWWWWWWWW",
                    "WEEEEEEEEEEEEEEEEW",
                    "WEEEEEEEWEEEEEEEEW",
                    "WEEEEEEEWEEEEEEEEW",
                    "WEEEEEEEWEEEEEEEEW",
                    "WEEEEEEEWEEEEEEEEW",
                    "WEEEEEEEWEEEEEEEEW",
                    "WEEEEEEEEEEEEEEEEW",
```

```
                        "WEEEEEWWEEWWEEEEEEEW",
                        "WEEEEEWRRRRWEEEEEEEW",
                        "WEEEEEWRRRRWEEEEEEEW",
                        "WEEEEWWWWWWWWWWEEEEW",
                        "WEEEEEEEEEEEEEEEEEEW",
                        "WEEEEEEEEWEEEEEEEEEW",
                        "WEEEEEEEEWEEEEEEEEEW",
                        "WEEEEEEEEWEEEEEEEEEW",
                        "WEEEEEEEEWEEEEEEEEEW",
                        "WEEEEEEEEWEEEEEEEEEW",
                        "WEEEEEEEEEEEEEEEEEEW",
                        "WWWWWWWWWWWWWWWWWWW"
                };

    public static final int PACMAN_EATING_DURATION = 5;

    private Configuration() {
    }
}
```

Code F.2 – Classe Configuration

F.3 Énumération Direction

```
package visuml_pacman.elements;

public enum Direction {
    UP,
    RIGHT,
    DOWN,
    LEFT
}
```

Code F.3 – Énumération Direction

F.4 Énumération ElementType

```
package visuml_pacman.elements;

public enum ElementType {
    ENTITY,
    CELL
}
```

Code F.4 – Énumération ElementType

F.5 Interface Movable

```
package visuml_pacman.elements;

public interface Movable {
    void move();
}
```

Code F.5 – Interface Movable

F.6 Classe Point

```
package visuml_pacman.elements;

public class Point {
    private int x = 0;
    private int y = 0;

    public Point() {
        this(0, 0);
    }

    public Point(Point point) {
        this(point.getX(), point.getY());
    }

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }

    public Point computeNewPosition(Direction direction) {
        Point newPoint = new Point(this);
        int newX = newPoint.getX(), newY = newPoint.getY();
```

```
        switch (direction) {
            case UP:
                newY--;
                break;
            case RIGHT:
                newX++;
                break;
            case DOWN:
                newY++;
                break;
            case LEFT:
                newX--;
        }

        newPoint.setX(newX);
        newPoint.setY(newY);

        return newPoint;
    }
}
```

Code F.6 – Classe Point

F.7 Classe Element

```
package visuml_pacman.elements;

public class Element {
    public static int elementId = 0;

    public int id;

    public ElementType elementType;

    protected int x = 0;
    protected int y = 0;

    protected Point position = new Point(0, 0);

    public Element(ElementType elementType) {
        this(elementType, 0, 0);
    }

    public Element(ElementType elementType, int x, int y) {
        this.elementType = elementType;

        this.id = Element.elementId++;

        this.setX(x);
```

```
        this.setY(y);
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
        this.setPosition(new Point(this.x, this.y));
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
        this.setPosition(new Point(this.x, this.y));
    }

    public Point getPosition() {
        return position;
    }

    protected void setPosition(Point position) {
        this.position = position;
        this.x = position.getX();
        this.y = position.getY();
    }
}
```

Code F.7 – Classe Element

F.8 Classe Grid

