# Une première Morph

Dr. Ducasse

ducasse@iam.unibe.ch
http://www.iam.unibe.ch/~ducasse/

Ce mois-ci nous allons définir une première Morph: une tortue à la LOGO. Cet exemple extrêmement simple montre comment on définit une classe et des méthodes en Smalltalk. De plus, dans un des premiers articles, nous vous avons parlé de la façon incrémentale de développer en Smalltalk et de la notion de proximité que le développeur a avec ses objets. Ici nous allons illustrer cela à l'extrême et montrer ce que l'environnement de développement de Smalltalk permet de faire. Nous allons partir de la plus petite classe fonctionnant et ajouter des méthodes une à une tout en étant capable de tester à tout moment les méthodes ainsi définies. Notez que meme si l'exemple peut paraître exagérer, la plupart des développeurs Smalltalk développe de cette façon : incrémentale et interactive. Il arrive meme de définir les méthodes directement dans un debugger. Tout d'abord, nous passons en monde Morphic (**open, Morphic Project**) puis **enter**.

## 1. Comprendre le Browseur

Squeak comme tous les Smalltalk propose différentes façons de parcourir du code à l'aide de browseurs de hiérarchies (hierarchy browser), de classes (class browser), classes implémentant certaines méthodes (implementors), classes invoquant certaines méthodes (senders), ... Nous allons principalement utiliser le browseur de classes. Ensuite **Open**... puis **Browser**. Pensez à changer le profondeur de l'écran (**appearance**, **set display depth**... 32) et à changer ka façon dont les scrollbars fonctionnent si vous détestez comme moi les scrollbars flottants (**help**, **preferences**, **scrolling**, **inboard**). Dans le menu de la liste de gauche utilisez **find class** avec `Point` pour positionner le browseur la classe `Point`, puis dans la troisième liste sélectionnez converting et dans la quatrième extent:.Vous devez obtenir la figure 1.

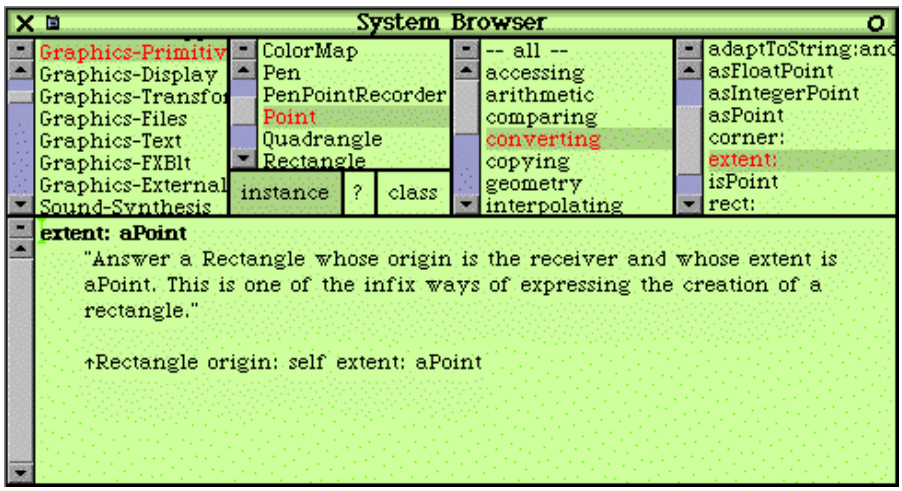



**Figure 1** Un browseur de classe montrant la méthode `extent:` définie dans le protocole de méthode `converting` de la classe `Point` contenue dans la

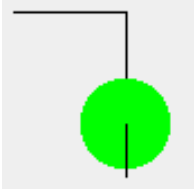
Un browseur de classes est constitué des cinq parties distinctes. De la gauche vers la droite nous avons :

- les catégories de classes qui sont justes une façon de ranger des classes ensemble. Notez qu'il n'y aucune sémantique associée. Il s'agit juste de sortes de dossiers. Dans la figure 1 la catégorie de classe sélectionnée est Graphics-Primitive.
- les classes associées à la catégorie sélectionnée.
- les protocoles de méthodes qui sont le pendant de catégories de classes pour les méthodes, ici converting.Les protocoles n'ont aucune sémantique et sont juste là pour aider à parcourir le code.
- la liste des noms, appelés sélecteurs, de méthodes contenues dans le protocole sélectionné, ici extent: est sélectionné.
- la partie inférieure montre la méthode : son nom avec arguments, commentaire et sa définition.

