IUT Belfort-Montbéliard

UNIVERSITÉ DE FRANCHE-COMTÉ

INFO DUT Informatique IUT Belfort-Montbéliard

THALES

Réalisation d'un outil de prototypage IHM pour composants

RAPPORT TECHNIQUE

Du 06 avril au 11 juin 2021

Lisa DOYEN

Entreprise : Thales DMS Brest

DUT Informatique
2ème année

Tuteur en entreprise :

Pierre LABORDE

Senior UX & UI designer

Tuteur enseignant :

Jean-Claude CHARR

Enseignant

Sommaire

Introduction

Ce stage a été réalisé au sein de l'entreprise Thales DMS à Brest, du 6 avril au 11 juin 2021. L'application développée est un outil de prototypage IHM *Open Source* pour composants avec les solutions de développement Pharo. Pouvoir rechercher et visualiser rapidement les composants[1] font partie des objectifs de cet outil.

Ce rapport a pour but d'expliquer précisément la partie technique du projet. Tout d'abord, les prérequis seront introduits. Ensuite, la présentation technique du sujet sera expliquée au travers des réalisations. Une conclusion clôturera ce rapport.

[1] **Composants** : voir rapport général : section 1.4

1 Prérequis Techniques

1.1 L'environnement Pharo

Pharo est un langage de programmation orienté objet, en SmallTalk. C'est aussi un environnement de développement en continuelle évolution.

Ce langage nait d'un laboratoire de recherche d'une société de Recherche et Développement appelé Xerox. Cette société est notamment populaire pour ses copieurs et imprimantes. Elle avait également une grosse unité de recherche appelé PARC qui a débouché sur le début du langage SmallTalk. Une première version de ce langage sort en 1972. Après plusieurs itérations, une version stable a été industrialisée à partir des années 1980 appelée SmallTalk 80.

Une première version libre et gratuite est sortie en 1997 sous le nom de Squeak. Sa finalité est principalement éducative et expérimentale. Puis, en 2008, vient une nouvelle version libre issue de Squeak appelé Pharo notamment utilisé dans l'industrie.

SmallTalk est un langage puissant qui a été pensé pour les programmeurs. Les syntaxes et concepts sont réduits au minimum. Son environnement est capable d'inspecter et de modifier les objets et les programmes pendant l'exécution.

Ce langage est simple à apprendre, il est proche du langage naturel. Il y a peu de mots et caractères spéciaux. Le typage est dynamique et les concepts sont uniformes. Tout est objet et une action équivaut à un message.

1.1.1 Le *Pharo Launcher*

La toute première étape pour commencer à développer en Pharo passe par le *Pharo Launcher*. Son nom signifie en français « Lanceur de Pharo ». Cet outil permet de créer et de gérer des projets Pharo.

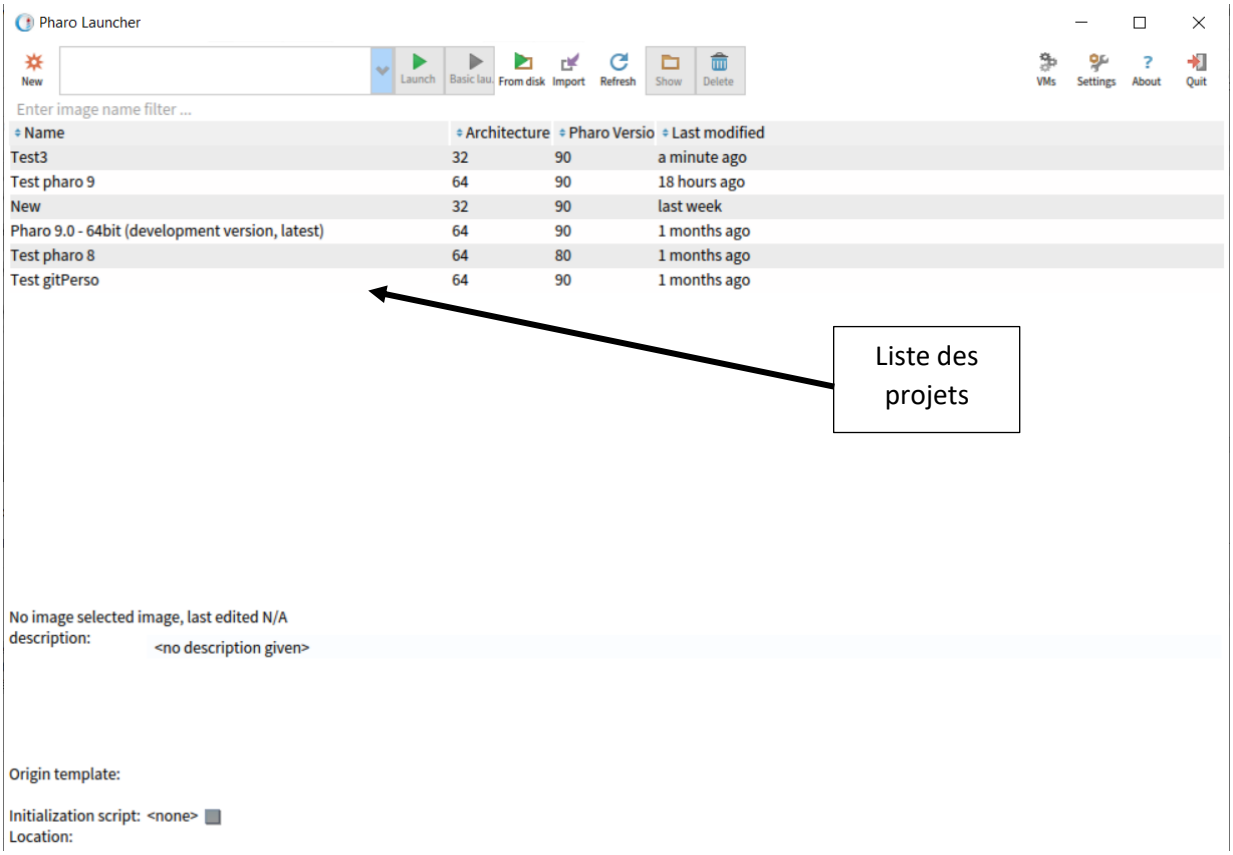

Figure 1 : Fenêtre du Pharo Launcher

Sur le *Pharo Launcher*, la liste des différents projets est présente (voir figure 1). Il s'agit de différentes images Pharo avec pour chacune son nom, son architecture, sa version et la date de sa dernière modification.

Lorsque l'utilisateur clique sur le bouton *New* en haut à gauche, il va pouvoir créer une nouvelle image Pharo correspondant à un nouveau projet.

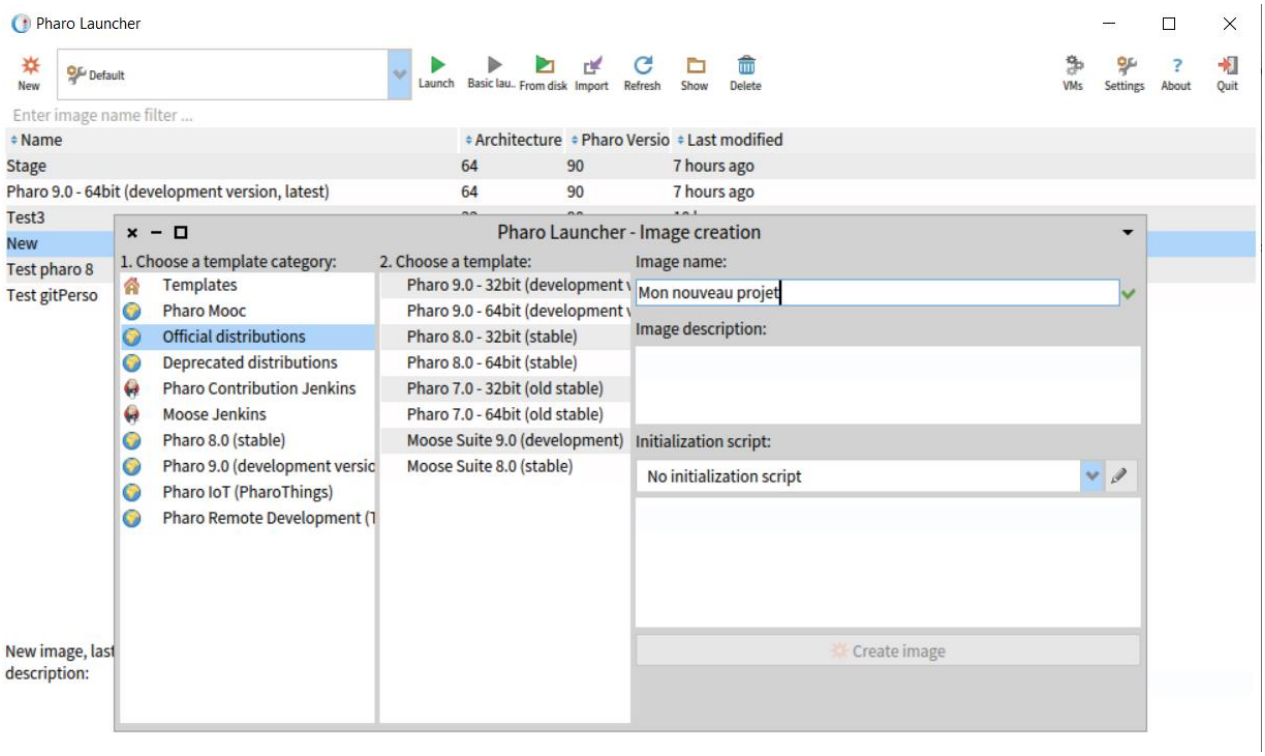

Figure 2 : Pharo Launcher avec la fenêtre permettant de créer une nouvelle image Pharo

Sur la figure 2, une fenêtre s'ouvre permettant de créer une nouvelle image Pharo. Dans celle-ci, différentes versions de Pharo sont proposées. Actuellement, la version stable est la huitième, cependant, j'ai utilisé la neuvième pour développer l'outil. Pour créer une image, l'utilisateur choisit la version, saisit le nom de l'image et la crée.

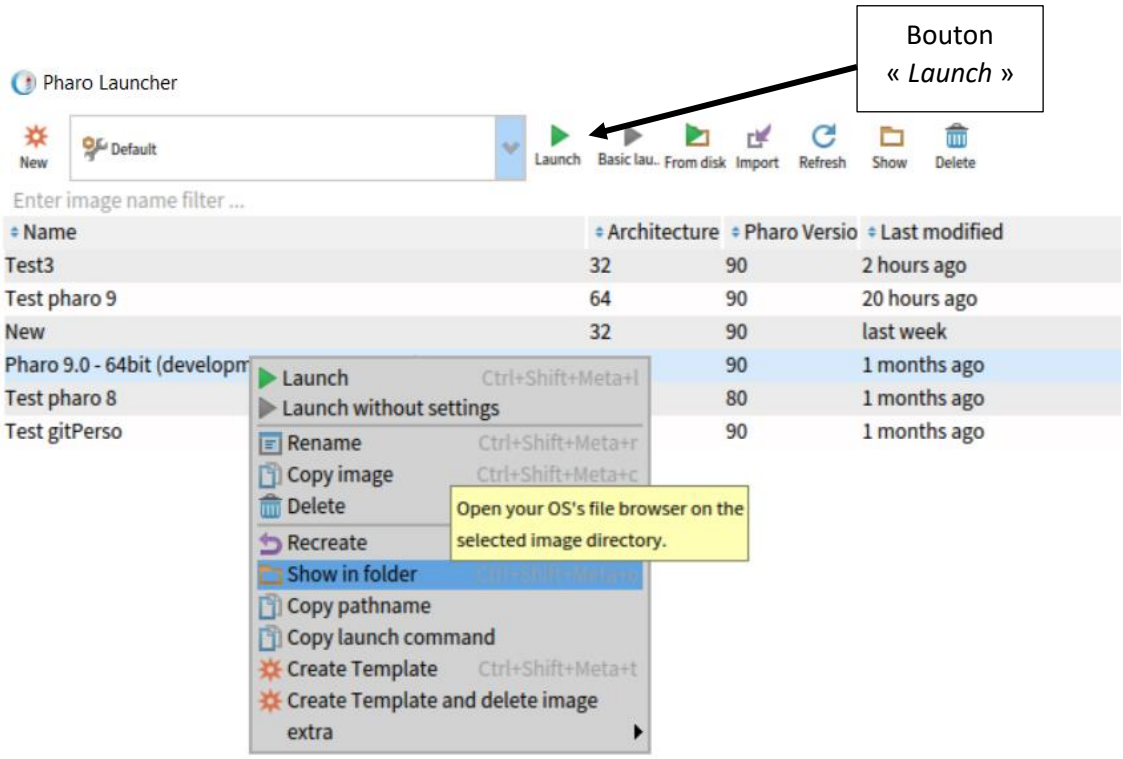

Figure 3 : Menu contextuel à partir d'une image sélectionnée

En effectuant un clic droit sur une image Pharo, un menu contextuel va apparaitre (voir figure 3). L'utilisateur peut cliquer sur le menu « *Show in folder* ».

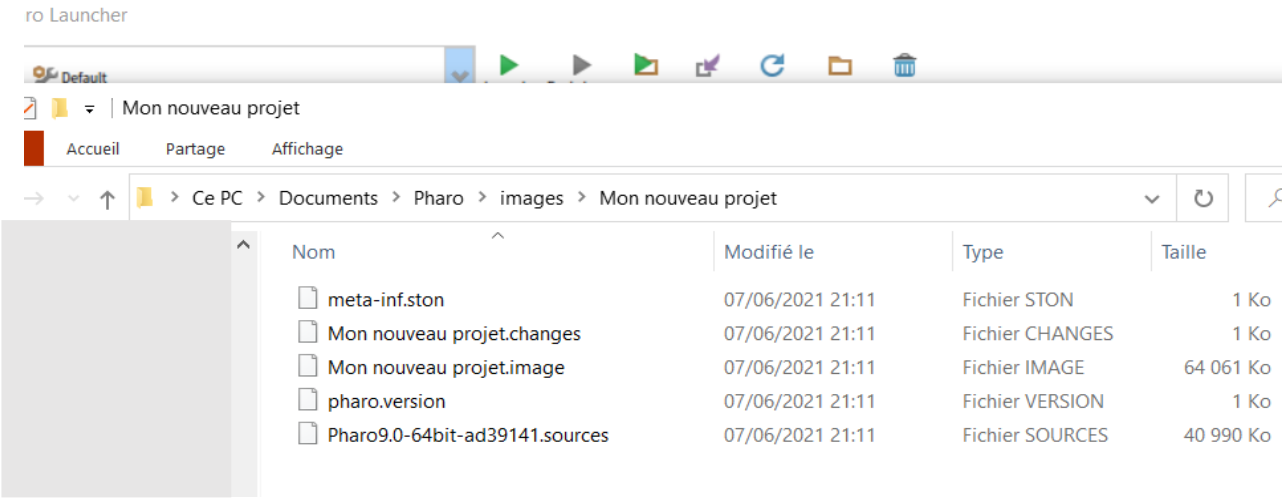

Figure 4 : Dossier d'une image Pharo

Le dossier du projet va alors s'afficher (voir figure 4). Dans ce dossier, nous retrouvons le fichier SOURCE, le fichier IMAGE, le fichier VERSION, le fichier CHANGES et le fichier STON.