```
package visuml_pacman.elements.grid;

import visuml_pacman.Configuration;
import visuml_pacman.elements.entities.Entity;
import visuml_pacman.elements.entities.Food;

import java.util.List;

public class Grid {
    private int width;
    private int height;

    private Cell[][] cells;
```

```
public Grid(int w, int h) {
    this.width = w;
    this.height = h;

    this.cells = new Cell[h][w];
    for (h = 0; w < this.height; h++) {
        for (w = 0; w < this.width; w++) {
            this.cells[h][w] = new EmptyCell();
        }
    }

    this.initializeMaze();
}

public boolean isCellAccessible(int x, int y) {
    Cell cell = this.getCell(x, y);
    if (cell == null)
        return false;

    return cell instanceof EmptyCell;
}

public Cell getCell(int x, int y) {
    if (x > 0 && x < this.width && y > 0 && y < this.height)
        return this.cells[y][x];

    return null;
}

public Cell[][] getCells() {
    return cells;
}

public void initializeMaze() {
    for (int y = 0; y < this.height; y++) {
        String line = Configuration.GRID_MAZE[y];
        for (int x = 0; x < this.width; x++) {
            String cellType = "" + line.charAt(x);
            Cell cell = getCellFromType(cellType);
            this.cells[y][x] = cell;
        }
    }
}

private Cell getCellFromType(String type) {
    Cell cell;
    switch (type) {
        case "W":
```

```
                cell = new Wall();
                break;
            case "R":
                cell = new RepopCell();
                break;
            case "E":
            default:
                cell = new EmptyCell();
        }

        return cell;
    }

    public void fillWithFood(List<Entity> entities) {
        for (int y = 0; y < this.height; y++) {
            for (int x = 0; x < this.width; x++) {
                Cell cell = this.getCell(x, y);
                if (cell instanceof EmptyCell && cell.getEntity() ==
                    null) {
                    Food food = new Food(x, y);
                    cell.setEntity(food);
                    entities.add(food);
                }
            }
        }
    }

    public Cell findAvailableRepopCell() {
        for (int y = 0; y < this.height; y++) {
            for (int x = 0; x < this.width; x++) {
                Cell cell = this.getCell(x, y);
                if (cell instanceof RepopCell && cell.getEntity() ==
                    null) {
                    return cell;
                }
            }
        }

        return null;
    }
}
```

Code F.8 – Classe Grid

F.9 Classe Cell

```
package visuml_pacman.elements.grid;

import visuml_pacman.elements.Element;
```

```
import visuml_pacman.elements.ElementType;
import visuml_pacman.elements.entities.Entity;

public class Cell extends Element {
    private Entity entity;

    public Cell() {
        super(ElementType.CELL);
    }

    public Entity getEntity() {
        return entity;
    }

    public void setEntity(Entity entity) {
        this.entity = entity;
    }
}
```

Code F.9 – Classe Cell

F.10 Classe EmptyCell

```
package visuml_pacman.elements.grid;

public class EmptyCell extends Cell {

    public EmptyCell() {

    }
}
```

Code F.10 – Classe Empty

F.11 Classe RepopCell

```
package visuml_pacman.elements.grid;

public class RepopCell extends Cell {

}
```

Code F.11 – Classe RepopCell

F.12 Classe Wall

```
package visuml_pacman.elements.grid;
```

```
public class Wall extends Cell {

    public Wall() {

    }
}
```

Code F.12 – Classe Wall

F.13 Classe Entity

```
package visuml_pacman.elements.entities;

import visuml_pacman.elements.Element;
import visuml_pacman.elements.ElementType;

public class Entity extends Element {
    public static int entityId = 0;

    public int id;

    public EntityType type;

    public Entity(EntityType type) {
        this(type, 0, 0);
    }

    public Entity(EntityType type, int x, int y) {
        super(ElementType.ENTITY, x, y);

        this.id = Entity.entityId++;
        this.type = type;
    }

    public boolean equals(Entity entity) {
        return this.id == entity.id;
    }
}
```

Code F.13 – Classe Entity

F.14 Énumération EntityType

```
package visuml_pacman.elements.entities;

public enum EntityType {
    GHOST,
```

```
    PACMAN,
    FRUIT,
    FOOD
}
```

Code F.14 – Énumération EntityType

F.15 Classe Food

