Mon premier programme

Hilaire Fernandes - cd40.tice-bordeaux.fr
Stéphane Ducasse – ducasse@iam.unibe.ch

Dans ce deuxième article sur Squeak, nous vous présentons quelques aspects liés au développement. Plus précisément, nous nous intéressons à la syntaxe du langage Smalltalk et à son environnement de développement. Comme prétexte à notre exploration, nous nous posons le problème simple de jouer des notes de musique à partir de la notation Do, Ré, Mi,...

Découverte de l'environnement de développement

Pour commencer, nous créons un nouveau projet Morphic vide. Dans Squeak, un projet est un espace dans lequel nous pouvons travailler sans interférer avec le reste de l'environnement. Un **`clic-gauche`** sur le fond de l'espace Squeak fait apparaître le menu du ***Monde***, choisissez l'entrée ***ouvrir*** puis ***nouveau projet morph***. Une petite fenêtre s'affiche alors, c'est le point d'entrée, la vue vers votre nouveau projet. Par défaut, il est nommé ***Sans titre1***, changer son nom en ***dorémi*** par exemple. Pour rentrer dans le projet, cliquez directement dans la petite vue du projet. Ça y est vous êtes dans votre premier projet. À tout moment, vous pouvez sauver l'état de votre travail, par un clic sur ***enregistrer*** dans le menu du ***Monde***. Cette opération fait une sauvegarde de toute l'image et préserve exactement l'environnement dans l'état où il est.

Pour commencer, nous allons nous familiariser un peu avec la syntaxe de Smalltalk. Pour ce faire rien de tel que l'utilisation d'un outil simple pour éditer et évaluer in situ de petits morceaux de code. Sur la droite se trouve un onglet ***Outils***, tirez le vers la gauche, et faites un tirer-déposer de l'outil ***Espace de travail*** dans le fond de votre projet. L'espace de travail (***workspace*** dans la langue de Shakespeare) est un éditeur dans lequel des programmes ou expressions Smalltalk peuvent être édités et exécutés.

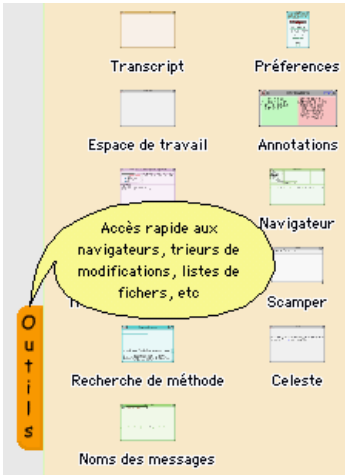

Figure 1 – Onglet de quelques outils de Squeak

Quelle heure est-il doc ?

Dans la nouvelle fenêtre de l'éditeur, saisissez l'expression suivante :

WatchMorph new openInWorld.

Sélectionnez l'expression puis faites la combinaison des deux touches **`[Alt]+d`**. Une montre apparaît alors.

Que c'est-il passé ?

Smalltalk est un langage **où tout est** objet (chaînes de caractères, souris, éditeur, compilateur, écran, nombres, booléens, ... tout quoi), même les classes. De plus la moindre instruction revient à envoyer des messages à des objets : les objets communiquent entre eux exclusivement par envoi de messages. Quand un objet reçoit un message, la méthode ayant le même nom est cherchée dans la classe de l'objet et ses superclasses, puis exécutée.

Ici la première expression **`WatchMorph new`** est exécutée. La classe **`WatchMorph`** (qui est un objet représentant une classe), reçoit le message **`new`**. Cela a pour conséquence d'invoquer la méthode **`new`** qui crée une montre -- un objet (ou instance de la classe **`WatchMorph`**) et la retourne. Celle-ci reçoit alors comme message **`openInWorld`** qui a pour effet de l'afficher.

La commande **`[Alt]+d`** demande à Squeak d'exécuter l'expression où est placé le curseur. Il est aussi possible de sélectionner un morceau de code à la souris et de demander son exécution avec cette commande. Le menu contextuel de l'éditeur s'appelle par un clic sur le bouton du milieu de la souris, depuis ce menu l'entrée ***Exécuter*** permet aussi d'exécuter un morceau de code Smalltalk. Pour exécuter une expression dans l'espace de travail, il faudra toujours utiliser l'une ou l'autre de ces méthodes.

