Amélioration du design et des fonctionnalités du debugger Pharo

Adrien Vanègue

Formation: **Licence 3 Informatique parcours Info**

Stage déroulé du *11 avril au 31 août 2022*

Entreprise:
Institut National de Recherche en Informatique et en Automatique ;
Park Plaza, Parc scientifique de la Haute-Borne,
40 Av. Halley Bât A,
59650 Villeneuve-d'Ascq

Université:
Université de Lille ;
42 Rue Paul Duez,
59000 Lille

Tuteur d'entreprise: M. Steven Costiou

Tuteur universitaire: M. Romain Rouvoy

Remerciements

Je tiens tout d'abord à remercier M. Ducasse, responsable de l'équipe RMOD, de m'avoir accueilli comme stagiaire au sein de son équipe.

Merci également à M. Costiou, mon tuteur de stage au sein de l'entreprise, animateur de l'équipe de debugging de RMOD, pour ses conseils avisés et la confiance qu'il m'a accordée tout au long de mon stage.

Je tiens aussi à remercier M. Rouvoy, mon tuteur universitaire de stage, pour l'attention qu'il a bien voulu apporter aux questions que j'ai pu avoir durant ce stage.

Toutes ces personnes ont contribué, par leur disponibilité, à rendre mon stage enrichissant et motivant.

Résumé

J'effectue mon stage à Inria dans l'équipe RMOD, spécialisée dans la remodularisation de logiciels orientés-objet. Ma contribution s'inscrit dans le projet OCRE dont l'objectif est de construire la première génération de debuggers centrés sur les objets. Ces outils puissants demandent une infrastructure solide. Dans ce cadre, j'ai réalisé quatre contributions pour améliorer l'infrastructure du debugger Pharo.

J'ai séparé les préoccupations du debugger Pharo car certaines des fonctionnalités de son API étaient codés dans son interface graphique (GUI). Ainsi, il n'était pas possible de tester l'API du debugger indépendamment du GUI et inversement, ce qui est nécessaire pour assurer leur fiabilité. J'ai séparé l'API du debugger de son GUI, en migrant toute l'API du debugger vers un modèle intermédiaire entre le GUI et le modèle du debugger. En conséquence, le debugger Pharo est maintenant plus extensible. En effet, la délégation de l'UI vers un objet intermédiaire permet de respecter le principe de responsabilité unique. Ainsi, il est maintenant possible d'avoir des debuggers sans UI pour tester séparément l'API ; tout comme il est possible de changer l'UI du debugger sans avoir à réimplémenter des parties de son API.

J'ai aussi amélioré et corrigé une extension du debugger proposant une API pour écrire des scripts de debugging. Du code ne fonctionnait plus car cet outil était à l'abandon pendant des années alors que Pharo évoluait pendant ce temps. J'ai actualisé ce code de manière à utiliser la nouvelle API de Pharo. Certaines commandes de debugging manipulant du code bas-niveau avaient un comportement indéterminé. Je les ai analysé en profondeur afin de définir les différents cas d'utilisation, pour proposer des implémentations qui couvrent au mieux ces cas.

J'ai pu amélioré l'interface du debugger, jusque-là statique, en la rendant dynamique pour faire évoluer l'apparence du debugger lorsqu'une extension est (dés)activée. J'ai aussi réalisé un outil expérimental de coffre-forts permettant de stocker depuis n'importe quel code Pharo des objets qui peuvent être chargés dans le debugger.

Enfin, j'ai même pu participer à la rédaction d'un papier de recherche sur un debugger objet-centrique permettant de voyager dans le temps.

Abstract

I am doing my internship at Inria, in the RMOD team specialized in the remodularization of object-oriented software. My contribution is part of the OCRE project whose objective is building the first generation of object-centric debuggers. Such powerful tools require a solid infrastructure. In this context, I have made four contributions to improve the infrastructure of the Pharo debugger.

I have separated concerns in the Pharo debugger as some features of its API were coded in its graphical interface (GUI). So, it wasn't possible to test its API separately from its GUI or vice versa, which is necessary to ensure its reliability. I have separated the debugger API from its GUI, by migrating the entire debugger API to an intermediate model between the GUI and the debugger model. Consequently, the Pharo debugger is now more extensible, as its GUI delegates all debugger model accesses to an intermediary model, which complies with single responsibility principle. It is now possible to have debuggers without GUIs. In addition, it is possible to change the debugger GUI without implementing again parts of its API.

I have also improved and fixed a debugger extension, offering an API to write debugging scripts. Some code was not working anymore as it had not been maintained for years while Pharo evolved. I have updated this code so that it uses the new Pharo API. Some debugging commands using low level code had undefined behaviors. I have deeply analyzed them to define all use cases in order to build implementations that cover at best these cases.

I have improved the debugger GUI, that was static until then, to make it dynamic so that the debugger appearance evolves when an extension is enabled or disabled. I have also rebuild an experimental tool, allowing the storage of any object from anywhere in Pharo into chests that can be accessed from the debugger to load their objects into the debugger.

Finally, I have even taken part in the writing of a research paper about object-centric time-travelling debuggers.

Option DPP: Enjeux environnementaux du numérique

Dans le cadre de l'UE DPP, j'ai suivi l'option "Enjeux environnementaux du numérique" qui m'a permis de participer à la fresque du numérique [3]. Dans cette section, je traite de ce qui m'a interpellé dans cette fresque, de comment ma vision du numérique a changé puis je finirai par parler des actions effectuées à Inria.

Mes découvertes lors de la fresque du numérique Participer à la fresque du numérique m'a permis d'ouvrir les yeux sur les impacts du numérique sur l'environnement. En effet, au-delà du fait que les équipements numériques, infrastructures réseaux et data centers consomme de l'électricité qui est produite en consommant de l'énergie fossile, je n'imaginais pas d'autres conséquences. Par exemple, j'ai découvert que la fabrication d'un ordinateur de 2kg nécessite un sac à dos écologique impressionnant : il faut 200 kg d'énergies fossiles et 600 kg de minéraux, principalement pour l'extraction et le raffinage des métaux, ce qui constitue 800 kg de ressources et c'est sans compter plusieurs milliers de litres d'eau douce utilisées pour la fabrication. Savoir cela et savoir que cela entraîne une pénurie de ressources qui peut déboucher sur des tensions géopolitiques voire des conflits, cela fait réfléchir.

Un autre aspect qui m'a interpellé dans cette fresque est l'impact social et éthique qu'il y a derrière les impacts environnementaux du numérique. Effectivement, le fait de ne pas utiliser du matériel encore fonctionnel tout en achetant du matériel toujours plus récent à la place entraîne énormément de déchets électroniques et seulement 17% d'entre eux son recyclés alors que le reste est enfoui, incinéré ou géré par des circuits illégaux. 60% de ces déchets sont gérés par des circuits illégaux dans lesquels les travailleurs sont traités dans des conditions désastreuses pour extraire quelques éléments avant d'abandonner les restes toxiques dans des décharges. Savoir cela fait aussi réfléchir.

Ainsi, des actions, comme acheter un ordinateur, ont des conséquences graves que je n'imaginais même pas.

Ma vision de l'utilisation du numérique après avoir participé à la fresque La fresque du numérique a changé ma vision du numérique sur certains points. En effet, je faisais déjà attention à mes usages du numérique et à mon matériel de manière général. Notamment, j'éteignais déjà tout appareil numérique non utilisé pour réduire la consommation électrique et je prenais déjà soin déjà soin de mon matériel sans acheter de nouvel équipement tant que le mien fonctionnait afin d'éviter le gaspillage. La fresque du numérique a vraiment changé ma vision du numérique au niveau des appareils obsolètes et/ou ne fonctionnant plus. Connaissant maintenant le sac à dos écologique de la fabrication d'un nouvel ordinateur ainsi que les conséquences humaines et environnementales des déchets électroniques d'un appareil inutilisé ; je penserai désormais à essayer de réparer mon matériel avant tout si c'est possible, puis à acheter d'occasion plutôt que neuf tout en recyclant ce qui est recyclable dans mes anciens appareils, dans le but de limiter ces conséquences.

Les actions de mon entreprise pour réduire l'impact environnemental du numérique Ce stage m'a permis de discuter de l'impact du numérique sur l'environnement, avec les employés d'Inria. L'entreprise entreprend des actions pour sensibiliser au numérique. Dans ce cadre, un MOOC [5] a été ouvert le 22 novembre 2021 dans le but de sensibiliser les lycéens, les enseignants et le grand public à l'empreinte écologique du numérique. Découpé en quatre parties, il propose des activités, des vidéos et des ressources complémentaires destinées à faire prendre conscience à chacun de la matérialité des mondes virtuels et des enjeux associés au développement très rapide du numérique.

D'autres équipes de recherche d'Inria entreprennent des actions pour une informatique plus écologique, comme l'équipe Spirals avec son projet *PowerAPI* [7] qui fournit des applications permettant de mesurer la consommation énergétique d'un programme sur une ou plusieurs machines. *PowerAPI* donne aussi la possibilité de créer son propre compteur de consommation énergétique, en reliant à une base de données externe deux composants :

- Un logiciel indépendant qui collecte les données brutes corrélées à la consommation énergétique du logiciel dont on mesure la consommation, et qui stocke ces données dans la base de données externe,
- un logiciel indépendant qui calcule une estimation de la consommation énergétique du logiciel, à partir des données collectées par le premier logiciel indépendant.

Table des matières

1 Introduction

Mon stage se déroule à Inria, au sein de l'équipe RMOD. Dans cette section, j'explique le contexte dans lequel s'inscrit mon stage, les objectifs du stage ainsi que mes motivations.

Présentation du contexte du stage. Inria est l'institut national de recherche en sciences et technologies. La recherche de rang mondial, l'innovation technologique et le risque entrepreneurial constituent son ADN. Au sein de 200 équipes-projets, pour la plupart communes avec les grandes universités de recherche, plus de 3 900 chercheurs et ingénieurs y explorent des voies nouvelles, souvent dans l'interdisciplinarité et en collaboration avec des partenaires industriels pour répondre à des défis ambitieux [4].

Le but de RMOD est de perpétuellement faire évoluer des systèmes afin qu'ils ne deviennent jamais obsolètes. Pour cela, l'équipe analyse des logiciels industriels avec un long cycle de vie afin de développer des outils pour les améliorer. Afin d'améliorer des outils utilisés tous les jours par les développeurs, comme les debuggers, ainsi que pour revisiter et concevoir des nouvelles fonctionnalités de développement, l'équipe développe Pharo, un langage réflexif dérivé de Smalltalk dans lequel tout est objet, dans le but de pouvoir explorer des solutions traitant ces problèmes [8].

Le debugging centré sur les objets est une technique récente soutenant que la concentration du debugging sur des objets spécifiques facilite le suivi et la compréhension de bugs. Mon stage s'inscrit dans le projet **OCRE** dont l'objectif est de construire la première génération de debuggers centrés sur les objets afin d'identifier et d'évaluer les bénéfices de la technique pour le debugging de systèmes à objets. Un des objectifs est de transférer ces prototypes de debuggers avancés vers l'open-source et vers le monde industriel [6]. Afin de concevoir de tels outils, il est nécessaire d'avoir une infrastructure de debugger solide. Toutes mes contributions lors de ce stage convergent vers cet objectif.

Les objectifs du stage. Dans ce cadre, un objectif de mon stage est d'améliorer le design du debugger. Pour l'instant, certaines parties de l'implémentation de l'API du debugger sont codées dans son interface graphique, ce qui ne respecte pas le principe de responsabilité unique. Séparer les préoccupations permettra de le rendre plus extensible et modulaire. Ainsi, pour changer l'interface graphique du debugger, il faut recoder les parties de l'implémentation de l'API du debugger dans sa nouvelle interface graphique, ce qui est source de bugs et perte de temps. Avoir un modèle de debugger à part entière, qui implémente l'API, permettra de tester en profondeur l'API du debugger afin d'en assurer la fiabilité. De plus, documenter les classes du debugger facilitera leur compréhension, afin de s'en servir comme base des debuggers centrés sur les objets.

Ensuite, *Sindarin* est un outil du debugger proposant une API pour écrire des scripts de debugging. Cette API n'a pas été conçue pour répondre à des besoins des utilisateurs, est donc peu utilisée et n'a pas été maintenue. Cet outil propose aussi des commandes de debugging qui sont instables, pouvant être la cause de plantage ou de comportements indéfinis. OCRE a besoin de cette API et de ces commandes afin de pouvoir customiser les outils de debugging. Un autre objectif du stage est donc d'améliorer la stabilité de l'outil, puis d'améliorer son API pour qu'elle soit plus facile à utiliser.

Ce stage me permettra aussi de participer à l'amélioration d'outils utilisés et développés par l'équipe de debugging, comme *Chest* qui est un gestionnaire de coffres-forts permettant de stocker des objets dans des coffres et d'y accéder depuis n'importe quel code Pharo. Son implémentation peut être améliorée car elle identifie les objets par des numéros, qui ne sont pas intuitifs pour des humains.

Enfin, étant en stage dans une équipe de recherche, ce stage est l'occasion pour moi de découvrir le milieu de la recherche en participant à la rédaction d'un petit papier de recherche sur un prototype de debugger centré sur les objets permettant de voyager dans le temps.

Mes motivations à réaliser ce stage Personnellement, j'ai toujours été quelqu'un de perfectionniste. Dès que je trouve un défaut dans ce que je fais, je me sens obligé de l'améliorer. C'est entre autres pour

cela que la conception orientée objet m'a charmé : car elle permet de faire les choses correctement et durablement dans le temps, si elle est bien pensée.

Dans notre société de consommation actuelle, il est facile de constater que mon credo est à l'opposé des normes. Cela se voit particulièrement bien dans le domaine des jeux-vidéos, par exemple. En effet, vous pourrez facilement constater qu'il est très rare de rencontrer des bugs dans les jeux sortis autrefois. A l'inverse, aujourd'hui, le but est de sortir les jeux rapidement, puis de continuellement proposer du contenu sur ces jeux. La quantité est privilégiée à la qualité et cela se voit à la quantité impressionnante de bugs que l'on rencontre dans les jeux de notre époque. Cela témoigne du fait que le temps qui aurait dû être pris pour faire les choses correctement, n'a pas été pris car plus on va vite, plus on gagne d'argent vite.

Ainsi, lorsque j'ai entendu parler de l'équipe RMOD spécialisée dans la remodularisation de logiciels orientés-objet, cela a été un peu une aubaine. Effectivement, car une équipe qui fait les choses biens dans le domaine d'informatique est rare. De plus, réaliser mon stage dans le domaine du debugging est un plus que cela: c'est donner l'occasion aux autres de faire les choses correctement, à l'aide d'un outil que j'aurais amélioré : le debugger.

