## Université de Montréal


# Abstraction de comportement de haut niveau à l'aide de la visualisation interactive


par

**Dorian Vandamme**

Département d'Informatique et de Recherche Opérationnelle
Faculté des arts et des sciences


Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Génie Logiciel

19 novembre 2021

# Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

## Abstraction de comportement de haut niveau à l'aide de la visualisation interactive

présenté par

## Dorian Vandamme

a été évalué par un jury composé des personnes suivantes :

*Eugene Syriani*
(président-rapporteur)

*Houari Sahraoui*
(directeur de recherche)

*Pierre Poulin*
(codirecteur)

*Derek Nowrouzezahrai*
(membre du jury)

# Résumé

Comprendre le comportement de haut niveau des programmes est nécessaire pour effectuer différentes tâches dans le développement et la maintenance des logiciels. Pour cela, on utilise régulièrement des traces d'exécution du système, enregistrées pendant des scénarios d'utilisation typiques. Cependant, une trace standard peut contenir un volume très important d'évènements, ce qui rend son interprétation difficile. Nous proposons d'utiliser une métaphore visuelle et un ensemble de filtres et d'outils pour assister le développeur dans la compréhension du comportement de haut niveau d'un logiciel. Notre environnement de visualisation interactive est basé sur une métaphore de traces lumineuses animées pour rejouer la trace d'exécution. Cette animation est accompagnée avec un ensemble de filtres et d'outils pour manipuler et réduire à l'essentiel les informations affichées. Nous démontrons l'utilité de notre approche au moyen de deux études de cas qui présentent des traces enregistrées sur un jeu d'échecs et un logiciel d'édition de diagrammes UML.

***Mots-clés*** **: Visualisation du logiciel, interaction, trace d'exécution, filtre, edge bundles.**

# Abstract

Understanding high-level behavior of programs is necessary to perform various tasks in software development and maintenance. This is usually done by analyzing execution traces extracted from typical scenarios. However, average execution traces consist of huge volumes of events and information that make it difficult to develop good insights from these traces. We propose to exploit a visualization metaphor and a set of filters and tools to assist developers grasping high-level behaviors of programs. Our interactive visualization is based on a metaphor of traces of light as part of an animation to explore execution scenarios. The animation is augmented with a set of structural and temporal filters to reduce the volume of information displayed. We showcase our visualization environment on two case studies featuring programs of a chess game and a UML diagram editor.

***Index Terms*** **: Software visualization, interaction, execution trace, filter, edge bundles.**

# Table des matières

**Résumé** . . . 3

**Abstract** . . . 4

**Table des figures** . . . 7

**Remerciements** . . . 9

**Chapitre 1. Introduction** . . . 10

1.1. Contexte . . . 10

1.2. Problématique . . . 11

1.3. Contributions . . . 12

**Chapitre 2. Travaux connexes** . . . 13

2.1. Abstraction du comportement et réduction de traces d'exécution . . . 13

2.2. Visualisation de traces d'exécution . . . 14

2.3. Techniques de visualisation . . . 16

2.3.1. Représentation de la hiérarchie . . . 17

2.3.2. Représentation des liens d'adjacence . . . 18

**Chapitre 3. Vue globale** . . . 22

3.1. Collecte des données . . . 22

3.1.1. Extraction de la structure du système . . . 22

3.1.2. Enregistrement des traces d'exécution . . . 22

3.2. Représentation graphique du logiciel . . . 23

3.3. Animation interactive des traces d'exécution . . . 23

**Chapitre 4. La visualisation logiciel avec VERSO** . . . 24

4.1. Représentation de la structure des programmes . . . 24

4.1.1. Placement *Padded Treemap* . . . 24

4.1.2. Représentation des métriques . . . 25

4.2. Visualisation des liens statiques . . . 26

4.2.1. Création des *B-splines* . . . 26

4.2.2. L'algorithme *Hierarchical Edge Bundles* (HEB) . . . 28

**Chapitre 5. Abstraction du comportement à partir d'une trace d'exécution** 30

5.1. Représenter la trace d'exécution . . . 30
5.1.1. Création des liens d'appel . . . 31
5.1.2. Animation de la trace d'exécution . . . 31
5.2. Interactions et filtrage de la trace . . . 32
5.2.1. Filtres structurels . . . 33
5.2.2. Interactions et filtres temporels . . . 35
**Chapitre 6. Études de cas** . . . 36
6.1. Premier cas : le jeu d'échecs . . . 36
6.2. Second cas : l'éditeur de diagrammes UML . . . 42
6.3. Synthèse . . . 46
**Chapitre 7. Conclusion** . . . 47
**Références bibliographiques** . . . 49

# Table des figures


2.1 Diagramme des phases d'exécution hiérarchisées pour une exécution de *Javac*. Diagramme tiré de [**1**]. . . . . 14

2.2 Figure tirée de l'approche proposée par Dugerdil et Alam [**2**]. On voit ici des immeubles représentant des classes, reliés par des courbes, représentant certains types de liens entre les classes. . . . . 15

2.3 Exemple de la visualisation proposée par *Extravis*. On observe en même temps que la représentation de la structure du logiciel une vue séquentielle de la trace d'exécution à droite de l'image. Figure tirée de [**3**]. . . . . 16

2.4 La figure (A) représente une arborescence et la figure (B) représente un arbre radial. Le problème avec ces représentations dans notre cas est que les liens entre les noeuds sont déjà utilisés pour représenter la hiérarchie et ne sont donc pas disponibles pour représenter d'autres relations. . . . . 17

2.5 Exemple de visualisation avec la méthode *Sunburst*. L'image est tirée de [**4**]. . . . . 18

2.6 Exemple de visualisation avec la méthode *Circle Packing*. L'image est tirée de [**5**]. 19

2.7 Exemple de la visualisation *Flow Map Layout*. On voit ici une carte de migration de population aux États-Unis. L'image est tirée de [**6**] . . . . . 20

2.8 Exemple de la visualisation *FDEB*. Les liens marqués en rouge indiquent une très forte densité. L'image est tirée de [**7**]. . . . . 20

2.9 Exemple de la visualisation *HEB*, ici utilisée sur un *Treemap*. L'image est tirée de [**8**]. . . . . 21

3.1 Schéma des flux de travail amenant à la création d'une animation. . . . . 22

4.1 Passage d'un placement *Treemap* par partition à un *Treemap* par imbrication. . . . 25

4.2 Exemple de visualisation de métriques dans VERSO. Tirée de [**9**]. . . . . 25

4.3 Exemple d'un ruban en vue du dessus. La couleur qui va du rouge au blanc indique le sens du lien. On peut noter que les faces avant et arrière du ruban sont coloriées de la même façon. . . . . 26

4.4 Exemple de représentation utilisant des lignes droites dans une version antérieure de VERSO [**9**]. La couleur allant du noir au jaune indique le sens du lien. . . . . 27

4.5 Exemple de placement des points de contrôle au dessus des paquets dans VERSO. Tirée de [**9**]. . . . . 28

4.6 Exemple de représentation utilisant l'algorithme HEB pour orienter les courbes. Tirée de [**9**]. . . . . 29

5.1 Exemple de photographie de nuit d'une autoroute, prise avec un grand temps d'exposition. On observe que les phares des voitures créent des traces lumineuses rouges ou blanches en fonction du sens de circulation du véhicule. ..... 30
5.2 On peut observer sur les classes représentées les deux zones de départ et d'arrivée. On notera aussi que les points de départ des liens sont affectés par un petit décalage, qui permet d'éviter qu'ils ne se superposent. ..... 32
5.3 Observons ici les cartes de chaleur sous les classes, qui pour certaines sont apparentes malgré l'absence de liens partant ou arrivant dans la classe, puisque nous utilisons le mode d'animation fenêtré. ..... 33
5.4 Vue complète de l'interface. En rouge (1) sont encadrées les interactions propres à l'animation et en vert (2) l'arbre d'appels. ..... 34
5.5 Vue du menu contextuel associé aux éléments de l'arbre d'appels. ..... 34
5.6 Capture de la barre de contrôle de l'animation. ..... 35

6.1 Le système récupère l'état du plateau stocké dans la classe *FenNotation* entourée en jaune. ..... 37
6.2 La classe *Chessboard2D* génère un grand nombre d'appels. ..... 37
6.3 Les classes *PawnBehavior*, *RookBehavior*, *KnightBehavior* et *BishopBehavior*, en jaune, sont assez actives dans l'appel à la méthode *paintComponent*. Les classes correspondant aux différentes pièces, en rouge, sont également mobilisées. ..... 38
6.4 La classe *MoveHistory* est rapidement utilisée après le second clic de souris. ..... 39
6.5 Un peu après *MoveHistory*, la classe *Move* qui devient active. ..... 39
6.6 La classe *Chessboard2D* exécute un second appel à *paintComponent*. ..... 40
6.7 Comparaison des différents appels à la méthode *paintComponent*. Le premier appel est en haut, et le second en bas. ..... 41
6.8 Au début de la trace, la zones active, à gauche de l'image, correspond aux classes dédiées au fonctionnement de l'interface ..... 42
6.9 Au moment où l'on commence à dessiner le lien, plusieurs zones deviennent actives. La zone 1 correspond à l'interface utilisateur, la zone 2 gère le diagramme et les classes, la zone 3 représente les liens et leur affichage, et la zone 4 gère l'arrière plan. ..... 43
6.10 Pendant le tracé de la flèche, deux nouvelles classes (entourées en jaune) viennent se rajouter aux classes déjà actives présentées dans la figure précédente, notamment *ContentBorder* et *ContentInsideRectangle*. ..... 43
6.11 Capture du rendu dans l'éditeur UML après notre scénario. Les classes sont ici de simples rectangles reliés par une flèche coudée. ..... 44
6.12 On observe ici les classes actives lors de la fin du scénario, c'est à dire la création du lien définitif. Les quatre classes les plus actives sont : *ClassNode* (cercle 1), *InheritanceEdge* (cercle 2), *ContentBackground* (cercle 3) et *Direction* (cercle 4). 45

# Remerciements

Je tiens à remercier en premier lieu Houari Sahraoui et Pierre Poulin pour leur aide et leurs conseils pendant toute la durée de ce projet. Je tiens également à remercier mes parents et les proches qui m'ont soutenu, ainsi que tous mes amis, notamment celles et ceux de l'OSEUM.

# Chapitre 1

# Introduction

## 1.1. Contexte

Le développement de logiciels constitue une part importante de l'activité du secteur informatique. Né il y a une quarantaine d'années, le paradigme orienté objet a peu à peu pris de l'importance pour devenir actuellement le paradigme dominant dans l'industrie. Cependant, la taille ainsi que la complexité des systèmes allant croissant, il a fallu mettre au point en parallèle de nouvelles méthodes de développement pour garantir des logiciels fiables. Ainsi des techniques comme le développement dirigé par les modèles, ou encore le développement dirigé par les tests, ont vu le jour et tendent à être de plus en plus utilisés. Cependant, un problème récurrent rencontré par les développeurs est le manque de documentation. Dans le cadre du développement d'un logiciel, et plus particulièrement dans des contextes de maintenance, c'est à dire la phase qui suit la mise en production du logiciel, il est important pour le développeur de comprendre le fonctionnement du logiciel, pour mener à bien différentes tâches, telles que du débogage ou du réusinage [10]. Lorsque la documentation n'est pas disponible, ou incomplète, il existe des techniques pour analyser le comportement d'un logiciel, séparées principalement en deux catégories, l'analyse statique de comportements et l'analyse dynamique de comportements.

L'analyse statique consiste à se placer à un point d'entrée dans le système et de dérouler l'exécution du code à partir de ce point de départ. On peut alors tracer plusieurs chemins, en fonction des différentes alternatives qui se présentent au fur et à mesure de l'exécution. Ainsi l'on obtient un graphe qui représente l'ensemble des méthodes et objets atteignables à partir d'un certain point de départ. Cependant, dans un contexte où l'on voudrait déterminer quels objets et quelles méthodes sont impliqués dans la réalisation d'un scénario d'utilisation du logiciel, l'analyse statique ne peut pas répondre précisément au problème, car elle ne permet pas de dire quel chemin en particulier a été utilisé parmi l'ensemble des possibilités.

À l'inverse, l'analyse dynamique de comportements consiste à étudier une seule exécution du système, à l'aide de traces d'exécution. Les traces d'exécution sont l'enregistrement de la suite des divers évènements (appels de méthodes, instanciations d'objets, accès à des variables...) qui correspondent au comportement du logiciel durant une certaine période de temps. Ainsi, lorsqu'on enregistre le comportement durant l'utilisation d'une certaine fonctionnalité haut niveau, il est possible ensuite pour le développeur de parcourir la trace afin de comprendre quels ont été les composants du logiciel impliqués dans la réalisation de la tâche. On peut alors cibler précisément le code source utilisé par la fonctionnalité

enregistrée et ainsi réduire le corpus de code à étudier pour le développeur. Cependant, même pour des logiciels de taille moyenne, il n'est pas rare que la taille des traces à analyser devienne conséquente voire ingérable, de l'ordre de plusieurs dizaines ou centaines de milliers d'évènements. Il devient alors très difficile pour le développeur de donner du sens à cette quantité d'information sans utiliser des outils permettant d'abstraire, de filtrer, d'organiser cette masse d'information [11].

Pour exploiter le contenu des traces d'exécution, il existe principalement deux types d'approches. La première consiste à utiliser des heuristiques pour automatiser la création d'une représentation abstraite des traces, consistant par exemple à détecter des motifs [12], des fonctionnalités [13], des services [14], des phases d'exécution [15] ou encore créer des diagrammes comportementaux [16, 17]. Ces techniques ont produit des résultats intéressants, mais de part l'utilisation automatique d'heuristiques, elles manquent de flexibilité pour répondre aux besoins très variables des développeurs, en fonction de ce qu'ils veulent explorer et comprendre d'un logiciel.

L'autre technique pour aider le développeur à comprendre une trace d'exécution est l'utilisation de la visualisation interactive de logiciels. Cela consiste à associer à un logiciel et une trace d'exécution une représentation graphique. On va alors transformer un ensemble de données textuelles en des images permettant de représenter sous une forme plus familière la masse d'information que constitue une trace d'exécution d'un programme. Un ensemble varié de métaphores ont été explorées, telles que les *edge bundles* [3], les cartes de chaleur [18] ou encore les signaux [19]. Comme les développeurs doivent gérer des programmes complexes et des traces d'exécution très grandes, l'efficacité de ces approches dépend grandement de leur scalabilité. L'utilisation de la visualisation de logiciels pour faciliter la compréhension de grandes traces d'exécution sera l'objet de ce mémoire.

## 1.2. Problématique

Il existe déjà des outils permettant de représenter graphiquement des logiciels développés en orienté objet, mais ces outils se limitent en général à représenter la structure du système, les relations entre les objets (comme l'héritage, les invocations de méthodes, ...) et des métriques. Pour ce qui est des traces d'exécution, il existe aussi des outils pour les exploiter et obtenir des graphiques. Cependant à notre connaissance il n'existe pas de système permettant de visualiser à la fois la structure du logiciel ainsi qu'une trace d'exécution de manière interactive et à grande échelle. Ensuite, un autre problème des traces d'exécution est la diversité d'information qu'elles contiennent. Ces informations sont de plusieurs natures : appels de méthodes, instanciations d'objets, créations d'un fil d'exécution, etc. Comme nous voulons associer la trace d'exécution avec la structure du logiciel, c'est à dire un ensemble d'objets, nous choisissons de limiter nos traces aux appels et retours de méthodes, à la manière d'un diagramme de séquence. De plus, les traces peuvent, même sur des systèmes de taille modeste, comprendre plusieurs dizaines voire centaines de milliers d'évènements, ce qui les rend difficiles à appréhender pour un développeur.

Notre premier problème consistera donc à intégrer la représentation des traces d'exécution à un système de visualisation du logiciel. Il nous faudra trouver une métaphore qui soit

capable de faciliter la compréhension de la trace et puisse s'adapter aussi bien à des petits qu'à des grands systèmes. Nous devrons ensuite offrir des outils pour permettre à l'utilisateur de naviguer aussi bien dans la trace d'exécution que dans la structure du logiciel, et notamment permettre de réduire la taille de la trace en autorisant l'utilisateur à cacher ou retirer des éléments qui ne lui sont pas utiles.

## 1.3. Contributions

Le système que nous avons développé permet de visualiser une trace d'exécution et la structure du logiciel associées sous la forme d'une animation contrôlée dans un environnement 3D. La représentation de la structure du logiciel, sous forme d'un *Padded Treemap*, l'utilisation de courbes avec la technique des *edge bundles*, ainsi que la représentation de métriques, sont basées sur VERSO. VERSO fournit un environnement 3D interactif permettant de visualiser la structure d'un logiciel orienté objet, des métriques ainsi que les liens statiques entre les classes (héritages, invocations de méthodes...).

En fait, une tâche majeure de notre travail a été de réimplanter la plupart des fonctionnalités offertes par VERSO dans une nouvelle version du système, principalement afin de faciliter l'ajout des nouvelles fonctionnalités décrites ci-dessous.

La première fonctionnalité ajoutée est celle permettant de jouer une trace d'exécution sous forme d'animation. L'élément de base de l'animation est l'appel de méthode, qui se traduit graphiquement par une ligne reliant la classe appelante à la classe appelée. L'animation consiste alors en l'accumulation de liens, c'est à dire qu'à l'appel suivant, un lien est créé. Nous avons défini deux modes pour l'animation, un qui offre un aperçu global, et un autre qui offre un aperçu local pour chaque moment de la trace.

Ensuite nous avons ajouté plusieurs outils pour permettre à l'utilisateur d'interagir avec la trace. Tout d'abord pour contrôler l'animation : on peut avancer et reculer, définir une fenêtre temporelle, et ainsi exclure des parties de la trace de l'animation. Au niveau de la structure, on peut choisir de cacher certaines classes et certaines méthodes, ou au contraire de ne regarder qu'un sous-ensemble de celles-ci.

