3

Premiers développements en Squeak

Nous allons à présent aborder le développement en Squeak, et en manier les premiers éléments de syntaxe en mettant en œuvre un petit programme Squeak. Nous prendrons pour prétexte le développement d'un composant d'affichage d'horloge qui réutilise une classe de Squeak fournie en standard. Il s'agira plus précisément de doter cette horloge de la possibilité de passer sur demande d'un affichage au format américain à un affichage au format français, et de modifier la couleur de son affichage.

La création de classes

Pour commencer, nous allons créer une première classe. En Squeak, toute classe appartient à une *catégorie*, une sorte de dossier de classes qui les regroupe logiquement. Pour créer une telle catégorie, dans la zone catégories de classes d'un System Browser, utilisez l'option <add item> du menu contextuel. Appelez-la par exemple `Livre Squeak-Exemples`. Il n'y a pas de convention particulière à respecter pour les noms de catégorie, si ce n'est qu'il vaut mieux les choisir assez explicites.

Après avoir sélectionné la catégorie créée, la zone 5 du browser laisse apparaître le code suivant :

```
Object subclass: #NameOfSubclass
       instanceVariableNames: ''
       classVariableNames: ''
       poolDictionaries: ''
       category: 'Livre-Exemples'
```

Ce code est le canevas de création de classes, à partir duquel une nouvelle classe, sous-classe d'une classe existante, peut être créée.

Rappel sur l'héritage

Le mécanisme d'héritage permet de créer une classe en réutilisant les définitions de structure et de méthodes d'une autre classe. Si C1 et C2 sont liées par une relation d'héritage telle que C2 hérite de C1, on dira que C2 est une *sous-classe* de C1, tandis que C1 est la *superclasse* de C2. Lorsqu'une classe ne peut hériter que d'une seule classe, on parle d'héritage simple et le graphe d'héritage est un arbre. C'est le cas en Squeak. Nous aborderons en détail les mécanismes d'héritage de Squeak dans le chapitre suivant, « Le modèle objet de Squeak ».

L'exemple que nous allons développer part de la classe `ClockMorph` de Squeak, que nous allons réutiliser par héritage. Cette classe permet l'affichage d'une petite horloge sous la forme d'un item graphique mobile, appelé en Squeak un « morph », du nom de la classe `Morph` dont sont issus tous les items de ce type.

Remarque sur les morphs

Squeak propose deux types d'interfaces graphiques différentes : les interfaces de type MVC (Modèle-Vue-Contrôleur), héritées des anciennes versions de Smalltalk, et les interfaces de type morph, issues du langage Self initialement développé par Sun. Les morphs sont des objets qui appartiennent à l'une des sous-classes de la classe `Morph`. Le mécanisme d'affichage et de boucle d'activation des morphs sera détaillé dans le chapitre 10, consacré au développement d'interfaces graphiques.

Pour voir un exemple d'horloge `ClockMorph`, exécutez :

```
ClockMorph new openInWorld
```

Nous allons maintenant créer une sous-classe de `ClockMorph` que nous appellerons `MyClockMorph`. Cette nouvelle sous-classe nous permettra de modifier les caractéristiques de `ClockMorph`.

Méthodologie de développement

La réutilisation par héritage ou délégation de classes, ou de frameworks complets, est l'une des possibilités très intéressantes qu'offre la programmation orientée objet. Cette réutilisation est facilitée en Squeak par la qualité de l'environnement de développement, par le non-typage des variables, et par le nombre des classes déjà existantes. Avant de développer quelque code que ce soit, le développeur Squeak cherchera d'abord à identifier ce qui peut être réutilisé, que cela soit dans l'environnement standard, ou dans l'une des innombrables contributions des membres de la communauté des développeurs Squeak. (voir par exemple http:// www.squeak.org).

Pour créer `MyClockMorph`, dans la zone de saisie de code du System Browser que nous venons d'ouvrir sur la catégorie `Livre Squeak-exemples`, modifiez le canevas de création de classe de la façon suivante, en précisant la catégorie dans laquelle la nouvelle sous-classe doit être ajoutée (ligne 5) :

```
ClockMorph subclass: #MyClockMorph
           instanceVariableNames: ''
           classVariableNames: ''
           poolDictionaries: ''
           category: 'Livre-Exemples'
```

Validez ensuite cette modification en sélectionnant l'option <accept> du menu (bouton droit, ou Alt+s au clavier).