De plus, j'apprécie manipuler du code complexe et bas-niveau, ce que Sindarin m'a permis de faire en manipulant le bytecode.

Les contributions explicitées dans ce rapport seront énoncées à la première personne du pluriel car cela constitue un travail d'équipe. Le rapport se structure comme suit :

1. nous expliquerons le fonctionnement du debugger Pharo qui est au coeur de mon stage(section 2),
2. nous expliquerons comment nous avons amélioré le design du debugger (section 3),
3. nous expliquerons comment nous avons corrigé des commandes de debugging complexes dans l'outil *Sindarin* (section 4),
4. j'expliquerai de quelles manières je me suis intégré à l'équipe (section 5).

2 Contexte technique : le debugger Pharo

La première étape pour améliorer le design du debugger a été d'identifier les problèmes qu'il y a dans le debugger. J'ai donc analysé le code source du debugger Pharo afin de le comprendre. Dans cette section, nous expliquerons le fonctionnement du debugger Pharo ainsi que les objets clefs à son fonctionnement.

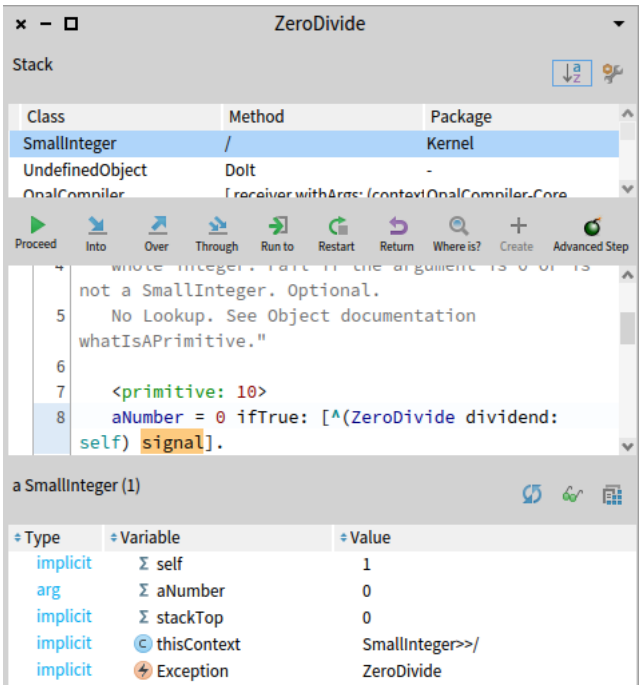

Figure 1: Vue d'ensemble du debugger.

L'interface utilisateur du debugger. Le debugger Pharo est constitué de plusieurs éléments décrits ci-dessous et visibles sur la Figure 1 :

- La pile de contextes du programme débuggué. Un contexte est un objet décrivant l'état dynamique associé à l'exécution d'une méthode ou d'un bloc (= fonction anonyme). Dès qu'une méthode ou un bloc est exécuté, un nouveau contexte est créé et si un contexte A se trouve juste au-dessus d'un contexte B dans la pile de contextes, alors le contexte A est appelé par le contexte B. Sur la Figure 1, la méthode `/` est appelée par la méthode `DoIt`.
- La barre d'outils, qui définit des opérations à appliquer sur le contexte sélectionné dans la pile, notamment des opérations de stepping pour exécuter le programme pas-à-pas. Le menu *Advanced Step* propose des commandes de debugging plus avancées, définies dans l'entension `Sindarin` du debugger qui permet d'écrire des scripts de debugging.
- Le code du programme débuggué, que l'utilisateur peut modifier et recompiler de suite, auquel cas le contexte sélectionné redémarre de zéro.
- Un inspecteur affichant toutes les variables du contexte sélectionné ainsi que leur valeur.

Le debugger possède un système d'extension qu'il est possible d'activer dans les paramètres. Dans ce cas, les extensions activées sont affichées dans une colonne à droite du debugger (Figure 2).

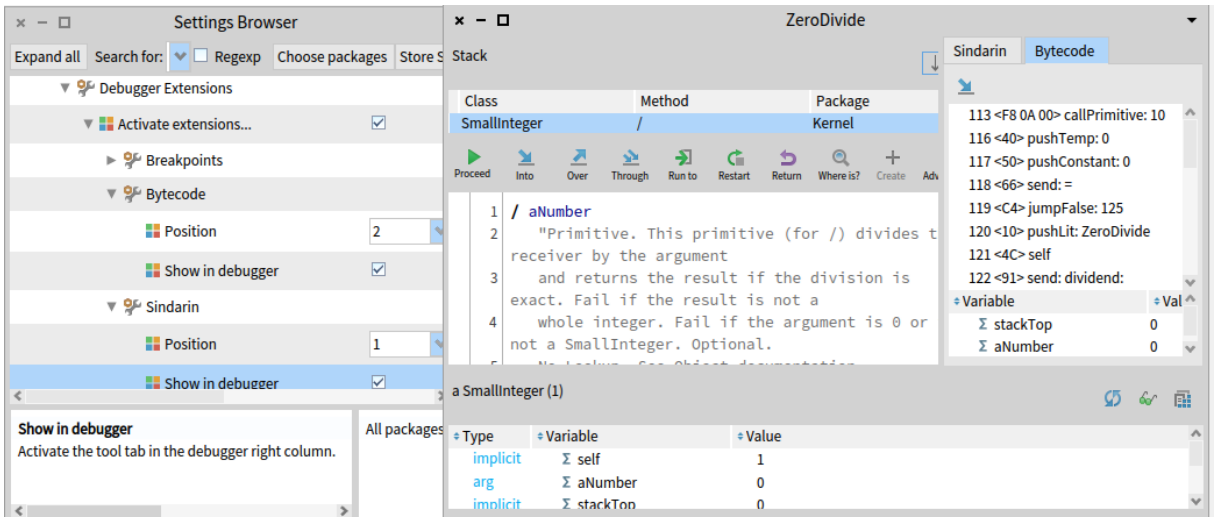

Figure 2: Vue du debugger avec au moins une extension activée

Analyse du code du debugger. J'ai utilisé *ClassBlueprint* [1], un outil faisant l'objet de la thèse d'une doctorante de l'équipe. *ClassBlueprint* est un outil aidant à la compréhension de code car il permet, entre autres, d'accéder au code source d'une classe entière (ce qu'il n'est pas possible de faire dans Pharo sans naviguer toutes les méthodes) ainsi que de visualiser grâce à des diagrammes : les dépendances entre méthodes/classes/packages, le code mort, le nombre de lignes de codes, etc. Cet outil donne aussi des métriques comme le donnent un outil comme *Sonarqube* [9], par exemple.

J'ai analysé plusieurs packages qui contiennent le code relatif au debugger afin de le comprendre ; notamment le package contenant le code des entités principales du debugger, dont l'interface graphique problématique qui contient des parties de l'implémentation de l'API du debugger.

J'ai pu identifier 3 classes constituant le coeur du debugger, ainsi que le rôle qu'elles sont censées avoir et le rôle qu'elles ont réellement :

- `DebugSession` est la classe implémentantant l'interface de contrôle du modèle du debugger. Entre autres, toutes les opérations classiques de stepping sont définies dans cette classe mais on y trouve aussi la pile de contextes du programme en train d'être débuggué.
- `StDebuggerActionModel`, que l'on appellera **action model** par la suite. Cette entité a pour rôle de communiquer avec la debug session. Notamment, elle devrait être un modèle intermédiaire entre l'IHM du debugger et le modèle du debugger. Elle est capable de dire, selon le contexte courant, quelles actions/commandes peuvent être exécutées sur ce contexte.
- `StDebugger`, que l'on appellera **debugger** par la suite. C'est le GUI du debugger. Y sont définis :
  - l'IHM du debugger dont certains éléments du GUI manipulent directement la session alors qu'elle devrait être accédée seulement dans l'action model.
  - un système d'écoute d'évènements, émis par sa debug session à chaque fois qu'une action est effectuée mais aussi par le système lorsqu'une nouvelle méthode a été compilée. Lorsqu'un évènement est reçu, l'UI se met à jour en conséquence en forçant la mise à jour de son action model alors que l'action model doit servir d'intermédiaire entre la session et l'UI.

J'ai identifié des responsabilités dans le code du GUI devant être migrées :

- La debug session ne doit pas être accédée directement depuis le GUI, mais seulement depuis l'API,
- l'action model ne doit pas dépendre de l'UI à chaque action, mais doit être automatique.

3 Vers un debugger extensible

Cette section traite des améliorations que nous avons apportées au coeur du debugger afin de le rendre plus extensible et maintenable, c'est-à-dire :

- La migration des accès à la debug session du GUI vers l'action model,
- l'automatisation des mises à jour de l'action model après une action de la debug session ou après qu'une nouvelle méthode soit créée,
- la transformation du layout du debugger, jusque-là statique en le rendant dynamique,
- l'affichage des extensions du debugger de manière dynamique, à leur (dés)activation.

3.1 Les accès à la debug session

L'action model est un objet qui sert d'intermédiaire entre la session et le debugger. Notamment, l'action model sert à effectuer des actions sur la session et accède à cette session. Pourtant, le debugger a aussi un accès direct à la session grâce à un objet qui s'appelle un `sessionHolder` qui sert à encapsuler la session, permettant entre autres de déclencher un évènement lorsque la session a été modifiée. Ainsi, le debugger accède à la session via ce `sessionHolder`, alors qu'elle est déjà contenue dans l'action model.

Ceci est problématique car le GUI ne devrait jamais manipuler directement la session. En effet, si le but de l'action model est d'effectuer des actions sur cette session, alors tous les accès à la session dans le GUI doivent passer par l'action model car l'action model doit pouvoir être réutilisable par d'autres debuggers et les accès à la session doivent être effectués, indépendamment du GUI utilisé.

J'ai identifié deux types d'accès à la session dans le debugger : les accès aux attributs de la session ainsi que les accès aux méthodes de la session.

Le debugger accédait aux attributs suivants de la session, via le `sessionHolder` :

- La pile de contextes,
- l'exception signalée si elle existe,
- le processus interrompu,
- le contexte interrompu.

J'ai donc refactoré ces accès en faisant en sorte qu'ils soient effectués via l'action model au lieu du `session holder`. Pour ce faire, j'ai créé accesseurs pour ces attributs, dans le debugger et dans l'action model. Par exemple, dans le debugger, pour accéder à la pile de contexte (`stack`) : au lieu d'écrire `self session stack`, nous écrivons `self stack` qui appelle `self debuggerActionModel stack`, l'action model déléguant lui-même l'accès à l'attribut `stack` vers sa session (voir code avant et après refactoring aux Figures 24 et 25).

Ces accès sont testés de manière à ce que tous les attributs accédés dans l'action model soient les mêmes que ceux accédés dans la session et tous les attributs accédés dans le debugger soient les mêmes que ceux accédés dans l'action model (voir Figure 26). Avoir des accesseurs dans chaque classe permet de rendre ces accès plus modulaires et surtout de respecter la loi de Déméter qui affirme qu'un objet devrait faire aussi peu d'hypothèses que possible à propos de la structure de quoi que ce soit d'autre, y compris ses propres sous-composants. En effet, avoir directement dans le debugger : `self debuggerActionModel session stack` est une mauvaise pratique car cette expression dépend fortement de la structure de l'action model et de la debug session. De plus, cela surcharge le code et le rend moins lisible.

En ce qui concerne les accès directs aux méthodes de la session dans le debugger, ceux-ci ont été corrigés de la même manière. Les méthodes concernées sont les suivantes :

- Une méthode pour recompiler une méthode dans un contexte,
- une méthode pour obtenir quelle sera la prochaine instruction exécutée,
- une méthode pour récupérer la pile de contexte avec un nombre maximum de contextes.

De manière similaire aux attributs, la solution à ce problème était de définir des methodes dans le debugger et dans l'action model déléguant les accès à ces méthodes pour les mêmes raisons. Ainsi, pour chacune de ses méthodes, le debugger définit une méthode du même nom qui délègue le message à son action model qui lui-même délègue à sa debug session (voir code avant et après refactoring, Figures 27 et 28).

Ces méthodes sont testées unitairement. Par exemple, pour tester la méthode permettant de recompiler une méthode dans un contexte, alors nous vérifions que tous les contextes qui sont au-dessus de ce contexte soient enlevées de la pile, car l'exécution redémarre depuis ce contexte (voir Figure 29). Dans cet exemple, avant l'exécution du test, l'action model manipulé a pour contexte courant une méthode qui appelle la méthode `squared`.

3.2 La gestion d'évènements

Nous avons identifié le fait que le debugger provoque la mise à jour de son action model lorsque des évènements sont envoyés par la debug session ou par le système. Un action model doit pouvoir exister par lui-même sans GUI alors qu'un GUI ne peut exister sans action model. Actuellement, l'action model ne peut pas être utilisé sans GUI car sa mise à jour dépend de son GUI. Recoder un autre GUI nécessitera de recoder dans le nouveau GUI la mise à jour de l'action model. Pour corriger le problème, l'action model doit être abonné à la session et au système pour ensuite mettre à jour son UI.

Les problèmes du système d'évènements

Cette section explique les problèmes que nous avons identifiés dans le système d'abonnements du debugger.

Plusieurs méthodes de mise à jour avaient le même code source. La debug session déclenche un évènement différent selon l'action effectuée sur cette session. Pour chaque évènement, une méthode différente était appelée sur l'UI pour se mettre à jour. Toutes ces méthodes n'étaient en réalité que des *hooks*, des points d'accroche pour permettre des traitements différents à des évènements différents. Les hooks sont une bonne pratique de programmation puisque des traitements différents pour chaque évènement offre plus de possibilité. Dans le debugger, ils sont mal utilisés car les hooks appelaient tous la même méthode de mise à jour : `updateStep`. Cette méthode mettait à jour le GUI et forçait la mise à jour de l'action model (voir Figure 30).

Pour maîtriser quand faire la mise à jour, l'utilisateur devait manuellement désabonner puis réabonner l'UI à la session. Un autre défaut du système d'évènement actuel est que certaines commandes des extensions du debugger forçaient le désabonnement de l'UI à la session avant d'effectuer plusieurs actions avant de réabonner l'UI à la session. Ceci est une manière de ne faire qu'une seule mise à jour de l'UI après plusieurs actions plutôt que plusieurs petites mises à jours, qui sont indésirables si nous n'avons pas besoin des états intermédiaires successifs du debugger. Par exemple, la commande permettant d'exécuter le code jusqu'à la prochaine création d'instance utilise ce mécanisme. Ceci peut être cause d'erreurs car ce n'est plus le debugger qui gère ses abonnements mais l'utilisateur lui-même.