Enfin, nous avons évalué notre système à l'aide de deux études de cas sur des logiciels de petite et grande taille, avec différents objectifs. Le but étant de montrer les différentes utilisations de notre outil et comment il peut être utile pour analyser des traces d'exécution et ainsi comprendre le comportement d'un logiciel.

# Chapitre 2

# Travaux connexes

Avant de présenter notre approche à proprement parler, nous aborderons les différentes contributions de la communauté au sujet de la compréhension du comportement d'un logiciel à partir de traces d'exécution. Pour cela nous allons diviser ce chapitre en deux parties, qui correspondront aux deux grandes catégories d'approches concernant ce sujet : (1) celles qui traitent les traces d'exécution, pour les réduire ou abstraire des comportements, et (2) celles qui transforment les traces en un environnement graphique, interactif ou non, afin de permettre à l'utilisateur de les explorer ou au moins de les visualiser différemment que par du texte.

## 2.1. Abstraction du comportement et réduction de traces d'exécution

La compréhension du comportement de haut niveau d'un logiciel à partir de ses traces d'exécution peut s'aborder de deux façons, soit par l'abstraction de ces traces d'exécution, soit par leur réduction aux seuls évènements significatifs. Comme la littérature sur ce thème est relativement abondante et que le sujet de ce mémoire est la visualisation, nous n'aborderons que des exemples représentatifs des travaux effectués dans ce domaine.

La plupart des contributions antérieures se concentrent sur l'abstraction ou combinent à la fois réduction et abstraction. Cette abstraction se fait souvent automatiquement, par l'utilisation de différentes heuristiques. Par exemple, Alimadadi et al. [12] utilisent un algorithme ad hoc pour déterminer des motifs d'exécution récurrents, pour en tirer un modèle comportemental du logiciel. Pour cela, ils commencent par enregistrer plusieurs traces d'exécution de scénarios semblables, qui sont ensuite filtrées automatiquement pour retirer les détails de bas niveaux, comme des appels à des bibliothèques externes ou encore des détails spécifiques à l'implantation du système. Une fois réduites aux évènements porteurs de sens, ils cherchent des motifs récurrents dans les traces en utilisant des techniques inspirées des algorithmes de recherche de similarités dans des séquences ADN ou ARN. La partie visualisation de leur approche offre à l'utilisateur un ensemble de tableaux et diagrammes permettant d'observer les résultats obtenus lors de la recherche des motifs.

De la même manière, de nombreux travaux de recherche combinent la segmentation des traces et des heuristiques de recherche pour identifier des éléments qui facilitent la compréhension d'un programme. Dans ces travaux, la méthodologie est relativement semblable à celle présentée dans le paragraphe précédent. Il s'agit tout d'abord de filtrer la trace selon

certains critères, d'appliquer un ou plusieurs algorithmes pour repérer les éléments ciblés, et enfin de présenter les résultats sous forme de tableaux et de diagrammes. Parmi ces éléments on distingue les fonctionnalités [13], les concepts dynamiques [20] ou encore les phases d'exécution [15]. Les phases d'exécution hiérarchisées ont également été utilisées et ont généré un intérêt certain [10, 1]. Le principe de la hiérarchisation permet de créer différents niveaux d'abstraction. L'exemple donné par Alanazi et al. dans la figure 2.1 illustre bien ce principe. On voit le résultat de leur approche, qui présente les différents niveaux d'abstraction des phases d'exécution lorsque l'on compile un programme avec *Javac*. Le premier niveau (*Level 1*) est le niveau le plus abstrait et correspond à l'opération que veut réaliser l'utilisateur. Ensuite, plus on descend dans la hiérarchie, plus la phase de compilation sera découpée en plusieurs phases plus petites et plus détaillées, jusqu'au niveau 6 (*Level 6*) où l'on peut voir les détails des différentes étapes de la compilation (tokenisation, *parsing*...).

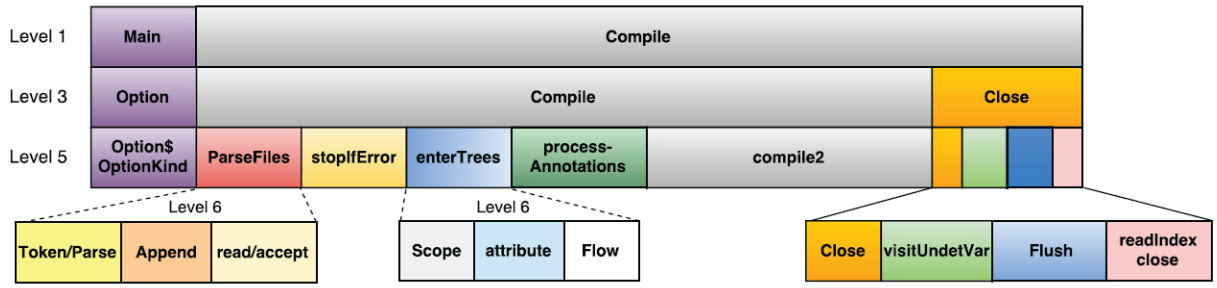


FIGURE 2.1. Diagramme des phases d'exécution hiérarchisées pour une exécution de *Javac*. Diagramme tiré de [1].

Les traces d'exécution sont également exploitées pour trouver des éléments provenant d'autres paradigmes dans des programmes en orienté objet, à des fin de compréhension ou de migration. Ceci a particulièrement été exploré pour les services [14] ou les composants [21]. Finalement, un des champs les plus explorés est l'ingénierie inverse de modèles comportementaux abstraits à partir de traces d'exécution [22]. Les modèles les plus ciblés par la recherche sont les diagrammes de séquences [16] et d'activités [17].

Bien que les approches automatisées aient mené à des résultats intéressants, la diversité des besoins des développeurs et le manque d'informations contextuelles dans les programmes créent un intérêt pour des approches semi-automatiques impliquant les développeurs. C'est pour cela que nous avons choisi de nous pencher sur ce genre d'approche, en utilisant un environnement de visualisation interactive.

## 2.2. Visualisation de traces d'exécution

La compréhension d'un programme avec des techniques de visualisation du logiciel peut se faire à travers la détection de patrons dans les traces d'exécution. En suivant ce concept, De Pauw et al. [23] proposent une représentation des traces basée sur un arbre pour détecter ces patrons. Leur outil permet de chercher des patrons en utilisant des filtres textuels et des opérateurs, permettant notamment d'effondrer ou encore d'aplatir l'arbre. Renieris et

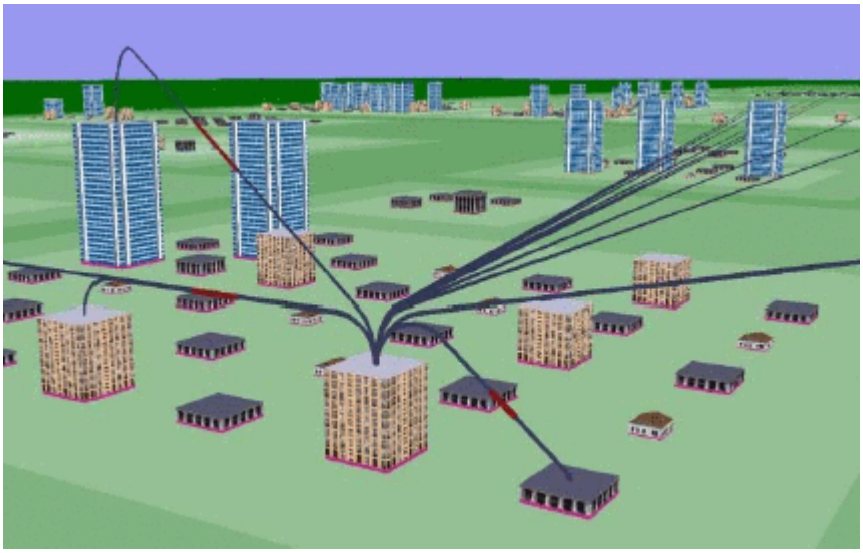

FIGURE 2.2. Figure tirée de l'approche proposée par Dugerdil et Alam [2]. On voit ici des immeubles représentant des classes, reliés par des courbes, représentant certains types de liens entre les classes.

Reiss [24] combinent une vue globale d'une trace d'exécution, sous la forme d'une spirale, avec une vue locale linéaire permettant l'affichage d'une portion spécifique de la trace. Ici encore, les outils offrent des opérateurs d'interaction pour aider à la compréhension de l'exécution, comme par exemple le zoom avant/arrière, l'accès au code source, ou encore à la pile d'appels. Dans la même lignée, Bohnet et al. [25] proposent une technique pour représenter dans une vue linéaire, des traces d'exécution réduites. Pour réduire la taille de la trace, les auteurs utilisent une classification des appels de méthodes et une identification de motifs répétés basée sur leur similarité.

Les trois méthodes que l'on a mentionnées utilisent la même représentation des arbres, à savoir des diagrammes en stalactites (*icicle plots*), avec quelques variations. D'autres équipes ont préféré expérimenter avec des métaphores. Par exemple, Trümper et al. [26] étendent la représentation en stalactites des traces en les plaçant dans des tubes, permettant ainsi de comparer différentes exécutions. Kuhn et Greevy [19] utilisent des signaux pour représenter les traces, permettant d'exploiter des techniques de traitement du signal pour analyser les exécutions. De la même façon, Dautriche et al. [27] s'appuient sur une méthode appelée *Slick Graphs* en combinaison avec d'autres techniques de visualisation.

Dans une perspective différente, Dugerdil et Alam [2] visualisent les traces d'exécution en les transformant en une ville animée. Ce travail est assez similaire au nôtre, dans la mesure où les auteurs utilisent une représentation comparable de la structure du logiciel, des outils de filtrage, et un affichage des appels basé sur des courbes aériennes. Cependant leur représentation ne permet pas de visualiser un grand nombre d'appels sans obstruer totalement la vue globale. Le travail de Cornelissen et al. [3] est également proche du nôtre. Dans leur outil, *ExtraVis*, les auteurs combinent une vue séquentielle de la trace au complet avec

une vue circulaire représentant les différents appels (figure 2.3). Cet outil offre des interactions permettant d'analyser les traces, par exemple pour détecter les phases d'exécution ou encore faire de la localisation de fonctionnalité. La détection et la visualisation de phases d'exécution sont également étudiées par Reiss [28] avec l'outil *JIVE*. Enfin, Benomar et al. [15] proposent un cadre de visualisation unifié pour explorer et analyser des exécutions et les évolutions d'un logiciel. La partie visualisation utilise des cartes de chaleur et permet entre autres d'identifier des phases d'exécution et de déterminer le rôle des différentes classes impliquées dans les scénarios d'exécution.

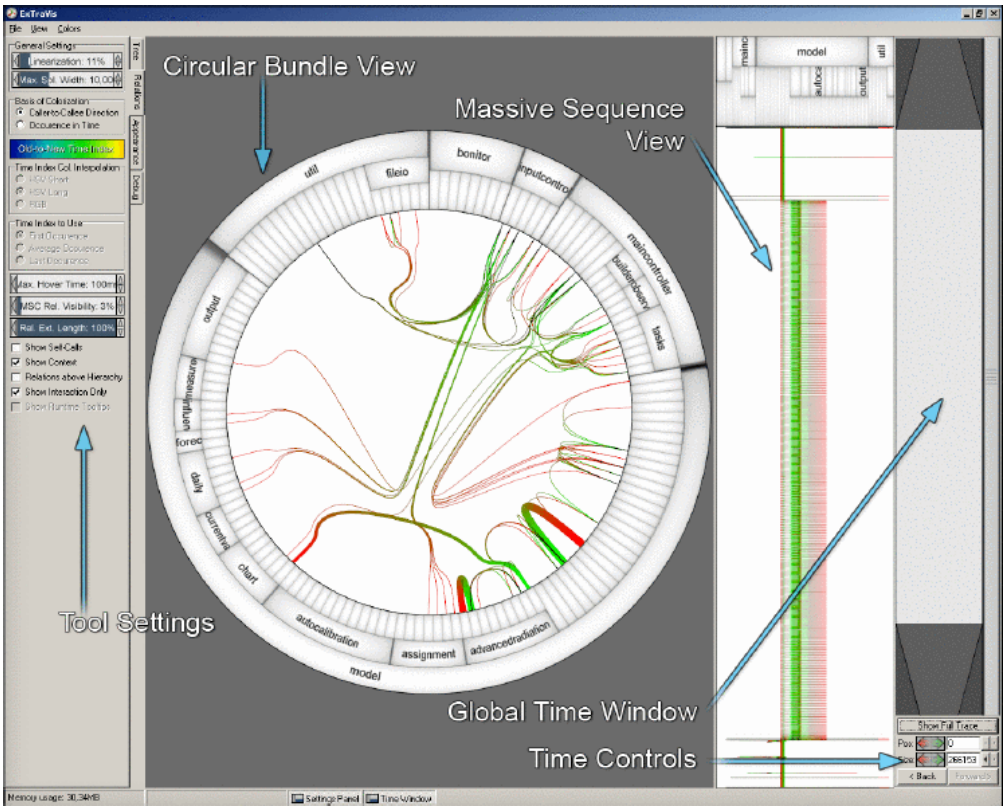


FIGURE 2.3. Exemple de la visualisation proposée par *Extravis*. On observe en même temps que la représentation de la structure du logiciel une vue séquentielle de la trace d'exécution à droite de l'image. Figure tirée de [3].

## 2.3. Techniques de visualisation

Nous avons présenté différentes façons dont des traces d'exécution peuvent être exploitées. Nous allons maintenant aborder les techniques de visualisation. Nous parlerons principalement des méthodes de représentation noeuds-liens.

Les méthodes de représentation noeuds-lien sont probablement les méthodes les plus répandues en visualisation, même en dehors du génie logiciel, lorsqu'il s'agit de représenter des relations entre des éléments. La méthode de base pour représenter graphiquement des

liens entre des éléments est sans doute le graphe. Cependant il ne faut pas perdre de vue le fait que dans notre cas, les noeuds représenteront les différents éléments du logiciel observé. Ces éléments, que ce soient des paquets, des classes, ou des méthodes, s'inscrivent dans une hiérarchie, qui ne peut pas être aisément représentée à l'aide de simples graphes.

### 2.3.1. Représentation de la hiérarchie

La structure des programmes dans le paradigme orienté objet est hiérarchisée. Il nous faut donc, pour la représenter graphiquement, utiliser des structure permettant de retranscrire cette hiérarchie. Pour cela, l'arborescence ou l'arbre radial sont des candidats naturels. Cependant, ils ont un défaut majeur dans notre cas : ce sont déjà des représentations noeuds-liens. Cela signifie que nous ne pouvons pas utiliser les liens pour représenter des relations d'adjacences, à moins d'en ajouter de nouveau, en plus de ceux représentant la hiérarchie.

Il existe d'autres techniques que des représentations noeuds-liens pour visualiser des structures hiérarchiques : ce sont les techniques dites par inclusion. Une des premières façons de représenter une structure hiérarchique par inclusion est la méthode du *Treemap* développée par Scheiderman et al. [29]. Cette représentation consiste en une surface initiale qui représente l'élément racine dans la hiérarchie, surface qui est ensuite divisée en autant d'éléments enfants que possède la racine. En alternant le sens de la division, on peut alors représenter des structures hiérarchique dans un espace fixé à l'avance. Nous parlons plus en détail de cette technique dans le Chapitre 4. Une autre technique utilisée est la méthode *Sunburst*, développée par Stasko et Zhang [4]. Les différentes strates de la hiérarchie sont représentées par différents anneaux concentriques, comme le montre la figure 2.5. Le cercle au centre est le niveau le plus élevé de la hiérarchie, et les niveaux inférieurs sont répartis de l'intérieur vers l'extérieur. S'il existe plusieurs éléments au même niveau hiérarchique, alors l'anneau est séparé en plusieurs portions, dont la taille dépend du nombre de descendants

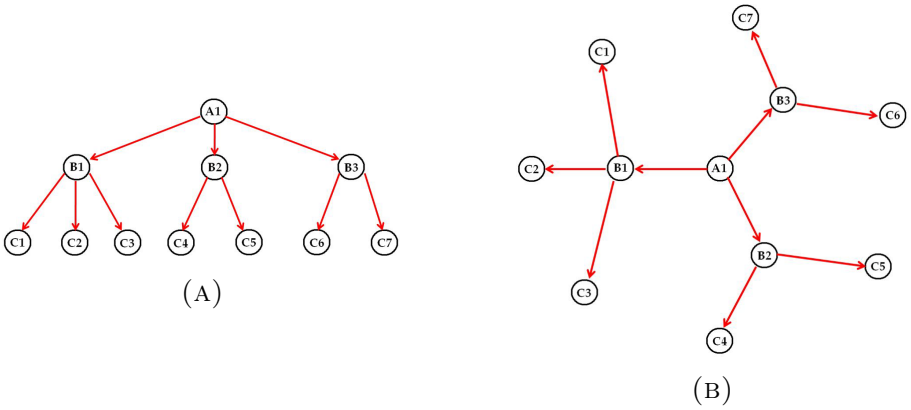


FIGURE 2.4. La figure (A) représente une arborescence et la figure (B) représente un arbre radial. Le problème avec ces représentations dans notre cas est que les liens entre les noeuds sont déjà utilisés pour représenter la hiérarchie et ne sont donc pas disponibles pour représenter d'autres relations.