Pour lancer un projet, l'utilisateur le sélectionne et clique sur le bouton « *Launch* » (voir bouton sur la figure 3). Il entre ensuite dans l'image et accède à l'environnement de Pharo.

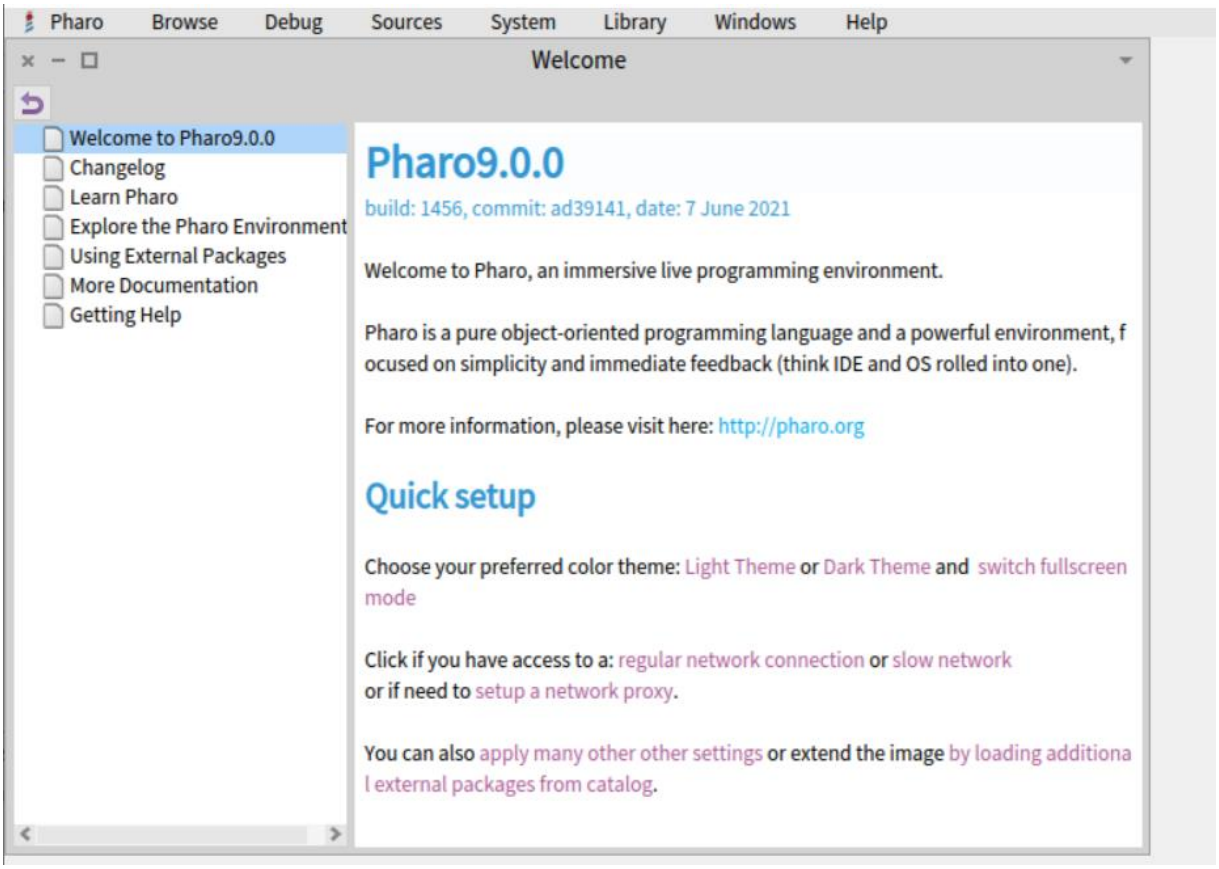

Figure 5 : Fenêtre d'accueil dans une image Pharo 9

Une fois arrivée dans le projet, par défaut, la fenêtre d'accueil est ouverte. Elle contient plusieurs informations notamment sur les réglages possibles (voir figure 5). Le thème par défaut est blanc pour la neuvième version.

Pour pouvoir naviguer dans l'image, un clic gauche fait apparaitre un menu contextuel appelé « *World* » avec différents sous-menus. Ce menu est également visible en haut de l'image.

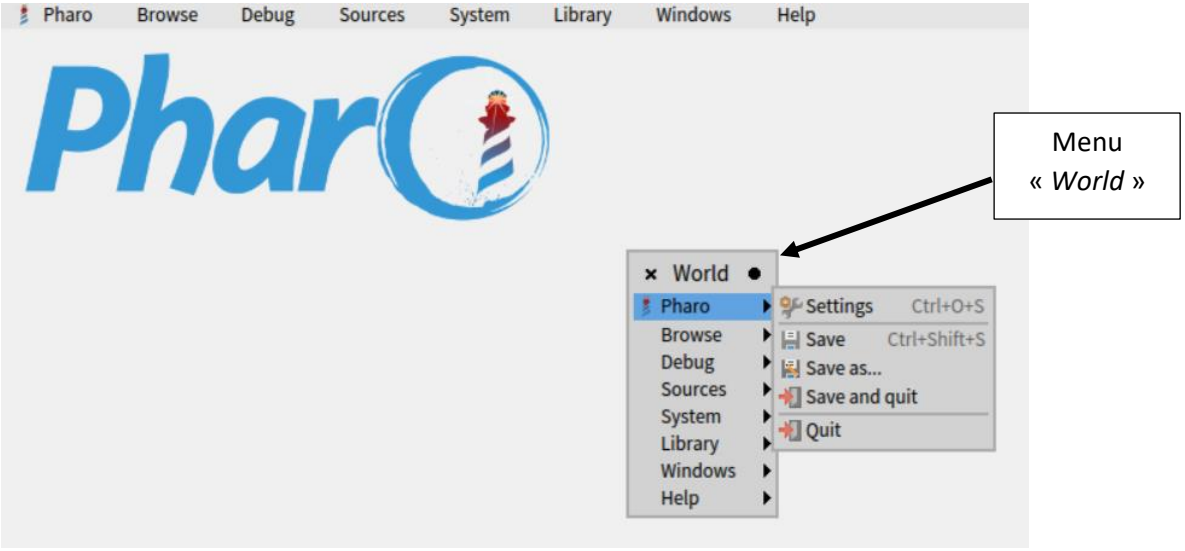

Figure 6 : Menu World accessible dans l'image Pharo

Ce menu permet de découvrir les différentes actions possibles dans l'environnement Pharo (voir figure 6). Par exemple, dans le menu « Pharo », l'utilisateur peut avoir accès aux paramètres de l'image via le sous-menu « *Settings* ». Il peut également sauvegarder son image et/ou l'arrêter.

1.1.2 Le *System Browser*

L'outil de développement de Pharo est le « *System Browser* ». Pour l'ouvrir le développeur y accède via le menu « *Browse* » puis le sous-menu « *System Browser* ».

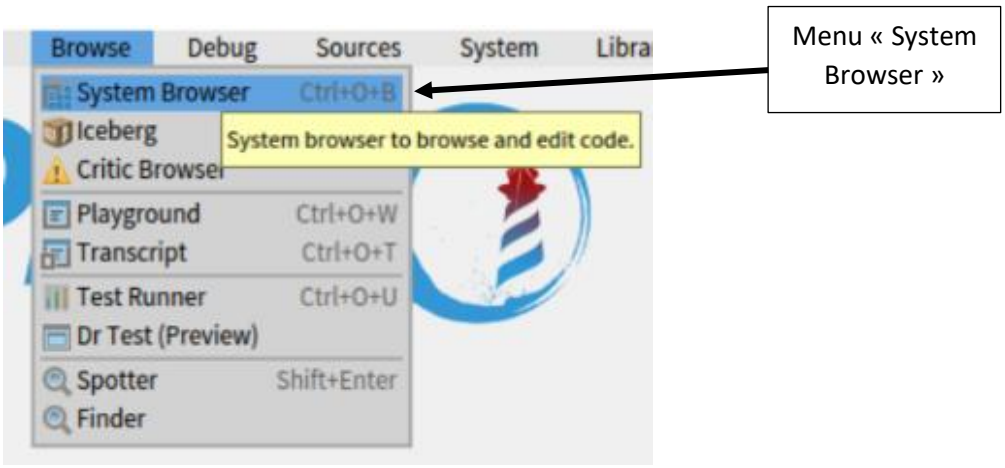

Figure 7 : Point d'entrée pour accéder au System Browser

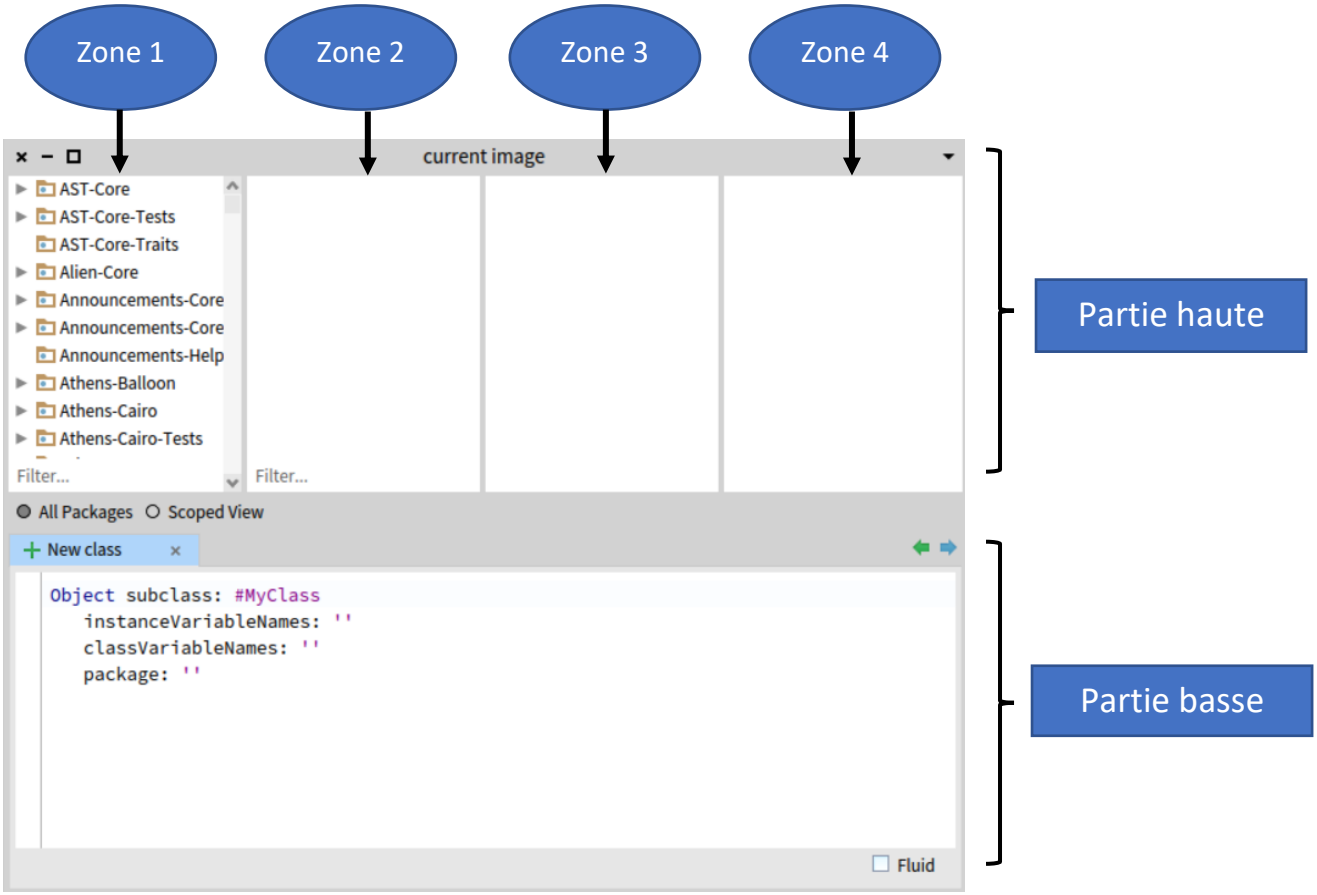

Figure 8 : System Browser dans l'environnement Pharo

Une fois cette action réalisée, la fenêtre du *System Browser* s'ouvre (voir figure 8). Elle est composée d'une partie haute et d'une partie basse.

Concernant la partie haute, elle est divisée en quatre zones :

- La première zone (zone 1 sur la figure 8) affiche les *packages*. Pour pouvoir accéder à un paquet spécifique, une barre de recherche est disponible.