```
package visuml_pacman.elements.entities;

public class Food extends Entity {
    public Food() {
        super(EntityType.FOOD);
    }

    public Food(int x, int y) {
        super(EntityType.FOOD, x, y);
    }
}
```

Code F.15 – Classe Food

F.16 Classe Fruit

```
package visuml_pacman.elements.entities;

public class Fruit extends Entity {
    public Fruit() {
        super(EntityType.FRUIT);
    }
}
```

Code F.16 – Classe Fruit

F.17 Classe Ghost

```
package visuml_pacman.elements.entities;

public class Ghost extends Entity {
    private boolean isScared = false;

    public Ghost() {
        super(EntityType.GHOST);
    }

    public Ghost(int x, int y) {
        super(EntityType.GHOST, x, y);
```

```
    }

    public boolean isScared() {
        return isScared;
    }

    public void setScared(boolean scared) {
        isScared = scared;
    }
}
```

Code F.17 – Classe Ghost

F.18 Classe Pacman

```
package visuml_pacman.elements.entities;

import visuml_pacman.Configuration;
import visuml_pacman.Game;
import visuml_pacman.elements.Direction;
import visuml_pacman.elements.Movable;
import visuml_pacman.elements.Point;
import visuml_pacman.elements.grid.Cell;
import visuml_pacman.elements.grid.Grid;

public class Pacman extends Entity implements Movable {
    private Direction direction = Direction.UP;
    private boolean canEatGhost = false;
    private int canEatGhostRemainingTick = Configuration.
       PACMAN_EATING_DURATION;
    private boolean isAlive = true;

    public Pacman() {
        super(EntityType.PACMAN);
    }

    @Override
    public void move() {
        Game game = Game.getGame();
        Grid grid = game.getGrid();

        Point currentPosition = this.getPosition();
        Point newPosition = currentPosition.computeNewPosition(this.
           direction);

        if (grid.isCellAccessible(newPosition.getX(), newPosition.
           getY())) {
            Cell previousCell = grid.getCell(this.getX(), this.getY()
               );
```

```
            Cell cell = grid.getCell(newPosition.getX(), newPosition.
                getY());

            previousCell.setEntity(null);

            this.setPosition(newPosition);

            Entity entity = cell.getEntity();
            this.handleEntity(entity);

            if (this.isAlive())
                cell.setEntity(this);
        }

        this.checkEatingDuration();
    }

    private void handleEntity(Entity entity) {
        if (entity instanceof Food) {
            Game.getGame().getEntities().remove(entity);
            Game.getGame().upScore();
        }
        if (entity instanceof Fruit) {
            Game.getGame().getEntities().remove(entity);
            this.setCanEatGhost(true);
            this.canEatGhostRemainingTick = Configuration.
                PACMAN_EATING_DURATION;
        }
        if (entity instanceof Ghost) {
            if (this.canEatGhost()) {
                Game.getGame().getEntities().remove(entity);
                Game.getGame().upScore(5);
            } else {
                this.die();
            }
        }
    }

    private void checkEatingDuration() {
        this.canEatGhostRemainingTick--;
        if (this.canEatGhostRemainingTick <= 0)
            this.setCanEatGhost(false);
    }

    public Direction getDirection() {
        return direction;
    }

    public void setDirection(Direction direction) {
```

```
        this.direction = direction;
    }

    public boolean canEatGhost() {
        return canEatGhost;
    }

    public void setCanEatGhost(boolean canEatGhost) {
        this.canEatGhost = canEatGhost;
    }