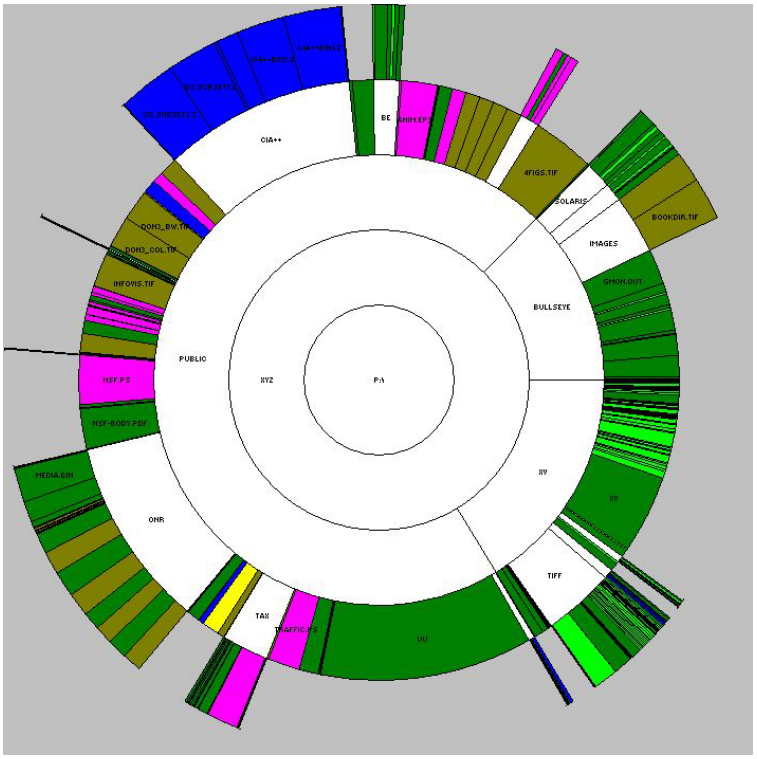


FIGURE 2.5. Exemple de visualisation avec la méthode *Sunburst*. L'image est tirée de [4].

de l'élément considéré. Les portions définies à un certain niveau serviront à limiter au niveau suivant la place occupée par les éléments enfants de cette portion. On remarquera que dans la figure 2.5 le dernier anneau n'est pas complet. Cela se produit lorsque les branches de l'arbre (si la structure que l'on représente est un arbre) ne sont pas toutes de la même taille. Il arrive alors, dans notre cas, un moment où certaines branches sont terminées lorsque d'autres peuvent encore avoir des feuilles, ce qui rend cet anneau incomplet. Enfin on citera une troisième technique qui est celle dite par imbrication de cercles ou *Circle Packing* en anglais. Cette technique s'inspire du *Treemap* mais en utilisant des cercles imbriqués plutôt que des rectangles, comme l'illustre la figure 2.6. Comme dans le *Treemap* le cercle le plus grand est la racine, et chaque élément enfant se retrouve à l'intérieur du cercle représentant son parent. Les éléments frères sont alors placés de façon à former un amas compact à l'intérieur du parent, en cherchant le plus possible à agencer de façon circulaire. L'avantage de cette méthode comparée au *Treemap* est qu'elle laisse plus d'espace entre les différents éléments de la figure et permet ainsi une meilleure perception de la hiérarchie.

### 2.3.2. Représentation des liens d'adjacence

Nous allons maintenant nous intéresser aux techniques utilisées pour représenter des liens d'adjacences sur des visualisations noeuds-liens. La façon la plus intuitive de relier

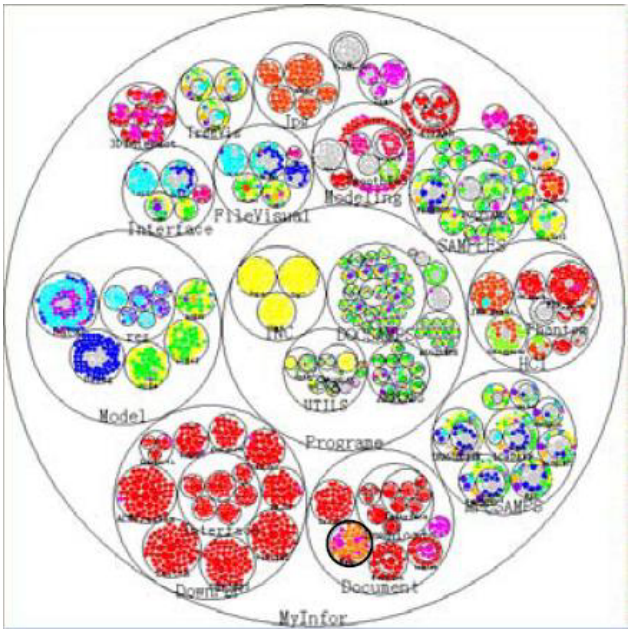


FIGURE 2.6. Exemple de visualisation avec la méthode *Circle Packing*. L'image est tirée de [5].

deux noeuds liés est de tracer une ligne droite entre ces deux noeuds. Cependant, même si cette méthode fonctionne pour un petit nombre d'éléments, plus le nombre de noeuds et de liens à représenter augmente, plus le nombre de liens qui vont se croiser ou pire, se confondre va devenir important et rendre la figure illisible. Pour résoudre ce problème, plusieurs techniques de regroupement et d'organisation des liens ont été proposées. Une première vient des techniques cartographiques : la carte de flux. Une carte de flux permet de représenter le transit d'objets ou de personnes entre deux lieux. Si deux flux empruntent un trajet (ou une portion de trajet) commun, alors le lien qui les représente est fusionné et épaissi sur leur partie commune, puis re-séparés si leur trajet diverge à nouveau. Cette technique est utilisée notamment sous la forme U *Flow Map Layout* proposé par Phan et al. [6], dont un exemple est présenté dans la figure 2.7.

Une autre technique pour organiser un ensemble de liens est la méthode *Force-Directed Edge Bundling* (*FDEB*) proposée par Holten et Van Wijk [7]. Avec cette méthode on commence par relier les noeuds par des lignes droites. Ensuite ces lignes droites sont découpées en plusieurs segments et les points ainsi créés vont permettre de déformer les droites. Pour cela chaque point sera déplacé en fonction de la force qu'exercent ses voisins sur lui. Cela aura pour effet de regrouper les droites des zones denses en faisceaux plus gros, tandis que les droites dans des zones de faible densité ne seront que peu déformées, comme le montre la figure 2.8. Il existe aussi d'autres méthodes donnant des résultats similaires, comme par exemple la méthode *Geometry-Based Edge Clustering* proposée par Weiwei et al. [30].

Cependant ces méthodes fonctionnent avec des structures quelconques et ne profitent pas de la hiérarchie présente dans les représentations que nous avons évoquées à la section

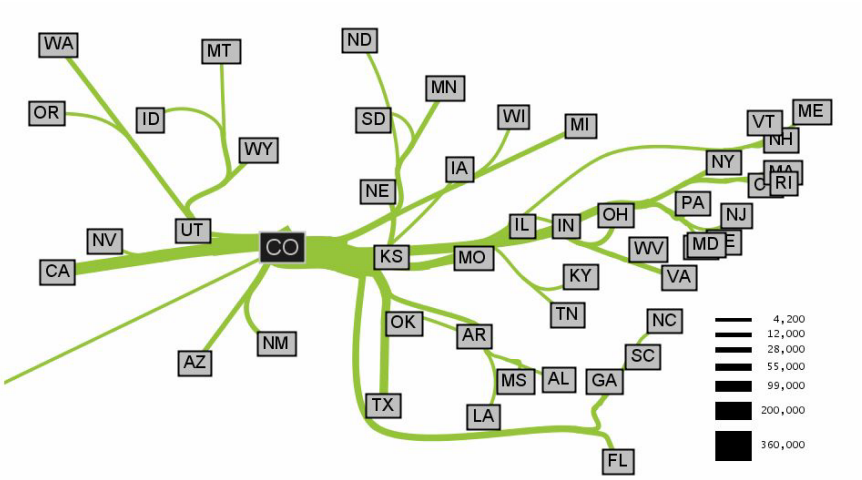


FIGURE 2.7. Exemple de la visualisation *Flow Map Layout*. On voit ici une carte de migration de population aux États-Unis. L'image est tirée de [6]

FIGURE 2.8. Exemple de la visualisation *FDEB*. Les liens marqués en rouge indiquent une très forte densité. L'image est tirée de [7].

précédente. Pour exploiter la hiérarchisation Holten a proposé la technique des *Hierarchical Edge Bundle* [8]. Cette technique permet de regrouper des liens en paquets, à la manière des deux autres méthodes évoquées précédemment, mais elle se sert de la hiérarchie présente dans la structure pour orienter les trajets. Ainsi les liens sont regroupés à chaque fois qu'ils sortent d'un niveau de hiérarchie pour passer au niveau suivant et sont séparés au moment où ils descendent dans la hiérarchie, comme le montre la figure 2.9. Nous aborderons plus en détail cette technique dans le chapitre 4.

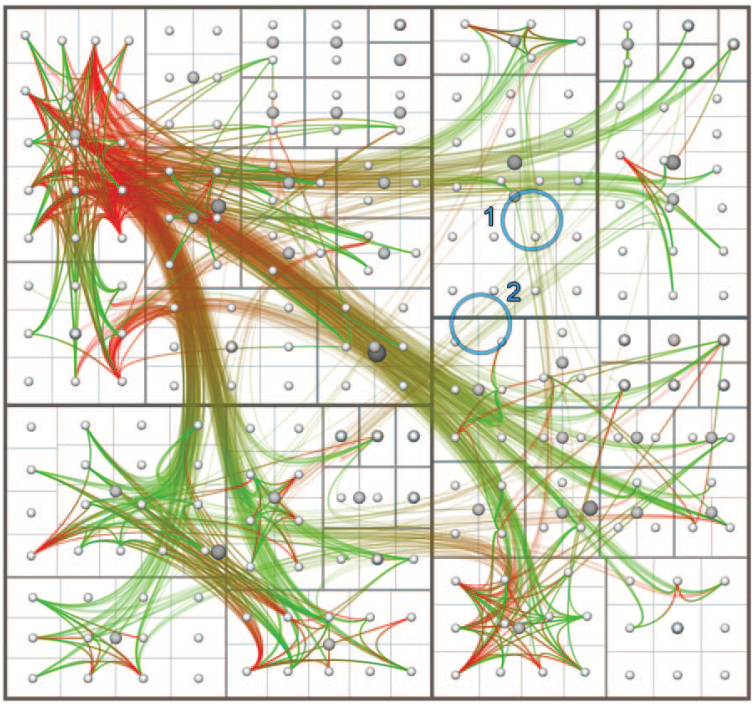


FIGURE 2.9. Exemple de la visualisation *HEB*, ici utilisée sur un *Treemap*. L'image est tirée de [8].

# Chapitre 3

# Vue globale

Dans ce court chapitre nous allons voir le fonctionnement global de notre approche, principalement en illustrant les deux flux de travail qui permettent de représenter graphiquement un scénario d'exécution.

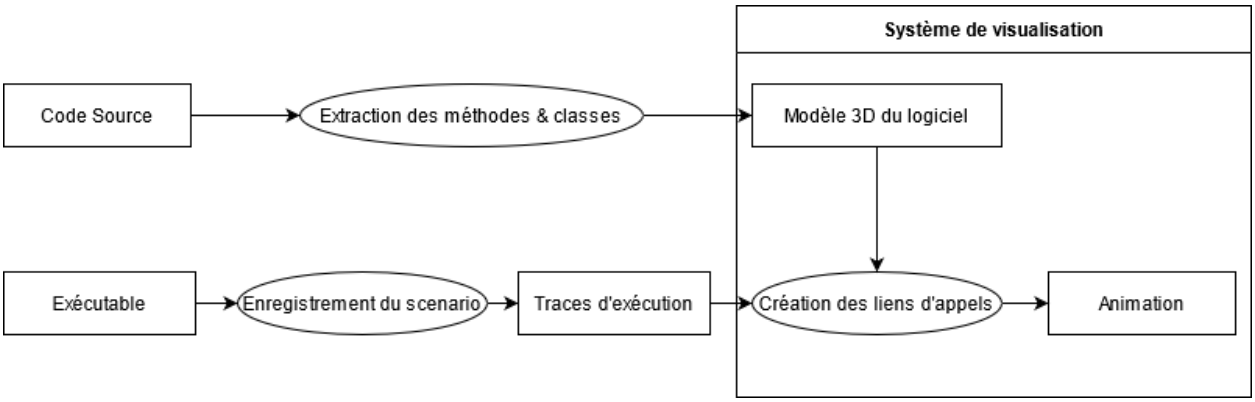


FIGURE 3.1. Schéma des flux de travail amenant à la création d'une animation.

## 3.1. Collecte des données

### 3.1.1. Extraction de la structure du système

Pour extraire la structure du système nous utilisons ses sources et un compilateur. Nous arrêtons cependant la compilation à la génération de l'Abstract Syntax Tree (AST). À partir de cet arbre, nous obtenons l'ensemble des paquets, classes et méthodes, ainsi que certaines informations (paquet parent, liens d'héritage, ...), que nous stockons ensuite sous forme de fichier CSV (*comma separated values*).

### 3.1.2. Enregistrement des traces d'exécution

Comme nous nous limitons pour l'instant au langage Java, la collecte des traces d'exécution se fait uniquement au travers de la *Java Virtual Machine (JVM)*. Pour cela nous avons réutilisé et amélioré un outil développé au préalable par Omar Benomar. C'est un outil disposant d'une petite interface graphique et qui permet d'enregistrer des scénarios d'exécution

dans un fichier de texte.
Avant d'enregistrer une trace, l'utilisateur doit déterminer un scénario à jouer. Un scénario consiste en une liste d'actions à effectuer pour réaliser une ou plusieurs fonctionnalités du logiciel que l'on veut observer. Par exemple sur un client courriel, un scénario possible serait "Écrire et envoyer un courriel" ou encore "Ouvrir un courriel". Une fois le ou les scénarios choisis, voici la façon dont on enregistre une trace avec l'outil :

— on commence par lancer le logiciel que l'on veut explorer depuis l'outil ;
— on créé un fichier qui contiendra le scénario que l'on va exécuter ;
— on indique les noms des paquets qui nous intéressent ;
— on lance l'enregistrement de la trace ;
— on joue le scénario que l'on veut enregistrer ;
— on arrête l'enregistrement une fois le scénario complété.

L'utilisation de l'outil se veut assez simple. Concernant l'indication des noms de paquets, cela permet de limiter la taille des fichiers créés. Nous avons en effet constaté que si nous récupérions les évènements liés à toutes les classes impliquées dans le scénario, les fichiers devenaient vite très volumineux, de l'ordre de plusieurs centaines de mégaoctets. Ainsi nous avons ajouté ce filtre sur les paquets pour permettre d'exclure des bibliothèques natives et/ou externes, qui ne sont pas utiles pour comprendre le comportement du logiciel exploré. Nous avons modifié l'outil pour récupérer les informations nécessaires à notre tâche. Pour cela nous utilisons un *java agent*. Cela consiste en une bibliothèque dynamique chargée au lancement de la JVM, qu'elle va ensuite appeler au déclenchement de certains évènements. Dans notre cas les évènements qui nous intéressent sont les entrées et sorties de méthodes. Ainsi à chaque appel de méthode, nous enregistrons le type d'évènement (entrée/sortie), le nom de la méthode et le nom de la classe de l'objet qui appelle la méthode. Ainsi une fois l'enregistrement terminé, on obtient un fichier contenant la succession d'appels et de retours des différentes méthodes, qui nous permettra ensuite d'en tirer l'évolution de la pile d'appels. Nous détaillerons dans un prochain chapitre comment est exploitée cette trace pour la transformer en une animation interactive dans VERSO.

## 3.2. Représentation graphique du logiciel

Pour représenter la structure du logiciel ainsi que les différentes données qui vont nous intéresser, nous utilisons plusieurs techniques, comme les *Padded Treemaps* ou encore les *Hierarchical Edge Bundles*. Le détail est abordé dans le chapitre 4.

## 3.3. Animation interactive des traces d'exécution

Pour visualiser les traces d'exécution nous avons décider d'utiliser une animation. Le but de l'animation va être de rejouer la trace d'exécution dans l'environnement 3D. Pour cela nous allons associer à chaque appel de méthode un lien, et l'animation consistera en l'affichage successif de ces liens, en fonction de l'ordre des évènements dans la trace d'exécution. Le détail est donné dans le chapitre 5.

# Chapitre 4

# La visualisation logiciel avec VERSO

## 4.1. Représentation de la structure des programmes

VERSO est un ensemble d'environnements de visualisation du logiciel qui existent et évoluent depuis une dizaine d'années. Plusieurs étudiants ce sont succédés en apportant chacun des modifications à VERSO, en rapport avec leur problématique de recherche, telles que la visualisations de métriques, de dépendances et d'architecture [31, 9], la détection de patrons [32] ou encore l'évolution d'un logiciel [33]. Le but de cette section est de présenter les éléments et approches conservés dans la version de VERSO que nous utiliserons, pour cette fois représenter des traces d'exécution.

### 4.1.1. Placement *Padded Treemap*

Nous avons choisi d'utiliser le placement que nous appelons *Padded Treemap* pour ce qui concerne la représentation de la structure du logiciel. À l'origine la représentation *Treemap* a été pensée pour représenter une arborescence (*tree*) en partitionnant un plan (*map*) [29]. Cette méthode est intéressante pour notre problématique car d'une part la structure d'un programme en Java, et plus généralement dans le paradigme orienté objet, est une arborescence, et d'autre part cette représentation permet, avec quelques adaptations, d'obtenir une meilleure répartition des éléments dans l'espace, ce qui nous sera utile pour la visualisation des liens hiérarchiques et d'adjacences [9], puis des traces d'exécution.
Comme indiqué plus haut, le principe de base du *Treemap* est de découper un espace pour représenter une arborescence. Sans entrer trop dans le détail, voici les règles de construction d'un *Treemap*, du moins celles que nous utilisons :

- La surface complète correspond à la racine de l'arbre.
- Pour chaque noeud de l'arbre, la surface qui lui est associée est découpée en autant de parties que le noeud a de branches. Ainsi si un noeud a trois branches, alors la surface du noeud est divisée en trois parties.
- La taille de chaque division dépend du poids de la branche associée. Dans notre cas la fonction de poids est le nombre de classes dans la branche.
- Enfin, à chaque descente d'un niveau dans l'arbre, le sens de coupe pour les divisions est modifié. Ainsi on peut commencer à découper selon l'axe vertical pour les branches de la racine, puis selon l'axe horizontal pour le niveau suivant, et ainsi de suite 4.1.