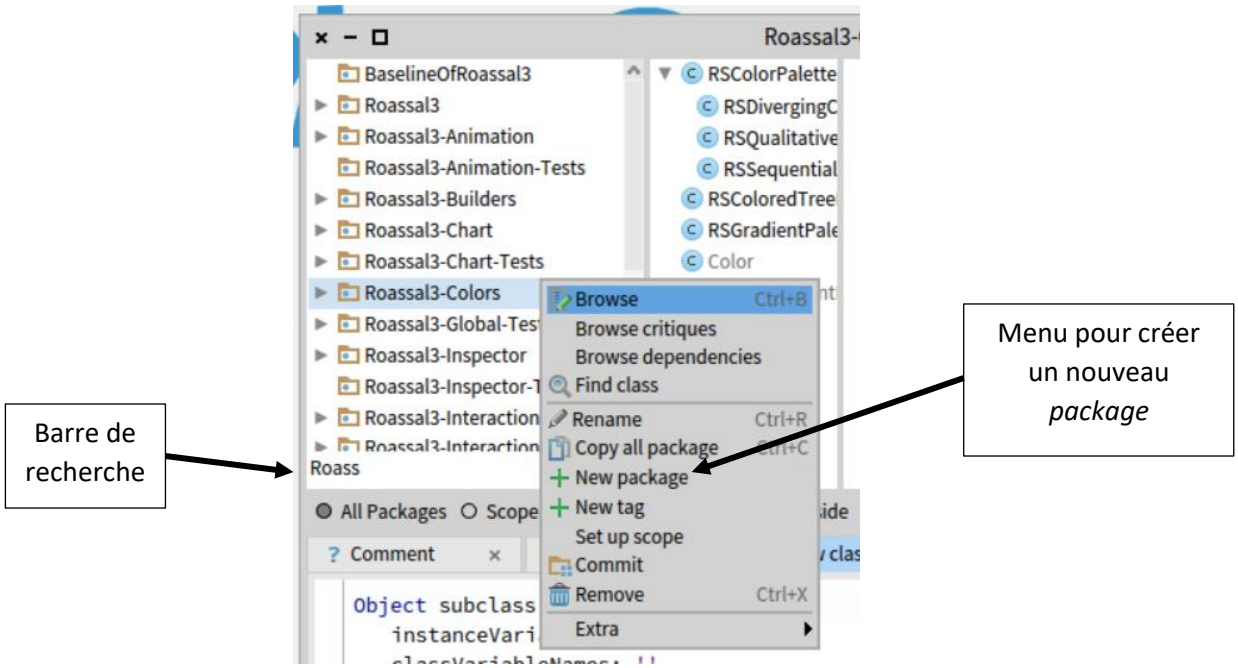

Figure 9 : Menu pour créer un nouveau package

Pour pouvoir créer un *package*, il suffit de faire un clic droit sur un *package* existant et de sélectionner le menu « *New package* » (voir figure 9). L'utilisateur peut ensuite entrer son nom et le créer.

Les autres zones vont s'actualiser en fonction de la sélection du *package*.

- Dans la seconde zone (zone 2 sur la figure 8) se trouve la liste des classes contenues dans le *package* sélectionné. Pour plus de lisibilité, chaque classe est identifiée par une icône. Par exemple, une classe abstraite va avoir une icône différente d'une classe non abstraite.

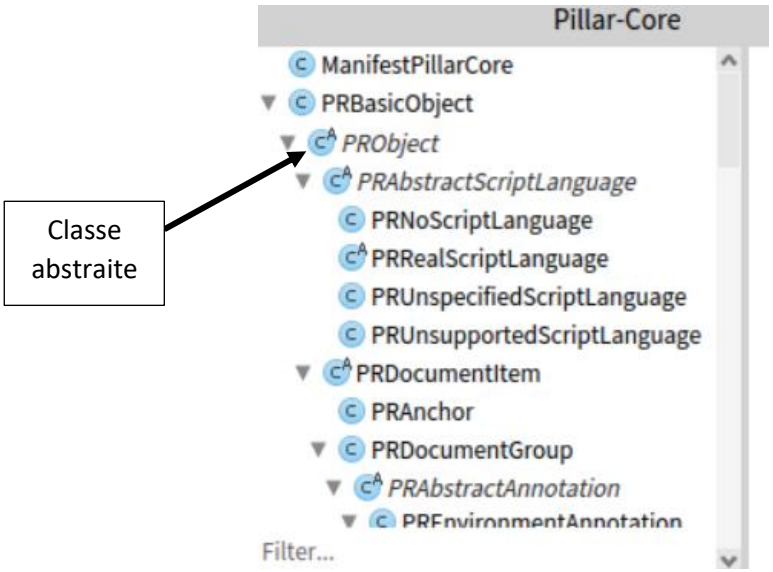

Figure 10 : Liste des classes contenues dans un package

Sur la figure 10, la classe appelée *PRObject* est une classe abstraite alors que *PRBasicObject* n'en est pas une, cela se distingue par le « A » sur l'icône.

Lorsqu'une classe va être sélectionnée par l'utilisateur, la troisième et quatrième zone s'actualisent.

- La troisième zone (zone 3 sur la figure 8) contient les classifications/protocoles des méthodes de la classe. Elle sert de documentation. Quand une classe possède beaucoup de méthodes, cette zone permet de les ranger et de les retrouver plus facilement.

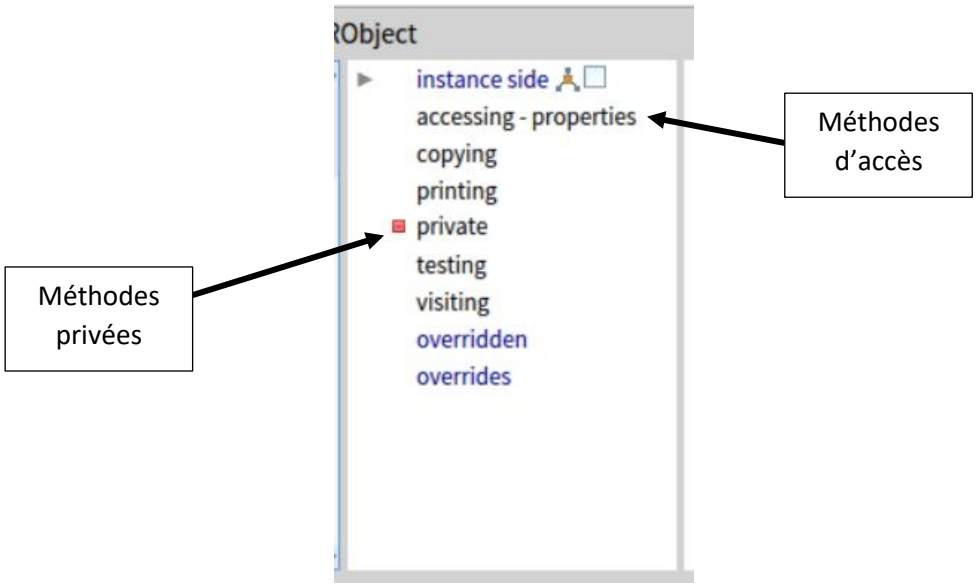

Figure 11 : Liste des classifications/protocoles des méthodes

Par exemple, sur l'image 11, certaines méthodes sont des méthodes d'accès (plus connus sous les noms de *setters* et *getters*) appelé ici « *accessing - properties* », d'autres sont des méthodes privées référencées comme « *private* ».

- Lorsque l'utilisateur clique sur une classification, la quatrième zone (zone 4 sur la figure 8) s'actualise. Elle contient la liste des méthodes de la classe correspondante à la classification choisie.

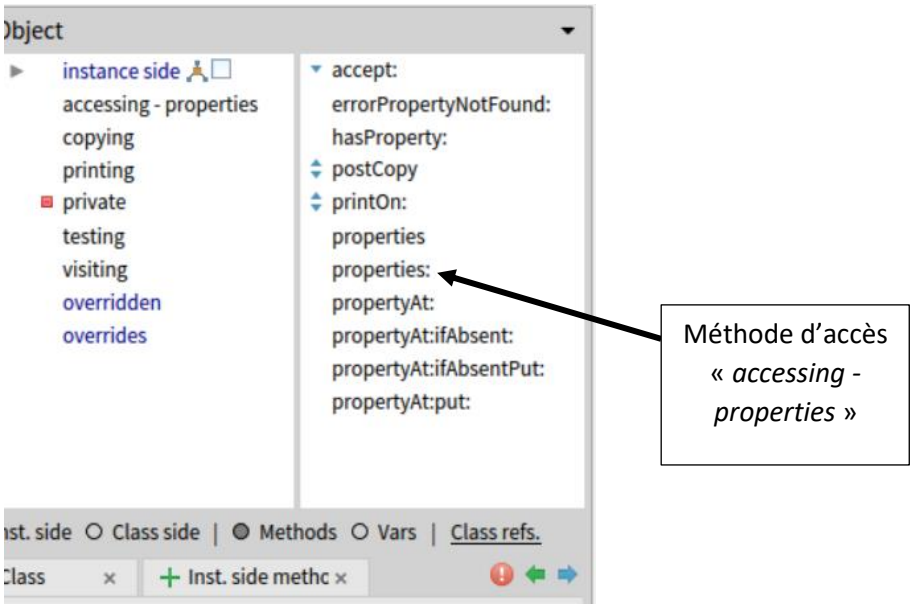

Figure 12 : Liste des méthodes d'accès

Par exemple, sur la figure 12, l'utilisateur affiche la liste des méthodes correspondante à la classification « *accessing – properties* ».

Par défaut, aucune classification n'est choisie ce qui permet d'afficher l'intégralité des méthodes.

Concernant la partie basse, des onglets donnent la possibilité à l'utilisateur de choisir ce qu'il affiche. C'est à l'intérieur de ces onglets que le développeur écrit du code informatique.

Si le développeur sélectionne un *package*, les onglets disponibles sont :

- Le premier onglet, nommé « *Comment* », permet d'ajouter des commentaires sur le *package*.

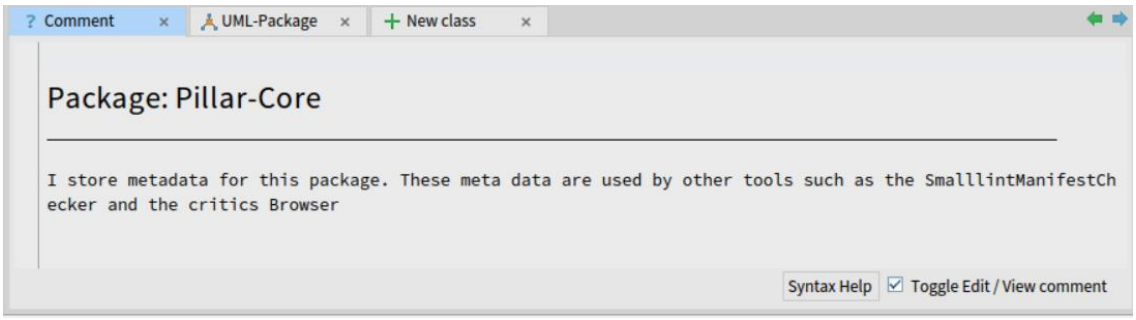

Figure 13 : Informations sur le package Pillar-Core

Sur la figure 13, des informations sur le *package* « Pillar-Core » sont présentes. Les informations sont en anglais car c'est la langue la plus utilisée en informatique.

- Le second onglet appelé « *UML-Package* » va permettre de visualiser le contenu du package sous forme de diagramme UML.

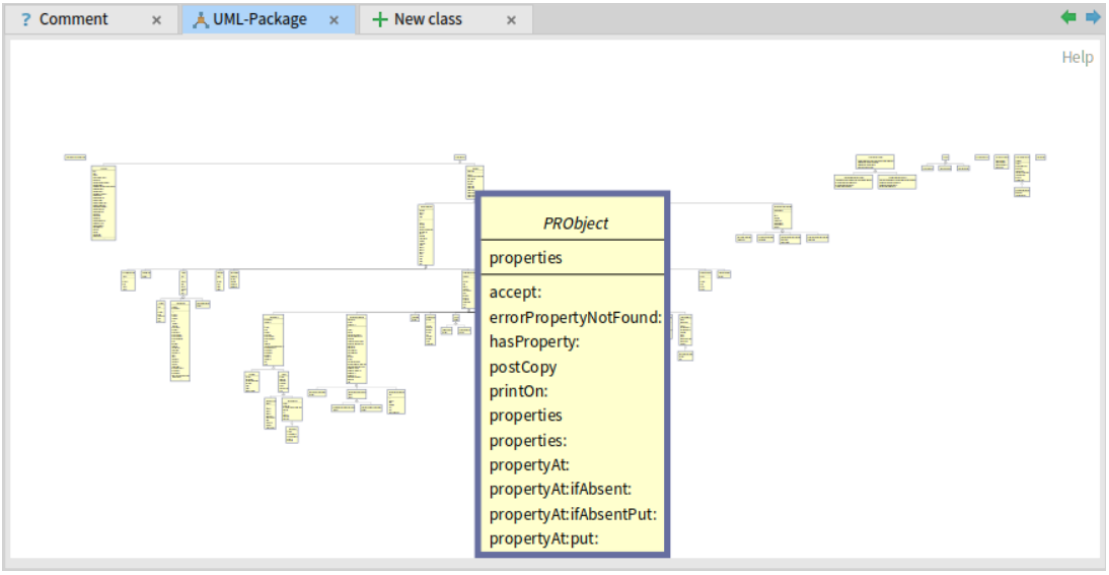

Figure 14 : Digramme UML du package Pillar-Core

Sur la figure 14, le diagramme UML du *package* « Pillar-Core » est affiché. Au passage de la souris sur une classe, un *tooltip* grossit cette classe afin de visualiser le contenu de la classe plus facilement.

- Le troisième onglet va permettre de créer une nouvelle classe.

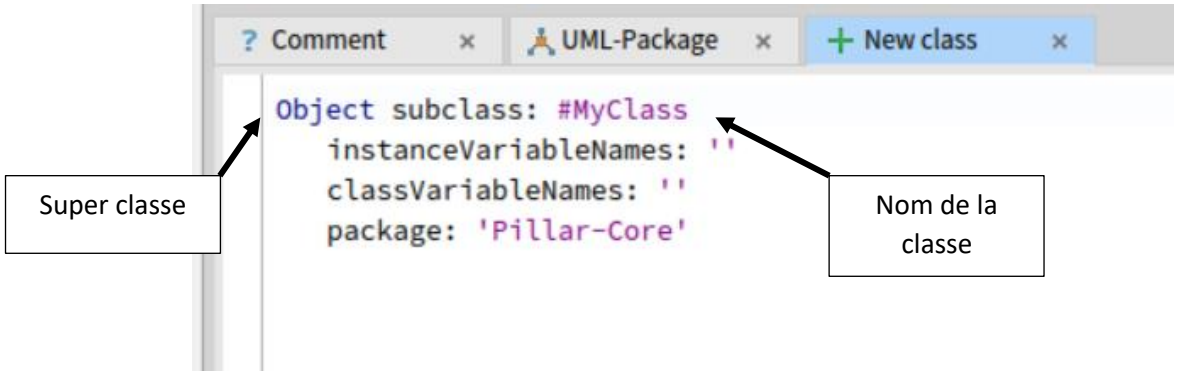

Figure 15 : Editeur permettant de créer une classe

Pour créer une classe, il faut lui attribuer un nom et une super classe, puis, l'enregistrer. Par défaut, sa super classe est la classe *Object*.