La résolution des problèmes du système d'évènements

Cette section explique comment nous avons résolu les problèmes décrits dans la section précédente.

Dissociation des mises à jour de l'UI et de l'action model. Pour pallier ces problèmes, nous avons migré le système d'évènement de l'UI vers l'action model en supprimant les hooks inutilisés et en utilisant un système d'**update guard**, un système d'abonnement effectuant des mises à jour uniquement si une variable booléenne `guard` est à `false` sinon la mise à jour sera effectuée lorsque ce booléen repassera à false. L'action model est maintenant abonné aux mêmes évènements de la session auxquels était abonné le debugger. Lorsque l'action model reçoit un de ses évènements, il exécute la méthode `updateIfAble` qui appelle la méthode `update` s'il en est capable, c'est-à-dire si sa variable booléenne `guard` est à `false` (ce qui est le cas par défaut). La méthode `update` effectue alors la mise à jour de l'action model, que le GUI forçait dans sa méthode `updateStep`, et puis appelle la méthode `updateStep` du GUI qui lui est abonné. Désormais, la méthode `updateStep` du debugger met à jour uniquement le GUI (voir Figure 31). La mise à jour de l'action model est automatisée et il n'y a plus de points d'accroche inutilisés (figure 3).

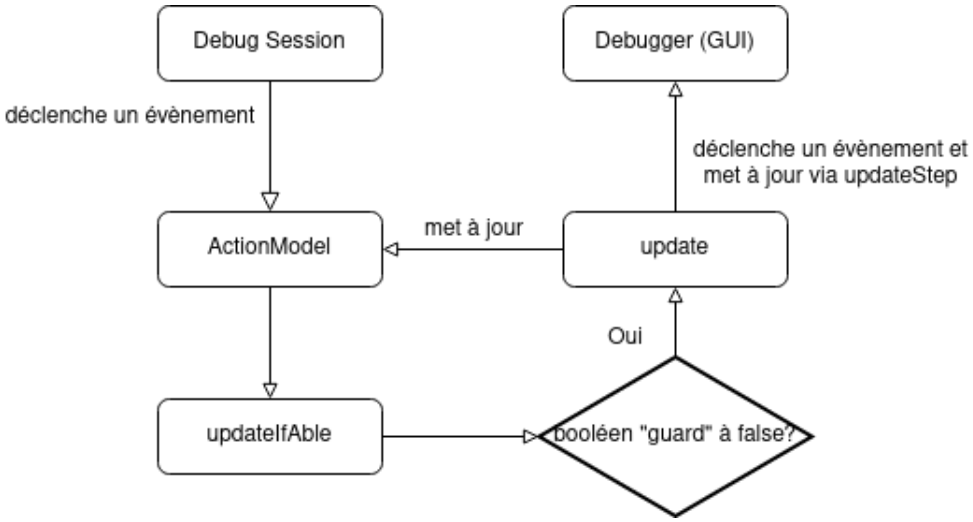

Figure 3: Schéma du système d'abonnement de l'action model et du debugger, après refactoring

L'utilisateur maîtrise quand faire la mise à jour, sans désabonnement/réabonnement. En remplacement du système de désabonnement et de réabonnement, l'action model propose une méthode `preventUpdateDuring:` qui prend une fonction anonyme en paramètre. Cette méthode met le booléen `guard` à `true` avant d'exécuter la fonction anonyme. Si cette dernière déclenche des évènements de la session, alors la méthode `updateIfAble` ne déclenchera pas la mise à jour de l'action model. Après l'exécution de la fonction anonyme, le booléen `guard` est repassé à `false`, et la méthode `update` est appelée pour mettre à jour l'action model et son éventuelle UI (voir Figures 4 et 32).

(Dés)abonnements lorsque la session est changée. Lorsque nous changeons la session grâce à son setter, ce qui arrive uniquement dans les tests, nous désabonnons l'action model de l'ancienne session avant de l'abonner à la nouvelle. Ainsi, cette migration d'évènements vers l'action model permet de supprimer complètement le `sessionHolder` du debugger : le debugger ne contient plus directement la debug session et l'action model joue pleinement le rôle d'intermédiaire entre le GUI et le modèle du debugger.

Migration de l'évènement système lorsqu'une nouvelle méthode a été créée. Enfin, la migration de l'évènement du système lorsqu'une méthode a été ajoutée s'est fait de la même manière, excepté que la mise à jour se fait indépendamment de la valeur de la variable booléenne `guard`. Ainsi, la méthode `updateAfterMethodAdded`, qui mettait à jour l'UI en forçant la mise à jour de l'action model, a

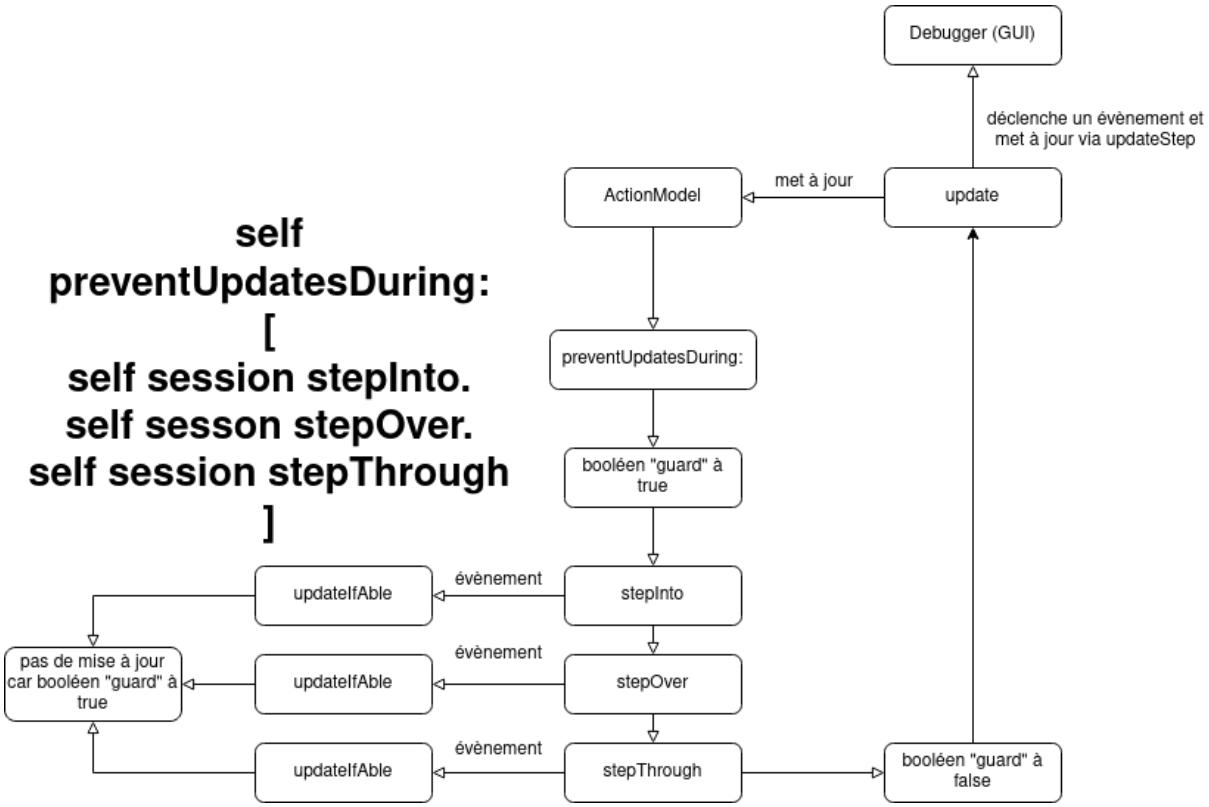

Figure 4: Schéma du fonctionnement du système empêchant les mises à jours intermédiaires, après refactoring

été migrée vers l'action model qui se met à jour tout seul avant de provoquer la mise à jour de l'UI qui lui est abonnée. Cet évènement met à jour l'action model pour savoir quelles commandes peuvent être exécutées dans le contexte courant. Pour cela, le processus interrompu de la session est manipulé pour récupérer le contexte interrompu. Or, lorsqu'une session est terminée, ce processus devient `nil`, ce qui cause des exceptions si l'action model est encore abonné à l'évènement système. Si l'utilisateur termine la session mais ne désabonne pas l'action model, l'action model ne sera pas récupéré par le *garbage collector* et il y aura une fuite mémoire (voir Figure 5). En nettoyant la session sans désabonner l'action model dans les tests, l'action model n'est pas récupéré par le `garbage collector` et plus on exécute les tests, plus on obtient l'erreur, car plus il y a d'action models qui ne sont pas nettoyés du système. Nous avons corrigé tous les tests utilisant des action model sans UI, en désabonnant l'action model dès qu'une session est nettoyée. Pour les action model avec UI, nous avons corrigé le problème en désabonnant l'action model lors de la fermeture de l'UI. Notre refactoring a introduit le problème car l'abonnement était au niveau de l'UI qui nettoyait la session et se désabonnait du système lorsque le debugger se fermait car fermer le debugger sous-entend qu'il ne sera plus utilisé. Pour un debugger sans UI, il n'y a d'autres choix que de le "fermer" programmatiquement. L'apparition de ce problème était donc prévisible.

3.3 Amélioration du *layout du debugger*

Le *layout* de l'interface graphique du debugger est défini en utilisant une bibliothèque graphique appelée `Spec`. Le *layout* du debugger a été conçu de manière statique car, au moment où il a été conçu, `Spec` ne proposait de définir le *layout* d'un élément graphique que de manière statique, côté classe. Cela a ses limitations car cela oblige à utiliser un nombre fixe de widgets, de la création d'un debugger jusqu'à sa destruction. Par exemple, si nous (dés)activons une extension du debugger, nous devons ouvrir un nouveau debugger pour la voir s'afficher (ou disparaître).

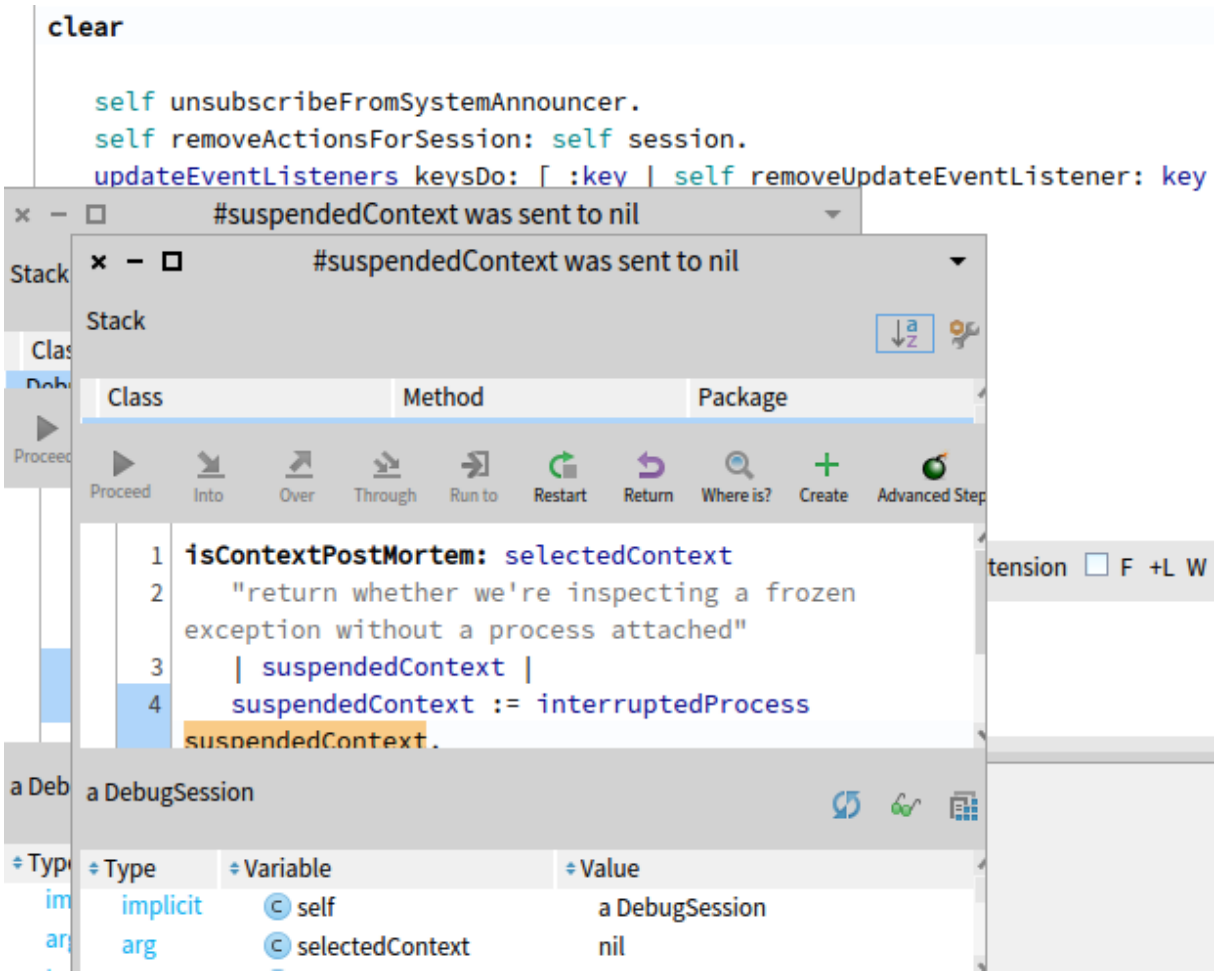

Figure 5: Illustration d'une fuite mémoire lorsque tous les tests de l'action model sont lancés sans désabonner chaque action model créé

Plus récemment, une version `Spec2` permet de définir le layout d'un élément graphique de manière dynamique, côté instance. Le debugger ayant un layout différent selon que ses extensions soient activées ou non, cela permettra à long terme d'afficher les extensions activées en direct, sans rouvrir de nouveau debugger.

Côté classe, pour construire le *layout* d'un debugger, la classe du debugger demande à une instance d'une autre classe de choisir un des deux *layouts* du debugger côté classe, c'est-à-dire celui sans extensions et celui avec extensions (Figure 6). Cet objet cherche dans les paramètres de Pharo si l'utilisateur autorise les extensions et si au moins une des extensions est activées, auquel cas le *layout* avec extensions est appliqué, sinon le *layout* sans extensions est appliqué.