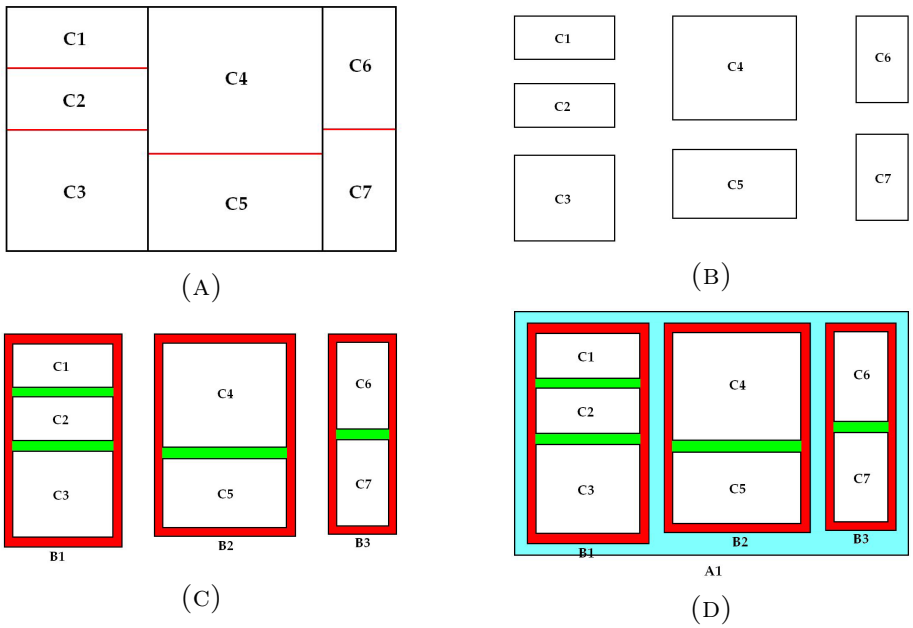


FIGURE 4.1. Passage d'un placement *Treemap* par partition à un *Treemap* par imbrication.

### 4.1.2. Représentation des métriques

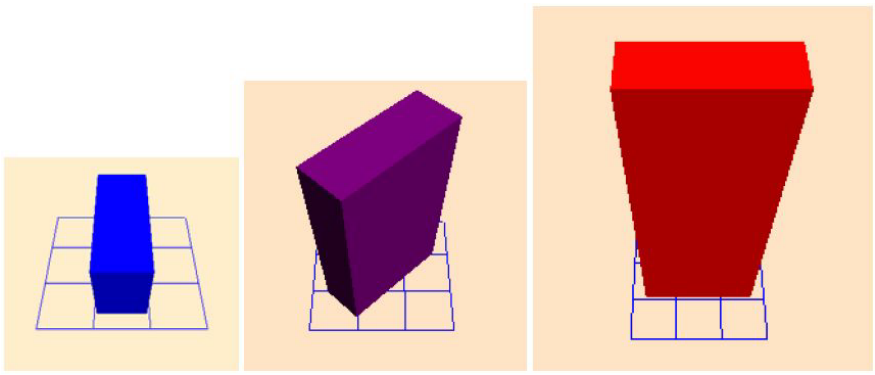

FIGURE 4.2. Exemple de visualisation de métriques dans VERSO. Tirée de [9].

En plus de la structure, un autre type d'information représenté par le modèle 3D sont différentes métriques, calculées au niveau des classes. Nous utilisons trois caractéristiques visuelles, chacune associées à une métrique différente, permettant de la quantifier et d'avoir un moyen simple de faire des comparaisons entre les classes. Ici nous avons choisi d'associer la hauteur de la classe à sa complexité (WMC). Sa couleur, allant du bleu au rouge, est associée à son couplage (CBO). Enfin sa rotation, sur un angle de 90 degrés, représente sa cohésion (LCOM5). La présence des métriques permet d'obtenir des informations supplémentaires sur

les classes, pouvant aider par la suite à déterminer le rôle et la façon dont la classe a été pensée, une fois mise en relation avec les autres classes du système.

## 4.2. Visualisation des liens statiques

Finalement la dernière heuristique que nous avons reprise de VERSO est la représentation de liens à l'aide de courbes *B-splines*, qui survolent la représentation 3D du système pour relier différentes classes. Dans VERSO, les liens servent à représenter des relations telles que l'héritage ou des appels statiques [9], mais ils serviront dans notre approche à représenter les appels dynamiques. Dans cette section nous allons en premier lieu voir comment sont tracées les *B-splines*, puis comment elles sont organisées au sein de la visualisation.

### 4.2.1. Création des *B-splines*

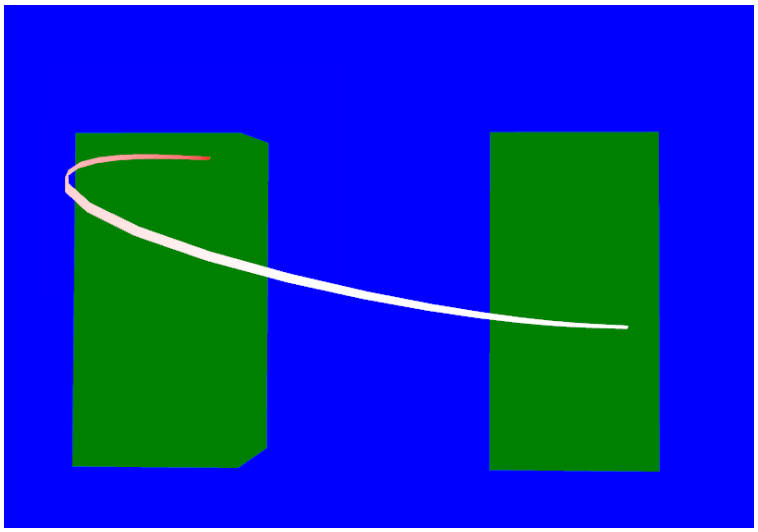

FIGURE 4.3. Exemple d'un ruban en vue du dessus. La couleur qui va du rouge au blanc indique le sens du lien. On peut noter que les faces avant et arrière du ruban sont coloriées de la même façon.

Les courbes *B-splines* sont des courbes paramétriques similaires aux courbes de Bézier. Comme ces dernières, elles sont définies par un ensemble de points de contrôle qui forment une polyligne de contrôle à l'intérieur de laquelle réside la courbe paramétrique, de degré 3 dans notre cas. Une différence de la *B-spline* est que la courbe n'intersecte pas nécessairement les points de départ et d'arrivée. Sans entrer dans le détail de sa construction et de ses propriétés, elle offre globalement plus de flexibilité qu'une courbe de Bézier, comme le nombre plus général de points de contrôle et la continuité $C^2$ en tout point de la courbe.

Comme nous nous situons dans un environnement en trois dimensions, nous ne pouvons pas utiliser simplement une ligne tracée dans l'espace pour représenter une courbe, celle-ci serait trop peu visible et affectée par des effets comme l'aliassage. Le tracé de la courbe va servir de squelette à un modèle 3D : un cylindre dans les versions précédentes et dans

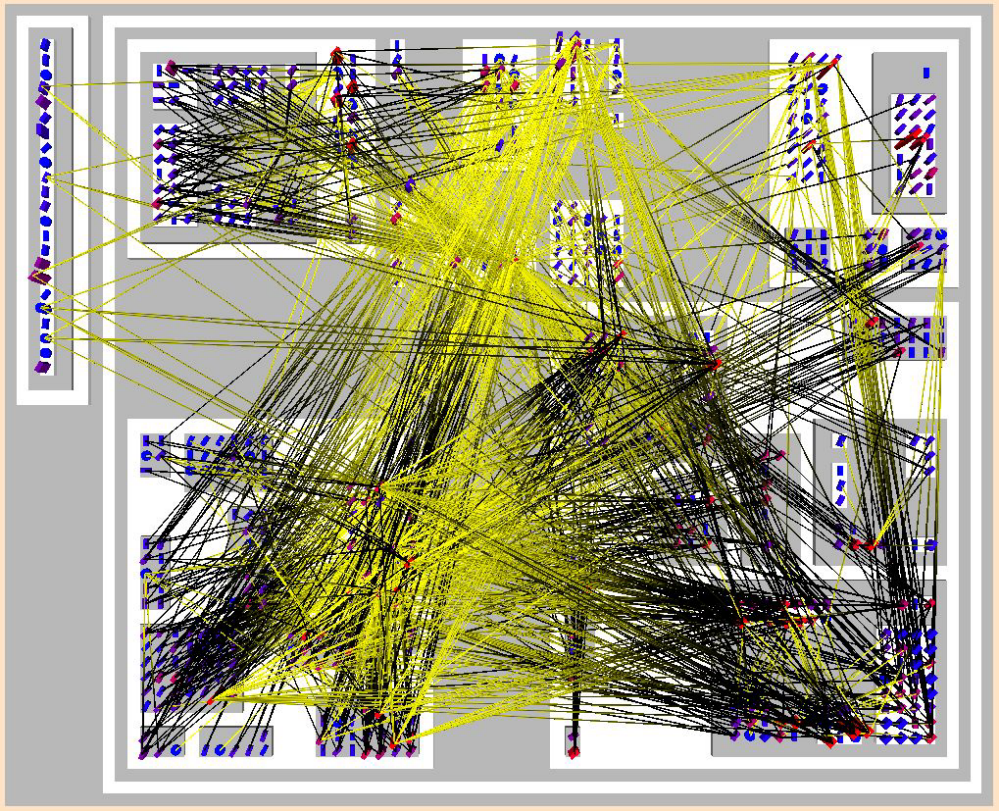

FIGURE 4.4. Exemple de représentation utilisant des lignes droites dans une version antérieure de VERSO [9]. La couleur allant du noir au jaune indique le sens du lien.

notre version, un ruban. Nous avons choisi d'utiliser des rubans pour représenter les liens car ceux-ci sont moins coûteux à afficher que des cylindres et nous permettent donc d'en afficher un plus grand nombre en même temps. Cependant, si l'angle de vue sur la scène 3D montre la "tranche" des rubans, alors cela revient à regarder une ligne, avec les mêmes problèmes que nous évoquions plus tôt. Mais nous estimons qu'il est peu probable que l'utilisateur se place avec un tel angle de caméra, la scène étant pensée pour être vue principalement du dessus, ou au moins avec un angle plongeant.

Pour créer les rubans on commence par échantillonner la courbe à intervalles réguliers dans l'espace paramétrique. Cela nous permet d'obtenir une suite de points sur la courbe qui servira de squelette au ruban. Plus nous utilisons de points, plus la courbe aura une apparence lisse et continue, mais elle prendra aussi plus de temps à tracer. Ensuite, pour chacun de ces points on récupère la dérivée de la courbe en ce point (soit sa tangente, en l'assumant non nulle) et on calcule le produit vectoriel de cette dérivée avec le vecteur vertical ($up$). On normalise ensuite le vecteur obtenu à la taille souhaitée pour établir la largeur du ruban. Pour chaque point de la liste on va alors le sommer avec le vecteur qui lui est associé. Cela nous donnera une nouvelle liste de points, qui suivra une courbe identique à la première

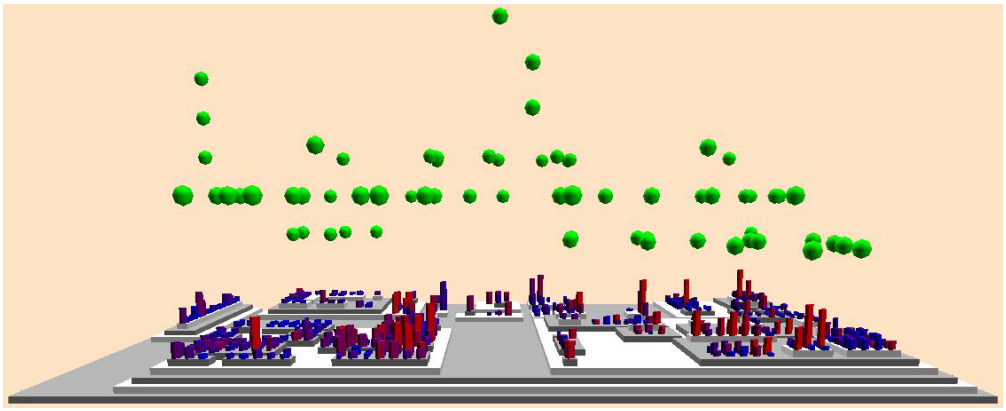

FIGURE 4.5. Exemple de placement des points de contrôle au dessus des paquets dans VERSO. Tirée de [9].

mais avec simplement un décalage qui correspond à la largeur du ruban. Il ne nous reste alors plus qu'à relier les points des deux listes pour créer les faces du ruban. Le fait d'utiliser le produit en croix du vecteur vertical avec la dérivée de la courbe en différents points nous permet de garantir que le ruban sera orienté face vers le haut. La figure 4.3 est un exemple des rubans que nous utilisons. Le seul cas à éviter est lorsque la tangente est dans la même direction que la verticale, où nous utilisons alors le vecteur allant du point de départ au point d'arrivée de la courbe. Enfin, nous utilisons les valeurs paramétriques entre 0.0 et 1.0 pour obtenir une interpolation de la couleur entre rouge (0.0) et blance (1.0) pour chaque point correspondant à un sommet du ruban.

Ces courbes sont utilisées pour représenter différents types de relations entre des éléments d'un logiciel, que ce soit entre des classes, des paquets ou des méthodes. Pour relier deux éléments, la façon la plus simple consiste à faire passer la courbe au dessus du modèle représentant le logiciel, en ligne droite. Cette technique peut fonctionner lorsqu'il n'y a pas beaucoup de liens à représenter, mais dès lors qu'il commence à y en avoir un certain nombre, la représentation devient alors difficilement lisible, comme le montre la figure 4.4.

### 4.2.2. L'algorithme *Hierarchical Edge Bundles* (HEB)

Pour limiter l'enchevêtrement des liens, la solution qui a été retenue est celle des *Hierarchical Edge Bundles* [8]. Le principe de cette technique va être de regrouper les liens en paquets, à la manière de câbles électriques. Pour cela on commence par trouver le chemin dans l'arborescence des classes et paquets qui relie la classe de départ à la classe d'arrivée. Ce chemin va alors être une liste de paquets qui vont devenir des points de contrôle de la courbe. Dans l'espace 3D, ces points de contrôle sont des points que l'on place au dessus du centre du paquet, comme le montre la figure 4.5. On remarque également que les points ont différentes hauteurs : les paquets les plus haut dans la pyramide ont les points de contrôle les plus bas, et inversement, les paquets les plus bas dans la pyramide ont les points de contrôle les plus hauts, jusqu'à la racine qui a le point culminant. Cela fera en sorte que plus un lien

remonte dans la hiérarchie des paquets plus il remontera également dans l'axe $Y$ de l'espace 3D.

Ajouter ces points de contrôle dans la courbe va avoir pour effet de regrouper les liens qui sortent ou entrent dans le même paquet, formant comme des paquets de câbles. Le rendu, dont la figure 4.6 est un exemple, est alors plus clair qu'auparavant. On ne peut certes pas distinguer la trajectoire d'un lien en particulier, mais l'utilisation de cet algorithme permet d'avoir un aperçu des différents "axes de communication" présents et de leur importance en terme de nombres de liens.

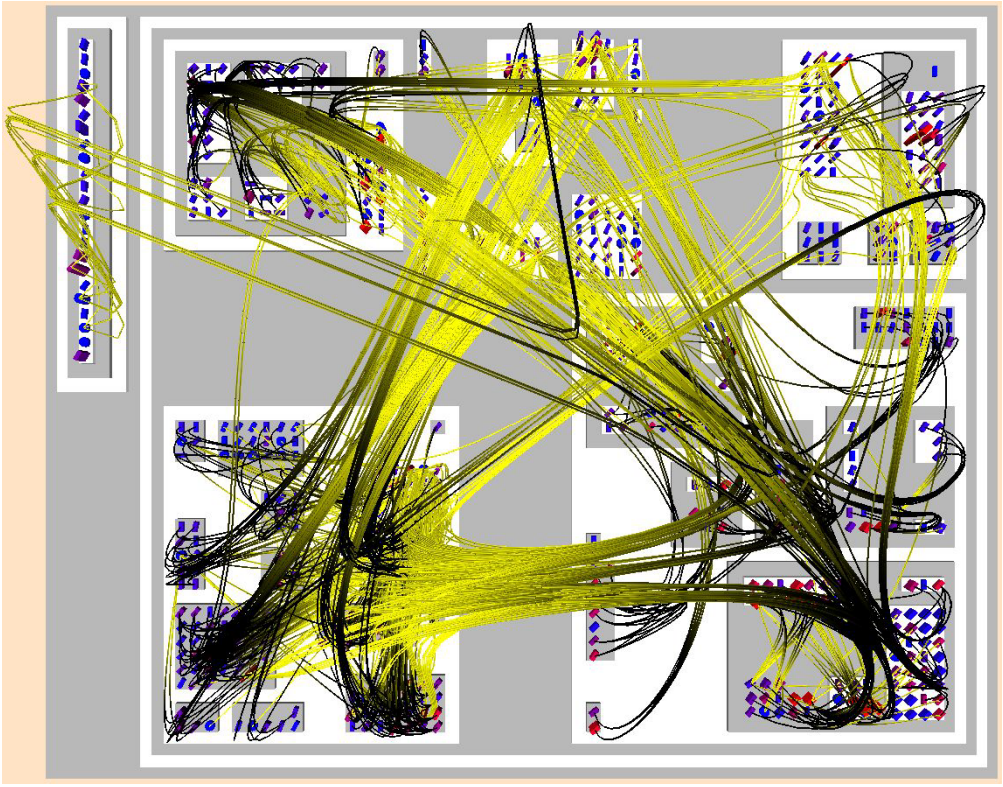

FIGURE 4.6. Exemple de représentation utilisant l'algorithme HEB pour orienter les courbes. Tirée de [9].

# Chapitre 5

# Abstraction du comportement à partir d'une trace d'exécution

Dans ce chapitre, nous allons détailler les fonctionnalités que nous avons ajoutées à la version originale de VERSO, qui sont principalement l'animation de la trace d'exécution et les différentes interactions et mécanismes de filtrage qui servent à manipuler et explorer la trace.