Si le développeur sélectionne une classe, les onglets disponibles sont :

- Le premier onglet, nommé « *Comment* », permet d'ajouter des commentaires sur la classe.
- Le second onglet, nommé par le nom de la classe, permet de voir la définition de la classe avec son nom, sa super classe, ses variables de classes et d'instances et le nom de son *package*.

  Il est également possible de voir les *Traits*[2] utilisés par la classe.

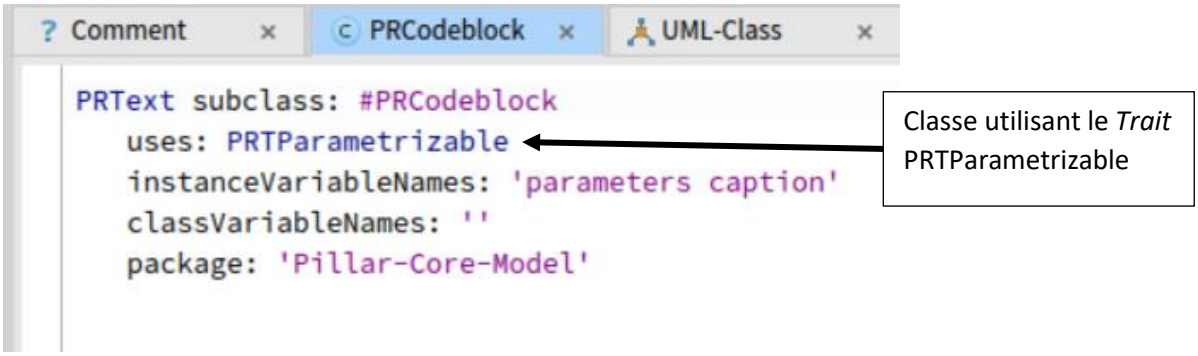

Figure 16 : Exemple de l'affichage de la classe PRCodeblock

[2] ***Traits*** : voir rapport générale section 1.4.1.2.3

Dans la figure 16, la classe PRCodeBlock utilise le *Trait* PRTParametrizable.

- Le troisième onglet peut être soit l'onglet appelé « *UML-Class* » permettant de visualiser le contenu de la classe en UML soit l'onglet contenant une méthode qui a été sélectionnée.
- Le dernier onglet, appelé « Inst.side method » ou « Class. side method », va permettre de créer une nouvelle méthode dans la classe. C'est à l'intérieur de cet éditeur de texte que le développeur écrit du code informatique.

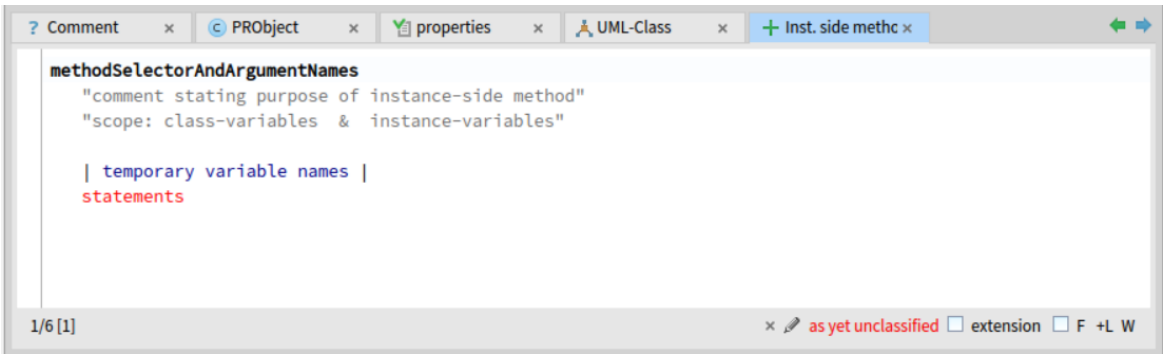

Figure 17 : Onglet permettant de créer une méthode

Entre les deux zones, des cases à cocher sont présentes pour personnaliser l'affichage des *packages* et des classes (voir figure 18).

Figure 18 : Cases à cocher pour modifier l'affichage

Par exemple, l'utilisateur peut afficher tous les paquets ou seulement celui sélectionné. Il peut aussi faire apparaitre la hiérarchie d'une classe sélectionnée. Il choisit également d'afficher soit les méthodes d'instance soit les méthodes de classe.

Dans l'environnement Pharo, le développeur a accès à de nombreux outils comme le débugueur, le *Playground* et la console appelée *Transcript*.

1.1.3 Le *Playground*

Le *Playground* (terrain de jeu en français) est un outil dans Pharo. Cet outil est accessible via le menu « *Browse* » puis le sous-menu « *Playground* » (voir figure 19).

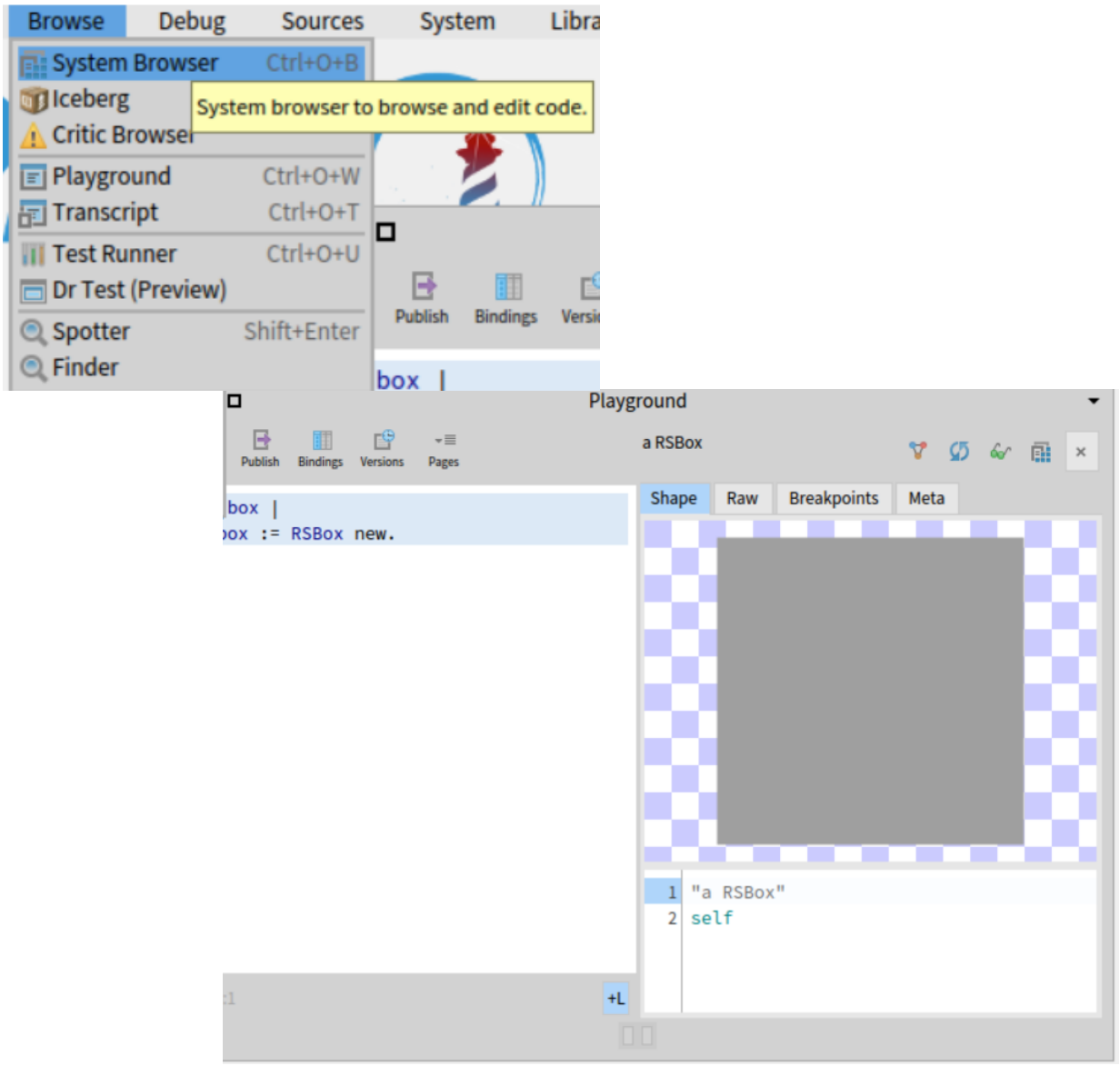

Figure 19 : Menu d'accès et fenêtre de l'outil Playground

Il est très pratique pour expérimenter de manière interactive du code. Le code écrit dans le *Playground* peut être compilé à la volée, puis exécuté. Les affectations sont mémorisées. Il peut être utile pour débuguer un programme par exemple.

Cet outil m'a permis d'exécuter et tester certains programmes. Il m'a notamment servi lorsque j'étais dans la phase de prise de connaissance du langage. Je reprenais des exemples de script que j'essayais de modifier pour découvrir les possibilités.

1.1.4 La console : *Transcript*

Le *Transcript* (la console en français) est un outil dans Pharo. Cet outil est accessible via le menu « *Browse* » puis le sous menu « *Transcript* » (voir figure 20).

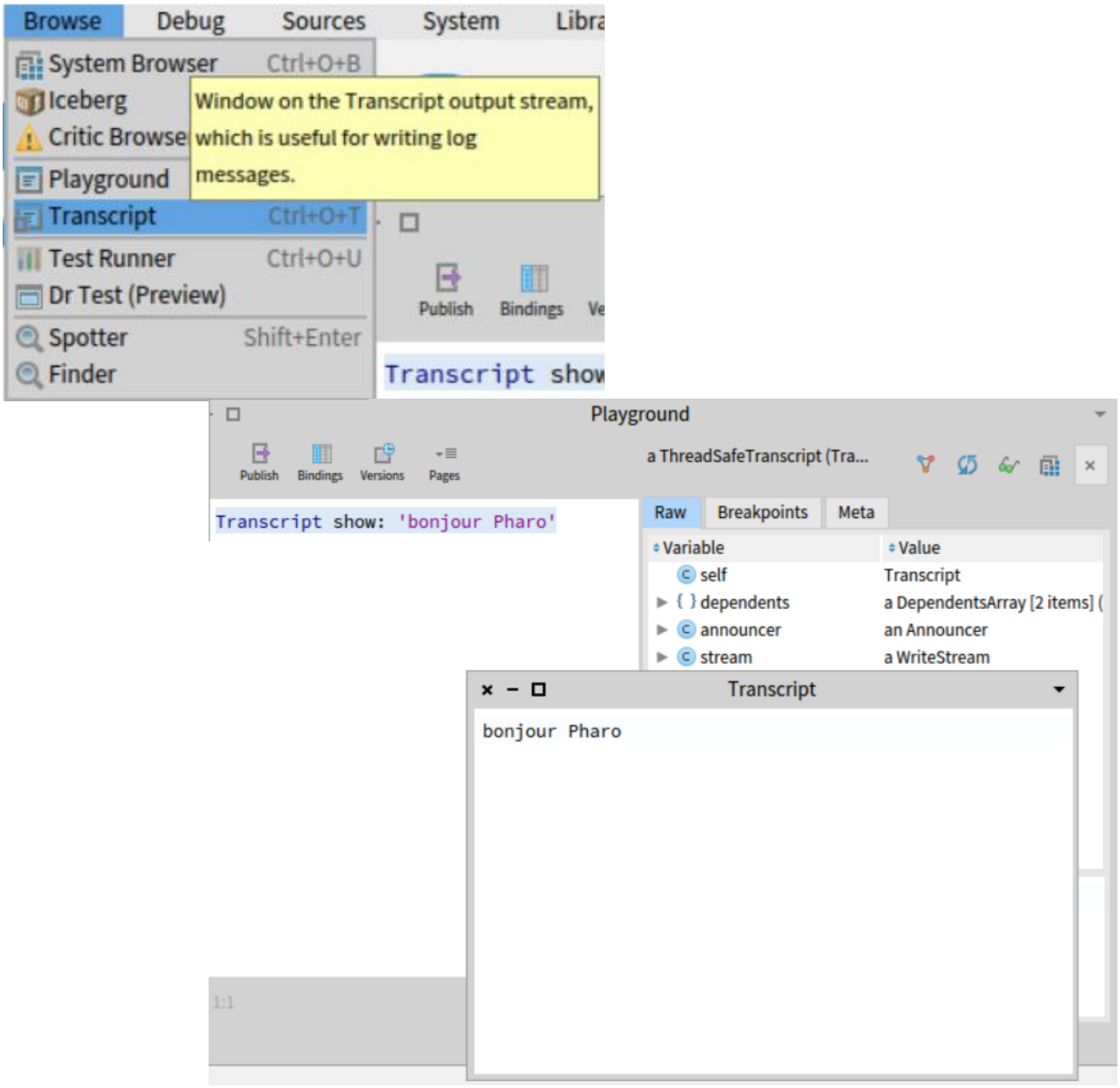

Figure 20 : Menu d'accès et affichage de la console Transcript

Cette console est très utile pour afficher ou écrire des informations. Elle peut être utilisée pour débuguer un programme par exemple. Le développeur peut exécuter du code mais, contrairement au *Playground*, les affectations ne sont pas mémorisées.

Pour mon projet, cette console m'a permis de tester rapidement du code et ainsi pouvoir détecter facilement où se situaient les erreurs.

1.1.5 Le débogueur

Le débogueur (ou débugueur) est un outil dans Pharo permettant d'identifier plus facilement les erreurs dans les programmes. Le développeur va pouvoir débuguer pas à pas le code qui vient d'être exécuté.

Cet outil est accessible via les points d'arrêt créés par l'utilisateur au sein d'un programme ou bien via un clic droit sur une ligne de code/expression via le menu « *Debug it* ».