Figure 6: Définition statique du layout du debugger

Pour migrer la définition du layout vers le côté instance, nous avons migré toutes les méthodes définissant les layouts des widgets du debugger (voir Figure 33). `Spec2` permettant de définir le layout par défaut de manière dynamique, nous avons fusionné les *layouts* avec et sans extensions en un seul *layout*, afin de factoriser du code. En effet, en comparant ces *layouts*, nous remarquons que la seule différence entre les deux réside dans le *layout* du container de la pile et du code. Sans extensions, ce container a un *layout* vertical simple, avec la pile au-dessus du code. Avec extensions, ce container a un *layout* horizontal avec la pile et le code à gauche, et les extensions à droite.

Ainsi, nous avons ajouté un attribut `stackAndCodeContainer` du côté instance afin de pouvoir unifier le *layout* du debugger dans son *layout* par défaut. Le *layout* dynamique de ce container est alors choisi selon que l'utilisateur a accepté l'utilisation d'au moins une extension ou non. Ce choix de *layout* est alors réalisé à l'initialisation du debugger. De cette manière, nous avons migré le *layout* du debugger du côté classe (statique) vers le côté instance (dynamique) en conservant le même comportement.

```
defaultLayout
    ^ SpPanedLayout newTopToBottom
      positionOfSlider: 65 percent;
      add: stackAndCodeContainer;
      add: #inspector;
      yourself
```

```
setStackAndCodeContainer

    stackAndCodeContainer := self class usesExtensions
                                ifTrue: [
                                self stackAndCodeWithExtensionsLayout ]
                                ifFalse: [ self stackAndCodeLayout ]
```

```
stackAndCodeWithExtensionsLayout

    ^ (SpPanedLayout newLeftToRight
        positionOfSlider: 65 percent;
        add: self stackAndCodeLayout;
        add: #extensionToolsNotebook;
        yourself)
```

Figure 7: Définition dynamique du layout du debugger

Un des avantages majeurs de ce refactoring est qu'il suffit qu'un évènement soit lancé lorsqu'une extension est (dés)activée afin de pouvoir mettre à jour automatiquement le layout du debugger.

4 *Sindarin*, une API pour scripts de debugging

Sindarin [2] est une extension du debugger de Pharo. Elle propose une API afin d'écrire des scripts de debugging ainsi qu'un ensemble de commandes avancées de debugging intégrées dans le debugger.

Cet outil n'a pas été maintenu donc certaines méthodes ne fonctionnent plus car le code n'a pas évolué depuis 3 ans, contrairement à Pharo. Nous avons donc corrigé certaines méthodes en les adaptant aux changements d'API qu'il a pu y avoir dans Pharo en 3 ans.

Certaines commandes ont été retirées du debugger car elles n'avaient pas le comportement attendu. Par exemple, la commande `skip`, utilisée pour sauter une instruction sans l'exécuter, saute parfois une instruction en trop lorsque nous voulons sauter une assignation. La commande `stepToReturn` est utilisée pour step jusqu'à ce que le contexte courant soit sur le point de *return*. Cependant, un contexte peut définir un **bloc** : une fonction anonyme qui, si elle *return*, *return* aussi dans le contexte qui l'a définie. `stepToReturn` ne permet pas de step jusqu'à un *return* dans un bloc.

Ces commandes manipulent des **bytecodes**, des instructions bas-niveau proches du code machine qui manipulent une pile de valeurs. Par exemple, un envoi de message utilise comme arguments les valeurs aux sommets de la pile, les retire de la pile et place le résultat au-dessus de la pile. Une assignation stocke la valeur au-dessus de la pile dans la variable souhaitée grâce à son index dans la pile, car les variables du contexte courant sont toujours placés en début de pile. Un bytecode peut être constitué de un ou plusieurs octets et une instruction Pharo peut être constituée de un ou plusieurs bytecodes.

Dans cette section, nous expliquerons les solutions apportées aux problèmes qu'impliquent ces deux commandes du debugger, ainsi que l'analyse en profondeur qui nous a mené à ces solutions.

4.1 Correction de la commande `skip`

La commande `skip` permet de sauter une instruction sans l'exécuter. Plus précisément, son implémentation est faite telle que la commande n'exécute pas l'instruction si c'est un message ou une assignation en plaçant `nil` au sommet de la pile de valeurs, sinon la commande `skip` ne fait rien. Pour sauter une instruction, le *program counter (= pc)*, qui est le nombre d'octets exécutés depuis le début du programme, est incrémenté de 1 dans le cas d'un message et de 2 dans le cas d'une assignation, en retirant au préalable tout ce qui doit être enlevé de la pile : receveur et arguments si c'est un message, ou la valeur à assigner si c'est une assignation (Figure 34).

Un message renvoie une valeur qui est placée au sommet de la pile car celle-ci peut être utilisée par d'autres instructions bytecodes (comme une assignation, un autre envoi de message, etc). Lorsque la commande `skip` passe un message alors une valeur de remplacement doit être mise sur la pile. Nous devons donc placer quelque chose sur la pile et `nil` est une valeur par défaut. En revanche, une assignation ne place une valeur de remplacement uniquement lorsque le résultat de l'assignation est utilisé par une autre instruction.

De même, sauter un octet pour sauter un message est correct car le bytecode d'envoi de message est unique et fait un octet, alors que sauter deux octets est arbitraire pour passer une assignation car il existe plusieurs bytecodes d'assignation de tailles différentes.

La commande skip ne doit pas systématiquement placer de valeur de remplacement d'une assignation La commande `skip` retire de la pile la valeur assignée puis incrémente le pc pour sauter l'assignation. Ensuite, elle place une valeur sur la pile. Si cette valeur n'est pas utilisée par la suite, elle restera inutilisée sur la pile. Dans l'exemple suivant, la variable `a` vaut 1 (Figure 8) puis en sautant l'assignation de 5 à `a`, `a` vaut toujours 1 mais la valeur `nil` a été placée sur la pile ; visible à droite sur les figures, indexée dont le sommet est la valeur la plus en bas, alors que cette valeur ne sera pas utilisée par la prochaine instruction (Figure 9).

Dans le cas d'une assignation dont le résultat est utilisée par une autre assignation, le résultat (qui est la valeur assignée) est placé sur la pile car ce résultat sera assigné dans la prochaine instruction. La différence entre les deux cas réside dans le bytecode utilisé pour stocker la valeur au-dessus de la pile : si le résultat de l'assignation est utilisé par la prochaine instruction alors le bytecode `storeIntoXXX` est

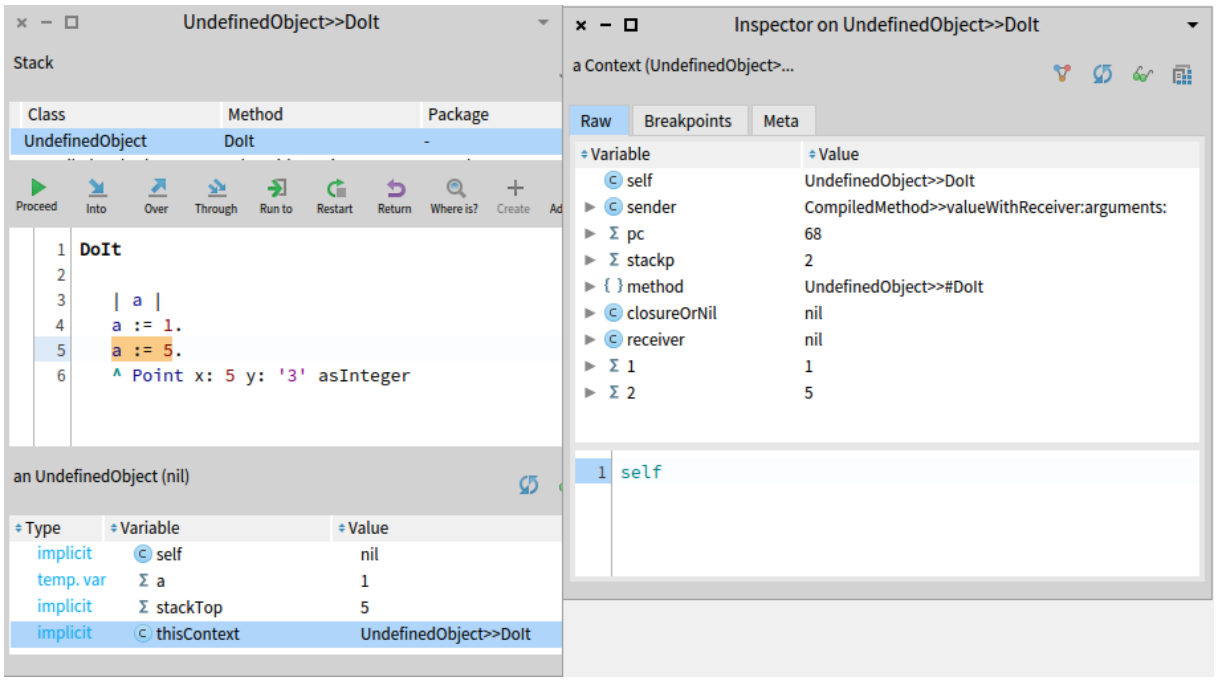

Figure 8: Etat de la pile avant de passer l'assignation de 5 à `a`

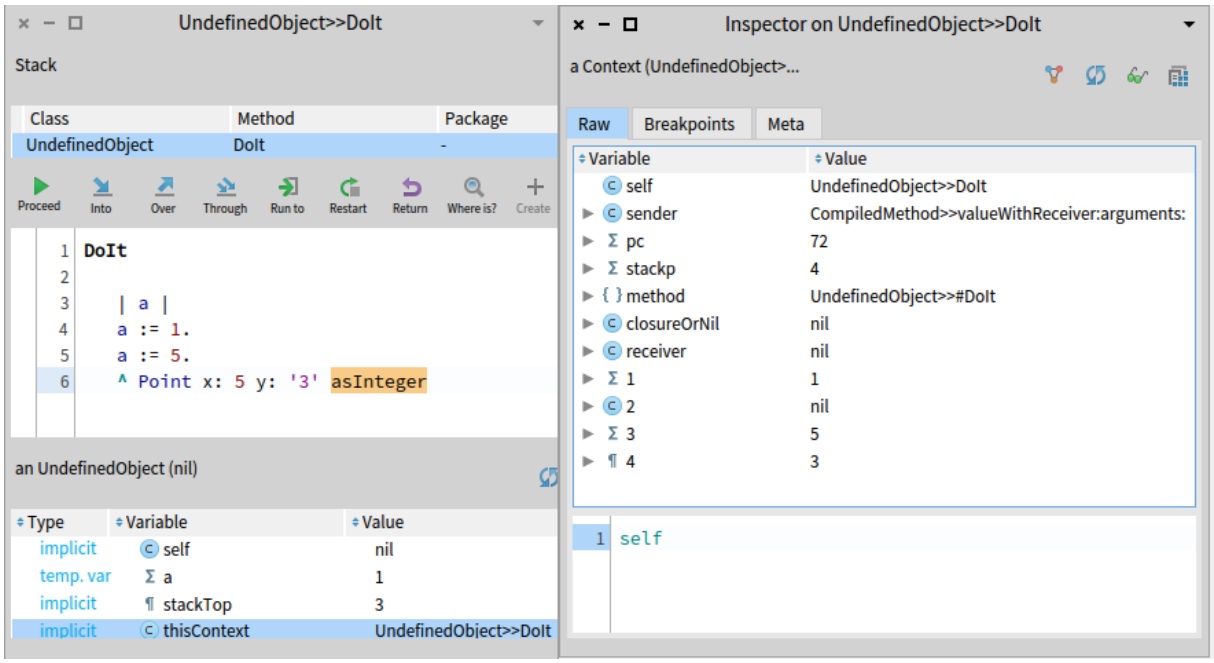

Figure 9: Etat de la pile après avoir passé l'assignation de 5 à `a`

utilisé sinon le bytecode `popIntoXXX`, qui retire la valeur assignée de la pile, est utilisé. Des variantes de ces bytecodes existent selon le type de la variable assignée. Les Figures 10 et 11 illustrent les bytecodes `storeIntoTemp` et `popIntoTemp` utilisés pour les assignations de variables temporaires. A droite, nous pouvons voir l'extension **Bytecode** du debugger qui montre la représentation du programme en train d'être debuggué sous la forme de bytecodes, le bytecode courant étant indiqué par une flèche bleue. Pour chaque bytecode du programme, cette extension indique : le pc correspondant, les octets correspondants et la description de ce que fait le bytecode. Cette extension permet aussi d'exécuter le programme bytecode par bytecode, ce qui est fait ici pour réaliser les assignations `b := 1` puis `a := (b := 1)`.