## 5.1. Représenter la trace d'exécution

Pour représenter la trace d'exécution, nous allons utiliser la métaphore des traces lumineuses visibles sur les photos de routes prises de nuit avec un grand temps d'exposition, comme le montre la figure 5.1. Pour décrire comment nous avons transcrit cette métaphore dans notre approche, nous allons d'abord détailler la création des liens, puis la façon dont ils sont animés dans le logiciel.

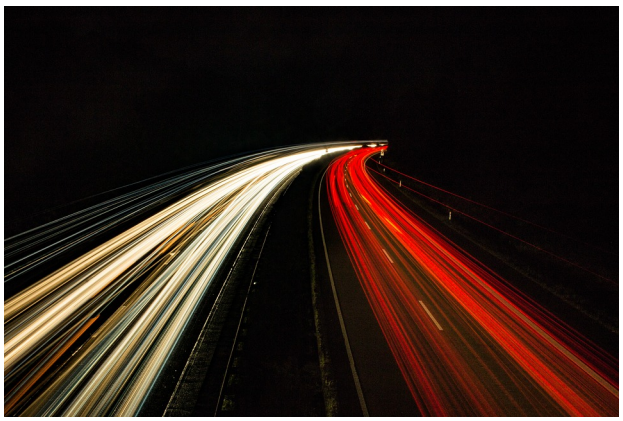

FIGURE 5.1. Exemple de photographie de nuit d'une autoroute, prise avec un grand temps d'exposition. On observe que les phares des voitures créent des traces lumineuses rouges ou blanches en fonction du sens de circulation du véhicule.

### 5.1.1. Création des liens d'appel

La trace d'exécution d'un scénario enregistré au préalable nous fournit une liste d'appels et de retours de méthodes. La première étape pour produire l'animation est de parcourir cette liste, qui va nous permettre de la transformer en une liste ordonnée de liens que l'on pourra ensuite afficher. Pour créer un lien nous prenons deux entrées consécutives de la trace et nous vérifions les conditions suivantes :

— Les deux entrées sont des appels de méthode, car nous avons choisi de ne pas représenter les retours de méthode.

— Elles appartiennent à deux classes différentes, car nous ne représentons pas graphiquement les liens internes aux classes.

Lorsque ces conditions sont respectées, un lien est créé dans le système et ajouté à une liste qui constituera la base de l'animation. Les liens seront représentés graphiquement par une courbe suivant le procédé décrit dans le chapitre précédent. Pour indiquer l'orientation de la courbe, nous texturons le ruban avec un gradient coloré, allant du rouge au blanc. La courbe sera rouge au départ de la classe appelante et blanche lorsque arrivée à la classe appelée. De plus, la boîte qui représente une classe est séparée en deux zone : une zone de départ et une zone d'arrivée. Comme leur nom le laisse deviner, les liens qui sortent de la classe partiront depuis la zone de départ, et vice-versa pour les liens qui arrivent dans la classe, voir la figure 5.2. Ces zones ne sont pas visibles sur les boites, elles ne se matérialisent que lorsque l'on observe la répartition des départs et arrivées des courbes.

### 5.1.2. Animation de la trace d'exécution

Une fois que la liste des liens d'appel a été créée, nous allons pouvoir les utiliser pour rejouer la trace d'exécution à travers une animation, qui va simplement être la succession de l'ensemble des appels effectués. Ainsi on ne va pas afficher l'ensemble des liens d'un seul coup. Ceux-ci vont s'afficher au fur et à mesure, et s'accumuler au dessus de la représentation du logiciel ; c'est le mode que nous appelons "cumulatif".

Notons au passage que pour les liens ayant la même origine et la même destination (mais ne représentant pas forcément le même appel de méthode) nous avons défini une limite au delà de laquelle nous n'afficherons pas plus de lien. En effet au delà de 300 liens dessinés, l'apparition d'une nouvelle courbe n'est quasiment pas perceptible et vient simplement ajouter des polygones à dessiner. Cette limite arbitraire de 300 liens n'est toutefois pas absolue ; la perception pouvant varier selon la largeur des rubans et la taille du logiciel représenté, mais ce choix s'adapte plutôt bien aux différents logiciels et scénarios que nous avons observés.

En plus du mode cumulatif, nous avons défini un mode qui utilise une fenêtre de temps (dans le sens de la liste ordonnée d'évènements) mobile. Au lieu d'accumuler les liens sans jamais en effacer aucun, nous choisissons une taille de fenêtre au delà de laquelle à chaque affichage d'un nouveau lien, le plus ancien lien affiché est effacé. Ainsi les liens visibles ne représentent que les appels effectués récemment, dans la limite définie par la fenêtre de temps, ce qui permet d'avoir une vision plus locale des évènements. On voit alors les zones actives se déplacer au fil de l'animation et la taille de la fenêtre étant paramétrable, on peut ajuster à tout moment ce qu'on considère être représentatif de la zone active.

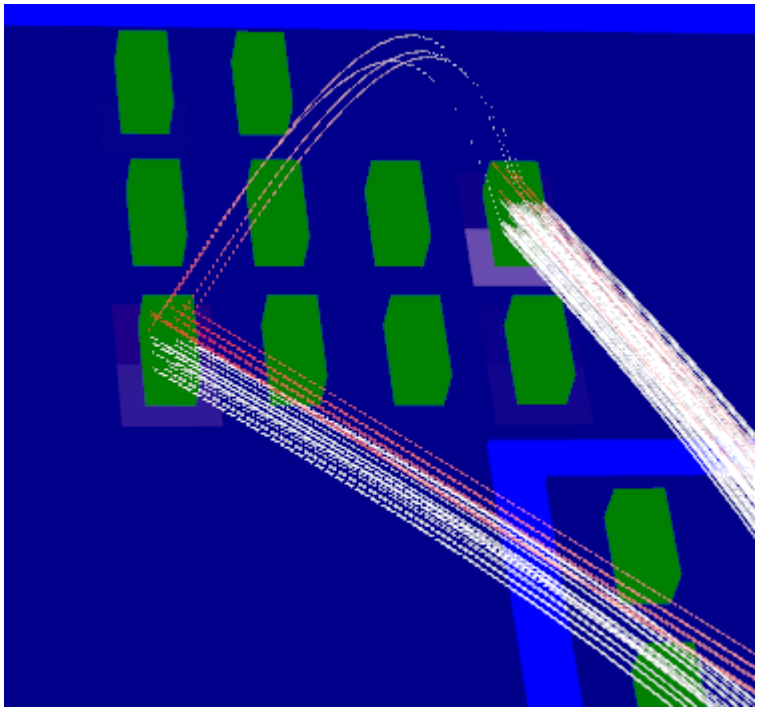

FIGURE 5.2. On peut observer sur les classes représentées les deux zones de départ et d'arrivée. On notera aussi que les points de départ des liens sont affectés par un petit décalage, qui permet d'éviter qu'ils ne se superposent.

En parallèle de l'affichage des courbes, l'animation génère aussi une carte de chaleur. Pour chaque classe, deux zones apparaissent en dessous de leur modèle qui vont de la couleur de leur paquet parent vers le blanc pour la zone d'arrivée, et vers le rouge pour la zone de départ. Cette carte de chaleur permet d'avoir en permanence un aperçu des classes qui ont reçu et ont effectué le plus d'appels, indépendamment du mode d'animation, voir la figure 5.3.

Les deux modes d'affichage sont complémentaires. Le premier permet d'avoir un aperçu global des évènements, qui devient inefficace lorsque l'utilisateur cherche à avoir une vue détaillée des évènements, étant donnée l'accumulation rapide d'un grand nombre d'informations visuelles. Le mode d'animation fenêtré permet alors de fournir une vue réduite, offrant même la possibilité de visualiser un unique lien à la fois, évitant ainsi la surcharge d'informations pour le système visuel.

## 5.2. Interactions et filtrage de la trace

Pour interagir avec la vue en 3D du système, nous avons ajouté une interface séparée en deux parties. La première consiste en une barre d'outils qui permet de contrôler l'animation à la manière d'un lecteur vidéo (voir encadré rouge de la figure 5.4). La seconde partie consiste en différents panneaux présentant différentes données, comme l'arborescence des classes et

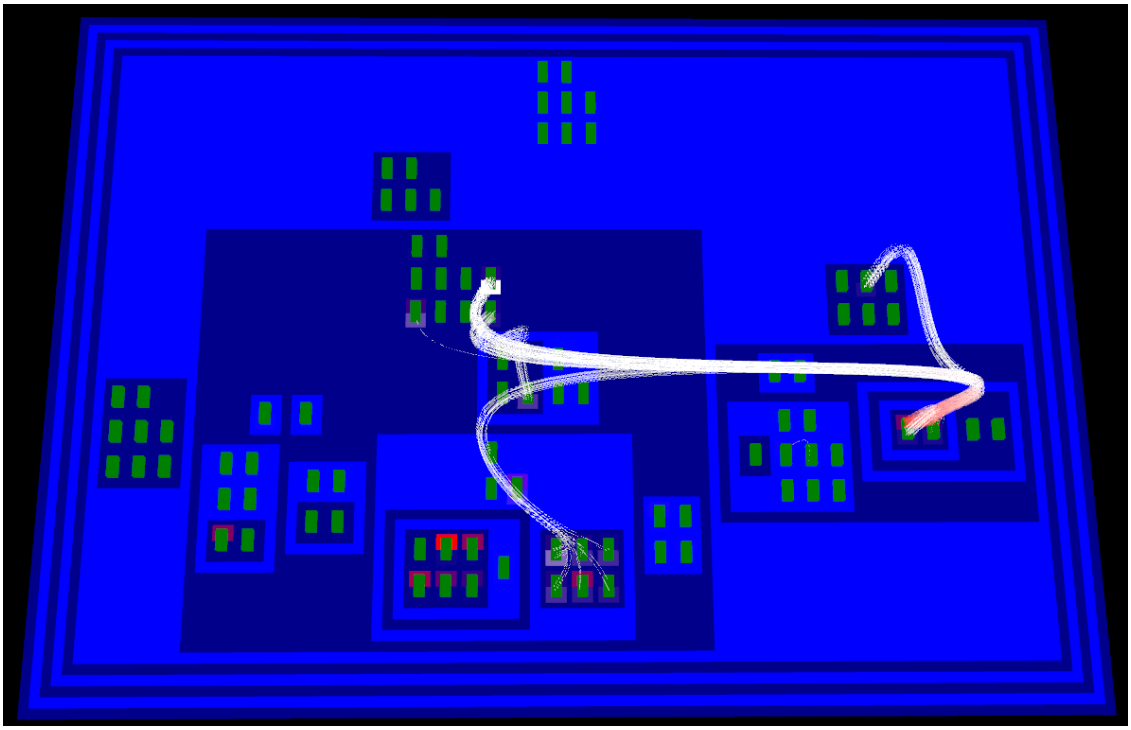

FIGURE 5.3. Observons ici les cartes de chaleur sous les classes, qui pour certaines sont apparentes malgré l'absence de liens partant ou arrivant dans la classe, puisque nous utilisons le mode d'animation fenêtré.

paquets, des détails sur la classe sélectionnée, l'ensemble des filtres et finalement l'arbre d'appel de la trace chargée, présenté dans l'encadré vert de la figure 5.4.

### 5.2.1. Filtres structurels

La première catégorie de filtre que nous allons aborder est ce que l'on a nommé les filtres structurels. Ce type de filtres s'applique sur les classes du système. L'utilisateur peut sélectionner un ensemble de classes, depuis le modèle 3D ou l'explorateur dans l'interface, et l'enregistrer dans un filtre. Ce filtre peut alors avoir deux états : visible ou invisible. Lorsqu'un filtre de classes est dans l'état invisible, les classes restent visibles mais ce sont les liens qui arrivent ou partent des classes filtrées qui sont masqués. Lorsque le filtre est dans l'état visible les liens sont visibles. Le but de ces filtres est de pouvoir masquer les évènements associés à des classes qui n'intéressent pas l'utilisateur pendant son exploration de la trace d'exécution.

Un deuxième type de filtres s'applique sur les méthodes, depuis l'arbre d'appels (voir l'encadré vert de la figure 5.4). L'utilisateur a alors plusieurs options pour filtrer depuis l'arbre. La première consiste à sélectionner un appel et à le cacher, comme montré dans la figure 5.5. Lorsque l'on choisit de cacher un appel, toute la succession d'appels qu'il aura générée sera également cachée. Cela se répercute dans l'animation par le retrait des

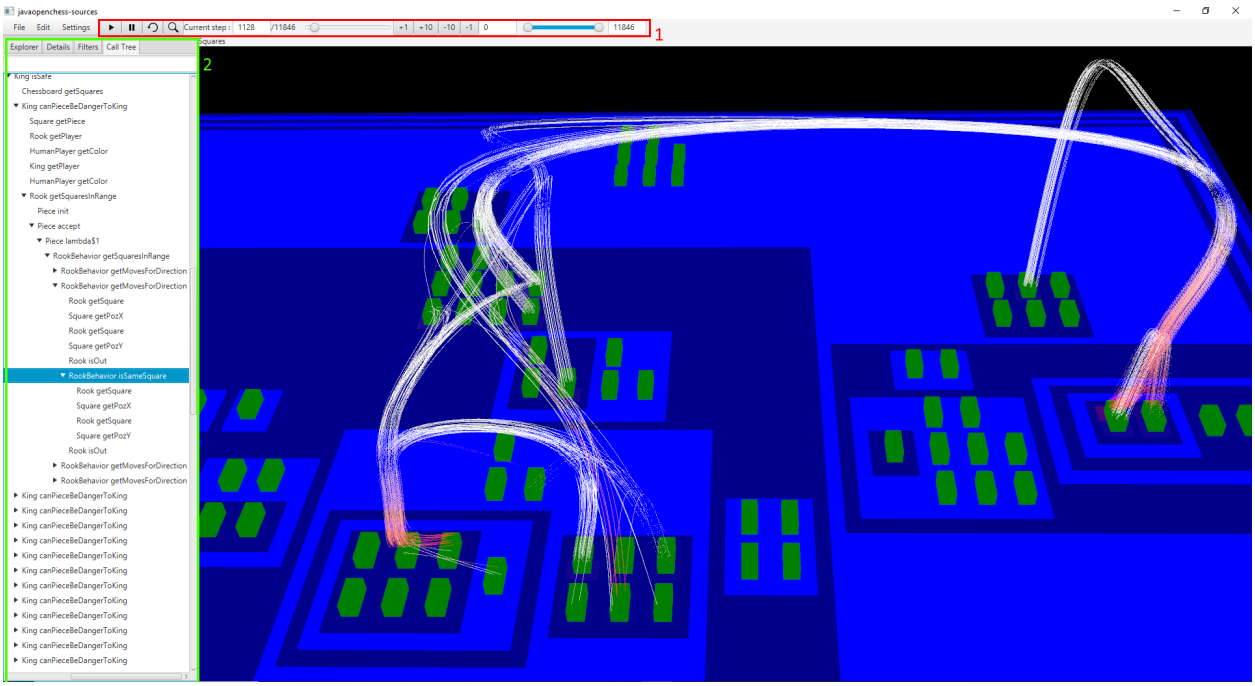


FIGURE 5.4. Vue complète de l'interface. En rouge (1) sont encadrées les interactions propres à l'animation et en vert (2) l'arbre d'appels.

liens associés aux appels cachés. Il est également possible avec l'option *Hide all* de cacher l'ensemble des appels de la méthode sélectionnée (en l'occurrence l'appel à une méthode *drawActiveSquare* dans la figure 5.5). Si l'utilisateur veut ensuite réafficher les appels qu'il a cachés, il peut utiliser l'option *Show All* pour réafficher tous les appels à une certaine méthode, ou bien l'option *Show* pour ne réafficher qu'une seule occurrence (l'option "Show" n'est visible dans le menu contextuel que si l'appel a été caché au préalable). Lorsque les appels sont réaffichés, l'ensemble des appels sous-jacents sont également réaffichés.

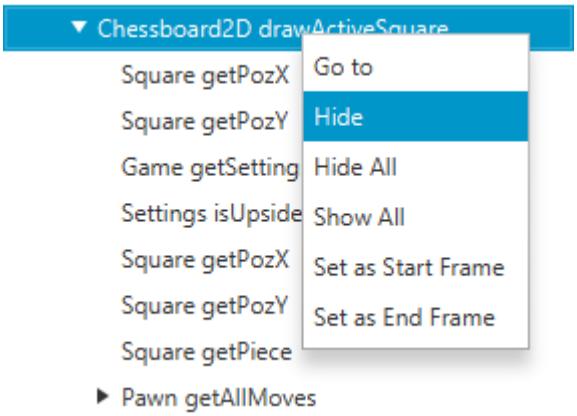


FIGURE 5.5. Vue du menu contextuel associé aux éléments de l'arbre d'appels.

### 5.2.2. Interactions et filtres temporels

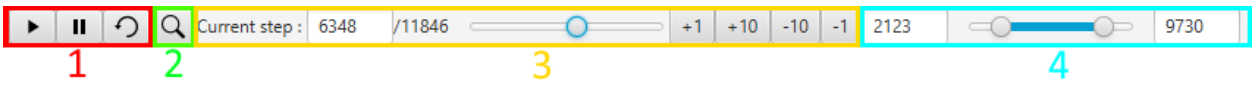


FIGURE 5.6. Capture de la barre de contrôle de l'animation.