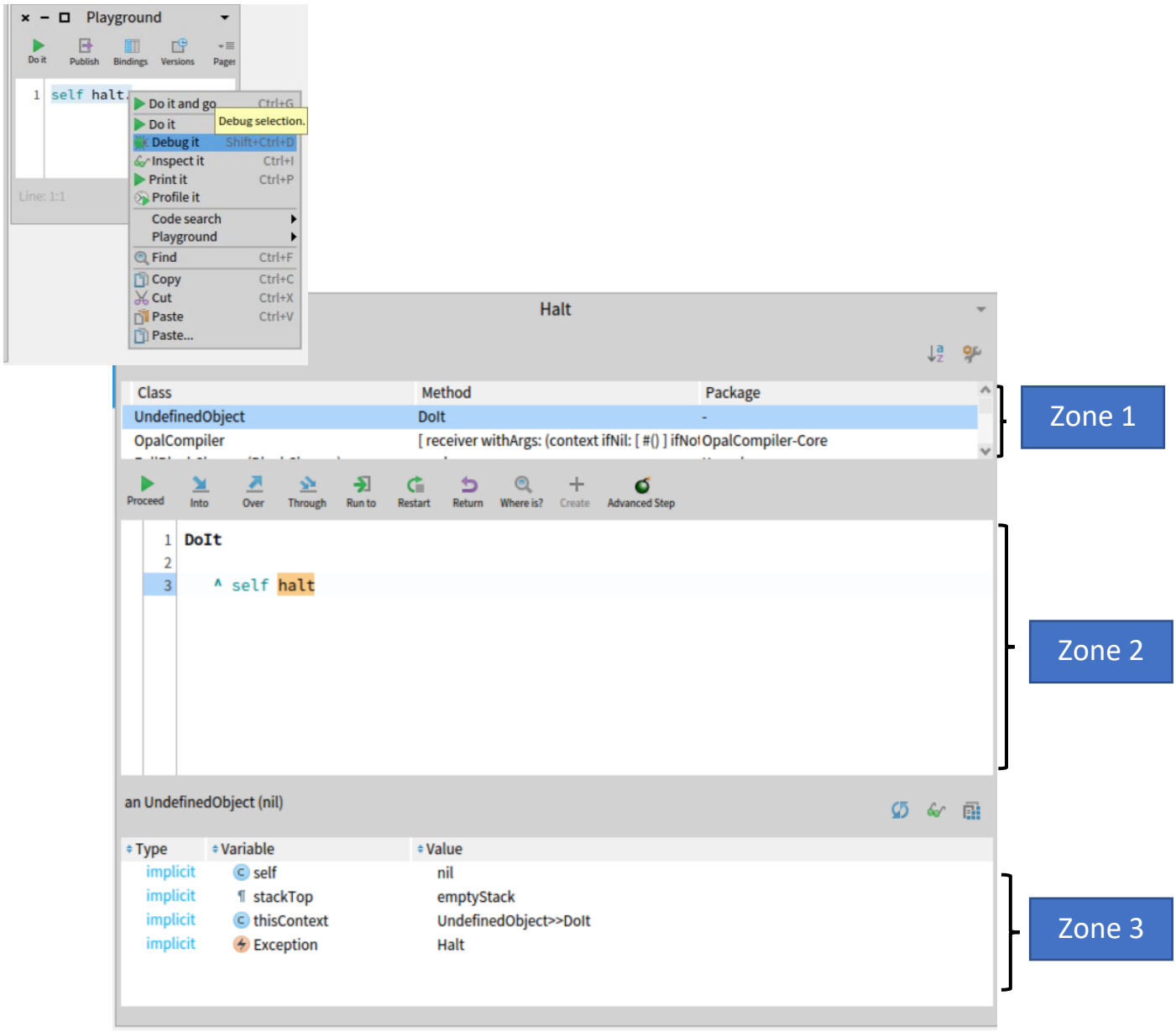

Figure 21 : Affichage du débogueur

Au sein de ce débogueur, trois zones horizontales se distinguent. La première zone (zone 1 sur la figure 21) contient une liste des différentes étapes lors de l'exécution du code. Le débogueur s'ouvre lorsque l'exécution est rendue à la ligne sélectionnée par le développeur. La deuxième zone (zone 2 sur la figure 21) affiche le code exécuté (correspondant à l'étape sélectionnée dans la liste dans la première zone). La dernière zone (zone 3 sur la figure 21) est représentée par un inspecteur permettant de connaitre notamment les valeurs des variables d'instances et/ou temporaires au moment de l'exécution. Dans cet inspecteur, l'objet courant indiqué par « *self* » est aussi indiqué (correspondant à l'objet « *this* » en Java).

Dans le cadre de mon stage, j'ai fortement utilisé cet outil pour comprendre précisément le cheminement du code. Cela m'a aussi permis de vérifier les valeurs des variables.

1.2 Roassal3

Roassal est une bibliothèque graphique directement intégrée dans Pharo. La version utilisée pour le stage est la troisième version. Roassal permet de construire des graphiques ou d'autres visualisations interactives avec l'utilisateur.

La version initialement présente sur Pharo ne représente pas la version complète de Roassal. En effet, il est possible de télécharger la version complète avec l'expression ci-dessous via le *GitHub* de Pharo. Cette commande peut être exécutée dans le *Playground*. Cette version contient plus d'exemples et de classes afin de créer de nouveaux éléments graphiques. Dans le cadre de mon stage, j'ai eu besoin de la version complète.

```
[Metacello new
    baseline: 'Roassal3';
    repository: 'github://ObjectProfile/Roassal3';
    load: 'Full' ] on: MCMergeOrLoadWarning do: [:warning | warning load ]
```

Figure 22 : Expression permettant de récupérer la version complète de Roassal

Le point d'entrée de Roassal se situe dans le menu principal. Via le menu « *Library* », un sous menu nommé « Roassal3 » apparait (voir figure 23). Il contient plusieurs menus traduisant différents accès.

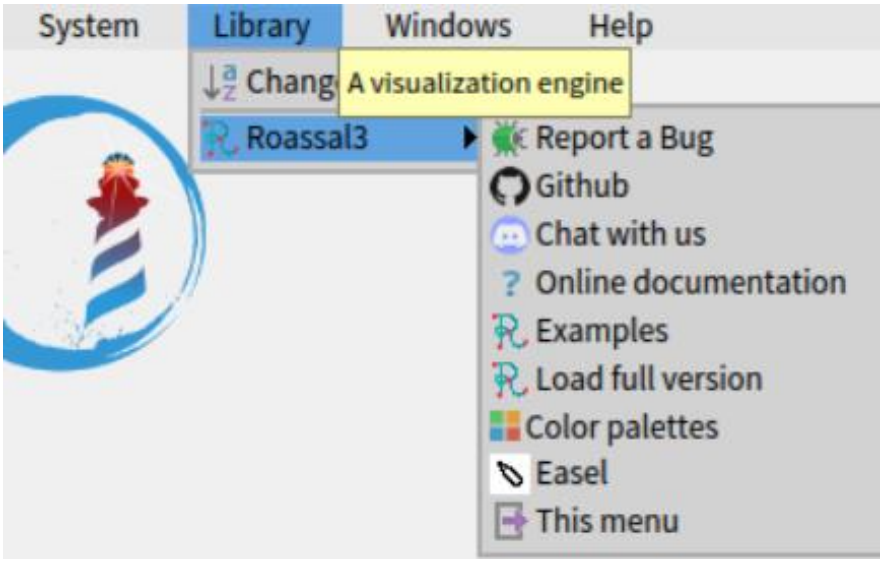

Figure 23 : Menu contextuel de Roassal

Parmi ces menus, l'accès à la communauté Roassal est possible via les menus nommés « *GitHub* », « *Chat with us* » et « *Report a bug* » afin de discuter et/ou reporter un éventuel bug.

Puis, un menu nommé « *Examples* » permet à l'utilisateur d'ouvrir les exemples Roassal au sein du *System Browser*. Ces exemples sont rangés par catégorie, par exemple, les animations et les formes basiques vont être séparées. Pour visualiser un exemple, il est possible de lancer le programme en appuyant sur l'icône à côté du nom de l'exemple.

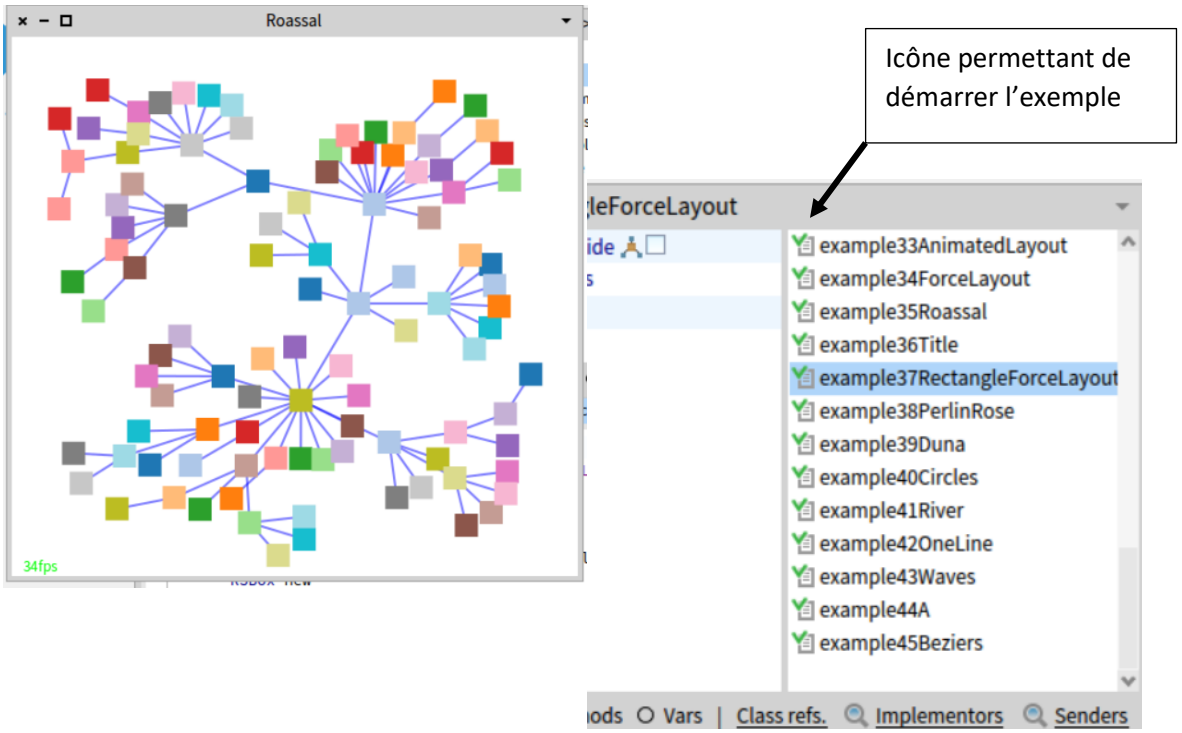

Figure 24 : Résultat d'un exemple Roassal

Dans cet exemple, la fenêtre contient un graphique (voir figure 24). Chaque petit carré de couleur représente une sous classe de la classe *Collection*.

L'utilisateur peut également consulter le script et voir quelles sont les classes qui ont été utilisées.

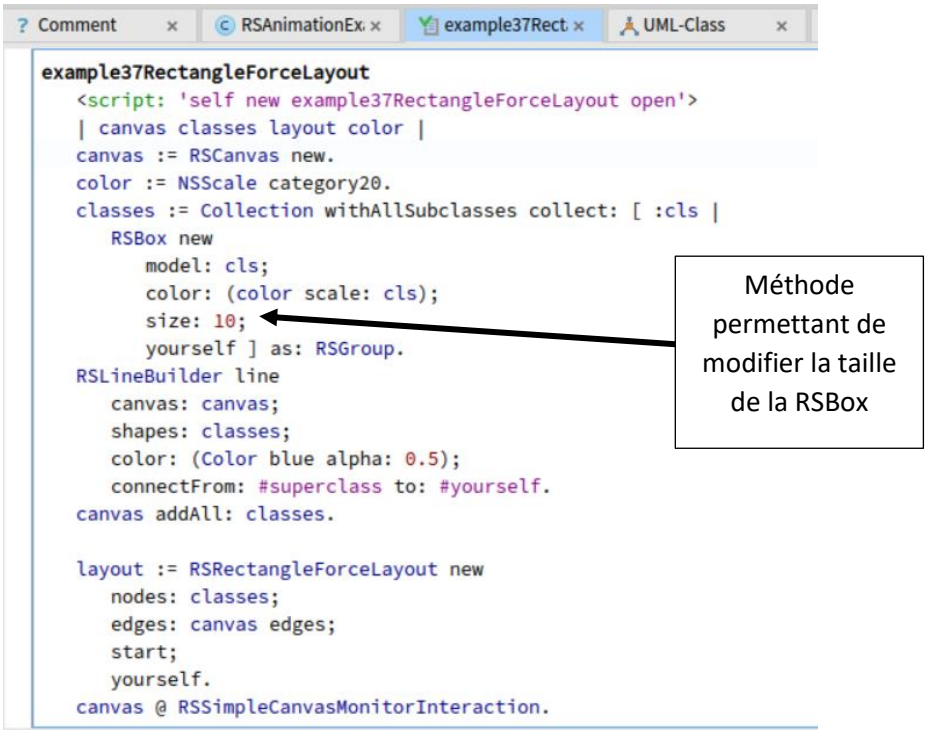

Figure 25 : Script de l'exemple en figure 24

Les classes appartenant à Roassal commencent toujours par l'abréviation « RS ». Il s'agit d'une norme d'écriture permettant d'identifier plus facilement ses classes. Dans le script visible sur la figure 25, les classes de Roassal utilisées sont RSBox, RSLineBuilder, RSGroup, RSCanvas, RSRectangleForceLayout et RSSimpleCanvasMonitorInteraction.

Le développeur peut aussi personnaliser un objet. Par exemple, l'utilisateur définit la taille de la RSBox (size : 10).

Aussi, nous observons que les noms des classes respectent la notation *CamelCase.*

1.3 Spec2

Spec2 est un *Framework* intégré dans Roassal permettant de décrire les interfaces. Par exemple, il est utilisé pour afficher des boutons, des listes et des tableaux.

Dans le menu principal via le menu « *Help* », un sous menu nommé « *Spec2 examples* » apparait. Il ouvre les exemples disponibles de Spec2.