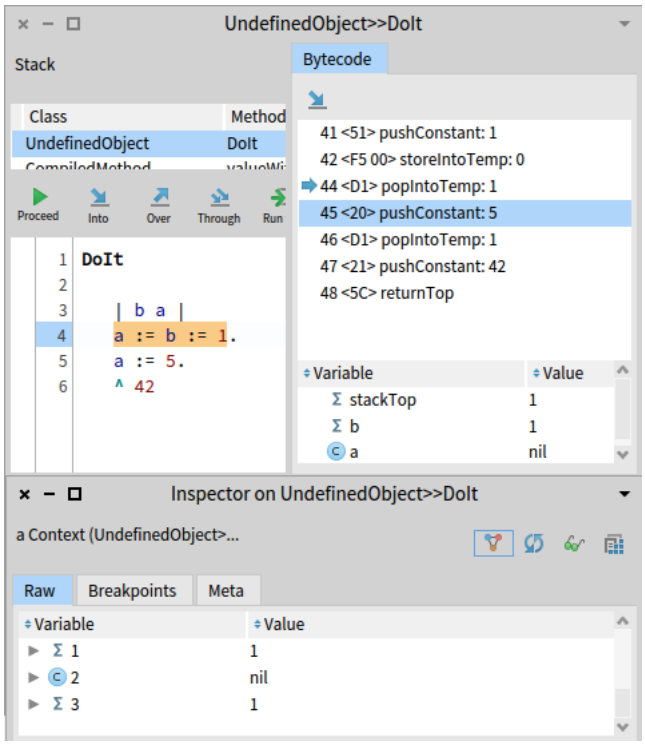

Figure 10: Etat de la pile après l'exécution du bytecode `storeIntoTemp`

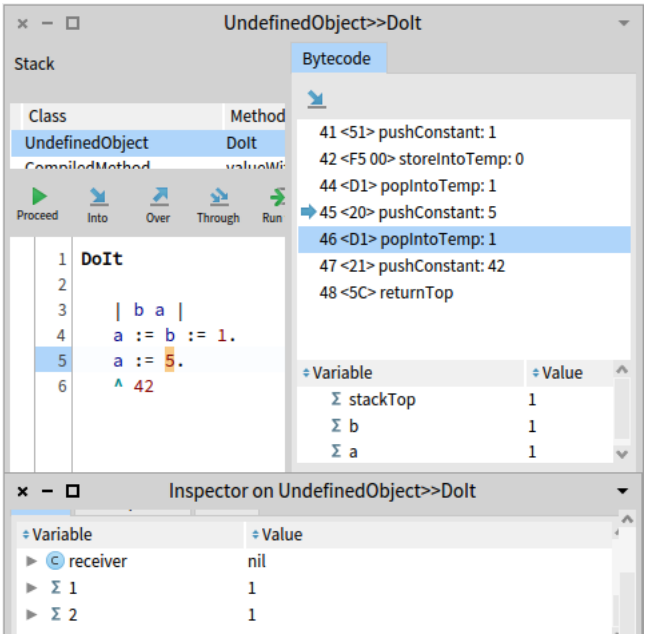

Figure 11: Etat de la pile après l'exécution du bytecode `popIntoTemp`

Afin que la commande `skip` place une valeur sur la pile uniquement si nécessaire dans le cas d'une assignation, nous avons réalisé deux versions de `skipAssignment` :

- une version qui réalise l'assignation mais avec une valeur de remplacement sur le haut de la pile et qui fait un step pour réaliser l'assignation avec cette valeur (Figure 12). Cette version est fonctionnelle car, le bon bytecode étant exécuté, on a la garantie que le résultat de l'assignation est placé

sur la pile seulement quand il le faut.

```
skipAssignmentNodeWith: replacementValue
   self context pop.
   "Pop the value to be assigned"
   "Push the replacement value on the context's value stack, to simulate
that the assignment happened and had value nil"
   self context push: replacementValue.
   self step.
   "Execute bytecodes the debugger usually executes without stopping the
execution (for example popping the return value of the just executed message
send if it is not used afterwards)"
   self debugSession
      stepToFirstInterestingBytecodeIn: self debugSession interruptedProcess
```

Figure 12: Version du skipAssignment avec valeur de remplacement

- une version qui passe complètement l'assignation en retirant la valeur à assigner de la pile et incrémentant le pc du nombre d'octets constituant le prochain bytecode afin de ne pas l'exécuter, tout en plaçant au sommet de la pile une valeur de remplacement pour l'assignation si le bytecode impliqué est `storeInto`. Cette version sera utilisée par la commande `skip` pour sauter une assignation et son implémentation fait l'objet des prochains paragraphes.

La commande `skip` passe un nombre arbitraire de bytecodes. Initialement, `skipAssignmentWith` sautait 2 octets pour sauter l'assignation. Cependant, nous pouvons voir sur la figure 11 que 2 octets sont plus que nécessaire pour le bytecode `popIntoTemp` qui fait un octet par exemple. Dans ce cas, des octets qui devraient être exécutés le sont.

A l'inverse, dans d'autres cas, certains octets qui ne devraient être exécutés le sont. C'est le cas, par exemple du bytecode `storeIntoTemp:inVectorAt:` qui fait 3 octets, les octets se trouvant entre les chevrons sur la Figure 13. Ce bytecode est utilisé dans le cas où l'assignation est faite dans une variable partagée entre plusieurs contextes, auquel cas sauter 2 octets ne saute pas l'assignation.

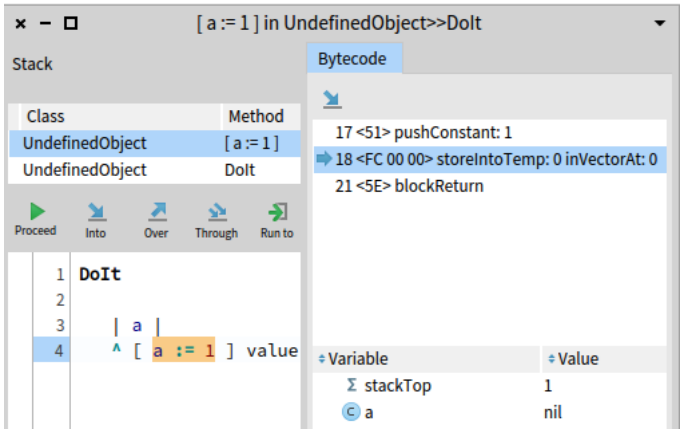

Figure 13: Assignation utilisant le bytecode `storeIntoTempInVectorAt`

Sauter deux octets est arbitraire et n'est pas une solution envisageable. Pour remédier à cela, il suffit de sauter un nombre d'octets égal au nombre d'octets constituant le prochain bytecode. Pour accéder au prochain bytecode, nous utilisons l'extension Bytecode du debugger, décrite page 18 . Une instance de

cet outil étant stocké dans le debugger *Sindarin* et connaissant le pc courant, nous pouvons connaître le bytecode courant pour le pc courant et donc le nombre d'octets du bytecode courant. En incrémentant le pc de ce nombre d'octets, nous obtenons alors une première version sautant complètement les assignations (Figure 14).

```
skipAssignmentNodeCompletely

   self context pop.
   "Pop the value to be assigned"
   "Increase the pc to go over the assignment"
   self context pc: (self context pc) + (self currentBytecode detect: [:each
| each offset = self context pc ]) bytes size.
   "Execute bytecodes the debugger usually executes without stopping the
execution (for example popping the return value of the just executed message
send if it is not used afterwards)"
   self debugSession stepToFirstInterestingBytecodeIn:
      self debugSession interruptedProcess
skip
   "If it is a message send or assignment, skips the execution of the
current instruction, and puts nil on the execution stack."
   self node isAssignment
      ifTrue: [ ^ self skipAssignmentNodeCompletely ].
   self skipWith: nil
```

Figure 14: Première version de `skipAssignment` passant complètement les assignations

Pour que cette version de `skipAssignment` saute correctement les assignations, il ne reste plus qu'à placer une valeur de remplacement sur la pile lorsque le bytecode impliqué est `storeInto`.

La commande skip ne place pas une valeur de remplacement pour les assignations lorsque cela est nécessaire. Pour l'instant, notre version de la méthode `skipAssignmentNodeCompletely` retire systématiquement de la pile la valeur qui allait être assignée avant de passer l'assignation. Nous devons maintenant placer une valeur de remplacement sur la pile après avoir retiré la valeur assignée lorsque le bytecode impliqué est un `storeInto`. Nous devons donc savoir comment identifier si le bytecode courant est de la famille `storeInto` et quelle valeur placer au-dessus de la pile comme résultat de l'assignation dans ce cas.

Tout bytecode a un identifiant qui correspond à son premier octet dans son tableau d'octets. Ainsi, en lisant le code source de l'interpréteur, nous pouvons voir que pour chaque bytecode, son identifiant est lu pour savoir quelle action effectuer (voir figure 35). Les noms de ces actions étant assez explicites, nous avons identifié quels identifiants correspondent aux bytecodes `storeInto`. Nous avons donc juste à comparer le premier octet du bytecode courant avec ces identifiants afin de savoir si nous devons mettre une valeur de remplacement sur la pile.

Pour ce qui est de la valeur de remplacement, nous pourrions mettre nil, comme nous le faisons pour passer un message. Nous avons décidé de mettre la valeur actuelle de la variable qui allait être assignée car c'est la valeur de la variable assignée après avoir sauté l'assignation et cette valeur offre plus de possibilités que `nil`, car envoyer n'importe quel message à `nil` déclenchera une exception.

Nous arrivons alors à une version fonctionnelle, sautant complètement et correctement les assignations et prenant en compte tous les cas expliqués précédemment (Figure 15).

```
skipAssignmentNodeCompletely

   | currentBytecode |
   currentBytecode := self currentBytecode detect: [ :each |
                        each offset = self context pc ].

   "Pop the value that will be assigned"
   self context pop.

   "If the assignment is a store bytecode and not a pop bytecode, we push the current value of the variable
that was going to be assigned."
   (#( 243 244 245 252 ) includes: currentBytecode bytes first) ifTrue: [
      self context push:
         (self node variable variableValueInContext: self context) ].

   "Increase the pc to go over the assignment"
   self context pc: self context pc + currentBytecode bytes size.
   "Execute bytecodes the debugger usually executes without stopping the execution (for example popping the
return value of the just executed message send if it is not used afterwards)"
   self debugSession stepToFirstInterestingBytecodeIn:
      self debugSession interruptedProcess
```

Figure 15: Version finale du skipAssignment passant complètement les assignations

Possibles améliorations. Ainsi, nous avons développé une commande `skip` qui a le comportement attendu. Nous pourrions essayer de l'améliorer car cette commande a été conçue afin de ne pouvoir sauter qu'un message ou une assignation mais nous pourrions réfléchir s'il ne serait pas pertinent de donner la possibilité de sauter d'autres instructions telles que des `return` ou la création de fonctions anonymes. Pour l'instant, si nous voulons sauter des instructions jusqu'à un point précis dans le code, nous ne pouvons pas si une instruction qui n'est ni un message ni une assignation se trouve entre le départ et le point d'arrivée car si c'est le cas, alors la commande skip ne fera rien lorsqu'elle arrivera sur une telle instruction, ce qui entraînera une boucle infinie si aucune gestion d'erreur n'est faite. Nous devons aussi nous demander si cela fait sens de sauter tout type d'instruction. Par exemple, quel est le sens de sauter la création d'une fonction anonyme ?

4.2 Correction de la commande `stepToReturn`

La commande `stepToreturn` permet de continuer l'exécution jusqu'à ce que le contexte lexical courant, c'est-à-dire le contexte de la méthode et le contexte de tous les blocs créés dans cette méthode, soit sur le point de `return` ou jusqu'à ce qu'une exception soit signalée. En Pharo, un bloc créé dans un contexte A constitue un contexte B à part entière, mais fait partie du même contexte lexical que le contexte A, car le bloc en question a été créé dans le contexte A. Des tests déjà écrits, spécifiaient le cahier des charges de la commande.

En Pharo, un bloc est une fonction anonyme, sauf que si un bloc `return`, cela `return` aussi dans tous les contextes constituant le même contexte lexical que le bloc. La commande `stepToReturn` doit donc permettre de s'arrêter aux `returns` non-locaux, dans les blocs par exemple (voir Figure 16). Dans cet exemple, comme le bloc est évalué avant que le contexte courant `return` et comme ce bloc `return`, `stepToReturn` doit step jusqu'au `return` dans le bloc.

La commande doit aussi s'arrêter lorsque certaines exceptions sont signalées mais pas toutes. Par exemple, la commande ne doit pas s'arrêter sur les points d'arrêt. La première version de cette commande respectait cette exigence (voir Figure 17).

La commande `stepToReturn` ne step pas jusqu'aux `return` dans les blocs. L'implémentation de la commande `stepToReturn` exécutait l'opération `stepOver`, qui exécute l'instruction sans rentrer dans le sous-contexte, tant que le contexte courant n'était pas sur le point de `return` ou tant qu'aucune exception n'a été signalée (Figure 18).

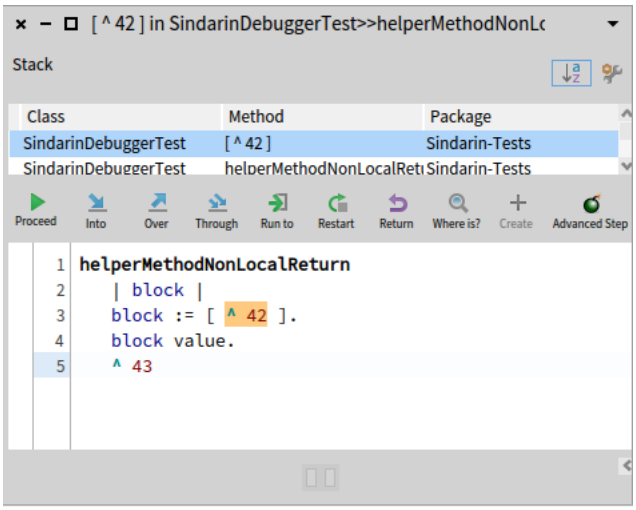

Figure 16: Etat d'un contexte après avoir `stepToReturn` à l'entrée d'une méthode évaluant un bloc qui `return`

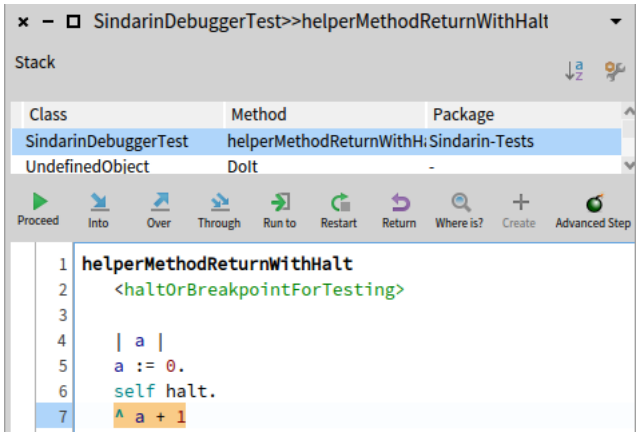

Figure 17: Etat d'un contexte après avoir `stepToReturn à l'entrée d'une méthode contenant un point d'arrêt`

```
stepToReturn

    [
    self context instructionStream willReturn or: [
        self hasSignalledUnhandledException ] ] whileFalse: [
        self debugSession stepOver ]
```

Figure 18: Première implémentation de la commande stepToReturn, utilisant stepOver

`stepOver` ne permet pas de rentrer dans des sous-contextes, comme les blocs par exemple. Dans le cas d'un `return` non-local (dans un bloc), cette version de `stepToReturn` remonte dans le contexte appelant, donc après le `return` du contexte lexical, ce qui n'est pas le comportement spécifié par son cahier des charges.

En conséquence, nous avons réalisé une deuxième implémentation utilisant une opération de stepping qui permet de rentrer dans les blocs : le `stepThrough`. D'après sa description, cette opération réalise

la même action qu'un `stepOver`, excepté qu'un `stepOver` ne rend le contrôle du debugger qu'après que la méthode invoquée ait `return` alors qu'un `stepThrough` redonne aussi le contrôle si un bloc du contexte lexical est sur le point d'être exécuté, ce que nous souhaitons, ou si un bloc dans la méthode invoquée est sur le point de `return`, ce que nous ne souhaitons pas car nous voulons uniquement rentrer dans les blocs du même contexte lexical.