## 2. La classe Turtle

Nous voulons définir une morph qui représente une tortue à la LOGO. Nous voulons pouvoir écrire le programme suivant et obtenir la figure suivante.

```
| t|
t := Turtle new.
t go: 100; turn: -90 ; go: 100
```



Etape 1 Définition de classe. Commençons par définir la catégorie `'MiniTurtle'` (sur la liste des catégories, liste de gauche, menu **addItem**). Le système crée donc une nouvelle catégorie ne contenant aucune classe et lorsqu'elle est sélectionnée il propose le patron suivant:

```
Object subclass: #NameOfClass
  instanceVariableNames: 'instVarName1 instVarName2'
  classVariableNames: 'ClassVarName1 ClassVarName2'
  poolDictionaries: ''
  category: 'MiniTurtle'
```

Une tortue est une Morph donc elle doit être une sous-classe directe ou indirecte de la classe `Morph`. Ici `Turtle` hérite de la classe `Morph`. Pour modéliser notre tortue, nous avons besoin des informations suivantes : sa position à l'écran, sa direction et si elle en mode écriture ou non. Toute morph a déjà une position par le biais de la variable `bounds` et un ensemble de méthodes comme `topLeft`, `center`...donc il nous suffit juste de définir deux variables d'instances `direction` et `penDown` comme montré ci-dessous. Remplissez le patron proposé puis du menu de l'éditeur choisissez le choix **accept**. Et vous avez compilé votre classe!!!

```
Morph subclass: #Turtle
  instanceVariableNames: 'direction penDown '
  classVariableNames: ''
  poolDictionaries: ''
  category: 'MiniTurtle'
```

Etape 2 Définition d'une méthode. Il suffit juste d'initialiser correctement les instances et nous allons pouvoir en créer de manière à programmer incrémentallement la classe `Turtle`. Sélectionnez la classe créée dans la liste de classes (deuxième liste) puis créez un nouveau

protocole de méthodes nommé `'initialize-release'` (troisième liste choix de menu **new category**, un protocole de méthodes et aussi appelé *method categorie*. Pour être plus clair nous avons choisi d'utiliser le terme protocole). Dans l'éditeur tapez la méthode `initialize` suivante :

```
initialize
    super initialize.
    self extent: 50 @ 50.
    penDown := true.
    direction := 0.
```

La méthode `initialize` que nous venons de créer invoque la méthode `initialize` qu'elle masque en utilisant la pseudo-variable `super`, ici `initialize` est aussi définie sur la classe `Morph` donc, c'est cette méthode qui sera invoquée. Puis elle définit des valeurs par défauts pour les instances.

Etape 3 Création d'instances. Maintenant nous pouvons créer une instance de la classe `Turtle` : tapez dans n'importe éditeur ou le Transcript (**open... Transcript**): `Turtle new openInWorld.` Vous devez obtenir un carré bleu en haut à gauche de votre écran. Si vous sélectionnez le halos rouge (Option Clique sur Mac et bouton du milieu sur PC) puis dans le choix **debug**, le menu **inspect**, vous obtenez un inspecteur sur la tortue ainsi créée comme illustré par la figure 2.

Un inspecteur est un petit outil de développement qui permet de voir et modifier les valeurs de variables d'instance d'une instance ou de lui envoyer des messages. Sélectionnez dans l'inspecteur la variable `bounds`. Maintenant lorsque vous prenez la tortue et la bougez, les valeurs `bounds` sont changées automatiquement. Notez qu'un inspecteur viole l'encapsulation. Il faut parfois regarder quelles sont les méthodes accédant une variable d'instance pour savoir comment interagir avec une instance.