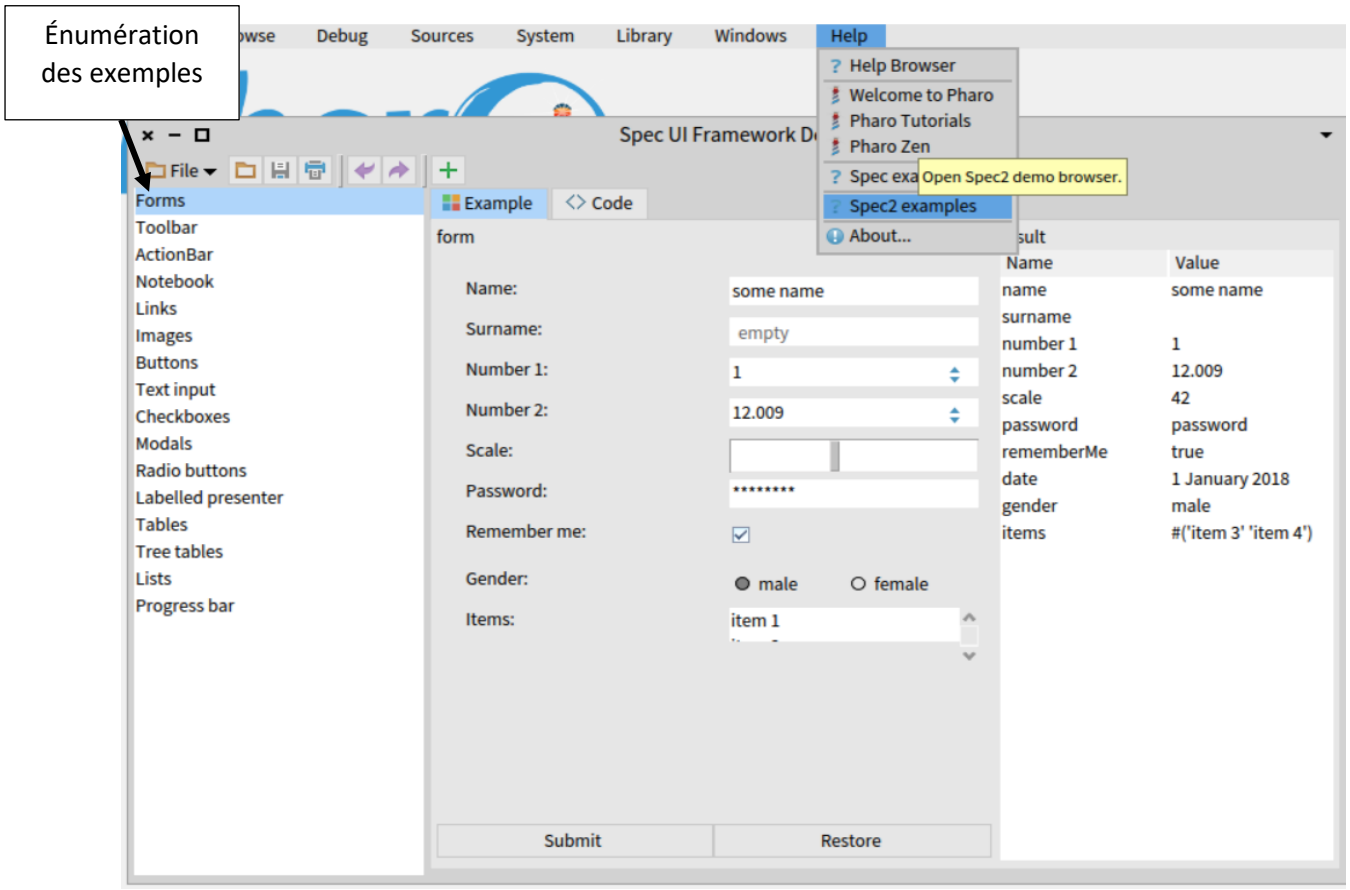

Figure 26 : Affichage de la fenêtre contenant les exemples Spec2

À gauche de la fenêtre, les exemples sont énumérés (voir figure 26). Parmi eux, des exemples de listes, de tableaux, de barres progressives, de champs de saisies ou encore de buttons sont mis à disposition. Ces exemples permettent au développeur de connaitre les différentes possibilités qu'offre Spec2.

2 Présentation technique du projet

2.1 Localisation du projet

Mon projet est intégré dans le GitHub de Molecule[3]. Pour télécharger Molecule, il suffit d'exécuter dans le *Playground* l'expression visible en figure 27.

```
Metacello new
    baseline: 'Molecule';
    repository: 'github://OpenSmock/Molecule';
    load.
```

Figure 27 : Expression permettant d'installer Molecule

Pour télécharger l'outil que j'ai développé pendant mon stage, il suffit d'exécuter dans le *Playground* l'expression visible en figure 28.

```
Metacello new
    baseline: 'MoleculeIncubator';
    repository: 'github://OpenSmock/Molecule';
    load.
```

Figure 28 : Expression permettant d'installer l'outil de mon stage

Concernant l'emplacement de l'outil, le projet se trouve dans deux *packages* ; le premier appelé « *Molecule-IDE-Incubators* » contient toutes les classes et le deuxième nommé « *Molecule-IDE-Incubators-Tests* » contient les tests.

[3] **Molecule** : voir rapport général : section 1.4

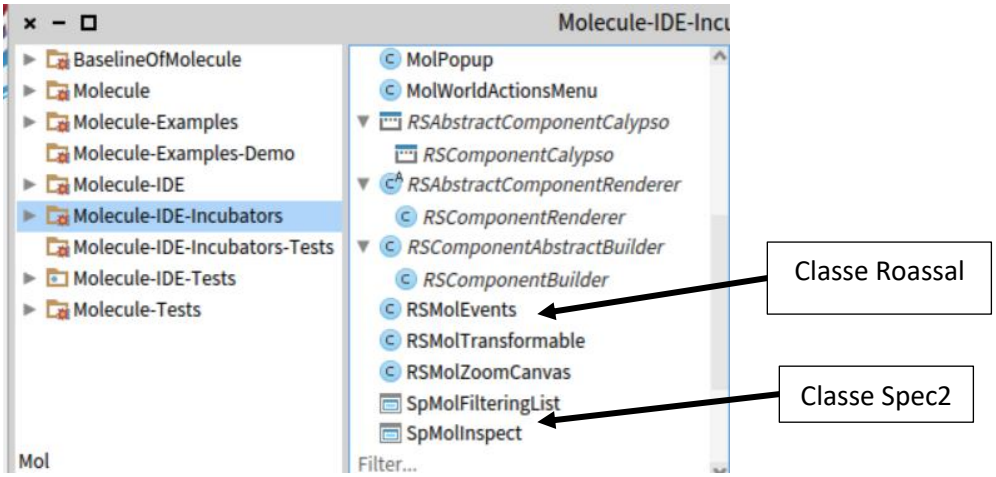

Figure 29 : Présentation du package "Molecule-IDE-Incubators"

À l'intérieur du premier *package*, j'ai défini une vingtaine de classes, dont sept sont des fenêtres Spec2 (classe avec l'icône fenêtre et l'abréviation « Sp » devant le nom) et neuf sont des classes Roassal (avec l'abréviation « RS » devant le nom).

Par exemple, sur la figure 29, la classe RSMolEvents est une classe Roassal et SpMolInspect est une classe Spec2.

Chaque classe est rangée dans une catégorie au sein du *package*. Les catégories sont Contract, Events, Icon, Interaction, Legend, Manifest, Menus, Popup, Tab Builder, Tab Calypso, Tab Renderer et Window Spec2.

À l'intérieur du deuxième *package*, j'ai créé des classes de tests (voir figure 30). Ces classes vont contenir des méthodes de tests effectuées sur les méthodes contenues dans les classes du premier *package*.

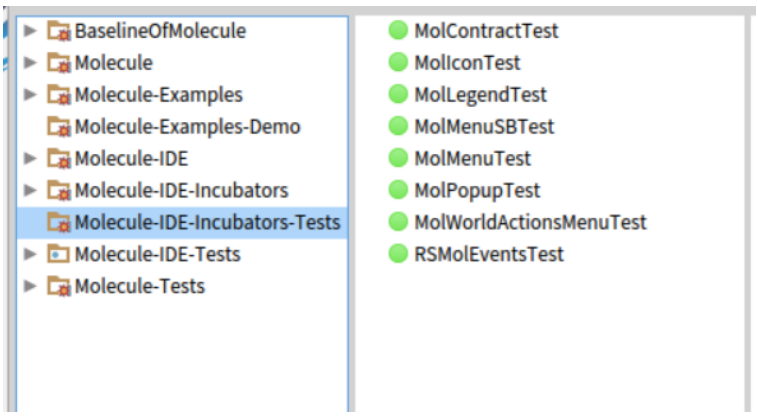

Figure 30 : Présentation du package "Molecule-IDE-Incubators-Tests"

Sur la figure 30, l'ensemble des tests sont verts, cela signifie qu'ils ont été réussis avec succès.

2.2 Fenêtre de recherche des composants

Dans le cadre du projet, j'ai réalisé une fenêtre permettant de rechercher et visualiser l'ensemble des composants.

Cette fenêtre est accessible via un menu contextuel appelé « *Search* » issu du menu principal « Molecule » appartenant au « *World* » menu (voir figure 31).

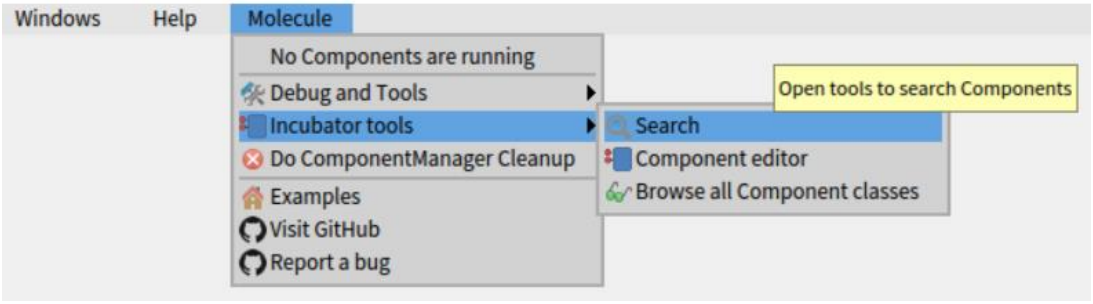

Figure 31 : Menu contextuel de l'outil

Le script de ce menu est le suivant. Il est présent dans la classe MolWorldActionsMenu.

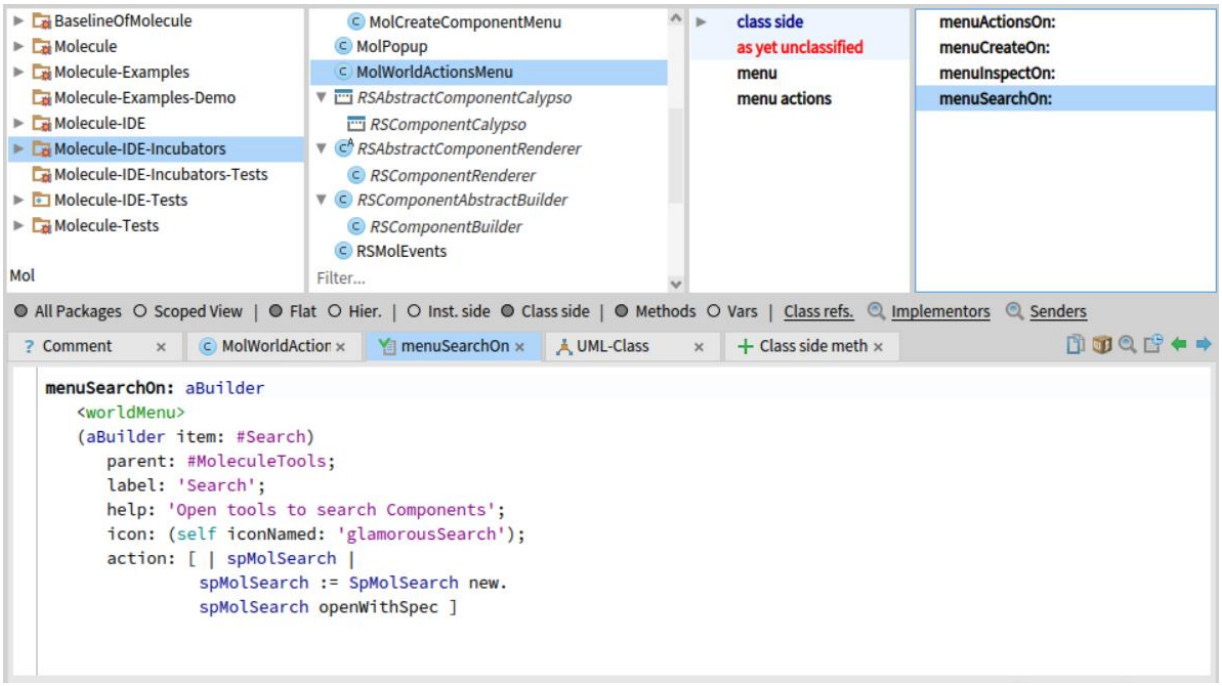

Figure 32 : Script du menu "Search"

Dans ce script (voir figure 32), le but est d'insérer un nouveau menu appelé « *Search* ». Pour cela, j'ai défini le menu parent (c'est-à-dire le menu « *Incubator tools* »), le label, l'icône, le *tooltip* (*help*) et l'action. Lorsque l'utilisateur clique sur ce menu, *spMolSearch*, une variable temporaire définie comme une instance de la classe SpMolSearch, se crée puis s'ouvre puisqu'il s'agit d'une fenêtre Spec2.