Notez que les noms de classes Squeak comprennent nécessairement une lettre initiale en majuscule et ne doivent contenir que des lettres (non accentuées), des chiffres ou le caractère de soulignement « _ ».

Conventions typographiques

Les parties significatives d'un nom de classe ou de méthode (par exemple, `My`, `Clock` et `Morph`) sont identifiées par des majuscules (plutôt que par des soulignés). Cela n'est pas imposé par la syntaxe, mais cette convention est respectée par tous les développeurs Squeak et rend le code plus lisible.

Une nouvelle classe, `MyClockMorph`, apparaît alors dans la zone Classes du browser. C'est sur cette classe que nous allons travailler.

À chaque étape son explorateur : l'explorateur hiérarchique

Les différents « explorateurs » Squeak

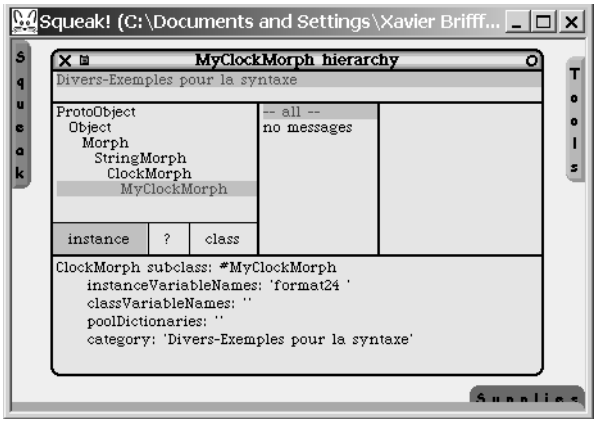
Il convient de bien distinguer les rôles et fonctions des différents outils d'exploration proposés par Squeak :

- les explorateurs de classe permettent d'accéder aux classes et aux méthodes, avec une structuration par catégorie (System Browser), ou en fonction de l'arbre d'héritage (Hierarchy Browser).
- les explorateurs d'objets permettent d'accéder à la structure interne des objets, que ce soit uniquement au premier niveau (Inspector), ou en ayant la possibilité de développer l'arbre des références aux objets (Explorer).

Il est maintenant utile, pour le développement de la classe `MyClockMorph`, d'en explorer la structure. Nous le ferons à l'aide d'un nouvel outil d'aide au développement, le Hierarchy Browser. Cet explorateur hiérachique permet d'accéder facilement à la hiérarchie d'une classe spécifique, et aux méthodes et variables définies dans ses superclasses.

Pour l'ouvrir, sélectionnez la classe `MyClockMorph` dans un System Browser, puis choisissez l'option <browse hierarchy> du menu contextuel (ou appuyez sur les touches Alt+h). Sur la figure 3-1 s'affiche la classe `MyClockMorph` et ses superclasses.

Figure 3–1
Un browser hiérarchique sur la classe `MyClockMorph`



Avec l'explorateur hiérarchique, comme avec tous les autres explorateurs, il est possible de créer de nouvelles méthodes. Il permet de surcroît un accès rapide aux méthodes existantes et à la structure des superclasses.

> **Environnement de développement**
>
> Design, développement architectural, optimisation d'une méthode, mise au point, débogage..., sont autant d'activités de développement nécessitant des outils différents. Squeak offre une palette d'outils répondant à ces besoins.

Spécialiser le comportement d'une classe

Telle qu'elle est implémentée, `ClockMorph`, la superclasse de `MyClockMorph`, affiche ses horloges en noir et au format anglophone (3:15 pm). Nous allons faire en sorte que `MyClockMorph`, la sous-classe de `ClockMorph`, puisse s'afficher dans d'autres couleurs et avec d'autres formats horaires. Pour ce faire, il nous faut d'abord comprendre comment est implémentée `ClockMorph`.

Zoom sur le comportement de la superclasse

En utilisant l'explorateur hiérarchique, il est facile de prendre connaissance des méthodes de `ClockMorph` (pour cela, cliquez sur la classe `ClockMorph`). Il y en a six. L'une d'elles va nous intéresser plus particulièrement. Il s'agit de la méthode `step` qui est responsable de l'affichage de l'heure dans le morph.