Maintenant, nous allons utiliser une variable pour manipuler interactivement notre objet. Détruisez d'abord la montre à l'écran (clic droit sur celle-ci puis poignée de fermeture), entrez ensuite le code suivant :

```
montre := WatchMorph new.
montre openInWorld.
```

Sélectionnez les deux lignes de code puis faites **`[Alt]+d`** pour exécuter le code. Une autre montre s'affiche. Le point en fin de ligne, sert à séparer deux expressions (affectation, envoi de messages). L'affectation se fait avec le symbole **`:=`**. Nous avons créé une instance de la classe **`WatchMorph`** et l'avons affectée dans la variable **`montre`**. En Smalltalk, les variables ne contiennent que des références vers des objets, un peu comme des pointeurs en C, mais bien plus sympathique à utiliser pour le développeur. Il est possible d'avoir plusieurs références vers un même objet. Enfin dès qu'un objet n'est plus référencé, celui-ci est nettoyé par le ramasse-miettes. Il n'y a donc pas besoin de se soucier de la libération des espaces mémoires, elle se fait automatiquement. Enfin, dans la deuxième ligne le message **`openInWorld`** est envoyé à l'objet référencé par la variable **`montre`**.

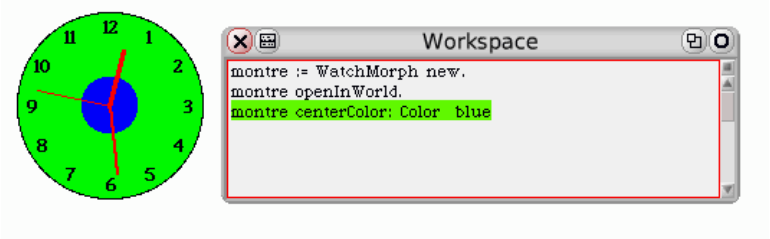

Figure 9 – Bricolons notre montre !

Démontons la montre !

C'est ici que cela devient intéressant. Maintenant que nous avons notre horloge, nous aurions envie d'en savoir un peu plus sur elle, histoire de s'amuser un peu avec. Comme en Squeak tout le code est disponible nous allons l'étudier. Pour cela nous utilisons un navigateur de classes (browser en anglais) un outil qui est le pivot central entre le développeur et Squeak.

Sélectionnez à la souris le nom de classe **`WatchMorph`** dans l'éditeur, puis faites **`[Alt]+b`**. Une nouvelle fenêtre s'affiche avec plusieurs colonnes, c'est le navigateur de classes qui s'est ouvert sur la définition de la classe **`WatchMorph`**.

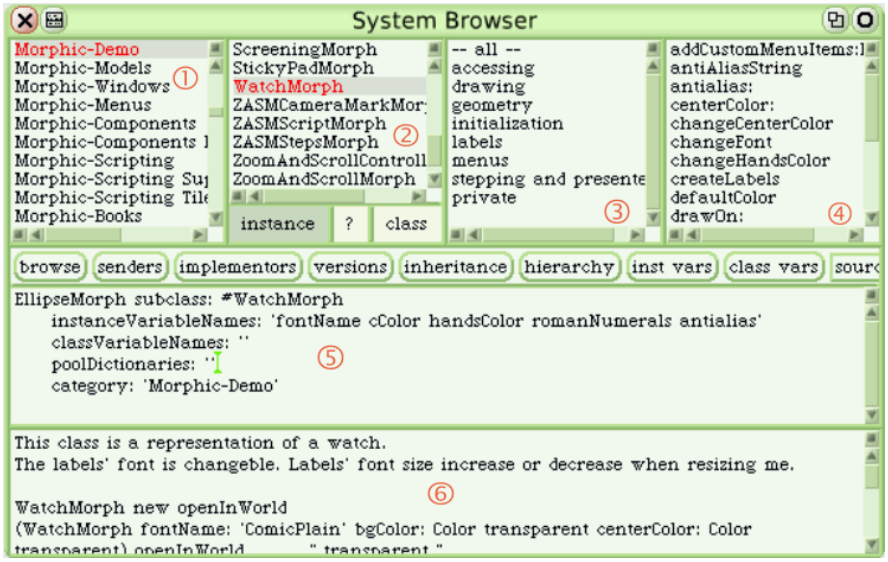

Figure 2 – Le navigateur de classe ouvert sur la définition de classe de ***WatchMorph***