    public boolean isAlive() {
        return this.isAlive;
    }

    public void die() {
        this.isAlive = false;
    }
}
```

Code F.18 – Classe Pacman

Annexe G

Actions disponibles dans VisUML

Sommaire

G.1 Diagramme de classe (+30 actions)

1. Filtrage (avec le menu gauche)
   1.1. Afficher uniquement les paquetages (« package only »)
   1.2. Afficher les paquetages, et les classes sans détail
   1.3. Afficher les paquetages, et les classes avec les attributs et méthodes, mais uniquement les éléments publiques et protégés (« Public & protected elements »)
   1.4. Afficher les paquetages et les classes avec les attributs et toutes les méthodes, mais sans les mots-clés (« Tags hidden »)
   1.5. Afficher les paquetages et les classes avec les attributs, toutes les méthodes et tous les mots-clés (« All details »)
   1.6. Afficher ou cacher les « getters » et « setters »
   1.7. Montrer le diagramme uniquement pour l'onglet actif dans l'IDE
   1.8. Afficher ou cacher les éléments associés

1.9. Choisir les paquetages qu'on veut afficher ou masquer (« Packages visibility »)

1.10. Choisir les éléments (classes, interfaces…) qu'on veut afficher ou masquer (« Elements visibility »)

2. Filtrage (avec la ligne de commande en bas)

2.1. Filtrage sur les Tags (par exemple « show class … with tag =… »)

2.2. Filtrage sur les informations du code source (par exemple « show class with method name contains … »)

3. Actions sur la représentation UML (changement de bordure, de couleur de fond, rotation, échelle…) par la ligne de commande en bas

4. Zoom sur le diagramme

4.1. Zoom pour tout afficher

4.2. Reset zoom (niveau de zoom à 100

4.3. Reset layout (recréation du diagramme)

4.4. Zoom par souris (CTRL+molette)

5. Navigation sur le diagramme

5.1. Navigation avec la souris (déplacement du diagramme par clic sur le fond)

5.2. Navigation avec la vue radar (déplacement du radar pour se déplacer sur le diagramme)

6. Interactions sur le diagramme

6.1. Ouvrir ou fermer un paquetage (simple clic)

6.2. Fermer une classe (simple clic)

6.3. Ouvrir ou fermer les attributs/méthodes/tags (simple clic sur triangle associé)

6.4. Ouvrir une classe associée et non ouverte dans l'IDE (simple clic)

6.5. Sélectionner une classe (simple clic)

6.6. Sélectionner une méthode (simple clic)

6.7. Sélectionner un attribut (simple clic)

6.8. Interactions sur les Tags (mots-clés)

i. Monter ou descendre un mot-clé

ii. Supprimer un mot-clé

6.9. Déplacement de classes et paquetages (drag&drop avec la souris)

6.10. Affichage des commentaires/javadoc/annotations d'une méthode (par rollover sur la méthode)
6.11. Affichage des commentaires et du nom du package du type de l'attribut (par rollover sur un attribut)
6.12. Affichage des mots-clés associés à un tag dans l'ontologie (par rollover sur le tag)
7. Exportation du diagramme de classe sous la forme d'un fichier SVG ou PNG (non encore implémentée)

G.2 Diagramme de séquence (+20 actions)

1. Sélection de la classe et de la méthode à visualiser
2. Zoom sur le diagramme
   2.1. Zoom pour tout afficher
   2.2. Reset zoom
   2.3. Bouton pour régler le niveau de zoom
   2.4. Zoom par souris
3. Navigation sur le diagramme
   3.1. Déplacement du diagramme avec la souris par un clic sur le fond
   3.2. Déplacement du diagramme avec la vue radar
   3.3. Scroll to top
   3.4. Center
4. Navigation dans les séquences UML
   4.1. Bouton pour régler le niveau de profondeur
   4.2. Bouton « + » pour afficher le détail d'une séquence
   4.3. Affichage du diagramme de séquence d'une méthode appelante ou appelée (par alt+clic)
5. Export PNG/SVG
6. Interactions sur le diagramme
   6.1. Sélectionner une séquence (simple clic)
   6.2. Sélectionner une ligne de vie/objet (simple clic)
   6.3. Sélectionner un bloc (simple clic)
   6.4. Sélectionner un bouton de fermeture/ouverture de bloc (simple clic)
   6.5. Sélectionner un appel parent (simple clic)
   6.6. Rollover sur méthodes (commentaire, javadoc, . . .)
7. Affichage en mode « Code Bubbles »

G.3 Interactions dans l'EDI (+3 actions)

1. Ouverture/fermeture de fichiers code source
2. Changement d'onglet