Code de la méthode `step` de la classe `ClockMorph`

```
step
    | time |
    super step.
```

```
time := String streamContents:
   [:aStrm | Time now print24: false
                                showSeconds: (showSeconds == true)
                                on: aStrm].
self contents: time
```

Nous allons analyser ce code, qui satisfait aux règles syntaxiques simples de définition de méthodes dans Squeak :

1. Le nom de la méthode, encore appelé *sélecteur*, est défini (ligne 1) .
2. Une variable locale est déclarée, entre des barres verticales (ligne 2). Notons que les variables ne sont pas typées : toute variable peut référencer tout type d'objet.
3. Commence ensuite le corps de la méthode :
   - En ligne 3, nous invoquons la méthode `step` sur `super` ; `super` est une variable qui représente le receveur du message `self` mais qui modifie la procédure de recherche de la méthode. En conséquence, cela revient à invoquer `step` sur `self`, l'objet lui-même, mais avec la définition de la superclasse.
   - En ligne 4, une chaîne (objet de classe `String`) est affectée à la variable `time` ; sa valeur est obtenue à partir du résultat renvoyé par le bloc de la ligne 5.
   - En ligne 5, on constate l'utilisation de la méthode `print24:` sur l'objet renvoyé par `Time now` (l'heure courante). Cette méthode détermine le format d'affichage. On remarque qu'il était prévu que le format d'affichage pût être paramétré, mais que `ClockMorph` ne fait pas usage de cette facilité car `print24:` a d'emblée comme valeur *`false`*.
   - En ligne 6, nous informons le morph du nouveau contenu qu'il doit afficher. La chaîne de caractères `time`, représentation sous forme de caractère de l'objet `Time now`, est affectée à `contents`.

Syntaxe

L'essentiel de la syntaxe de Squeak repose sur un principe simple : l'envoi de messages postfixés à des objets. Ici, le message `step` demande l'exécution de la méthode `step` à l'objet `super` ; de même, le message `now` est envoyé à la classe `Time` qui exécute alors la méthode `now` et renvoie l'heure courante. Réciproquement, toute opération est en fait un message qui déclenche une méthode.

Création de variables sur la classe MyClockMorph

Nous allons définir des variables d'instance dans `MyClockMorph`. Pour cela, nous allons définir dans notre classe `MyClockMorph` une nouvelle variable d'instance, `format24`, qui contiendra vrai ou faux selon que l'affichage doit se faire sur 24 heures ou sur 12. Pour définir cette variable d'instance, ouvrez la définition de la classe `MyClockMorph` dans un System Browser et modifiez-la comme suit :

Redéfinition de la classe `MyClockMorph`

```
ClockMorph subclass: #MyClockMorph
           instanceVariableNames: 'format24 '
           classVariableNames: ''
           poolDictionaries: ''
           category: 'Livre-Exemples'
```

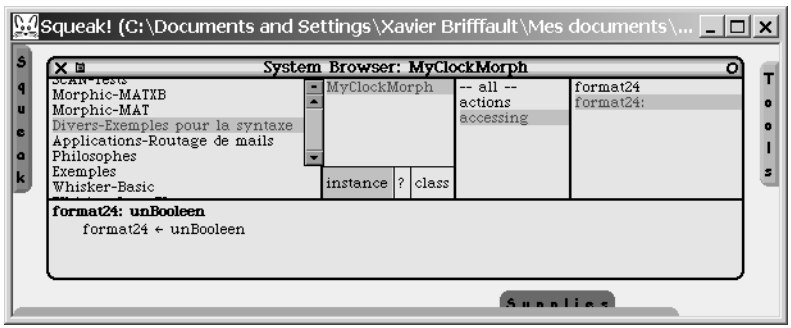
Enfin, validez les modifications.

Création de méthodes d'accès et d'affectation

Nous aimerions pouvoir modifier la valeur de `format24` à tout moment. Pour cela, nous allons définir une méthode d'accès (appelée *accesseur*), ainsi qu'une méthode d'affectation (appelée *mutateur*).

À l'instar des classes, les méthodes sont, nous l'avons vu, regroupées elles aussi dans des catégories logiques (protocoles), qui facilitent l'accès au code. Aussi faut-il d'abord créer la catégorie dans laquelle il convient de regrouper les deux méthodes à créer (voir figure 3-2).

Figure 3–2
Exemple de regroupement de méthodes dans un protocole



Pour créer une nouvelle catégorie de méthodes, cliquez avec le bouton droit sur la liste des protocoles de méthodes dans le System Browser, et sélectionnez l'option <new category>. Par convention, le nom de protocole choisi pour les méthodes d'accès est `accessing`. C'est donc le nom que nous allons utiliser. Le nouveau protocole apparaît dans la liste des protocoles.