Pour ne pas s'arrêter sur cette dernière condition, nous réalisons des `stepThrough` tant que le contexte de la méthode, qui est "le plus à l'extérieur" du contexte lexical dans lequel nous voulons `return`, ne s'apprête pas à `return` (Figure 19). Le contexte de la méthode n'est plus forcément le contexte courant car `stepThrough` peut rentrer dans un sous-contexte.

```
outerMostContextOf: aContext

   | currentContext oldContext |
   currentContext := aContext.
   oldContext := nil.
   [ currentContext ~= oldContext ] whileTrue: [
      oldContext := currentContext.
      currentContext := currentContext outerMostContext ].
   ^ currentContext
```

```
stepToReturn

   | oldContext |
   oldContext := self context.

   [
   (self context outerMostContext = oldContext and: [
       self context instructionStream willReturn ]) or: [
      self hasSignalledUnhandledException ] ] whileFalse: [
      self debugSession stepThrough: oldContext ]
```

Figure 19: Deuxième implémentation de la commande stepToReturn, utilisant stepThrough

Cette implémentation permet alors de step jusqu'aux `return` dans les blocs. Cependant, le `stepThrough` possède un effet indésirable non-mentionné dans la description de la commande : elle ne s'arrête pas sur les exceptions qui n'empêchent pas de poursuivre l'exécution du programme, comme la division par 0.

Les `stepThrough` ignorent certaines exceptions Désormais, la commande `stepToReturn` ne s'arrête plus sur certaines exceptions, comme l'exception signalée lors d'une division par 0 par exemple (figure 20).

```
helperMethodReturnWithException

   | a |
   a := 0.
   1 / 0.
   ^ a + 1
```

Figure 20: Utiliser stepToReturn ne s'arrête pas sur l'exception signalée lors de la division par 0

En explorant le code de `stepThrough:`, on voit qu'elle appelle la méthode `stepToHome:` qui réalise une gestion d'erreur. Elle capture toutes les exceptions pour reprendre l'exécution immédiatement après si l'exception capturée est *resumable*. *Resumable* est une propriété des exceptions indiquant qu'une exception est non-bloquante, c'est-à-dire que lorsqu'elles sont signalées, le programme s'arrête mais rien n'empêche de continuer le programme en ignorant l'exception. L'exception signalée lors de la division par 0 étant *resumable*, la commande `stepToReturn` l'ignore.

Notre logique permet de s'arrêter uniquement sur les `returns` du contexte lexical. Cependant, nous avons besoin d'une opération de stepping qui permet de rentrer dans des sous-contextes comme `stepThrough`, mais qui soit assez unitaire pour ne pas passer les exceptions contrairement au `stepThrough`. La dernière opération de stepping `stepInto` semble être adapté car elle permet de rentrer dans les contextes des méthodes appelées et s'arrête avant l'exécution de la première instruction du sous-contexte donc nous aurons le contrôle si une exception est signalée. Nous aboutissons alors à une troisième implémentation utilisant `stepInto`, utilisant toujours la même logique de la boucle

(Figure 21). Cependant, `stepInto` s'arrête sur toutes les exceptions, même les points d'arrêt, ce qui va à l'encontre du cahier des charges de la commande `stepToReturn`.

```
stepToReturn

    | oldContext |
    oldContext := self context.

    [
    (self context outerMostContext = oldContext and: [
        self context instructionStream willReturn ]) or: [
       self hasSignalledUnhandledException ] ] whileFalse: [
       self debugSession stepInto ]
```

Figure 21: Troisième implémentation de la commande stepToReturn, utilisant stepInto

Les `stepInto` s'arrêtent sur les Halt. L'opération `stepInto` s'arrête sur toutes les exceptions, même les points d'arrêt sur lesquels la commande `stepToReturn` ne doit pas s'arrêter. Nous avons essayé de comprendre pourquoi `stepOver` les ignore contrairement à `stepInto`. En regardant le code de l'interpréteur, nous pouvons voir que lorsqu'il interprète un `stepOver`, il ignore un ensemble d'exceptions. Les points d'arrêt faisant partie de ces exceptions, `stepOver` les ignore alors que `stepInto` s'arrête dessus.

Ainsi, `stepOver` permet de s'arrêter sur les exceptions tout en ignorant les points d'arrêt tandis que `stepInto` permet de rentrer dans n'importe quel contexte, dont les blocs du contexte lexical qui nous intéressent. Nous avons donc réalisé une implémentation de `stepToReturn` profitant des avantages qu'offrent `stepOver` et `stepInto` : tant que le contexte lexical n'a pas `return` ou tant qu'aucune exception n'a été signalée, alors nous réalisons un `stepOver`, sauf si le prochain message est susceptible d'exécuter un bloc du contexte lexical, auquel cas nous réalisons un `stepInto` (Figure 22). Si un bloc du contexte lexical va être exécuté lors d'un envoi de message, alors cela veut dire qu'il est passé en argument ou receveur de ce message. Notre implémentation réalise donc des `stepOver` sauf si un bloc du contexte lexical est receveur ou argument du message sur le point d'être exécuté.

Pour savoir s'il faut `stepInto`, il faut vérifier que la prochaine instruction est un message, qui a en argument ou en receveur un bloc ou une variable qui a pour valeur un bloc, auquel cas on récupère sa valeur plutôt que la variable. Il faut `stepInto` le message si un de ces blocs a été défini dans le contexte lexical de la méthode dans lequel nous voulons exécuter jusqu'au return. Pour vérifier qu'un bloc a été défini dans le contexte lexical, nous le passons en argument d'un visiteur d'AST (= Abstract Syntax Tree, représentant les instructions d'un programme sous forme d'arbre), qui, lorsqu'il visite un AST, cherche si l'AST du bloc correspond à un sous-arbre de la méthode de l'AST. L'avantage du design pattern **Visiteur** est qu'il permet de définir une opération à l'extérieur d'une classe, afin de ne pas surcharger la classe et de rendre l'opération réutilisable. Le visiteur visite un objet, ici un noeud, et lui demande de l'accepter. Selon le type de noeud visité, le noeud appelle une méthode sur le visiteur qui réalise un traitement spécifique au type de noeud visité. Pharo propose une implémentation de visiteur d'AST qui parcourt récursivement tout AST, en appliquant aucun traitement. Nous avons alors juste à comparer l'AST visité récursivement avec l'AST du bloc cherché, donné en paramètre, afin de savoir si le bloc cherché est dans le contexte lexical de la méthode dont l'AST est visité (Figure 23).

Désormais, la commande `stepToReturn` respecte sa spécification, en utilisant `stepOver` par défaut tant que le contexte lexical courant n'est pas sur le point de `return`, permettant ainsi de s'arrêter sur les exceptions tout en ignorant les points d'arrêt, tout en utilisant `stepInto` lorsque nécessaire pour rentrer dans les blocs.

```
stepToReturn

    | oldContext methodAST |
    oldContext := self outerMostContextOf: self context.
    methodAST := self context method ast.

    [
    ((self outerMostContextOf: self context) = oldContext and: [
        self context instructionStream willReturn ]) or: [
       self hasSignalledUnhandledException ] ] whileFalse: [
       (self shouldStepIntoInMethod: methodAST)
          ifTrue: [ self debugSession stepInto ]
          ifFalse: [ self debugSession stepOver ] ]
```

```
13 [1]                                  × stepping - steps ☐ extension ☑ F +L
shouldStepIntoInMethod: aRBMethodNode

    | messageNode childrenOfMessageNode |
    messageNode := self node.
    messageNode isMessage ifFalse: [ ^ false ].
    childrenOfMessageNode := messageNode children.
    childrenOfMessageNode := childrenOfMessageNode
                                select: [ :child |
                                   child isBlock or: [
                                      child isVariable and: [
                                         (child variableValueInContext:
                                             self context) isBlock ] ] ]
                                thenCollect: [ :child |
                                   child isVariable ifTrue: [
                                      (child variableValueInContext:
                                          self context) startpcOrOuterCode
ast ] ].
    ^ childrenOfMessageNode anySatisfy: [ :child |
         (RBBlockDefinitionSearchingVisitor newToSearch: child) visitNode:
            aRBMethodNode ]
```

Figure 22: Quatrième implémentation de la commande stepToReturn, utilisant stepOver et stepInto

```
blockToSearch: aBlockNode

    blockToSearch := aBlockNode.
    isBlockFound := false
```

```
visitNode: aNode

    super visitNode: aNode.
    aNode = blockToSearch ifTrue: [ isBlockFound := true ].
    ^ isBlockFound
```

Figure 23: Implémentation simple du visiteur d'AST, comparant pour chaque sous-arbre à l'AST du bloc à trouver

5 Intégration dans l'équipe

Ce stage m'a aussi permis d'intégrer une équipe d'ingénieurs et de chercheurs, ce que je n'avais jamais fait auparavant, de plusieurs manières :

- Tous mes travaux ont subi un processus de validation,
- j'ai participé à l'amélioration des outils utilisés et développés par l'équipe de debugging comme *Chest*, un gestionnaire de coffre-forts pour objets,
- j'ai participé à la rédaction d'un petit papier de recherche,
- j'ai participé aux évènements organisés par l'équipe, comme les sprints tous les derniers vendredis du mois où nous corrigeons des bugs toutes la journée en pair-programming.

Cette section traite de ces activités qui m'ont permis de m'intégrer à l'équipe.

Validation de mes contributions. Participant au développement d'outils open-source utilisés par des milliers de personnes, mon travail doit subir un processus de validation afin d'en évaluer la qualité. Ainsi, dès que je trouvais un problème dans le debugger ou dans *Sindarin*, après en avoir parlé aux membres de l'équipe de debugging, j'enregistrais le problème sur le dépôt Github correspondant en créant un rapport de bug qui détaillait le problème, illustré par du code, et qui expliquait comment le résoudre. Après avoir résolu le problème sur mon dépôt *forké*, j'effectuais alors une *pull request (= PR)* afin que mes changements soient intégrés à Pharo.

Pour que mes changements soient acceptés, il faut que :

- Les changements ne soient pas en conflit avec la branche `master` du dépôt principal,
- le code déjà existant fusionné avec mes changements passe l'intégration continue (tous les tests doivent être verts),
- mes changements soient passés en revue et approuvés par un ou plusieurs problèmes de l'équipe, selon la criticité des changements.

Ainsi, toutes mes contributions cités précédemment ont été intégrées dans Pharo, sauf mon *refactoring* des évènements du debugger, qui à ce jour est encore passée en revue car elle est sensible. Au total, j'ai réalisé 11 PR, dont 8 ont été intégrées et dont 3 sont encore en review, et j'ai rédigé 14 rapports de bugs dont 8 ont été résolues par mes 8 PR.

Outillage expérimental du debugger : Chest. J'ai aussi pu participer à l'amélioration des outils utilisés et/développés par l'équipe de debugging. J'ai participé à l'amélioration de *Chest*, un outil de gestionnaire de coffres-forts qui permet de stocker des objets, afin de pouvoir y accéder par la suite depuis n'importe quel outil dans Pharo.

Chest n'a pas été maintenu pendant 3 ans donc j'ai corrigé du code qui utilisait l'ancienne API de Pharo afin que l'outil refonctionne à nouveau. Ensuite, j'ai repensé l'outil dans son entièreté afin de le rendre plus facile d'utilisation. Par exemple, pour accéder à un coffre-fort (resp. un objet dans un coffre-fort), l'utilisateur devait connaître l'indice du coffre dans la liste des coffres (resp. l'indice de l'objet dans le coffre), ce qui ne se retient pas facilement pour un humain, d'autant plus que les indices changent lorsqu'un coffre (resp. un objet dans un coffre) est supprimé. J'ai donc refactoré *Chest* afin de pouvoir accéder aux coffres et aux objets dans les coffres grâce à un nom, qui est plus parlant pour un utilisateur.

J'ai aussi repensé l'IHM de l'outil qui donnait trop peu de place aux coffres et aux objets dans les coffres eux-mêmes comparé à l'inspecteur qui envahissait deux tiers de la place (voir Figure 36). J'ai donc amélioré l'IHM pour donner plus de place aux coffres et aux objets dans le coffre sélectionné, tout en agrémentant ceux-ci d'un menu contextuel afin de pouvoir les renommer ou les charger dans le *playground* afin de pouvoir les manipuler (voir Figure 37).

Afin d'élargir le potentiel de *Chest*, je l'ai intégré au debugger en tant que plugin, tout en donnant la possibilité de pouvoir stocker un objet dans un coffre depuis un debugger. Inversement, il est désormais possible de charger n'importe quel objet d'un coffre dans le debugger (voir Figure 38). Ces fonctionnalités sont expérimentales mais un ingénieur de l'équipe les trouvent intéressantes car il y a déjà eu des demandes pour avoir ces fonctionnalités dans Pharo.

Introduction au domaine de la recherche. J'ai intégré une équipe de 3 chercheurs afin de rédiger un petit papier de recherche pour le workshop *International Workshop on Smalltalk Technologies (IWST)*, à propos d'un prototype, fait par un doctorant de l'équipe, de debugger objet-centrique permettant de naviguer une exécution en avant et en arrière dans le temps. Le but du papier est de montrer comment les debuggers objet-centriques à voyager dans le temps peuvent être des outils de debugging puissants, permettant d'identifier des objets particuliers.

J'ai coécrit avec le créateur du prototype un exemple d'utilisation permettant d'introduire au fur et à mesure certaines fonctionnalités de ce prototype notamment les requêtes afin de récupérer diverses informations liés à un ou plusieurs objets spécifiques comme les messages envoyés ou reçus par cet

objet et le moment de sa création. Chaque résultat d'une requête permet de voyager dans le temps pour retourner au moment de l'exécution où l'objet a envoyé/reçu le message ou au moment où l'objet a été créé. Avant de rédiger, j'ai manipulé l'outil sur des exemples afin de comprendre comment il fonctionne.

Ensuite, mon tuteur et le créateur de *Seeker* [10], un time-travelling debugger fait par RMOD et le prototype de debugger faisant l'objet du papier, ont écrit le reste du papier sur l'implémentation du prototype ainsi que sur les motivations et les difficultés que pourraient résoudre les debuggers objet-centriques à voyager dans le temps. Puis, l'équipe a terminé ce papier par une phase de relecture afin de reformuler certaines phrases et de supprimer des passages inutiles à la compréhension.