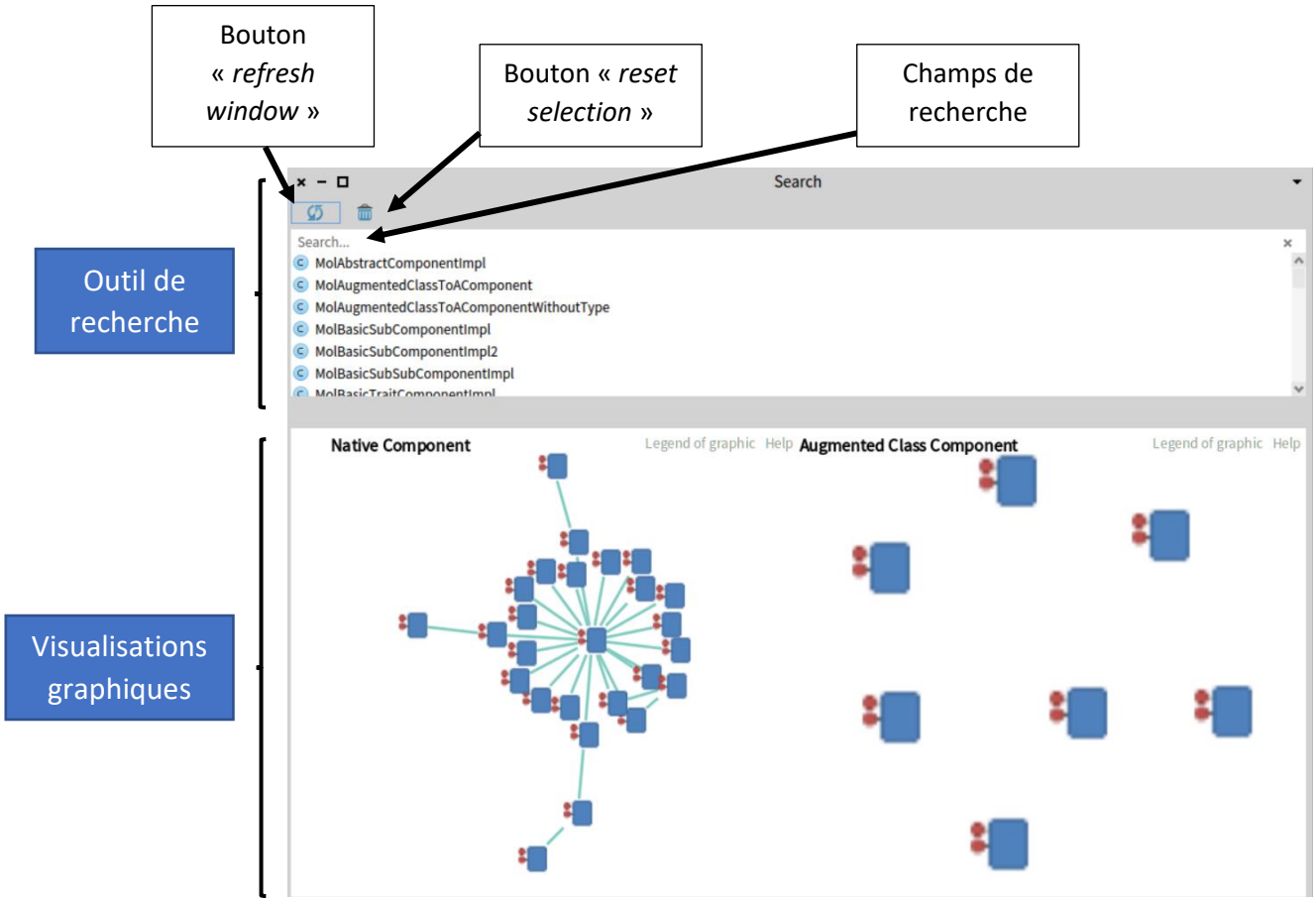

Figure 33 : Fenêtre de recherche des composants

Sur la figure 33, deux parties se dégagent dans la fenêtre.

La première représente un outil de recherche. L'utilisateur peut sélectionner dans la liste déroulante un composant. Il peut également rechercher le nom du composant via le champ de recherche, la liste s'actualisera en conséquence. Puis, deux boutons sont situés en haut de l'outil de recherche. Le premier sert à rafraichir la fenêtre c'est-à-dire que les données vont s'actualiser. Le deuxième permet de désélectionner le composant en cours de sélection.

La seconde partie contient deux graphiques. Ces derniers comportent des icônes représentant les composants. L'un affiche des composants dont leur super classe est

MolAbstractComponentImpl et l'autre affiche les composants issus d'une classe qui a été transformée. Cela résume les deux chemins d'implémentation d'un composant[4].

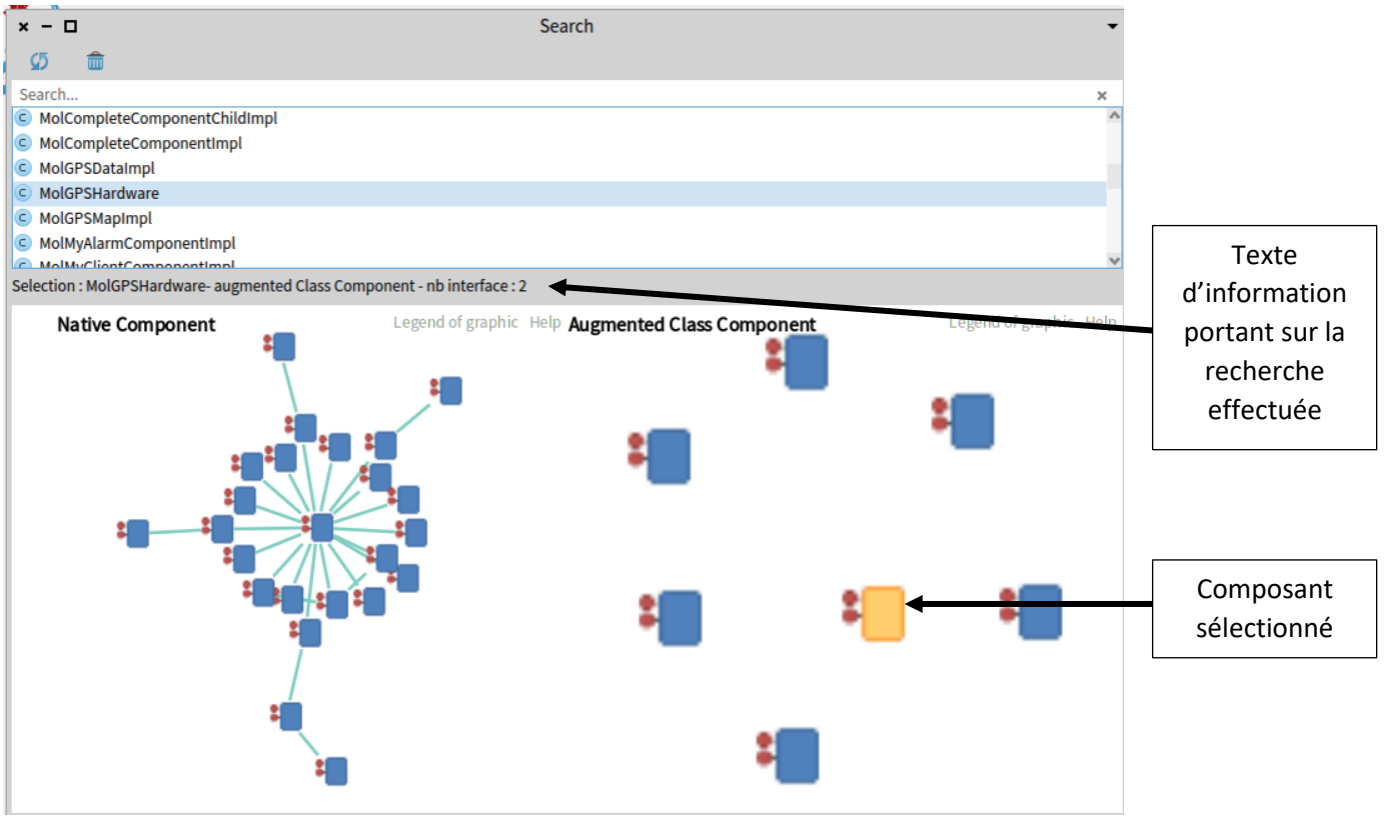

Figure 34 : Fenêtre de recherche des composants (action : un composant est sélectionné)

Lorsque l'utilisateur sélectionne un composant dans l'outil de recherche, un texte d'information sur ce composant apparait. Celui-ci permet de garder une trace de la recherche effectuée. Par ailleurs, les graphiques se modifient en fonction de la sélection. En effet, le composant sélectionnée change de couleur (il devient orange). Le contraste des couleurs va permettre à l'utilisateur d'identifier directement la localisation du composant sélectionné dans le graphique.

[4] **deux chemins d'implémentation d'un composant** : voir rapport général : figure 10

Le script suivant présente l'initialisation des éléments dans cette fenêtre (figure 33).

```
initializePresenters

    "define canvas"
    componentCanvas := self instantiate: SpRoassalPresenter.
    classAugmentedCanvas := self instantiate: SpRoassalPresenter.
    "define label"
    updatelabel := self newLabel.
    "define list and filtering input"
    droplist := self instantiate: SpMolFilteringList.
    filterInputPresenter := droplist filterInputPresenter.
    filterInputPresenter cursorPositionIndex.
    listPresenter := droplist listPresenter.
    "define toolbar"
    toolbar := self instantiate: SpToolbarPresenter.
```

Figure 35 : Initialisation des éléments de la fenêtre de recherche des composants

Dans ce script (voir figure 35), les deux variables *componentCanvas* et *classAugmentedCanvas* représentent les deux graphiques. La variable *updateLabel* concerne le texte d'information lorsqu'un composant est sélectionné (voir figure 33). L'outil de recherche est défini par les variables *droplist* et *listPresenter*. Enfin, la variable *toolbar* affiche les deux boutons.

Il est possible pour l'utilisateur d'effectuer un clic droit sur n'importe quelle icône. Un menu contextuel va alors apparaitre (voir figure 36).

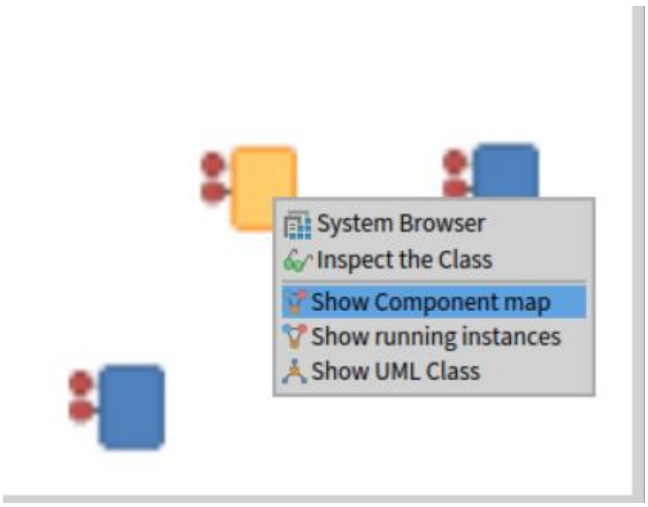

Figure 36 : Menu contextuel sur une icône

Parmi les menus proposés, le menu intitulé « *Show Component map* » va permettre d'afficher la prochaine fenêtre.

Ce menu est également accessible via un clic droit sur une classe d'un composant au sein du *System Browser* (visible en figure 37).

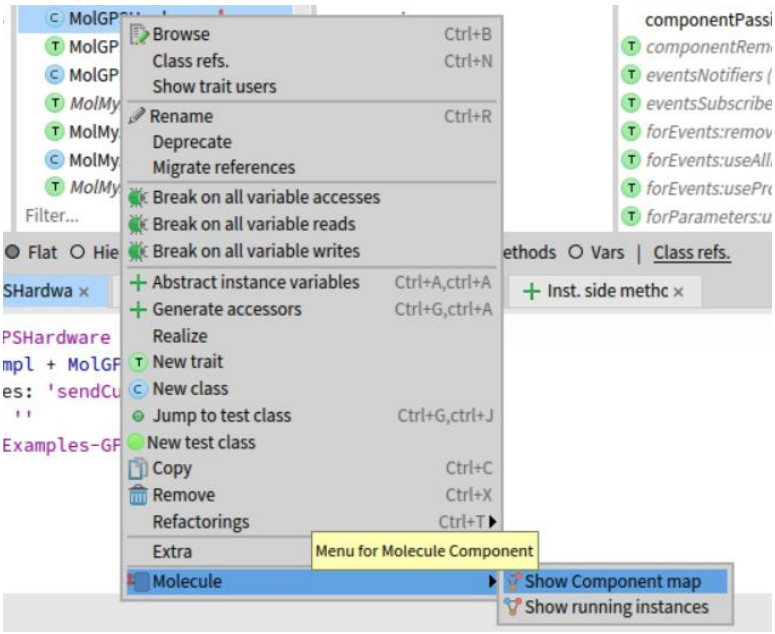

Figure 37 : Menu contextuel sur une classe de composants dans le System Browser

2.3 Fenêtre de visualisation d'un composant

2.3.1 Composant

Cette fenêtre permet d'accéder à la vue graphique détaillée du composant sélectionné lors de la vue précédente. Pour pouvoir récupérer la valeur du composant sélectionné, j'ai utilisé le débogueur.