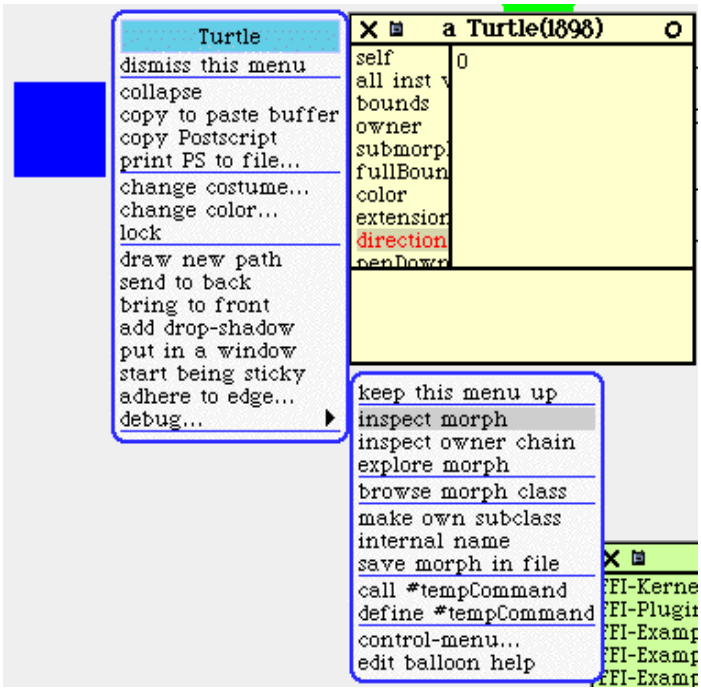



**Figure 2** Un inspecteur ouvert sur une instance de Turtle.

Etape 4 Définition de méthodes. Définissez la méthode `drawOn:` suivante dans la catégorie `'drawing'`.

```
drawOn: aCanvas
  | lengthOfHead center |
  lengthOfHead := self height // 2 - 2.
  center := self center.
```

```
aCanvas
   fillOval: (self bounds topLeft + (5 @ 5) extent: self height - 10)
   color: Color green.
aCanvas
  line: center
  to: center + (direction degreeCos @ direction degreeSin negated * lengthOfHead)
  width: 1
  color: Color black
```

Immédiatement après avoir défini cette méthode si vous pointez la souris sur le carré bleu vous devez voir une tortue verte avec un trait indiquant sa direction comme montré plus avant. La méthode `drawOn:` est la méthode responsable de la représentation visuelle d'une Morph. qui revient à écrire dans un canevas. Regardez la classe Canvas si vous voulez en savoir plus.

Maintenant si vous sélectionnez dans l'inspecteur la variable `direction`, tapez 90 dans la partie de gauche et sélectionnez le choix **accept**. La valeur est changée, pour voir le résultat, il faut tapez `self changed` dans la partie inférieure de l'inspecteur.

Maintenant, nous implémentons la méthode `goAt:` qui positionne une tortue à un point sur l'écran et laisse une trace si la variable `penDown` est vraie.

```
goAt: aPoint
  "Move the turtle at the position given by aPoint. If penDown is true, draw a line
  from the old place to the new one."
  penDown
    ifTrue: [self trailMorph
                  drawPenTrailFor: self
                  from: self center
                  to: aPoint].
  self changed.
  fullBounds := bounds := bounds translateBy: (aPoint - self center) rounded.
  self changed
```

Dans l'inspecteur, essayez `self goAt: 200@200` (**do it**).

Etape 4 autres méthodes. Maintenant, on peut définir la méthode `go:` qui fait avancer une tortue d'un certain nombre de pixels dans sa direction courante.

```
go: distance
  penDown := true.
  self goAt: (direction degreeCos @ direction degreeSin negated * distance)
               asIntegerPoint + self center
```

Tester votre méthode en utilisant l'inspecteur. Finalement, définissez la méthode `turn:` qui change la direction de la tortue d'un certain nombre de degrés.

```
turn: degrees
  "Change the direction by an amount equal to the argument, degrees. The positive
  sense is nonclockwise"
  direction := direction + degrees.
  self changed
```

Etape 5 Méthode de classe. Vous avez sûrement remarqué que nous devons invoquer la méthode `openInWorld` dans la séquence: `Turtle new openInWorld` pour voir la tortue.