Dans la partie haute du navigateur de classes se trouvent quatre zones. La zone 1, à gauche, affiche la liste des catégories de classes – une catégorie est juste un dossier contenant des classes pour permettre de les organiser. On y trouve les catégories définies par l'utilisateur mais aussi celles du système. La zone 2 regroupe les classes définies dans la catégorie sélectionnée en zone 1. Comme nous avons ouvert le navigateur sur la classe **`WatchMorph`**, celle-ci est sélectionnée dans la zone 2. Remarquez que cette classe est rangée dans la catégorie de classe **`Morphic-Demo`**. En zone 5 s'affiche la définition de la classe ou de la méthode sélectionnée. Ici sur la photographie écran (Fig. 2), c'est la définition de la classe. La zone 3 liste les catégories de méthode définies dans cette classe (comme pour les catégories de classes, c'est juste un dossier pour organiser les méthodes de la classe). Enfin dans la zone 4, la liste des méthodes de la catégorie sélectionnée, ou si aucune catégorie n'est sélectionnée l'ensemble des méthodes de la classe. Cliquez sur ces différentes rubriques pour constater les changements, essayez de les interpréter. La définition d'une classe comprend la classe ancêtre qui est raffinée et les variables d'instances ou attributs représentant l'état des objets que la classe créera. Ici la classe **`WatchMorph`** raffine la classe **`EllipseMorph`** et définit les variables d'instance **`fontName cColor handsColor`**...

Enfin cliquez sur la catégorie de méthode **`accessing`**. Dans cette catégorie sont très généralement regroupés les accesseurs aux variables d'instance des objets, ils permettent de modifier l'état interne de l'objet. Par exemple la méthode **`centerColor:`** permet de modifier la couleur du centre de la montre. Pour invoquer cette méthode, il faut envoyer un message ayant ce nom à une instance de la classe **`WatchMorph.`** Par exemple, tapez dans le workspace ouvert et exécutez l'expression suivante :

montre centerColor: Color red.

Cet exemple montre comment différents messages sont exécutés. En Smalltalk il y a plusieurs sortes de messages : les messages unaires, binaire et à mots clés. **`Color red`** est un message unaire. Ces messages sont toujours évalués en priorité, ils sont de la forme **`<objet> <nom du message>`**. Le résultat de ce message est un objet représentant la couleur rouge. Le deuxième message est à mots clés, il est de la forme **`<instance de WatchMorph> centerColor: <instance de couleur>`**. Ils sont moins prioritaires dans l'ordre d'évaluation. Pour définir nous même une couleur, nous pouvons envoyer le message **`r:g:b`** à la classe **`Color`** :

montre centerColor: (Color r:0.5 g:0.5 b:1).

Les parenthèses autour du message sont importantes, sinon Smalltalk essayerait d'envoyer le message **`centerColor:r:g:b`** envoyé à **`montre`**,`message qui échouerait.`

Un peu d'audio dans Squeak

Commençons par nous intéresser aux capacités audio de la bibliothèque de classes de Squeak. Dans un

espace de travail exécutez l'expression :

((FMSound organ1) soundForPitch: 440 dur: 0.5 loudness: 0.8) play.

Cette expression, qui est en fait la succession de trois messages, crée un instrument **organ1** et le joue à une fréquence de 440 Hz, sur une durée de 0.5 s et un volume de 0.8. Nous aurions pu l'écrire sur plusieurs lignes :

```
1: instr := FMSound organ1.
2: note := instr soundForPitch: 440 dur: 0.5 loudness: 0.8.
3: note play.
```

Pour exécuter ce code, sélectionnez les trois lignes et faites **[Alt]+d**. Les points en fin de chaque ligne sont importants, ils séparent les différentes expressions.

À la ligne 1, le message **organ1** est envoyé à la classe **FMSound**. Comme l'indique le nom de la classe, un échantillon sonore est généré par modulation de fréquence, nous l'affectons dans la variable **instr**. À la ligne 2, une note est créée à l'aide de ce son. Le message envoyé à l'instrument (instance de **FMSound**) est **soundForPitch:dur:loudness:.** Enfin notez que les parenthèses autour de **FMSound organ1** sont facultatives puisque c'est un message unaire, prioritaire sur le message suivant. Pour en savoir plus sur la classe **FMSound** rien de plus simple avec le navigateur, sélectionnez son nom, puis faites **[Alt]+b** et voilà. Allez donc voir dans la partie **class** la méthode **organ1**. Vous verrez qu'il y a d'autres instruments qui ne demande qu'à être essayés !