Je participerai à la présentation de ce papier de recherche en tant qu'étudiant volontaire à la conférence internationale *ESUG*, qui réunit chaque année la communauté Smalltalk pour donner la possibilité de mettre en avant ses projets devant la communauté.

Aussi, j'ai rencontré une autre équipe de recherche qui est située à l'université de Bruxelles et qui est associée à RMOD. Dans ce cadre, j'ai pu assister à plusieurs présentations sur des projets de recherche variés sur le debugging comme le debugging à distance, le debugging d'applications concurrentes et l'implémentation de points d'arrêt objets-centriques. Le but principal de ce déplacement était d'assister à la soutenance de thèse publique d'un doctorant de Bruxelles car sa thèse était co-encadrée par un membre de l'équipe RMOD.

Participation aux évènements de l'équipe. L'équipe organise des *sprints* le dernier vendredi de chaque mois, où tous les membres de l'équipe pair-programment pendant toute la journée pour corriger le plus d'*issues* possibles dans Pharo. Cela est l'occasion de programmer avec des personnes qui ont souvent une vision des problèmes différente de la nôtre tout comme c'est l'occasion de travailler avec des personnes extérieures à l'équipe de debugging, comme les personnes développant la machine virtuelle (= VM) de Pharo par exemple.

Avec un membre de l'équipe de VM, nous avons nettoyé des classes qui n'étaient utilisés que dans des tests, en faisant en sorte que ces classes et leurs méthodes soient générés automatiquement avant l'exécution des tests puis supprimés après l'exécution des tests. Nettoyer des classes dans Pharo permet de naviguer plus rapidement les classes dans le navigateur de classes Pharo.

Avec deux membres de l'équipe de debugging et un membre de la VM, nous avons aussi corrigé certaines commandes du menu contextuel de la pile de contextes dans le debugger qui ne fonctionnaient plus à cause d'une évolution de l'API de *Spec*, tout en ajoutant des raccourcis claviers pour ces commandes.

6 Conclusion

Pour conclure, lors de ce stage, j'ai contribué à 11 *pull requests* dont 8 ont été intégrées dans Pharo et 3 en cours de review.

Afin de complètement séparer l'API du debugger de son GUI, nous avons complètement migré l'implémentation de l'API du debugger vers l'action model, qui sert d'intermédiaire entre le GUI et le modèle du debugger. Pour cela, tous les accès à la debug session depuis le GUI sont délégués à l'action model. L'état de l'action model est désormais mis à jour automatiquement après chaque action réalisée sur la debug session et après qu'une méthode ait été ajoutée, ce qui le rend réutilisable par tout debugger. Désormais, chaque classe possède une responsabilité unique, ce qui les rends plus extensibles, maintenables et testables.

Nous avons corrigé des commandes complexes de debugging dans *Sindarin* en améliorant leur stabilité. En effectuant une analyse poussée des différents cas possibles, nous avons corrigé la commande `skip` qui passe désormais correctement les assignations, en passant le nombre exact d'octets et en plaçant de remplacement une valeur sur la pile selon le bytecode impliqué dans l'assignation. Cette commande, dont *Thales* a besoin, pourra être réintégrée dans Pharo. Nous avons aussi corrigé la commande `stepToReturn` en proposant une implémentation qui répond pleinement à son cahier des charges, ce qui a aussi nécessité une analyse approfondie.

Nous avons aussi amélioré *Chest* en améliorant son IHM afin que ses widgets occupent profite plus de la place offerte par la fenêtre. Nous avons rendu son utilisation plus intuitive pour les utilisateurs en donnant la possibilité de nommer les coffres et les objets. Nous avons aussi élargi son champs de possibilité en l'intégrant dans le debugger et en implémentant des commandes pour stocker des objets depuis n'importe quel code Pharo et inversement pour charger des objets d'un coffre dans n'importe quel code Pharo, et ces fonctionnalités ont déjà été demandées par des utilisateurs dans le debugger.

J'ai aussi participé à la rédaction d'un petit papier de recherche, m'introduisant ainsi à la recherche.

Ainsi, ce stage m'a appris, dans un premier temps les réels bénéfices de l'utilisation d'un debugger car je n'en avais utilisé qu'occasionnellement.

De plus, je sais désormais dans les grandes lignes comment sont implémentées les opérations basiques de stepping, tout comme j'ai été introduit à d'autres principes de debugging avancés comme le debugging à distance grâce à des présentations auxquelles j'ai assistées, lors d'un workshop de debugging et lors d'un déplacement à l'université de Bruxelles pour rencontrer une autre équipe de chercheurs.

Ensuite, j'ai appris comment était représenté un programme sous forme de bytecodes, ce qui m'a permis de me rendre compte qu'une instruction simple comme une assignation peut cacher plusieurs bytecodes différents. Cela implique plusieurs conséquences à prendre en considération qui sont invisibles au premier abord.

J'ai aussi appris comment participer au développement d'un projet open-source, en apprenant en profondeur le fonctionnement de *Git*, que l'on apprend que trop superficiellement en licence.

Enfin, j'ai découvert le domaine de la recherche. J'ai eu un aperçu de comment travaille un doctorant et je sais désormais qu'une thèse ne sert pas qu'à devenir chercheur, contrairement à ce que je pensais.

7 Bilan

Ce stage m'a permis d'avoir une première expérience professionnelle dans un domaine complexe.

J'ai découvert que le debugging est un domaine dans lequel il reste beaucoup à explorer, que ce soit du point de vue de l'ingénierie ou de la recherche. Ce domaine m'intéressant pour sa complexité et son efficacité à trouver et corriger des bugs, je vais certainement approfondir mes connaissances dans ce domaine par le biais d'un master en alternance.

Ce stage m'a aussi permis de découvrir ce que je n'apprécie pas faire, comme concevoir des interfaces graphiques car leur appréciation repose sur les goûts des développeurs, ce qui est subjectif.

Ce stage est en adéquation avec ma future formation, le master Informatique parcours Génie Logiciel, car il m'a permis de participer au développement d'une infrastructure solide pour rendre un outil, ici le debugger, extensible et maintenable. Il est aussi en adéquation avec ma volonté de bien d'avoir des logiciels avec moins de bugs, car proposer des outils de debugging puissant permettra aux développeurs de trouver et corriger plus efficacement les bugs.

Bibliographie

[1] Nour Djihan. *ClassBlueprint V2*. https://github.com/NourDjihan/ClassBlueprint. 2021.

[2] Thomas Dupriez et al. "Sindarin: A Versatile Scripting API for the Pharo Debugger". In: *International Symposium on Dynamic Languages (DSL'19)*. ACM, 2019, pp. 67–79. DOI: 10.1145/3359619.

[3] *FDN - Correction (V2.1)*. URL: https://app.mural.co/t/yvain4858/m/yvain4858/1586333351975/f5553779156629f8bc14e1866ca394591d5d4126?sender=88ee545d-e705-4ce6-832e-1a17b53744ab (visited on 06/10/2022).

[4] *Inria & son écosystème*. URL: https://www.inria.fr/fr/innovation-numerique-ecosysteme (visited on 06/10/2022).

[5] *Numérique, quel équilibre entre avancées majeures et impact environnemental ?* URL: https://www.inria.fr/fr/podcast-mooc-impact-environnement-numerique (visited on 06/10/2022).

[6] *OCRE: Lowering the cost of debugging with the first generation of object-centric debuggers – OCRE*. URL: https://anr.fr/Project-ANR-21-CE25-0004 (visited on 06/10/2022).

[7] *PowerAPI*. URL: http://www.powerapi.org/ (visited on 06/10/2022).

[8] *RMoD*. URL: https://rmod.gitlabpages.inria.fr/website/ (visited on 06/10/2022).

[9] SonarSource. *Sonarqube*. https://github.com/SonarSource/sonarqube. 2007.

[10] Maximilian Willembrinck. *Seeker OC*. https://github.com/Willembrinck/SeekerOC-2021/. 2021.

8 Annexes

Table des figures

```
newDebuggerContext

    ^ self class debuggerContextClass new
        exception: self session exception;
        yourself
```

Figure 24: Exemple d'accès aux attributs de la debug session dans le debugger, avant refactoring

```
newDebuggerContext

    ^ self class debuggerContextClass new
        exception: self exception;
        yourself
```

```
exception
    ^ self debuggerActionModel exception
```

```
exception
    ^ self session exception
```

Figure 25: Exemple d'accès aux attributs de la debug session dans le debugger et dans l'action model, après refactoring

```
testExceptionProvidesSameExceptionAsTheActionModelOne

    debugger := self debuggerOn: session.
    self
        assert: debugger exception
        identicalTo: debugger debuggerActionModel exception
```

```
testExceptionProvidesSameExceptionAsTheSessionOne

    self
        assert: debugActionModel exception
        equals: debugActionModel session exception
```

Figure 26: Exemple de test d'accès aux attributs de la debug session dans le debugger, après refactoring

```
recompileMethodTo: aString inContext: aContext notifying: aNotifyer

    aContext ifNil: [ ^ self ].
    self session
        recompileMethodTo: aString
        inContext: aContext
        notifying: aNotifyer
```

Figure 27: Exemple d'accès aux méthodes de la debug session dans le debugger, avant refactoring

```
recompileMethodTo: aString inContext: aContext notifying: aNotifyer

    ^ self debuggerActionModel
        recompileMethodTo: aString
        inContext: aContext
        notifying: aNotifyer
```

```
recompileMethodTo: aString inContext: aContext notifying: aNotifyer

    aContext ifNil: [ ^ self ].
    self session
        recompileMethodTo: aString
        inContext: aContext
        notifying: aNotifyer
```

Figure 28: Exemple d'accès aux méthodes de la debug session dans le debugger et dans l'action model, après refactoring

```
testRecompileMethodToInContextNotifyingUpdatesSourceCodeAndContext

    | oldStack contextChanged expectedNewStack rejectedFromOldStack |
    contextChanged := debugActionModel topContext.

    debugActionModel session stepIntoUntil: [ :currentContext |
        currentContext selector = #squared ].
    debugActionModel updateTopContext.
    oldStack := debugActionModel stack.
    rejectedFromOldStack := { debugActionModel topContext }.

    expectedNewStack := oldStack copyWithoutFirst.

    debugActionModel
        recompileMethodTo: contextChanged method sourceCode
        inContext: contextChanged
        notifying: nil.
    debugActionModel updateTopContext.

    self denyCollection: debugActionModel stack includesAny: rejectedFromOldStack.
    self assertCollection: debugActionModel stack hasSameElements: expectedNewStack.
    self assert: debugActionModel topContext identicalTo: contextChanged
```

Figure 29: Exemple de test d'un helper de l'action model appelant une méthode de la debug session après refactoring. Avant l'exécution du test, l'action model manipulé a pour contexte courant une méthode qui appelle la méthode `squared`

```
registerActionsForSession: aSession
    self flag: 'Rewrite it'.
    aSession
        ifNotNil: [ aSession
            when: #restart send: #updateRestart to: self;
            when: #resume send: #updateResume to: self;
            when: #stepInto send: #updateStepInto to: self;
            when: #stepOver send: #updateStepOver to: self;
            when: #stepThrough send: #updateStepThrough to: self;
            when: #contextChanged send: #updateContextChanged to: self
```

```
updateRestart
    self updateStep
```

```
StDebugger>>updateStep
Filter...   Filter...   instance side   updatePredica
All Packages  Scoped View | Flat  Hier.  Traits | Inst. side  Class side | Metho
? Comment   StDebugger   *updateStep
```

```
updateStep
    self flag: #DBG_UPDATE_OF_MODEL_SHOULD_BE_AUTOMATED.
    debuggerActionModel updateTopContext.
    debuggerActionModel updateContextPredicate.
    debuggerActionModel updateDebugSession.

    self updateStackFromSession: self session.
    self updateWindowTitle.
    self updateExtensionsFrom: self session.
    self updateToolbar.
    self flag: #DBG_INSPECTOR_UPDATE_BUG.
    inspector getRawInspectorPresenterOrNil ifNotNil: [ :p | p update ]
```

Figure 30: Code du système d'abonnement du debugger, avant refactoring

```
updateIfAble

    preventUpdate ifNil: [ ^ self ].
    preventUpdate ifFalse: [ self update ]
```

```
update

    self updateTopContext.
    self updateContextPredicate.
    self updateDebugSession.

    self raiseUpdateEvents
```

```
subscribeToActionModel

    self debuggerActionModel
        addUpdateEventListener: self
        whenUpdatedDo: [ self updateStep ]
```

```
updateStep

    self updateStackFromSession: self session.
    self updateWindowTitle.
    self updateExtensionsFrom: self session.
    self updateToolbar.
    self flag: #DBG_INSPECTOR_UPDATE_BUG.
    inspector getRawInspectorPresenterOrNil ifNotNil: [ :p | p update ]
```

Figure 31: Code du système d'abonnement de l'action model et du debugger, après refactoring

```
preventUpdatesDuring: aBlock

    "Sets the prevent update flag during the execution of a block. Restores
its value after the block   finishes"

    | oldEventFlag |
    oldEventFlag := preventUpdate.
    preventUpdate := true.
    [
    aBlock value.
    self update ] ensure: [ preventUpdate := oldEventFlag ]
```

Figure 32: Code du système permettant d'interdire les mises à jour intermédiaires dans l'action model, après refactoring

```
stackLayout
    ^ SpBoxLayout newTopToBottom
        add: #stackHeader
            expand: false
            fill: false
            padding: 5;
        add: #stackTable;
        yourself
```

/8 [1]

```
stackAndCodeLayout

    ^ SpPanedLayout newTopToBottom
        positionOfSlider: 30 percent;
        add: self stackLayout;
        add: self codeLayout;
        yourself
```

```
codeLayout

    ^ SpBoxLayout newTopToBottom
        add: #toolbar expand: false;
        add: #code;
        yourself
```

Figure 33: Méthodes définissant le *layout* du côté classe, s'appuyant sur des attributs côté instance.

```
skip
   "If it is a message send or assignment, skips the execution of the
current instruction, and puts nil on the execution stack."

   self skipWith: nil
```

SindarinDebugger>>skipWith:

```
skipWith: replacementValue
   "If it is a message-send or assignment, skips the execution of the
current instruction, and puts the replacementValue on the execution stack."

   "If the current node is a message send or assignment"

   (self node isMessage not
      and: [ self node isAssignment not ])
      ifTrue: [ ^ self ].
   self node isMessage
      ifTrue: [ ^ self skipMessageNodeWith: replacementValue ].
   self node isAssignment
      ifTrue: [ ^ self skipAssignmentNodeWith: replacementValue ]
```