Après l'avoir sélectionné, saisissez dans la zone de saisie de l'explorateur le code de la méthode d'accès `format24` présenté ci-après, puis validez les modifications :

Code de la méthode `format24` de la classe `MyClockMorph`

```
format24
      format24 isNil ifTrue: [format24 := false].
      ^format24
```

Toute méthode Squeak renvoie une valeur. Par défaut, c'est l'objet courant (`self`), qui est renvoyé. Mais ici, l'opérateur de renvoi `^` provoque la sortie de la méthode, en lui donnant comme valeur de sortie la valeur qu'il précède, en l'occurrence la valeur de la variable `format24`.

Dans le cas où la variable n'aurait pas été initialisée (elle est « `nil` »), la valeur `false` lui est affectée par défaut (ligne 2). Vous noterez la syntaxe du test de condition, qui se présente sous la forme d'un message envoyé à un objet booléen (`ifTrue:`).

Il convient en effet de rappeler qu'en Squeak, tout est objet. Les booléens ne font pas exception et sont des objets à part entière, comme l'est l'objet renvoyé par `format24 isNil`. C'est également le cas des blocs de code qui sont passés en argument, par exemple `[format24 := false]`.

Astuce

Notez bien la technique d'initialisation utilisée dans la méthode `format24`. Il s'agit d'une technique dite « d'initialisation paresseuse », dans laquelle un test systématique est effectué dans les méthodes d'accès à une variable avant de renvoyer sa valeur. Si la variable n'a pas encore été initialisée, elle l'est à ce moment. Cette technique garantit que la variable est correctement initialisée.

Saisissez à présent le code de la deuxième méthode, `format24:` puis validez les modifications :

Code de la méthode `format24:` de la classe `MyClockMorph`

```
format24: unBooleen
      format24 := unBooleen
```

La présence du caractère deux-points dans un nom de méthode implique qu'un argument est attendu lors de l'appel. Ce nom d'argument doit être placé après le caractère ':' et être suffisamment explicite pour indiquer quelle fonction va jouer cet argument.

Point de conception

Nous avons présenté ici les deux méthodes qui implémentent l'accesseur et le mutateur (`format24` et `format24:`) de la classe `myClockMorph`. Or, cette façon de procéder présente l'inconvénient de fournir un mutateur public, `format24:`. Puisque les variables ne sont pas typées, l'objet appelant peut fournir un argument erroné, qui ne soit pas un booléen. Une solution consisterait alors à tester systématiquement l'argument dans `format24:`, ce qui prémunit la méthode contre le passage de toute valeur incorrecte. Une autre solution, préférable, serait de ne pas fournir de mutateur, mais uniquement deux méthodes de bascule, `format24On` et `format24Off`, qui affecteraient à la variable les valeurs `true` et `false`, respectivement.

Spécialisation de la méthode step

Il ne reste maintenant plus qu'à définir la méthode `step` sur `MyClockMorph` (selon la méthode présentée pour la définition des méthodes `format24`).

Cette nouvelle méthode masquera la définition de la méthode `step` de `ClockMorph`, tout en la réutilisant, grâce à la pseudo-variable `super`, dont nous présenterons le fonctionnement en détail au chapitre 4 consacré au modèle objet de Squeak ; la méthode telle qu'elle est redéfinie utilisera la valeur contenue dans la variable `format24`, plutôt que de recourir systématiquement à la valeur `false` comme c'était le cas dans la définition initiale :

```
step
   | time |
   super step.
   Time := String streamContents:
      [:aStrm | Time now print24: self format24
                                  showSeconds: (showSeconds == true)
                                  on: aStrm].
   self contents: time
```

Exécutez maintenant dans un `Workspace`

```
(myClock := MyClockMorph new) openInWorld
```

L'instance créée par `MyClockMorph new` est affectée à la variable `myClock`, puis le morph correspondant est ouvert (`openInWorld`). Vous pouvez alors constater que… rien n'a changé. La couleur et le format d'affichage sont toujours les mêmes. Pourtant, si nous exécutons maintenant `myClock format24: true`, l'affichage passe immédiatement au format 24 h. Si nous exécutons à son tour `myClock color: Color red`, l'affichage passe bien au rouge.