Toujours depuis le navigateur et avec la classe **FMSound** sélectionnée, cliquez sur le bouton ***hierarchy***, un navigateur de hiérarchie s'ouvre alors sur cette classe, il vous montre les ascendants et descendants de la classe et il permet de naviguer très efficacement dans les différents éléments constituants de ceux-ci. Remarquez en particulier la classe **AbstractSound** qui est la classe mère de toutes les classes représentant les sons, visitez là ! Voyez la méthode +, qui permet de mixer deux notes.

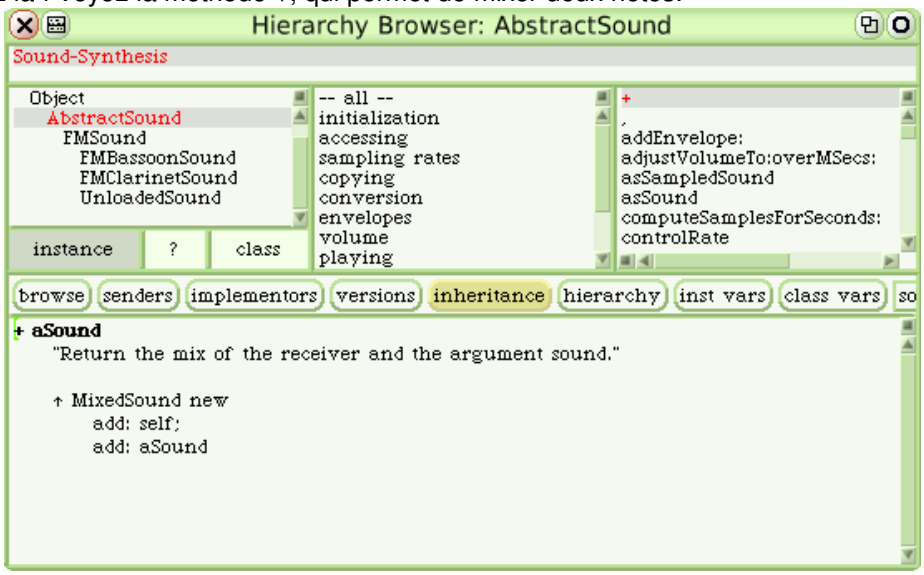

Figure 3 – Le navigateur de hiérarchie, un outil très utile pour s'y retrouver dans les relations entre classes.

Dans nos trois lignes de code précédentes, l'objet référencé par la variable **instr** est clairement identifiée quant à son type, mais que dire pour la variable **note**. Pour en savoir un peu plus sur cette variable, nous disposons d'un autre outil fort utile : ***l'inspecteur***. Sélectionnez la variable et faites **[Alt]+i** (vous pouvez appeler le menu contextuel de l'espace de travail par un **clic-bouton-milieu, puis** choisir le menu ***inspecter***). Une nouvelle fenêtre avec le titre **FMSound** s'affiche. Nous apprenons que **note** fait référence à un objet instance de la classe FMSound. Vous pouvez envoyer des messages à cet objet en utilisant la partie basse de l'inspecteur. Par exemple, vous pouvez obtenir la durée en exécutant l'expression **self duration** ou **self play.** Donc vous pouvez interagir directement avec l'objet que vous venez de créer.

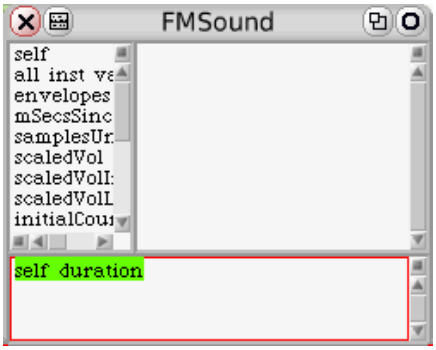

Figure 4 – Inspecter l'objet référencé par la variable ***note****.. Dans la partie basse, entrez* ***self duration*** *et exécutez/imprimez avec* ***[Alt]+p*** *pour connaître sa durée.*

Nous pouvons donc envoyer à **note** les messages des classes **FMSound** ainsi que celles de ses classes parentes. Dans l'espace de travail, vous pourriez par exemple définir une autre note dans une variable **note2** et jouer **(note+note2) play**, où **+** est le message défini dans la classe **AbstractSound**.