SindarinDebugger>>skipMessageNodeWith:

```
skipMessageNodeWith: replacementValue
   self node arguments do: [ :arg | self context pop ]."Pop the arguments of
the message send from the context's value stack"
   "Pop the receiver from the context's value stack"
   self context pop.
   "Push the replacement value on the context's value stack, to simulate
that the message send happened and returned nil"
   self context push: replacementValue.
   "Increase the pc to go over the message send"
   self context pc: self context pc + 1.
   "Execute bytecodes the debugger usually executes without stopping the
execution (for example popping the return value of the just executed message
send if it is not used afterwards)"
   self debugSession
      stepToFirstInterestingBytecodeIn: self debugSession interruptedProcess
```

SindarinDebugger>>skipAssignmentNodeWith:

```
skipAssignmentNodeWith: replacementValue
   self context pop.
   "Pop the value to be assigned"
   "Push the replacement value on the context's value stack, to simulate
that the assignment happened and had value nil"
   self context push: replacementValue.
   "Increase the pc to go over the assignment"
   self context pc: self context pc + 2.
   "Execute bytecodes the debugger usually executes without stopping the
execution (for example popping the return value of the just executed message
send if it is not used afterwards)"
   self debugSession
      stepToFirstInterestingBytecodeIn: self debugSession interruptedProcess
```

Figure 34: Implémentation problématique de la commande skip

```
InstructionStream>>interpretNext2ByteSistaV1Instruction:for:extA:extB:startPC:
            numArgs: (extB bitShift: 3) + (byte \\ 8)].

        bytecode = 236 ifTrue:
          [^client trapIfNotInstanceOf: (method literalAt: (extA bitShift: 8) + byte + 1)].
       bytecode = 237 ifTrue:
          [^client jump: (extB bitShift: 8) + byte withInterpreter: self].
        ^client jump: (extB bitShift: 8) + byte if: bytecode = 238 withInterpreter: self].
    bytecode < 243 ifTrue:
       [bytecode = 240 ifTrue:
          [^client popIntoReceiverVariable: (extA bitShift: 8) + byte].
        bytecode = 241 ifTrue:
          [^client popIntoLiteralVariable: (method literalAt: (extA bitShift: 8) + byte + 1)].
        ^client popIntoTemporaryVariable: byte].
    bytecode = 243 ifTrue:
       [^client storeIntoReceiverVariable: (extA bitShift: 8) + byte].
    bytecode = 244 ifTrue:
       [^client storeIntoLiteralVariable: (method literalAt: (extA bitShift: 8) + byte + 1)].
    bytecode = 245 ifTrue:
       [^client storeIntoTemporaryVariable: byte].
    "246-247 1111011 ixxxxxxxx UNASSIGNED"
    ^self interpretUnusedBytecode: client at: startPC
```

Figure 35: Code de l'interpréteur utilisant l'identifiant de bytecode pour savoir quelle méthode exécuter en conséquence.

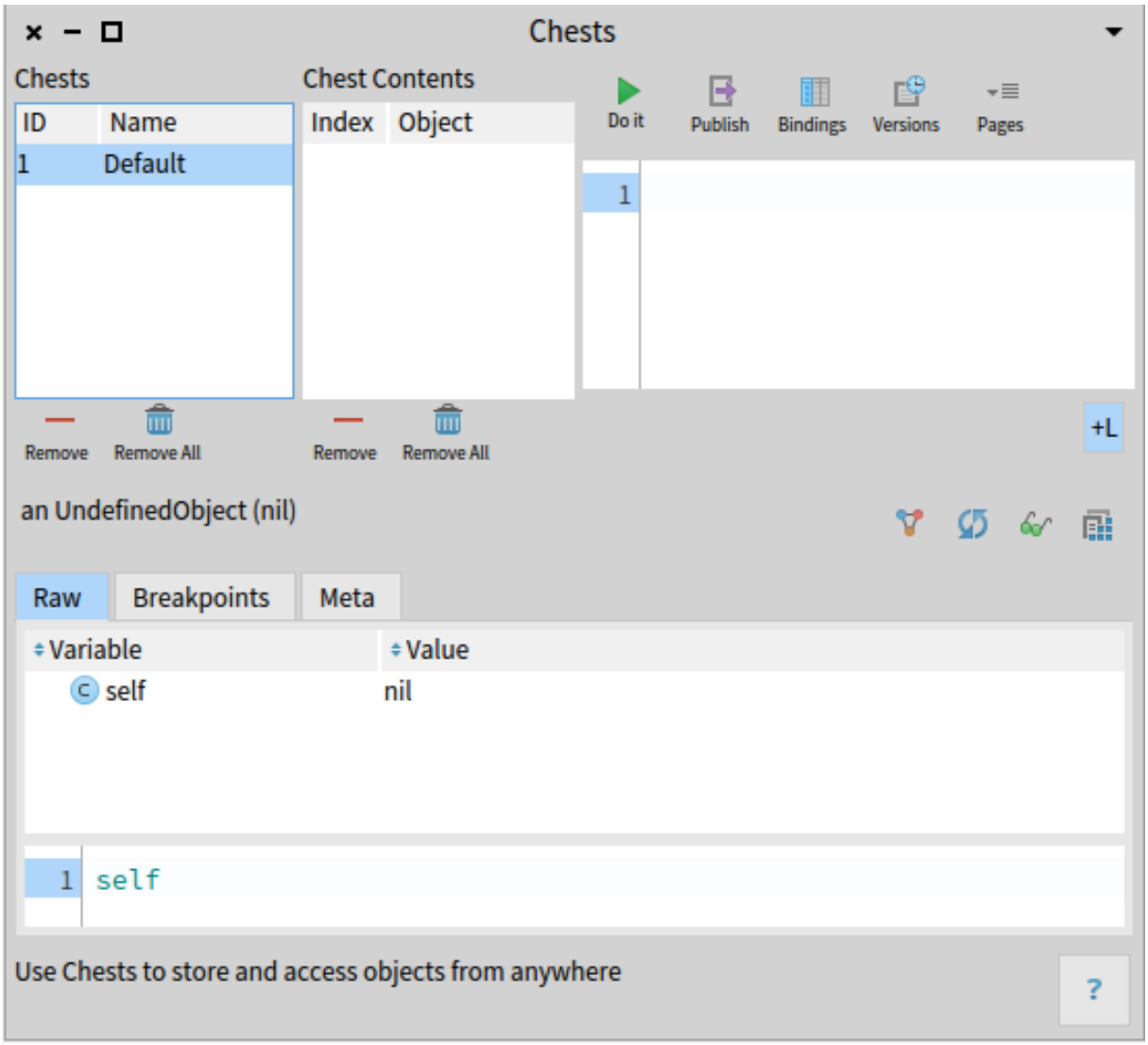

Figure 36: IHM de Chest avant *refactoring*, donnant peu de place aux coffres, leurs objets et le playground

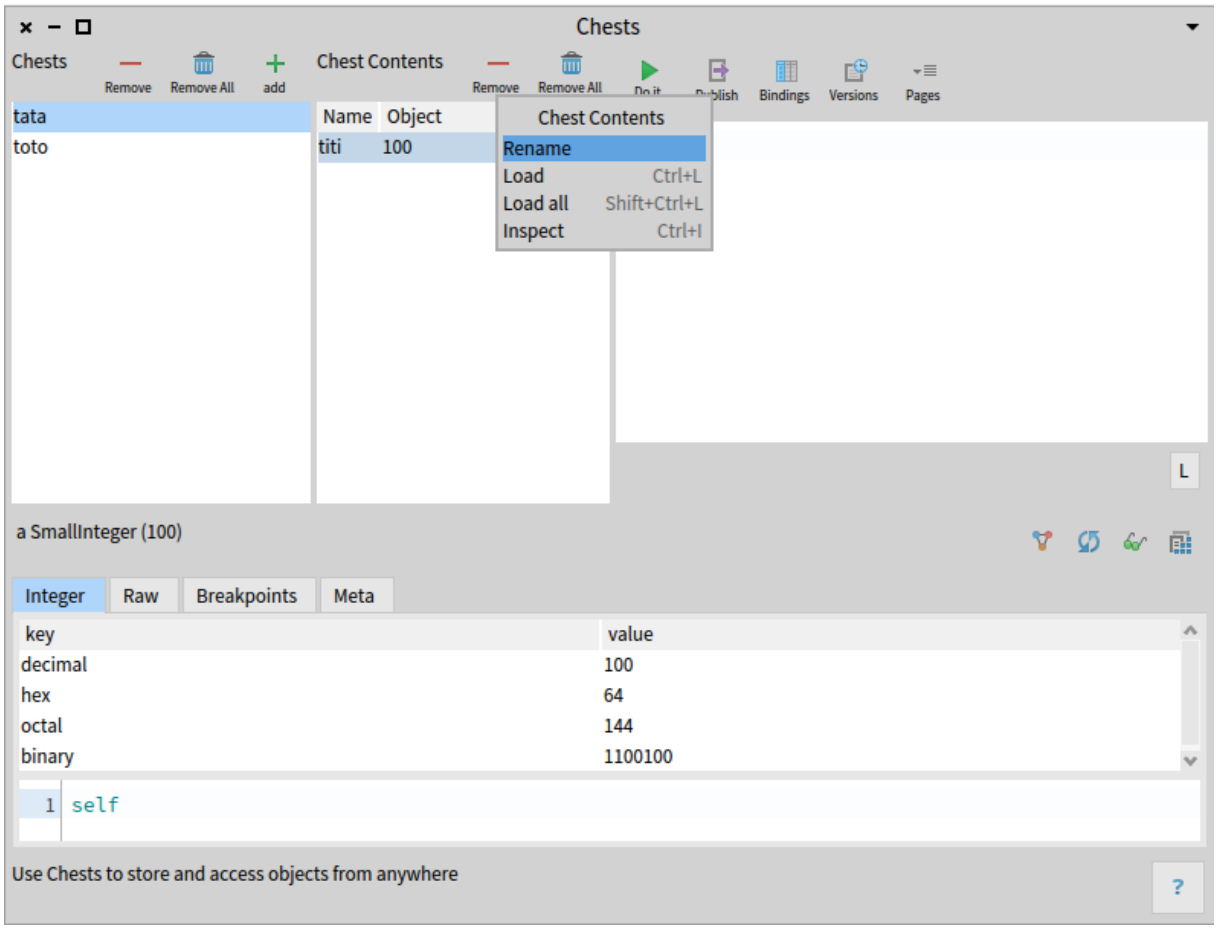

Figure 37: IHM de Chest après *refactoring*, accordant plus d'importance à la partie haute de l'interface.

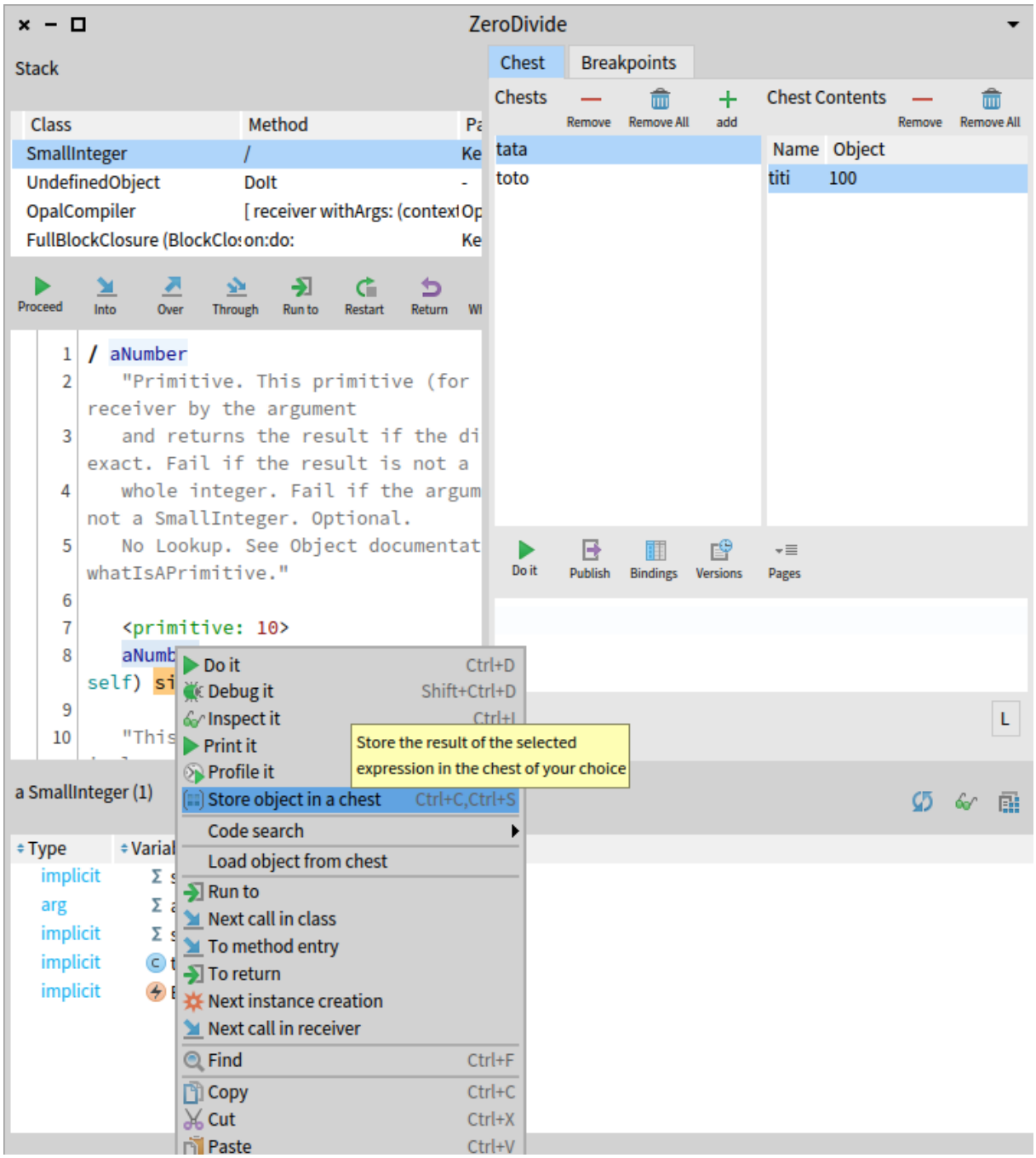

Figure 38: Intégration de Chest dans le debugger, proposant de stocker des objets depuis le debugger et de charger un objet d'un coffre directement dans le debugger.