3. Navigation dans un fichier (par clic ou clavier)

G.4 « Landing page » (4 actions)

1. Ouverture du diagramme de classe
2. Ouverture du diagramme de séquence
3. Ouverture de la télécommande
4. Informations sur le projet (nom de la session, URL WSE, nom du projet)

G.5 Palette contextuelle (+8 actions)

1. Créer un élément (classe, énumération, interface, paquetage)
2. Créer une classe mère
3. Déplacer dans un package
4. Supprimer un ou plusieurs éléments (de façon sécurisée ou forcée)

Table des matières

Améliorer la compréhension d'un programme à l'aide de diagrammes dynamiques et interactifs

Résumé

Les développeurs occupent une place prépondérante dans le développement logiciel. Dans ce cadre, ils doivent réaliser une succession de tâches élémentaires (analyse, codage, liaison avec le code existant...), mais pour effectuer ces tâches, un développeur doit régulièrement changer de contexte de travail (recherche d'information, lecture de code...) et analyser du code qui n'est pas le sien. Ces actions nécessitent un temps d'adaptation élevé et réduisent l'efficacité du développeur. La modélisation logicielle est une solution à ce type de problème. Elle propose une vue abstraite d'un logiciel, des liens entre ses entités ainsi que des algorithmes utilisés. Cependant, l'Ingénierie Dirigée par les Modèles (IDM) est encore trop peu utilisée en entreprise. Dans cette thèse, nous proposons un outil pour améliorer la compréhension d'un programme à l'aide de diagrammes dynamiques et interactifs. Cet outil se nomme VisUML et est centré sur l'activité principale de codage du développeur. VisUML fournit des vues (sur des pages web ou sur des outils de modélisation) synchronisées avec le code. Les diagrammes UML générés sont interactifs et permettent une navigation rapide avec et dans le code. Cette navigation réduit les pertes de temps et de contextes dues aux changements d'activités en fournissant à tout moment une vue abstraite sous forme de diagramme des éléments actuellement ouverts dans l'outil de codage du développeur. Au final, VisUML a été évalué par vingt développeurs dans le cadre d'une expérimentation qualitative de l'outil afin d'estimer l'utilité d'un tel outil.

Mots clés : ingénierie dirigée par les modèles (idm), interactions homme-machine (ihm), unified modeling language (uml), visualisation de programme, compréhension de programme, recherche d'informations

---

CRIStAL UMR 9189
Université Lille – Bâtiment M3 extension – Avenue Carl Gauss – 59655 Villeneuve d'Ascq Cedex – FRANCE

Enhance software comprehension with dynamic and interactive diagrams

Abstract

Developers dominate in software development. In this context, they must perform a succession of elementary tasks (analysis, coding, linking with existing code ...), but in order to perform these tasks, a developer must regularly change his context of work (search information, read code ...) and analyze code that is not his. These actions require a high adaptation time and reduce the efficiency of the developer. Software modeling is a solution to this type of problem. It offers an abstract view of a software, links between its entities as well as algorithms used. However, Model-Driven Engineering (MDE) is still underutilized in business. In this thesis, we propose a tool to improve the understanding of a program using dynamic and interactive diagrams. This tool is called VisUML and focuses on the main coding activity of the developer. VisUML provides views (on web pages or modeling tools) synchronized with the code.The generated UML diagrams are interactive and allow fast navigation with and in the code. This navigation reduces the loss of time and context due to activity changes by providing at any time an abstract diagram view of the elements currently open in the developer's coding tool. In the end, VisUML was evaluated by twenty developers as part of a qualitative experimentation of the tool to estimate the usefulness of such a tool.

Keywords: model driven engineering (mde), human–computer interaction (hci), unified modeling language (uml), software visualization, program comprehension, information retrieval