Ma première classe

Maintenant nous allons créer une classe **Gamme. D**ans celle-ci nous définissons une méthode **note:** qui reçoit comme argument un nom de note en notation française et retourne un objet note de type **FMSound**. La classe **Gamme** a deux variables d'instance : **tableEquivalence** et **instrument** pour contenir respectivement un dictionnaire d'équivalence entre la notation/fréquence des notes et un instrument utilisé par défaut pour définir les notes.

Nous allons commencer par créer une catégorie pour ranger notre classe, dans le menu (voir figure 5) des catégories, choisir ***add item...*** et saisissez par exemple ***ArticlesLinuxPratique***. Remarquez les autres entrées du menu.

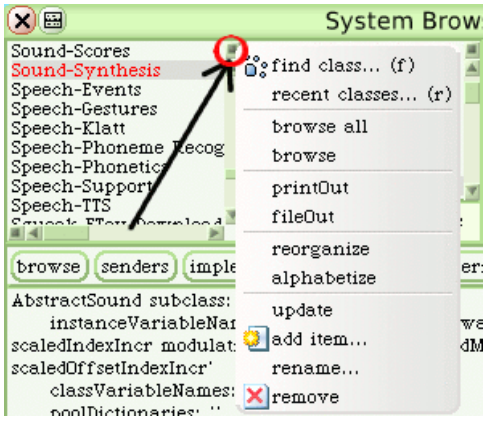

Figure 5 – Le petit bouton à droite des catégories de classe donne accès au menu

Dans la zone 5 du navigateur, vous avez un modèle de création d'une nouvelle classe. Si vous ne le voyez pas cliquez soit sur le nom de notre catégorie ou dans un espace vide de la zone 2 :

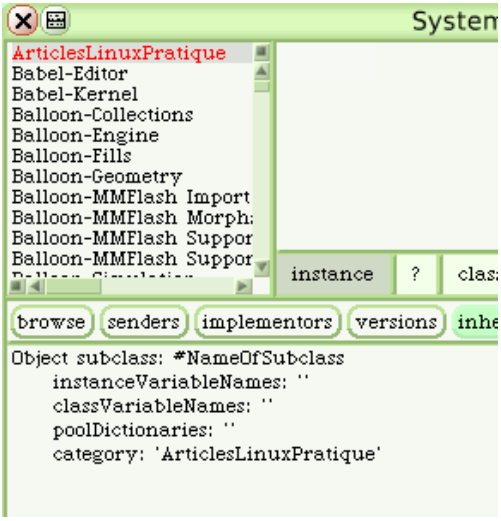

Figure 6 – Le modèle de création de classe

Dans le modèle, nous trouvons plusieurs éléments que nous substituons afin de créer la classe **`Gamme`**: **`Object subclass: #NameOfSubclass`** définit une sous-classe de la class **`Object`** qui se nommera **`#NameOfSubclass`**, ici nous remplaçons ce dernier par **`#Gamme`**. **`Object`** est la classe racine de la plupart des classes, comme nous ne souhaitons pas spécialiser une classe particulière nous laissons donc **`Object`**. La deuxième ligne du modèle permet de déclarer les variables d'instance (ou attributs), elles sont spécifiques à chaque nouvelle instance de la classe. Les variables de classe – ligne suivante – sont communes à toutes les instances (i.e., leur valeur est la même). Pour notre exemple nous nous contenterons de définir nos deux variables **`tableEquivalence`** et **`instrument`** comme variables d'instance. Au final, le modèle doit être édité comme ci-dessous :

```
Object subclass: #Gamme
        instanceVariableNames: 'tableEquivalence instrument'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'ArticlesLinuxPratique'
```

Pour valider et créer la classe faites ensuite **`[Alt]+s`** (sauver). À y regarder de plus près vous avez peut-être compris que le modèle est en fait l'envoi du message **`subclass:instanceVariableNames:classVariableNames:poolDictionaries:category:`** à la classe **`Object`**. Nous avons bien dit que Smalltalk est purement objet !