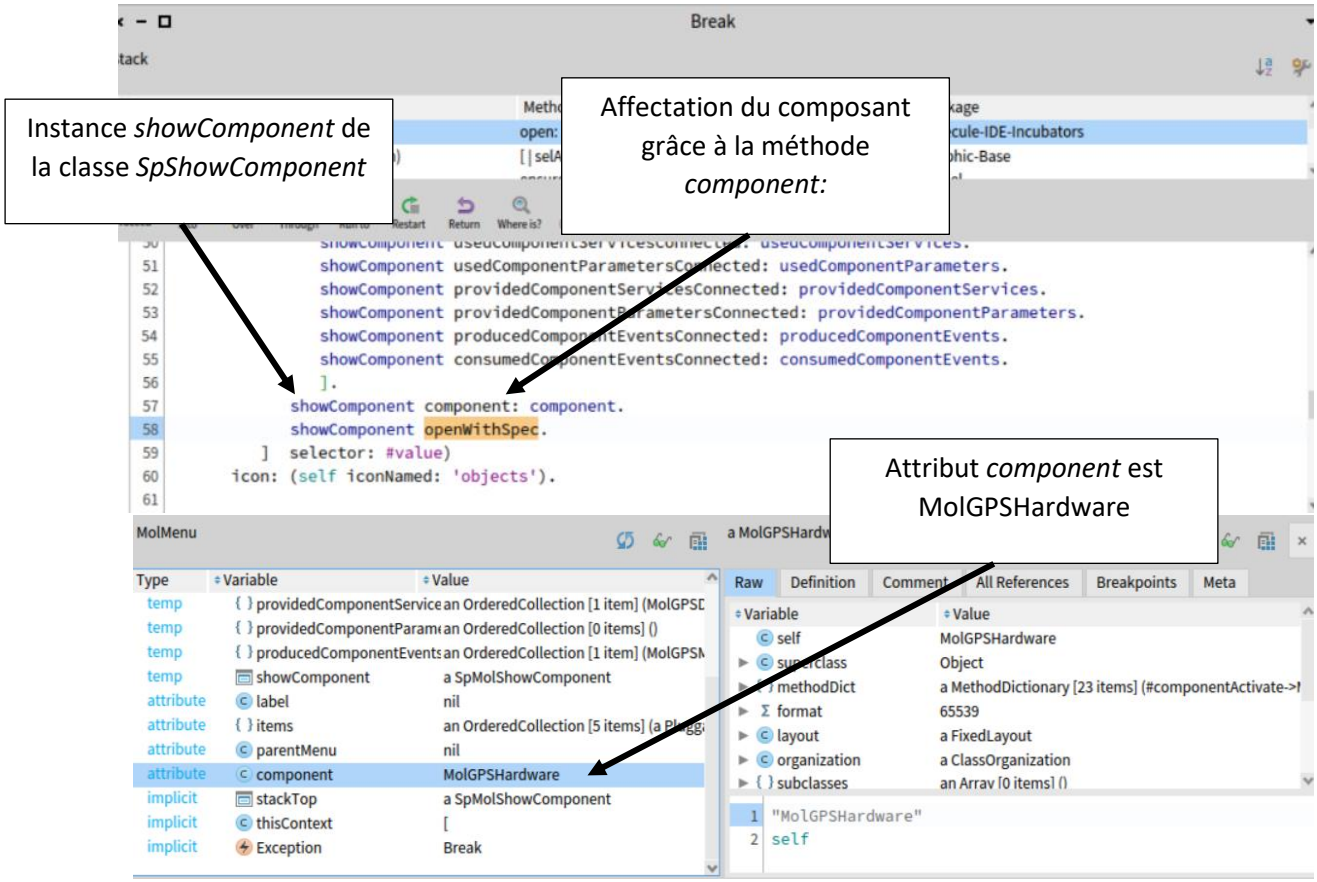

Figure 38 : Ouverture du debogueur d'un point d'arrêt

À partir du menu contextuel, je crée une instance de la classe *SpShowComponent* appelé *showComponent* (instance qui ouvre la vue détaillée du composant). J'appelle une méthode nommée *component:* qui va affecter la valeur du composant MolGPSHarware dans une variable appelé *component* présente dans l'instance *showComponent*. Afin de vérifier que le composant est bien réceptionné dans cette instance, j'effectue un point d'arrêt juste avant que la fenêtre ne s'ouvre. Dans le débugueur visible en figure 38, nous remarquons que l'attribut *component* possède la valeur du composant MolGPSHardware. Donc, le composant a bien été réceptionné, sa valeur est présente dans le menu et dans la fenêtre.

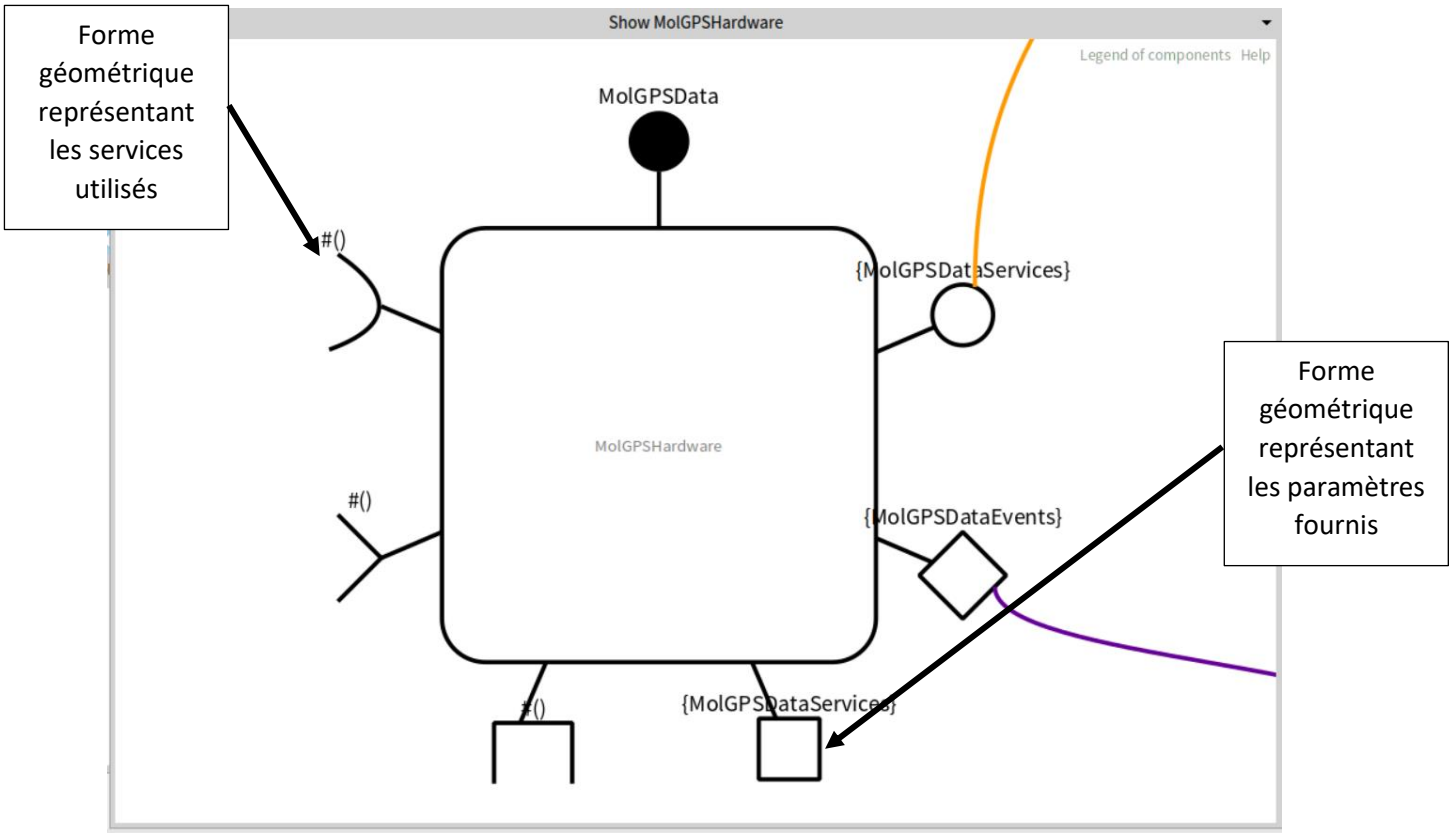

Figure 39 : Visualisation du composant MolGPSHardware

Sur cette fenêtre, le composant est affiché graphiquement (voir figure 39). Son *Type* et son contrat apparaissent. Ce modèle graphique existait déjà avant mon stage. J'ai reproduit ce modèle avec Pharo.

Deux méthodes permettant d'établir cette vue sont présentées ci-dessous.

Comment × SpMolShowCon × boxProvidedPai ×

```
boxProvidedParameters:model
    boxProvidedParameters := RSBox new.
    self
        shape: boxProvidedParameters
        color: Color white
        model: model
        translateTo: 15@35
        borderColor: Color black.
    self borderWith: boxProvidedParameters.
    boxProvidedParameters @ (self labeled).
```

Figure 40 : Affichage de la méthode boxProvidedParameters:

La première méthode permet de définir la forme géométrique des paramètres fournis (*ProvidedParameters*) visible sur la figure 40. Elle prend en paramètre une variable nommée *model*. Dans cette méthode, un nouvel objet de type RSBox appelé *boxProvidedParameters* est instancié. Cette méthode va appeler d'autres méthodes de la classe (acronyme « *self* » suivit du nom de la méthode) permettant de définir la couleur, la taille, les bordures et la position de l'objet. La variable *model* va également être affectée au modèle de l'objet.

Figure 41 : Affichage de la méthode svgUsedServices:

La deuxième méthode permet de définir la forme géométrique des services utilisés (*UsedParameters*) visible sur la figure 39. Elle prend également en paramètre une variable nommée *model*. Dans cette méthode, un nouvel objet de type RSSVGPath appelé *svgUsedServices* est instancié. Cet objet est ensuite personnalisé. Sa bordure est créée, il s'agit de l'objet RSBorder. Elle est de couleur noir et a une épaisseur de 0.5. Ensuite, la forme de *svgUsedServices* est décrite. La variable *model* va également être affectée au modèle de *svgUsedServices*.

2.3.2 Cartographie

Il est également possible pour l'utilisateur de visualiser les composants avec lesquels le composant sélectionné interagit (voir annexe 42). En effet, lorsque l'utilisateur diminue l'échelle de la fenêtre d'affichage, il peut apercevoir plus d'informations.

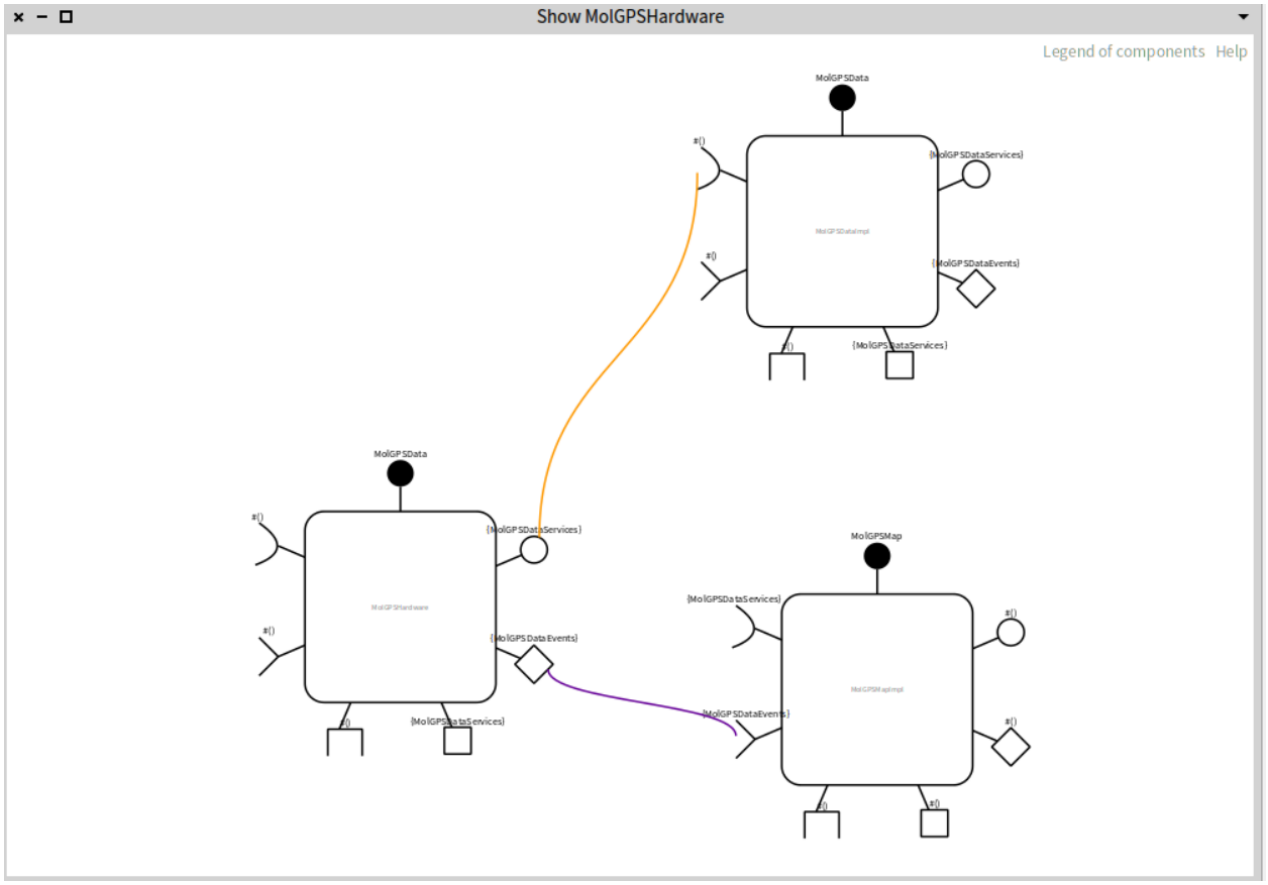

Figure 42 : Cartographie des composants

Les composants sont reliés grâce à leurs interfaces. Chaque lien est associé à un code couleur en fonction de l'interface. Ces différentes couleurs permettent de rendre la cartographie plus lisible.

Une partie du script permettant de créer la cartographie est présentée ci-dessous.

```
arrayOfComponents: interfaces shape: aShape collection: collection
   | composable array |
   array := RSComposite new.
   interfaces notNil ifTrue: [
      array shapes: ((1 to: interfaces size) collect: [ :n |
         composable := RSComposite new.
         composable
            model: (interfaces at:n);
            draggable;
            shapes: (self shapeModel: (interfaces at:n))
            yourself.
         self popup: (interfaces at:n).
         self event: composable.

         collection == toProvidedServices ifTrue: [
         composable position: (-250)+(n*(70))@(n*(-50)).
         composable shapes do: [:node | (node == ellipseProvidedServices) ifTrue: [ toProvidedServices
 add: node ]]].
```

Figure 43 : Script permettant de visualiser les composants et les liens associés

Dans le script visible en annexe 43, j'ai notamment utilisé les *Collections* (Liste) et les *Arrays* (tableaux). Ce sont des éléments pratiques pour afficher plusieurs fois la même chose mais avec des paramètres différents. Ici, les composants sont dans un tableau appelé *array* et les liens sont dans la liste appelé *collection*.

Conclusion

L’outil développé durant le stage est actuellement fonctionnel, cependant, des évolutions et améliorations sont envisageables.

Par exemple, concernant la vue détaillée des composants, les liaisons entres les composants pourraient être plus lisibles. Les formes géométriques correspondant aux interfaces pourraient s’assembler. Aussi, ces interfaces qui n’offrent rien ou qui ne demandent rien pourraient être supprimées graphiquement, cela faciliterai la compréhension.

Tables des illustrations