En effet, bien que nous n'ayons fait aucune modification pour la gestion des couleurs, une rapide analyse de la classe `Morph` nous montre que la gestion des couleurs est déjà prévue à ce niveau, qu'elle est donc héritée par les sous-classes de `Morph`, donc par `MyClockMorph`, et qu'il n'est en fait pas nécessaire de développer quoi que ce soit.

Afin de vérifier la définition de la méthode `color`, explorez les méthodes du protocole `accessing` de la classe `Morph`.

L'environnement graphique : quelques informations supplémentaires

Tous les objets graphiques de type `Morph` disposent, à l'instar des horloges de la section précédente, de menus et de « halos » (un ensemble d'icônes de contrôle) pour les contrôler. Le menu associé à chaque morph s'obtient en cliquant avec le bouton gauche tout en maintenant la touche Ctrl enfoncée.

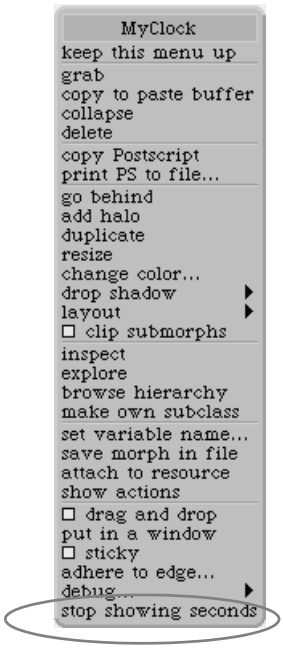
Voici par exemple le menu qui peut être obtenu sur une horloge `MyClockMorph` (voir figure 3-3).

Parmi les options disponibles, on trouvera les options de déplacement <grab>, de copie, de suppression, de gestion des couleurs, ainsi que des options relevant de la programmation (inspection et exploration, hiérarchie, actions associées, débogage).

On remarquera également le dernier item du menu, <stop showing seconds>, qui est spécifique du `ClockMorph`, à la différence des autres options, partagées par tous les morphs.

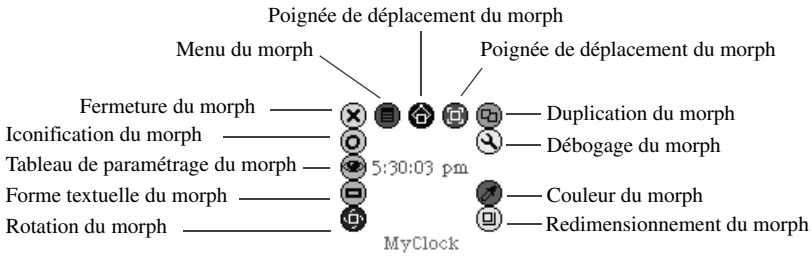
Le halo est une spécificité des interfaces Squeak, qui permet de contrôler les morphs.

Figure 3–3
Le menu associé à un `ClockMorph`



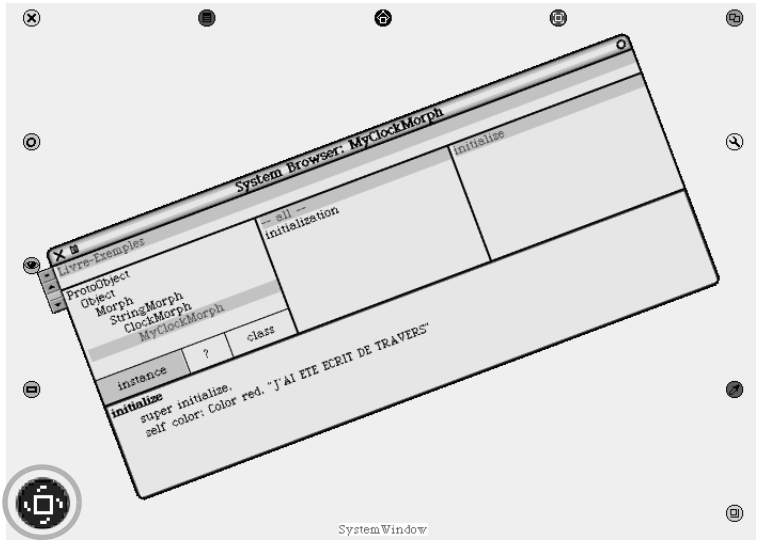
On obtient un halo par Alt+clic gauche sur le morph (voir figure 3-4) :

Figure 3–4
Un halo associé à un `MyClockMorph`*, avec les fonctionnalités correspondant aux icônes*