Passons maintenant à la définition des méthodes de la classe. La méthode initialisant toute nouvelle instance de classe se nomme **`initialize`**, cette méthode est appelée lorsque nous créons une instance en envoyant le message **`new`** à une classe. Nous allons y définir l'instrument de musique par défaut et la table d'équivalence note/fréquence. Dans le navigateur de classe cliquez dans la zone 3 jusqu'à obtenir le modèle de création de méthode ci-dessous :

```
message selector and argument names
        "comment stating purpose of message"
        | temporary variable names |
        statements
```

Remarquez la définition des variables locales, il suffit de placer les déclarations entre **`|x n|`**. Les commentaires sont placés entre **`"mon commentaire"`**

Remplacez l'ensemble de ces instructions par :

```
01: initialize
02:     super initialize.
03:     tableEquivalence := Dictionary new.
04:     tableEquivalence at: 'do' put: 264; at: 'do#' put: 275; at: 're' put: 297;
05:             at: 'mib' put: 316.80; at: 'mi' put: 330; at: 'fa' put: 352;
06:             at: 'fa#' put: 371.25; at: 'sol' put: 396; at: 'sol#' put: 412.5;
07:             at: 'la' put: 440; at: 'sib' put: 475.2; at: 'si' put: 495;
08:             at: 'dof' put: 528.
09:     instrument := FMSound organ1
```

À la ligne 2, nous appelons la méthode d'initialisation de la classe mère, **`super`** recherche la méthode à exécuter dans la superclasse de la classe contenant la méthode utilisant super. Ligne 3 nous créons notre dictionnaire, nous envoyons le message **`new`** à la classe **`Dictionary`** ce qui a pour effet de créer une nouvelle instance de **`Dictionary`** (pour en savoir plus sur cette classe, comme d'habitude, sélectionnez à la souris son nom et faites **`[Alt]+b`** pour ouvrir un navigateur de classe dessus). Pour peupler notre dictionnaire, nous utilisons la méthode **`at:put`**, pour avoir une clé **`'la'`** associée à la valeur **`440`** nous écrirons : **`tableEquivalence at: 'la' put: 440.`** Nous avons choisi des clés de type chaîne de caractères, nous les mettons donc entre **`' '`**. Pour retrouver une valeur dans le dictionnaire nous utilisons la méthode **`at:`** ce qui donne : **`tableEquivalence at: 'la'.`**

Il est possible d'envoyer à un objet plusieurs messages en enfilade, c'est exactement ce que nous faisons dans les lignes 04-08 à l'aide du point-virgule (;).

Pour enregistrer la méthode, encore une fois faites **`[Alt]+s`**.

Bien que notre classe ne soit pas complètement implémentée, nous pouvons déjà créer des instances de celle-ci, nous compléterons par la suite sa définition, que nous pourrons immédiatement tester dans nos instances déjà existantes. Cette possibilité de développement incrémentale donne une très grande souplesse pour la validation des modèles, c'est aussi très confortable pour le développeur.

Dans notre espace de travail, créez une instance de gamme :

`gamme := Gamme new.`

Placez le curseur sur la ligne est faites un **`[Alt]+d`** ou bien même un **`[Alt]+p`** pour évaluer et imprimer la valeur retournée par l'évaluation, dans ce dernier cas vous devez voir afficher **`Gamme.`**

Et si nous inspections notre instance **`gamme`** ? Rien de plus facile, sélectionnez la variable et ouvrez l'inspecteur (**`[Alt]+i`**), vous voyez alors que les variables **`instrument`** et **`tableEquivalence`** sont correctement initialisées :

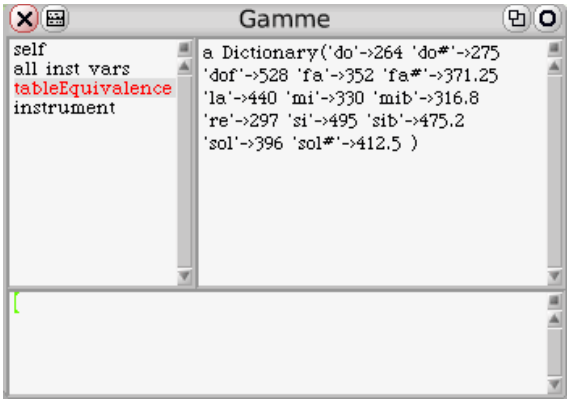

*Figure 7 – L'inspecteur sur une instance de **`Gamme`**. La variable d'instance **`tableEquivalence`** est correctement initialisée*

Retournons maintenant à notre navigateur de classe pour y ajouter notre méthode **`note:`**

```
01: note: maNote
02:     |frequence|
03:     frequence := tableEquivalence at: maNote ifAbsent: [100].
04:     ^ instrument soundForPitch: frequence dur: 0.5 loudness: 0.8
```

Enregistrez la méthode.