Avant d'aborder les filtres temporels, nous allons détailler les contrôles dont dispose l'utilisateur pour manipuler l'animation, qui sont présentés dans la figure 5.6. L'encadré rouge (1) présente les interactions de base, qui sont similaires à celles d'un lecteur vidéo. La flèche permet de lancer l'animation, la double barre de la mettre en pause et la flèche qui revient sur elle-même de recommencer l'animation depuis le départ. Dans l'encadré vert (2) figure une loupe. Celle-ci permet, lorsqu'on clique dessus, de mettre en évidence dans l'arbre d'appels (encadré vert de la figure 5.4) l'appel associé à l'évènement courant de l'animation. Dans l'encadré jaune (3) se trouve, de gauche à droite, l'évènement courant de l'animation, le nombre total d'évènements, un curseur qui indique également l'évènement courant, puis quatre boutons permettant d'avancer ou de reculer d'un ou dix évènements. Ces différents éléments permettent à l'utilisateur de se déplacer plus ou moins rapidement dans l'animation, sans avoir à la rejouer. Enfin une dernière façon de se déplacer dans l'animation est visible dans la figure 5.5, par le biais de l'option *Go to*. Cette option va alors déplacer le curseur de l'animation à l'évènement associé à l'appel sélectionné.

Le filtrage temporel consiste à réduire la taille de la trace en modifiant les points de départ et d'arrivée de l'animation. Le premier moyen pour le faire est d'utiliser les champs de texte ou les deux curseurs présents dans l'encadré bleu (4) de la figure 5.6. On peut utiliser le champ ou le curseur de gauche pour indiquer l'évènement initial de l'animation et le champ ou le curseur de droite pour indiquer l'évènement final. Une autre façon de modifier les bornes de départ ou d'arrivée est d'utiliser les options *Set as Start Frame* et *Set as End Frame* dans le menu contextuel de l'arbre d'appels (voir la figure 5.5). Ces interactions permettent de choisir comme point de départ ou d'arrivée un appel de méthode précis, ce qui est utile lorsque l'on veut regarder en détail le comportement d'un appel en particulier. Pour cela il suffit de choisir comme point de départ cet appel que l'on veut observer et comme point d'arrivée le prochain appel qui est au même niveau d'indentation dans l'arbre (le même niveau d'indentation signifiant que le deuxième appel n'est pas généré par le premier).

Maintenant que l'ensemble des outils et interactions que nous avons défini ont été présentés, nous allons les mettre en pratique à l'aide de deux études de cas, présentant deux logiciels de tailles différentes.

# Chapitre 6

# Études de cas

Dans ce chapitre nous allons présenter deux études de cas illustratives, sur deux logiciels différents. Elles ont pour but de montrer des utilisations possibles de notre approche et les informations que l'on peut en extraire. Dans les deux cas nous sélectionnerons différents points d'intérêt de l'animation pour les détailler, puisque traiter l'intégralité de la trace serait trop long. Les deux cas ont pour but d'illustrer différentes fonctionnalités de notre approche sur des logiciels de tailles différentes. Pour avoir un meilleur aperçu de comment nous utilisons notre outil, nous avons réalisé une vidéo de démonstration pour l'édition 2021 de la conférence VISSOFT, vidéo disponible à https://youtu.be/-8wMjzFGU40.

## 6.1. Premier cas : le jeu d'échecs

Dans cette première étude de cas nous explorons un scénario enregistré sur un jeu d'échecs [**34**]. Le scénario consiste simplement à déplacer un pion sur le plateau, ce qui nous a demandé deux interactions : un premier clic pour sélectionner la pièce et un deuxième pour la déplacer sur une case. Le logiciel observé est de taille relativement modeste avec environ 90 classes. Quant à la trace, bien que simple à réaliser, elle est composée d'un peu moins de 28 000 entrées.

Le but de ce premier cas est d'utiliser des filtres temporels pour comparer deux appels d'une même méthode et tenter d'interpréter les différences entre ces deux appels.

Lorsque l'on démarre l'animation, on observe que les premiers appels, déclenchés par le premier clic de souris, s'articulent autour de la classe *FenNotation* (voir la figure 6.1). *Fen* est l'abréviation de *Forsyth-Edwards Notation* ; elle est une notation utilisée pour indiquer la position d'une pièce sur un échiquier. Il semblerait donc que le logiciel commence par récupérer l'état de l'échiquier.

Ensuite c'est l'appel à la méthode *paintComponent* de la classe *Chessboard2D* qui va nous intéresser. Cette méthode va générer un grand nombre d'appels comme le montre la figure 6.2. Parmi ces appels, on note un bloc de classes qui va également en générer un nombre important, bloc composé des classes *PawnBehavior*, *RookBehavior*, *KnightBehavior* et *BishopBehavior*, entourées en jaune dans la figure 6.3. Les classes associées aux différentes pièces, en rouge sur la même figure, sont également actives. La classe *Chessboard2D* suggère par son nom qu'elle a un rôle dans l'affichage du jeu. Il paraît donc étonnant, surtout dans une méthode *paintComponent*, que celle-ci fasse des appels vers des classes dont le rôle semble plutôt dédié à la logique du jeu.

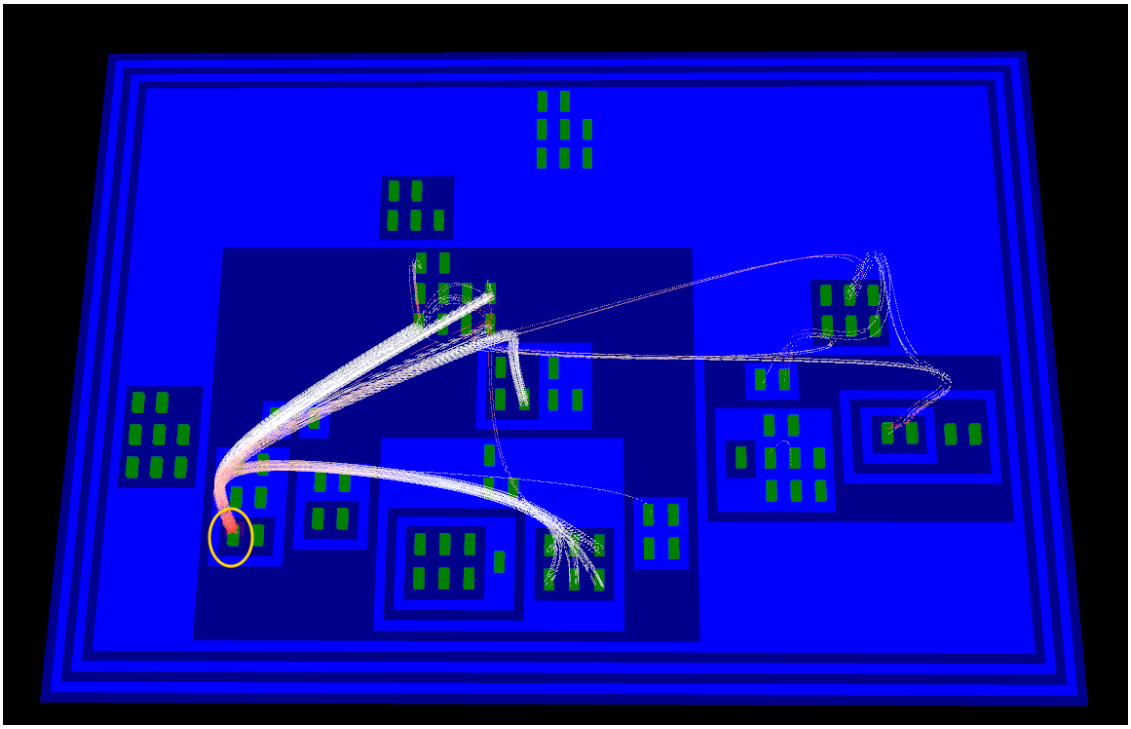

FIGURE 6.1. Le système récupère l'état du plateau stocké dans la classe *FenNotation* entourée en jaune.

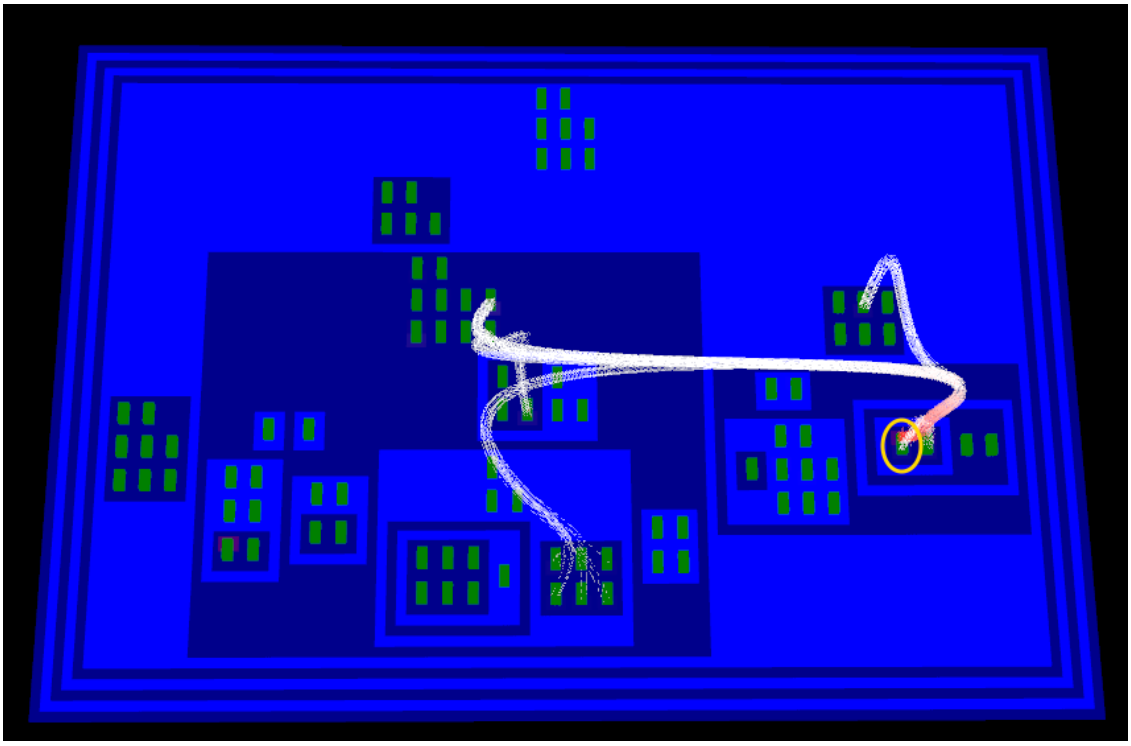

FIGURE 6.2. La classe *Chessboard2D* génère un grand nombre d'appels.

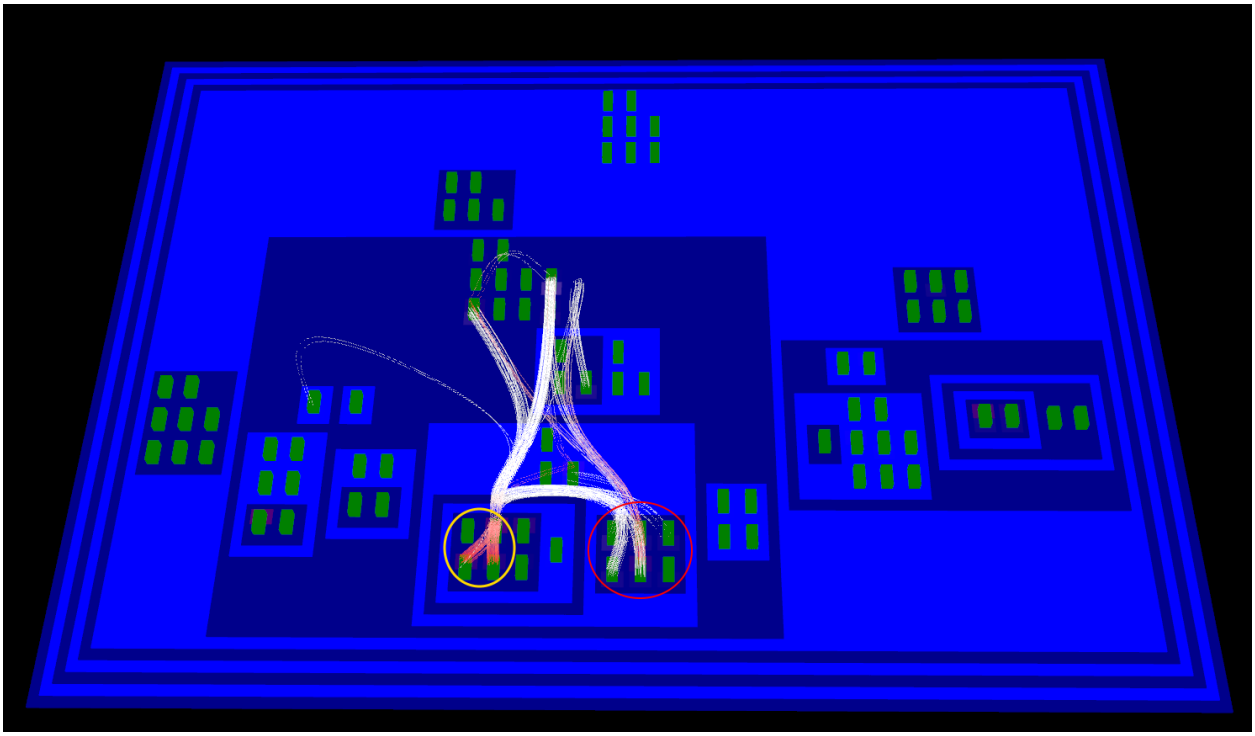

FIGURE 6.3. Les classes *PawnBehavior*, *RookBehavior*, *KnightBehavior* et *BishopBehavior*, en jaune, sont assez actives dans l'appel à la méthode *paintComponent*. Les classes correspondant aux différentes pièces, en rouge, sont également mobilisées.

L'appel suivant est celui généré par le deuxième clic de souris. On observe alors des appels autour de la classe *MoveHistory* (voir la figure 6.4) puis plus tard de la classe *Move* (figure 6.5), qui ont très probablement pour rôle d'effectuer le mouvement de la pièce au niveau de la logique du jeu, mouvement qui sera ensuite encodé par de nouveaux appels à la classe *FenNotation*. Dans le même temps, on remarque que les classes qui représentent les pièces et leur comportement (indiquées dans la figure 6.3) sont également très actives. En regardant les appels associés dans l'arbre d'appels on note que cette activité provient principalement de nombreux appels à une méthode *canPieceBeDangerToKing* de la classe *King* qui doit donc vérifier si, après le mouvement, le roi est en danger.

Ensuite, une fois l'appel résultant du clic de souris terminé, le système repasse à l'affichage en exécutant un second appel à la méthode *paintComponent* (voir la figure 6.6). On remarque cependant que celui-ci est beaucoup plus court que le premier, notamment parce que tous les appels vers les classes gérant le comportement des pièces (*PawnBehavior*, *RookBehavior*, ...) ne sont pas présents dans ce second appel. Pour pouvoir facilement comparer ces deux appels à la même méthode, nous pouvons limiter la trace à l'un puis l'autre des appels, tout en retirant la fenêtre glissante pour passer en mode cumulatif. Le résultat que l'on obtient est présenté dans la figure 6.7. Il apparaît très clairement que le premier appel à *paintComponent* est beaucoup plus long que le second. Si l'on se replace dans le scénario,

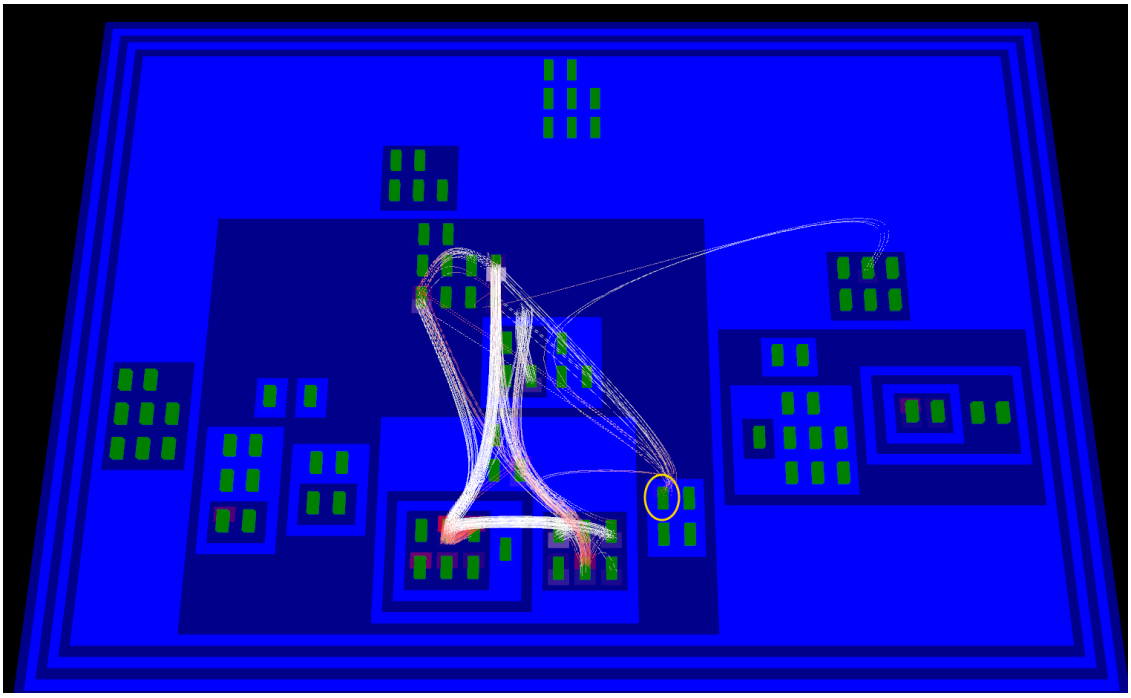

FIGURE 6.4. La classe *MoveHistory* est rapidement utilisée après le second clic de souris.