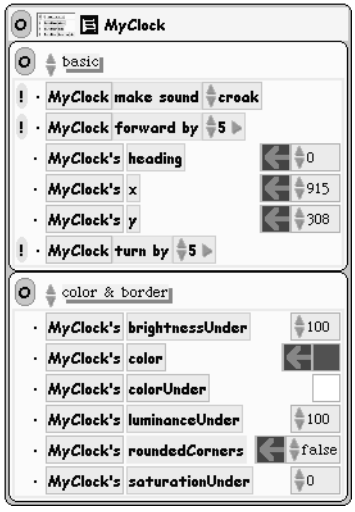
Les icônes qui composent le halo permettent, comme on peut le voir figure 3-4, d'effectuer certaines opérations associées au morph. Parmi celles-ci, on trouvera la rotation du morph. Tout morph, même retourné selon un angle quelconque, reste parfaitement utilisable. Voici par exemple un explorateur (voir figure 3-5), dans lequel il reste possible d'éditer du texte ! Indépendamment du fait que c'est très utile pour tout développement effectué sur des tables dont les pieds droits sont plus courts que les pieds gauches, c'est en soi une illustration des étonnantes possibilités graphiques de Squeak.

Figure 3–5
Exemple de rotation d'un `Morph` complexe : le system browser



Le panneau de paramétrage du morph (option <Viewer>) ouvre l'interface à partir de laquelle il est possible de définir tous ses paramètres : couleurs, taille, emplacement, comportement à l'ouverture, à la fermeture… (voir figure 3-6).

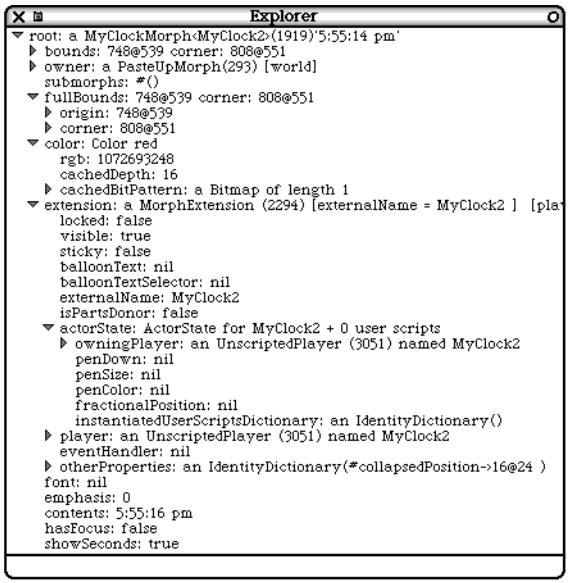
Figure 3–6
Le panneau de paramétrage d'un `Morph`



L'option de débogage ouvre un menu à partir duquel il est possible d'inspecter le morph, d'accéder à la chaîne des objets qui le référence, d'inspecter son modèle, d'ouvrir un explorateur sur sa classe, mais aussi d'ouvrir un explorateur sur la structure des objets qui composent le morph (voir figure 3-7).

Figure 3–7

Un explorateur hiérarchique de la structure d'un morph



Tous ces outils sont utiles non seulement en phase de mise au point, pour déboguer un morph, mais aussi en cours de développement pour mieux comprendre le comportement d'un morph que l'on ne connaît pas, et donc pouvoir le réutiliser.

Squeak dispose, on l'aura compris, de nombreuses facilités graphiques, très intéressantes, que nous n'avons jusqu'ici fait qu'effleurer.

En résumé

Ce chapitre a été l'occasion de vous initier quelque peu au développement en Squeak et de vous familiariser avec divers éléments de son environnement de développement.

À partir de l'exemple d'une classe existante, la classe `ClockMorph`, une petite horloge mobile, nous avons abordé les problèmes simples de réutilisation du code existant, en présentant comment on peut créer une classe à partir d'une classe existante, définir de nouvelles méthodes et variables d'instance qui permettent d'en spécialiser ou modifier le comportement. On a pu ainsi présenter les principes de base de la syntaxe de Squeak.

Nous avons également abordé un nouvel outil de l'environnement (l'explorateur hiérarchique) à partir duquel on peut accéder aux classes, méthodes et variables dont hérite une classe donnée. Les outils associés aux morphs, auxquels donne accès le « halo » qui leur est associé, ont également été rapidement présentés.