Nous déclarons une variable locale **`frequence`**. Lors de la déclaration d'une variable, il n'est pas nécessaire de préciser son type. C'est parfaitement inutile car toutes les variables font toutes la même chose : référencer un objet. Les variables sont en quelque sorte des pointeurs comme en C mais beaucoup moins archaïque, lorsque le programme sort de la portée d'une variable l'objet référencé par celle-ci est automatiquement nettoyé, si nécessaire, par le ramasse - miettes. Aussi cette gestion unifiée des variables, facilite énormément le travail du développeur. Beaucoup d'aspects qui dans d'autres langages sont gérés au niveau de la syntaxe le sont ici sous forme d'objets (donc dans les bibliothèques) : les chaînes de caractères (**`String`**), les nombres (**`Fraction`**, **`Integer`**, **`Float`**, **`Number`**,...), etc. La syntaxe de Smalltalk en est donc immédiatement plus simple et uniforme.

Retournons à notre méthode **`note:`**, la ligne 3 récupère la fréquence dans le dictionnaire, cependant au lieu

d'utiliser la méthode **`at:`** nous utilisons **`at:ifAbsent:`** . Comme vous vous en doutez, **`frequence`** est affectée du résultat de l'exécution des expressions contenues entre les **`[ ]`** si la note n'existe pas. Pour avoir une fréquence aléatoire, nous aurions pu écrire **`:`**

`frequence := tableEquivalence at: maNote ifAbsent: [(100 to: 150) atRandom].`

(Indice : naviguer dans la classe **`Random`**)

Maintenant retournons dans notre espace de travail, et créons une note : **`note1 := gamme note: 'do'.`** Pour jouer la note : **`note1 play`**. Vous pouvez créer plusieurs notes et les jouer à la suite : message '**`,`**' ou bien en polyphonie : message '**`+`**' . Bien sûr nous n'avons pas eu besoin de recréer une instance de **`Gamme`**. Nous avons utilisé celle déclarée précédemment, c'est la joie du développement incrémental.

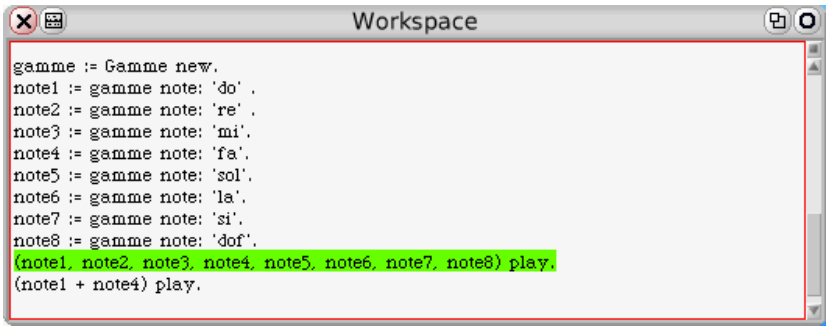

Figure 8 – Exemples d'utilisation de notre classe ***`Gamme`***

Vous devriez maintenant être capable de créer d'autres méthodes. Par exemple, il serait utile de pouvoir spécifier par une méthode **`note:dur:vol:`** la durée et le volume de la note. Changer d'instrument serait aussi intéressant.

Conclusions

Dans cet article, nous vous avons montré quelques un des outils de développement qui donne toute la puissance à cet environnement. Vous avez créé votre première classe et expérimenté le développement incrémental. Vous vous êtes frottés à la syntaxe de Smalltalk. Nous espérons que cet article vous a aidé à franchir le premier cap et qu'il vous donne envie d'aller plus loin dans votre exploration de cet environnement.

Dans notre prochain article, nous nous attarderons sur le syntaxe de Smalltalk et sa gigantesque bibliothèque de classes. Cela nous sera nécessaire pour ensuite poursuivre sur d'autres aspects excitants de Squeak comme le développement avec les Morphs et Seaside. D'ici là vous pouvez consulter quelques site d'informations sur le sujet.

Liens

Livres sur Smalltalk et Squeak : **www.iam.unibe.ch/~ducasse/FreeBooks.html**

Le site de Squeak: www.squeak.org/

Livre en français sur Squeak, X. Briffault et Stéphane Ducasse, Eyrolles, 2002

En particulier pour compléter cet article : *Smalltalk by Example: the Developer's Guide* et *DRAFS of Squeak : Open Personal Computing for Multimedia (le chapitre 2)*