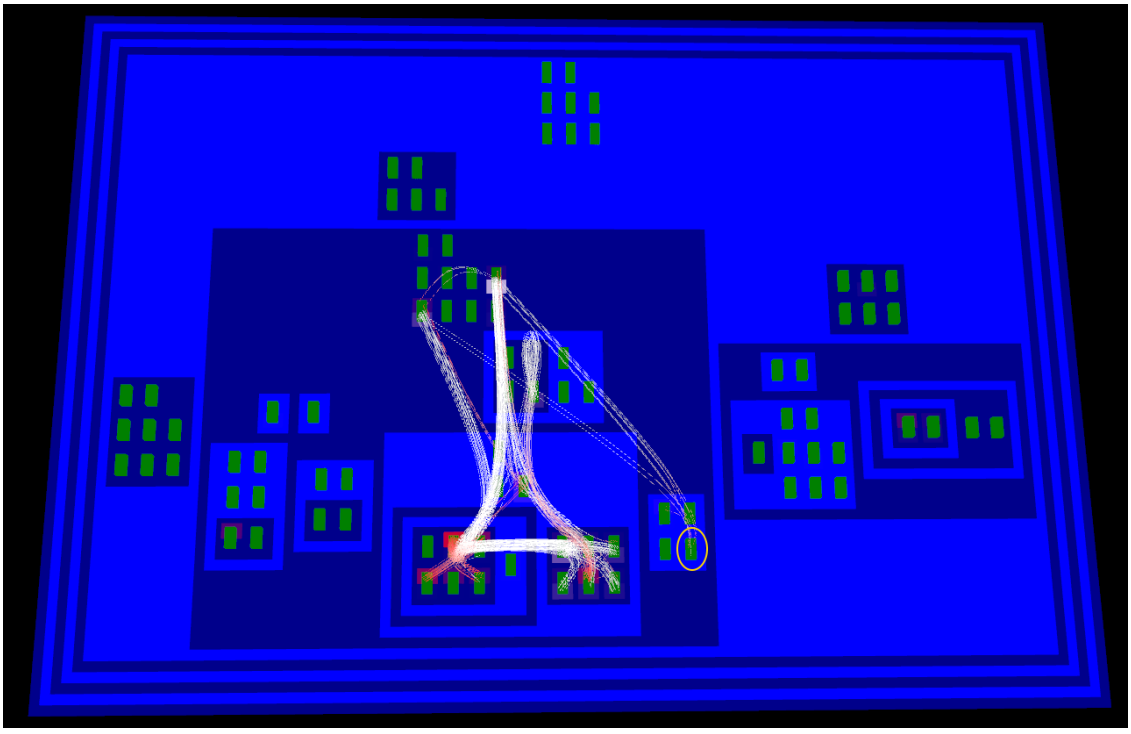

FIGURE 6.5. Un peu après *MoveHistory*, la classe *Move* qui devient active.

le premier appel arrive après le premier clic, lorsque le jeu indique au joueur quels sont les

coups possibles. En regardant le détail fourni par l'arbre d'appels, on trouve en effet un appel à une méthode *drawLegalMoves*. On peut alors en déduire que le comportement de la méthode *paintComponent* est à la fois contextuel, puisque cet appel à *drawLegalMoves* ne se retrouve pas après le second clic de souris, et n'est pas seulement dédié à de l'affichage, comme son nom pourrait le laisser supposer. On peut alors estimer qu'un réusinage de cette méthode pourrait être envisageable.

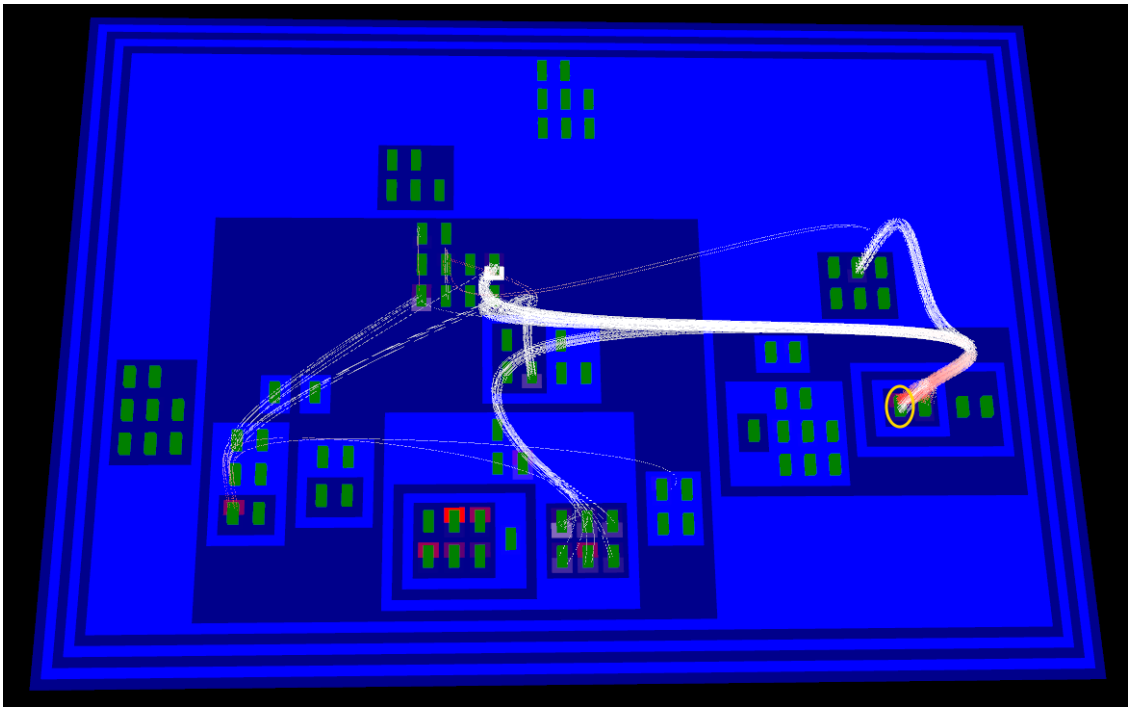

FIGURE 6.6. La classe *Chessboard2D* exécute un second appel à *paintComponent*.

Pour conclure cette première étude de cas, nous avons tenté de montrer comment utiliser un filtre pour isoler et comparer différentes parties de la trace d'exécution. Cela nous aura permis de trouver dans cet exemple un potentiel problème de conception, sans avoir eu à regarder une seule fois le code source du logiciel.

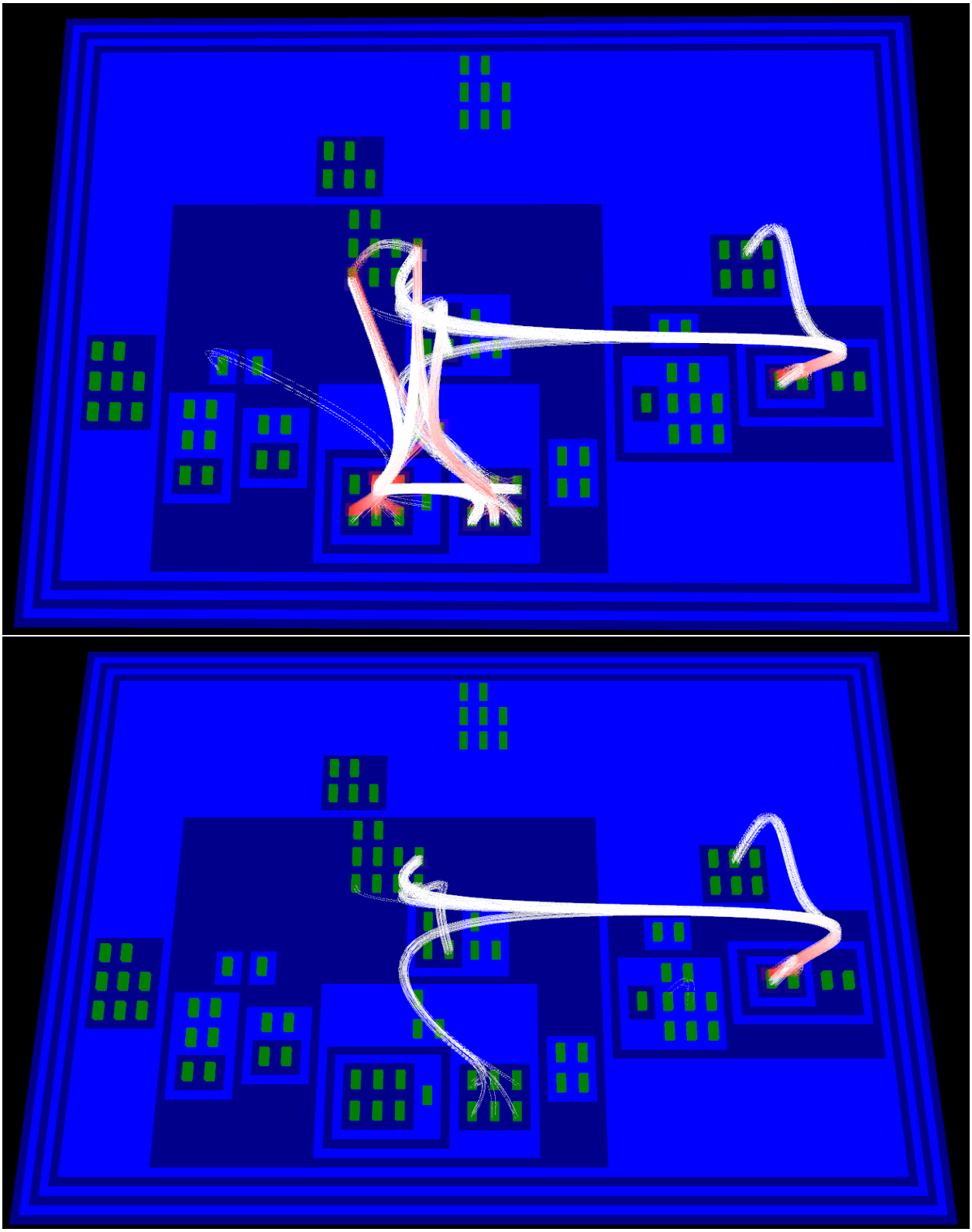

FIGURE 6.7. Comparaison des différents appels à la méthode *paintComponent*. Le premier appel est en haut, et le second en bas.

## 6.2. Second cas : l'éditeur de diagrammes UML

Dans cette seconde étude de cas nous nous intéressons à un logiciel de conception de diagrammes UML [35]. Le scénario consiste en la création d'un lien d'héritage entre deux classes préalablement ajoutées sur un diagramme de classes. Pour réaliser ce scénario, nous avons sélectionné dans un menu l'outil *Lien d'héritage* et réalisé un glisser/déposer depuis la première classe vers la seconde. Comparé au logiciel du scénario précédent, ce logiciel est plus grand en terme de nombre de classes, puisqu'il en compte 290. La trace associée est cependant de taille comparable avec environ 30 000 entrées.

Le but de cette seconde illustration est de montrer comment on peut utiliser les filtres pour réduire la taille de la trace en retirant les évènements non pertinents. Ensuite, nous détaillons les évènements intéressants de la trace pour essayer de comprendre comment le logiciel réalise le scénario que nous avons enregistré.

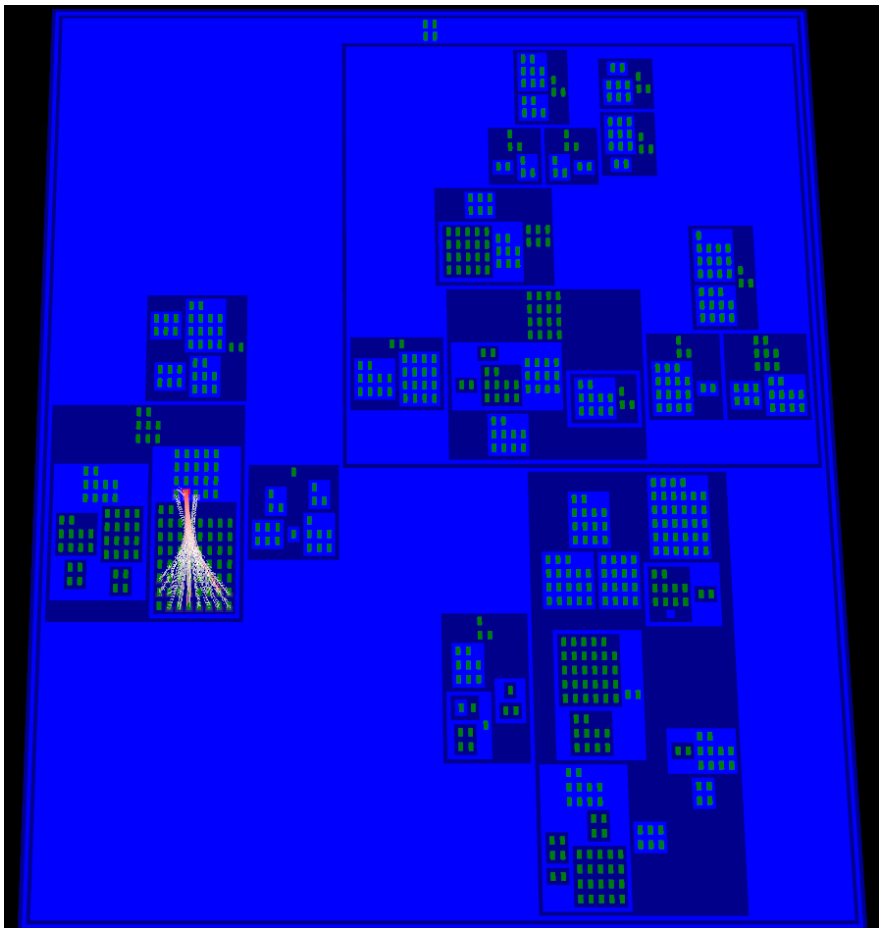

FIGURE 6.8. Au début de la trace, la zones active, à gauche de l'image, correspond aux classes dédiées au fonctionnement de l'interface

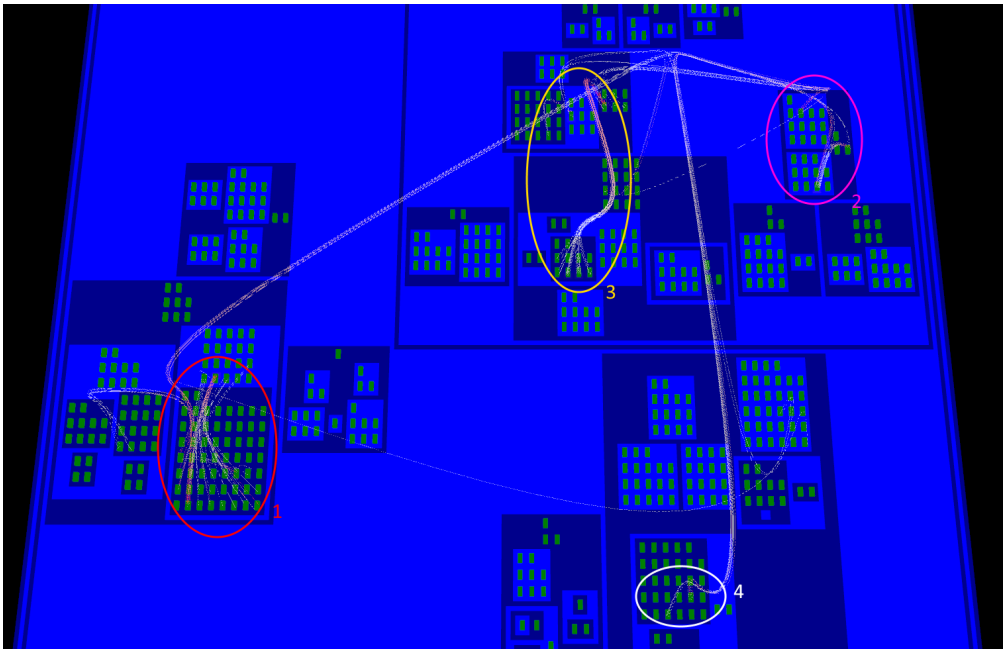


FIGURE 6.9. Au moment où l'on commence à dessiner le lien, plusieurs zones deviennent actives. La zone 1 correspond à l'interface utilisateur, la zone 2 gère le diagramme et les classes, la zone 3 représente les liens et leur affichage, et la zone 4 gère l'arrière plan.

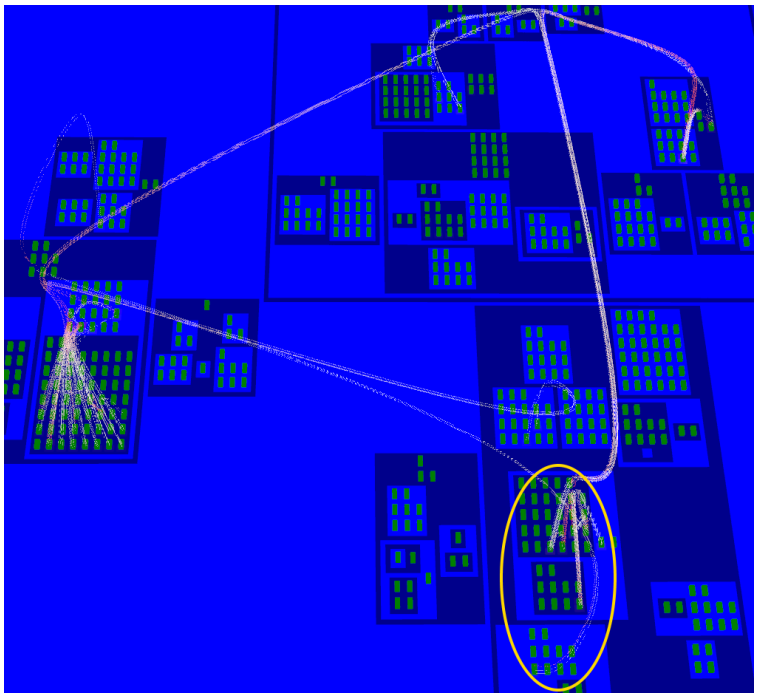

FIGURE 6.10. Pendant le tracé de la flèche, deux nouvelles classes (entourées en jaune) viennent se rajouter aux classes déjà actives présentées dans la figure précédente, notamment *ContentBorder* et *ContentInsideRectangle*.