Alors que nous voulions juste envoyé le message `new`. Pour cela, nous allons redéfinir la méthode `new`. Attention cette méthode est envoyée à la classe `Turtle` elle-meme et non à une instance donc c'est une méthode de classe. Pour définir cette méthode vous devez vous assurez que vous avez cliqué sur le bouton 'class' du browseur. Ensuite créez une catégorie `'instance creation'` puis définissez la méthode. Maintenant vous devez obtenir le comportement souhaité.

```
new
  "Creates a new instance of Turtle and open it in world"
  | t |
  t := super new.
  t openInWorld.
  ^ t
```

## 3. Visibilité et la dualité classe/instance

Dans les règles de visibilité de Smalltalk on retrouve la simplicité du modèle. Les règles de visibilité sont triviales:

- les variables d'instances sont protégées, i.e., qu'elles ne peuvent pas être directement accédées par des clients. Les sous-classes peuvent y accéder bien que cela soit souvent pas de bon style. Les variables d'instances commencent par une minuscule.
- Smalltalk définit aussi les variables de classe (*classVariables*) qui devraient s'appeler variables partagées. Celles-ci sont accessibles aussi bien depuis les méthodes de classes que celles d'instances. Elles sont aussi partagées par toutes les instances de la classe et des sous-classes.
- les méthodes sont publiques.

Un des aspects délicats de Smalltalk est la notion de variables d'instances et de méthodes de *classes*. En première approximation, variable d'instances et méthodes de classes peuvent être assimilées à des variables et méthodes statiques en Java. Les classes en Smalltalk sont des objets et les variables d'instances et méthodes de classes sont juste des variables d'instances et méthodes définis sur les classes qui représentent ces classes. Analysons la séquence suivante:

```
| t |
t := Turtle new.
t go: 100.
```

Commençons par la méthode go:. Elle est envoyée à t instance de Turtle donc elle doit être définie dans la classe de t, c'est-à-dire `Turtle`. Le raisonnement est similaire pour new. new est envoyée à `Turtle` instance de `Turtle class` (chaque classe X est instance d'une classe anonyme nommée `X class`) donc `new` doit être définie dans la classe de `Turtle`: `Turtle class`. Et c'est ce que nous avons fait quand nous avons cliqué sur le bouton 'class' du browseur.

Il n'y a pas de constructeurs avec des règles complexes, juste des méthodes de classes ayant la *meme* sémantique que les autres méthodes mais dont le résultat est une instance de la classe. Ceci implique que les méthodes de classes suivent les *mêmes* règles pour l'héritage que les méthodes d'instance (car il n'y a qu'une seule règle en fait). Une autre conséquence de cette uniformité est que une instance ne peut pas accéder directement les variables d'instances de sa classe car elles sont protégées et que l'instance joue le rôle d'un client. Les seules variables pouvant être accédées par les instances et par les classes sont les variables de classes (classVariable).

## 4. A propos de l'initialisation d'instances

Par défaut, la méthode `initialize` n'est pas automatiquement appelée par la méthode de classe `new`. Dans notre exemple, la classe `Morph` surcharge `new` (regardez la méthode `new` de la classe `Morph` en cliquant sur le bouton 'class') afin d'invoquer automatiquement la méthode d'instance `initialize`. `new` est invoquée sur une classe et non sur une instance. Donc il s'agit d'une méthode de classe. `super new` crée une instance et la méthode `initialize` est invoquée sur cette instance qui est ensuite retournée.

```
Morph class>>new
  ^ super new initialize
```

Notez que si l'on veut s'assurer que meme si la méthode `initialize` ne rend pas l'instance créée, la méthode `new` la rend. On peut code new de la meme façon que dans la classe Turtle comme montré plus haut ou comme suit:

```
Morph class>>new
  ^ super new initialize ; yourself
```

`yourself` est une méthode qui rend le receveur de la méthode ce qui couplé avec une cascade permet d'obtenir le premier receveur de la cascade.

## 5. Conclusion

Nous avons montré comment définir des classes, des méthodes et compiler de manière incrémentale du code. Si vous regardez la classe Morph vous serez perplexe par la taille de cette classe qui est trop grande. C'est aussi notre point de vue. Cette classe a beaucoup trop de responsabilités et a évolué de manière trop "sauvage". C'est un des problèmes de Squeak. Trop d'expérimentation et peu de génie logiciel. Les autres Smalltalk proposent un code de meilleure qualité, mais bien moins amusant.

Nous montrerons dans un prochain article comment trouver des informations en Squeak. Pour le moment, nous vous suggérons d'essayer les fonctions sender et implementors disponible sur le menu des sélecteurs de méthodes et le browseur de sélecteur (**open**, **method finder**).