En lançant le scénario on remarque que tout le début de la trace est constitué d'appels entre classes dédiées à la gestion de l'interface. En regardant dans l'arbre d'appels, on remarque que ces appels sont ceux à la méthode *mouseMoved* de la classe *EditorPart* qui est la classe qui apparaît en rouge sur la figure 6.8. Comme, d'après son nom, il s'agit d'une méthode appelée à chaque mouvement de souris sur le canevas, et qui ne semble pas modifier l'état du diagramme, puisqu'elle ne communique pas avec les classes qui le gèrent, nous avons alors choisi d'enlever de l'animation tous ces appels qui surviennent avant le premier clic de souris. En faisant cela nous retirons environ un tiers des appels de l'animation, qui commence maintenant à partir du premier clic de souris, c'est à dire au début du glisser/déposer dans le scénario. Au moment où l'on presse le bouton de la souris pour commencer le glisser/déposer, les classes associées à l'interface de l'éditeur (zone 1 sur la figure 6.9) vont alors appeler les classes gérant le diagramme pour récupérer la classe de départ et sa position dans le canevas (zone 2 sur la figure 6.9). Il crée ensuite une instance de la classe *IneritanceEdge*, l'associe à la classe de départ, et choisit ensuite la forme de la flèche qui sera dessinée (zone 3 sur la figure 6.9). Le système récupère également l'arrière plan et divers éléments visuels (zone 4 sur la figure 6.9).

Une fois la flèche créée, la suite de la trace comprend plusieurs appels chargés de répondre au déplacement de la souris en adaptant la taille de la flèche pour qu'elle suive le curseur. Sans entrer dans le détail, les seules nouvelles classes à entrer en jeu s'occupent principalement d'éléments d'affichage (par exemple *ContentBorder* et *ContentInsideRectangle* dans la zone entourée de la figure 6.10).

Ensuite nous arrivons au point d'intérêt suivant qui est le moment où l'on relâche la souris. L'éditeur va alors commencer par récupérer la classe d'arrivée et mettre à jour le lien d'héritage créé précédemment. Une fois faite, la création du lien va alors engendrer une série d'appels vers différentes classes, dans le but de déterminer la forme de la courbe comme le montre la figure 6.12. À ce moment là, quatre classes sont particulièrement impliquées : *ClassNode* (cercle 1), *InheritanceEdge* (cercle 2), *ContentBackground* (cercle 3) et *Direction* (cercle 4). On remarque également que les appels ne se font qu'à partir de *InheritanceEdge* et *ClassNode*. En jouant pas à pas cette partie de la trace on peut déduire la procédure suivante : le lien veut obtenir en premier la position des classes et leurs dimensions. Pour cela les instances de *ClassNode* font différents appels à *ContentBackground*. Ensuite le lien veut connaître les points de connexion des classes, qui vont alors faire plusieurs appels aux classes *BackgroundContent* et *Direction*, pour déterminer quel point de connexion le lien doit

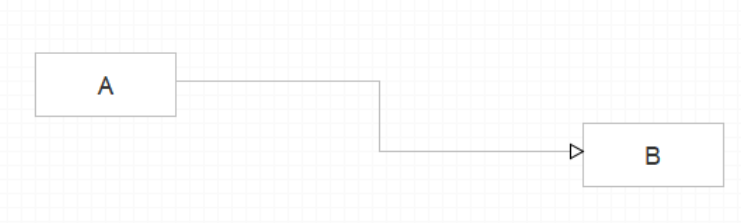


FIGURE 6.11. Capture du rendu dans l'éditeur UML après notre scénario. Les classes sont ici de simples rectangles reliés par une flèche coudée.

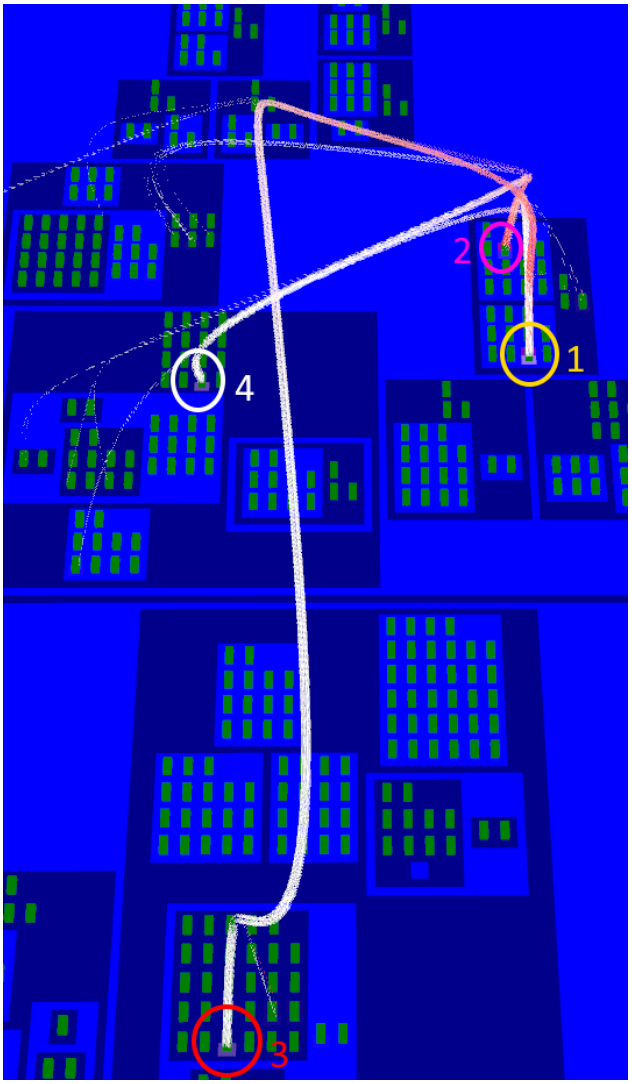


FIGURE 6.12. On observe ici les classes actives lors de la fin du scénario, c'est à dire la création du lien définitif. Les quatre classes les plus actives sont : *ClassNode* (cercle 1), *InheritanceEdge* (cercle 2), *ContentBackground* (cercle 3) et *Direction* (cercle 4).

utiliser. Ensuite, le lien récupère le type de courbe choisie dans la classe *BentStyleChoiceList*, fourni les données récupérées précédemment et détermine le tracé de la courbe. Par défaut dans cet éditeur, le type de courbe n'est pas une droite mais plutôt un chemin qui suit la grille selon un tracé appelé "HVH" dans l'éditeur, qui ressemble à ce que présente la figure 6.11. Finalement la trace se termine par un affichage de tous les éléments par l'appel à *paintImmediately* de *EditorPart*.

En conclusion, dans cette seconde étude de cas nous avons cherché à montrer une autre utilisation de notre approche, qui vise à détailler, un peu à la manière d'un diagramme de

séquence, la succession des différents appels effectués pour réaliser une fonctionnalité de haut niveau du logiciel observé.

## 6.3. Synthèse

Avec ces deux études de cas, nous avons cherché à illustrer les différentes fonctionnalités que nous avons implantées. La première montre comment on peut se servir des filtres pour comparer deux exécutions d'une même méthode. La seconde tente de reconstituer la succession des évènements importants de la trace en filtrant les évènements mineurs.

Ces études de cas ne constituent cependant pas une évaluation complète et suffisante pour juger de l'utilité et de l'efficacité de notre approche. Pour cela, il faudrait réaliser une ou plusieurs études utilisateurs, sur des logiciels connus ou non par les utilisateurs, avec différentes cas de figure et objectifs, comme des scénarios présentant une erreur (*bug*) et dont l'objectif serait de la repérer et d'en trouver la source, des scénarios dont le but serait de trouver des défauts de conception via les traces, ou encore des scénarios dont le but serait simplement de comprendre le comportement du logiciel et le rôle de ses composants.

En ce qui concerne les fonctionnalités, celles présentes constituent une base permettant de filtrer et d'explorer des traces. De nombreuses autres fonctionnalités restent envisageables, comme étiqueter différentes classes ou méthodes, avoir accès au code source, voir les communications intra-classe, extraire des diagrammes de séquence, etc.

# Chapitre 7

# Conclusion

Dans ce mémoire nous avons proposé une utilisation de la visualisation interactive du logiciel pour aider le développeur à comprendre le comportement d'un système à partir de traces d'exécution. Notre environnement combine une scène 3D et une métaphore de traces lumineuses qui représentent respectivement la structure du système et les évènements des traces d'exécution. L'environnement offre plusieurs méthodes d'interactions qui permettent l'exploration du système (recherche de classes, de paquets, ...) et de la trace d'exécution au moyen d'une animation qui peut être contrôlée à la manière d'un lecteur vidéo standard. La structure et la trace d'exécution sont également affichées dans une interface, via une vue arborescence, synchronisée avec la scène 3D, offrant à l'utilisateur plusieurs modes d'interactions. Un ensemble de filtres permettent la réduction de la trace par le développeur, lui permettant de retirer certains évènements, soit en les éliminant, soit en réduisant la fenêtre de l'animation, pour se focaliser sur une phase de l'exécution par exemple.

Nous avons évalué l'utilité de notre approche à travers deux études de cas, utilisant deux logiciels de tailles différentes. Nous avons cherché à montrer à travers ces exemples qu'il était possible d'explorer efficacement de grandes traces d'exécution pour comprendre comment se comportent certaines fonctionnalités d'un programme.

Cependant, bien que ces études de cas amènent des perspectives intéressantes, il sera nécessaire de mener des évaluations plus rigoureuses pour arriver à des conclusions plus définitives quant à ce type d'approches [15]. En particulier, des études utilisateur pour comparer notre environnement à d'autres approches de compréhension du logiciel, sur différentes tâches et aspects, constitueraient une validation bien plus complète.

Par rapport à l'environnement lui-même, nous avons des idées quant à de futurs ajouts pour améliorer notamment sa scalabilité et son efficacité. Par exemple nous pourrions pousser encore plus loin la métaphore des traces de lumière en faisant en sorte que les différents appels de méthodes soient représentés à la manières de feux avant et arrière de voitures sur une autoroute. Une texture animée placée sur un ruban pourrait permettre ce genre de rendu, et réduirait également le nombre de courbes à afficher. L'évaluation du nombre d'appels ne se ferait alors plus par la quantité de liens entre deux éléments mais par le nombre de "voitures" sur la texture.

Toujours en rapport avec le rendu graphique, les rubans fins que nous utilisons ont l'inconvénient de disparaître ou d'être très affectés par l'aliassage sous certains angles de vue. Ce problème pourrait être résolu en leur attribuant une taille minimale projetée, qui s'adapterait à l'angle de vue. Enfin, nous pourrions donner aux cartes de chaleur une plus grande

importance, ou alors les remplacer par des effets d'illumination globale entre les classe et paquets, en fonction du nombre de voitures arrivées ou parties d'une certaine classe par exemple.

Finalement, en ce qui concerne les fonctionnalités permettant d'étudier le comportement du logiciel, nous envisageons d'ajouter un moyen pour le développeur d'étiqueter les méthodes ou classes d'intérêts, afin d'extraire un diagramme de séquence condensé, à la manière de ce qu'ont fait Grati et al. [36]. Cela devra être combiné avec des approches automatisées, permettant par exemple d'identifier des structures de contrôle (boucles, appels récursifs, ...) ou encore de remplacer des objets concrets par leur type abstrait lorsque cela est nécessaire. On peut également penser à d'autres fonctionnalités pour assister le développeur, comme pouvoir facilement accéder au code source d'une méthode, ou encore fournir des indications quant à l'utilisation des ressources des différentes phases d'exécutions.

# Références bibliographiques


[1] Rakan Alanazi, Gharib Gharibi et Yugyung Lee : Facilitating program comprehension with call graph multilevel hierarchical abstractions. *Journal of Systems and Software*, 176, 2021.

[2] Philippe Dugerdil et Sazzadul Alam : Execution trace visualization in a 3d space. *In Fifth International Conference on Information Technology : New Generations*, pages 38–43, 2008.

[3] Bas Cornelissen, Andy Zaidman et Arie van Deursen : A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering*, 37(3):341–355, 2011.

[4] J. Stasko et E. Zhang : Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. *In IEEE Symposium on Information Visualization 2000. INFOVIS 2000. Proceedings*, pages 57–65, 2000.

[5] Weixin Wang, Hui Wang, Guozhong Dai et Hongan Wang : Visualization of large hierarchical data by circle packing. *In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, page 517–520, New York, NY, USA, 2006. Association for Computing Machinery.

[6] Doantam Phan, Ling Xiao, Ron Yeh, Pat Hanrahan et Terry Winograd : Flow map layout. *In Proceedings of the IEEE Symposium on Information Visualization*, pages 219–224, 2005.

[7] Danny Holten et Jarke J. Van Wijk : Force-directed edge bundling for graph visualization. *Computer Graphics Forum*, 28(3):983–990, 2009.

[8] Danny Holten : Hierarchical edge bundles : Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748, 2006.

[9] Simon Bouvier : Utilisation de la visualisation interactive pour l'analyse des dépendances dans les logiciels. Mémoire de maîtrise, Université de Montréal, 2011.

[10] Yang Feng, Kaj Dreef, James A. Jones et Arie van Deursen : Hierarchical abstraction of execution traces for program comprehension. *In 26th Conference on Program Comprehension*, page 86–96, 2018.

[11] Steven P. Reiss et Manos Renieris : Encoding program executions. *In 23rd International Conference on Software Engineering*, page 221–230, 2001.

[12] Saba Alimadadi, Ali Mesbah et Karthik Pattabiraman : Inferring hierarchical motifs from execution traces. *In 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 776–787, 2018.

[13] Qi Xin, Farnaz Behrang, Mattia Fazzini et Alessandro Orso : Identifying features of android apps from execution traces. *In 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 35–39, 2019.

[14] Wuxia Jin, Ting Liu, Yuanfang Cai, Rick Kazman, Ran Mo et Qinghua Zheng : Service candidate identification from monolithic systems based on execution traces. *IEEE Transactions on Software Engineering*, 47(5):987–1007, 2021.

[15] Omar Benomar, Houari Sahraoui et Pierre Poulin : Detecting program execution phases using heuristic search. *In International Symposium on Search Based Software Engineering*, pages 16–30, 2014.

[16] Kunihiro Noda, Takashi Kobayashi et Kiyoshi Agusa : Constructing object groups corresponding to concepts for recovery of a summarized sequence diagram. *Journal of Information Processing*, 29:305–320, 2021.

[17] R. Devi Sree et J. Swaminathan : Construction of activity diagrams from java execution traces. *In Ambient Communications and Computer Systems*, pages 641–655, 2018.

[18] Omar Benomar, Houari Sahraoui et Pierre Poulin : Visualizing software dynamicities with heat maps. *In 2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–10, 2013.

[19] Adrian Kuhn et Orla Greevy : Exploiting the analogy between traces and signal processing. *In 22nd IEEE International Conference on Software Maintenance*, pages 320–329, 2006.

[20] Fatemeh Asadi, Massimiliano Di Penta, Giuliano Antoniol et Yann-Gaël Guéhéneuc : A heuristic-based approach to identify concepts in execution traces. *In 2010 14th European Conference on Software Maintenance and Reengineering*, pages 31–40, 2010.

[21] Anas Shatnawi, Hudhaifa Shatnawi, Mohamed Aymen Saied, Zakarea Al Shara, Houari Sahraoui et Abdelhak Seriai : Identifying software components from object-oriented apis based on dynamic analysis. *In Proceedings of the 26th Conference on Program Comprehension*, page 189–199, 2018.

[22] A. Hamou-Lhadj, E. Braun, D. Amyot et T. Lethbridge : Recovering behavioral design models from execution traces. *In Ninth European Conference on Software Maintenance and Reengineering*, pages 112–121, 2005.

[23] Wim De Pauw, David Lorenz, John Vlissides et Mark Wegman : Execution patterns in object-oriented visualization. *In Proceedings of the 4th Conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 4*, 1998.

[24] Manos Renieris et Steven P Reiss : Almost : Exploring program traces. *In Proceedings of the workshop on new paradigms in information visualization and manipulation*, pages 70–77, 1999.

[25] Johannes Bohnet, Martin Koeleman et Juergen Doellner : Visualizing massively pruned execution traces to facilitate trace exploration. *In 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 57–64, 2009.

[26] Jonas Trümper, Jürgen Döllner et Alexandru Telea : Multiscale visual comparison of execution traces. *In 2013 21st International Conference on Program Comprehension (ICPC)*, pages 53–62, 2013.

[27] Rémy Dautriche, Renaud Blanch, Alexandre Termier et Miguel Santana : Traceviz : A visualization framework for interactive analysis of execution traces. *In Actes de La 28ième Conference Francophone Sur l'Interaction Homme-Machine*, page 115–125, 2016.

[28] Steven P. Reiss : Dynamic detection and visualization of software phases. *In Proceedings of the Third International Workshop on Dynamic Analysis*, page 1–6, 2005.

[29] Ben Shneiderman : Tree visualization with tree-maps : 2-d space-filling approach. *ACM Trans. Graph.*, 11(1):92–99, janvier 1992.

[30] Weiwei Cui, Hong Zhou, Huamin Qu, Pak Chung Wong et Xiaoming Li : Geometry-based edge clustering for graph visualization. *IEEE Transactions on Visualization and Computer Graphics*, 14(6): 1277–1284, 2008.

[31] Guillaume Langelier, Houari Sahraoui et Pierre Poulin : Visualization-based analysis of quality for large-scale software systems. *In 20th IEEE/ACM international Conference on Automated software engineering*, pages 214–223, 2005.

[32] Karim Dhambri, Houari Sahraoui et Pierre Poulin : Visual detection of design anomalies. *In 12th European Conference on Software Maintenance and Reengineering*, pages 279–283, 2008.

[33] Guillaume Langelier, Houari Sahraoui et Pierre Poulin : Exploring the evolution of software quality with animated visualization. *In IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 13–20, 2008.

[34] Java Open Chess. https://sourceforge.net/projects/javaopenchess/. Visité : 2021-02-12.

[35] Violet UML Editor . https://sourceforge.net/projects/violet/. Visité : 2021-02-26.

[36] Hassen Grati, Houari Sahraoui et Pierre Poulin : Extracting sequence diagrams from execution traces using interactive visualization. *In 2010 17th Working Conference on Reverse Engineering*, pages 87–96, 2010.