Supporting Objects as An Anthropomorphic View at Computation or Why Smalltalk for Teaching Objects?

Stéphane Ducasse and Roel Wuyts
Université of Berne
{ducasse, wuyts}@iam.unibe.ch
http://www.iam.unibe.ch/∼ducasse/

Abstract

In this paper we stress the fact that a language and an environment for teaching object-oriented programming should support the anthropomorphic metaphor promoted by the paradigm. We show that all the cultural aspects of Smalltalk, *i.e.,* the vocabulary and the syntax support the object metaphor. In addition, we stress that the programming environment should also support the metaphor. We show that Smalltalk environments offer an important property we named liveness or object proximity that promotes the anthropomorphic perception of objects. By providing excerpt from our forth coming book, we show how Squeak with the Morphic framework reinforces this ability to make object into living entities.

1 Introduction

The title of this position paper could have been: "Why Smalltalk is still one of the best languages (if not the best) to teach object-oriented programming?" but we refrained to explain again why Smalltalk is better from a syntactical and semantically point of view than Java. We point the reader to the papers of Thomas Kühne [Kue01] for a presentation of the main advantages of the Smalltalk language and its environment over Java for teaching. Guzdial and Soloway elaborate in [GS02] why the richness of the Squeak environment makes it a good tool to motivate Nintendo generation students to build multimedia systems.

As explained by the following definition, the term *anthropomorphism* represents the action we perform when we tend to think that physical objects or animal have human behavior.

> **Anthropomorphism** noun [U] *the showing or treading of animals, gods and objects as if they are human in appearance, character or behaviour*(From Cambridge International Dictionary of English)

As anthropomorphism is the essence of object-oriented programming, object-oriented languages and environments should support it. That is why in this paper we show why the Smalltalk language and its environment are an excellent environment that immerses the novice programmers in the object metaphor. Indeed Smalltalk culture, vocabulary, syntax, and pureness favors it. Then we show that the Smalltalk environments have a fundamental property we name liveness or object proximity that supports the perception of objects not simply as abstract data structures existing at execution time but as persistent living entities.

This paper first presents the cultural aspects of Smalltalk that supports the object metaphor, then presents how the environment reinforces this perception by the liveness of objects or the proximity a programmer can have with his objects.

We are currently writing a book to teach object-oriented programming to novices [DD00b], [DD00a], [Duc02]. Our approach starts from an objectified turtle à la LOGO with which elementary aspects such as variables, loops, abstractions, and abstraction compositoins are presented. Then object-oriented programming is approached using Joe the Box example [KG77], the construction of simple games based on the Morphic framework [1], and finally a robot environment.

In appendix we present two chapters of our forthcoming book to illustrate these points [Duc02]. The second chapter shows in particular that the Morphic environment developed for Self and now used in Squeak is a further step in this anthropomorphic view in the sense that it supports a direct interaction with graphical objects. We are currently using the MorphicWrapper package [MG01, Mor] developed on top of Morphic that allows one to send messages to morphs by typing the messages *directly* above the concerned objects. However, we do not have any material showing this aspect except the environment available at: http://www.iam.unibe.ch/∼ducasse/WebPages/Turtle.html. Therefore, we only show two screenshots illustrating these aspects.

2 Smalltalk: Promoting an Anthromorphic View of Abstraction

When teaching object oriented programming and design, the choice of the vocabulary is important. For example, we say that an object performs certain actions. We commonly use the objects as the subject of active sentences. In a similar way that we talk to a friend, we talk to objects that react by executing methods.

Smalltalk culture promotes this anthropomorphism in a constant manner from the vocabulary used, the syntax, and the conceptual uniformity of the language as we show now.

Vocalulary. In the Smalltalk jargon we do not invoke a method or call a function but *send a message to an object*. Sending a message implies two aspects: first the fact that the receiver is responsible to react to the message the way it wants. The analogy with the reception of a letter is again a strong incentive to think about objects as entities

[1]the Morphic framework has been developed for the language Self and ported to Squeak [MG01] by its creator, John Maloney.

protecting their private information while exhibiting some behavior. Second, invoking a method implies that a method defined in a specific class is invoked, while sending a message lets the door open to the way the message resolution actually happens. Therefore sending a message supports late binding understanding.

Syntax: Talking English. The Smalltalk syntax itself promotes the anthropomorphic aspect of object-oriented programming. Indeed, the syntax of the language has been designed with the idea in mind that kids would talk to the objects they create. Hence the result is that code tends to look like english.

The following example is taken from Squeak Alice [Ali] an authoring environment based on 3D ported to Squeak [GS02]. The line expresses that we ask the object, `a3DObject`, to turn to the left, making `2` turns at speed `2`. The keyword-based decomposition identifies clearly which argument plays which role. But more important is that we can read the expression aloud and it sounds like a sentence as shown by the following message send.

```
a3DObject turn: #left turns: 2 speed: 2.
```

As show below, the Java equivalent is less readable because we have to remember the order of the arguments and it does not read like english.

```
a3DObject.turnTurnsSpeed(#left, 2, 2);
```

Language Uniformity and Dynamic Typing. The fact that Smalltalk is pure, *i.e.,* every entity is an object promotes also the object metaphor.Basic language elements such as integer, boolean, string or array are first class objects. In a similar fashion the complete environment from code browsers, compiler to scheduler and concurrency support are objects. There is no difference between objects and primitive types, so the novice is not inclined to think in terms of the underlying data structures.

Moreover, dynamic typing combined with this uniformity is another step in that direction: the novice programmer does not have to lose his time to determine which number related types (float, signed int, double....) he should use when he declares a variable. In addition he does not have to coerce the types. The coercion is done automatically. There is no mixture between the physical representation of objects and their behavior. A novice thinks mainly in terms of objects. This aspect is also promoted by the fact that class methods are not static methods with specific rules but just methods sent to specific objects that are classes. The same rules that worked for instance apply for classes.

3 The Image: The Anthropomorphism at The Environment Level

The anthropomorphism of the objects is also promoted by the Smalltalk environment itself. The virtual machine execution model with its image –the image is a chunk of memory containing all the objects currently created and the byte-code of the currently loaded classes – represents a world in which objects appear as living entities. The environment acts as a *micro-world* even if nothing is specially dedicaced to novice and no specific metaphor have een developed.

We used the term *liveness* or *proximity* to refer to an important property that an environment should have to support the anthropomorphism. By liveness or proximity we refer to the fact that an environment lets novice programmer see and manipulate his objects: he can send messages to objects. An object can be assigned to a global variable or inspected by an inspector and can then receive messages while its class is been completed by defining method incrementally. Via such an approach the novice grab his objects and talk with them. The abstractness of the computation entity decreases and the objects become living entities. The inspector acts as a microscope by which the novice can see his own objects changing and reacting to messages. Note that an inspector is a dangerous tool because it violates the encapsulation. In a similar fashion that dedicaced debuggers for teaching purpose have been developed in LearningWorks [Gol95] where the stack was filtered, it is straightforward to develop dedicated inspectors that would not violate encapsulation and let only send messages to objects.

The incremental compilation coupled with the presence of an image hosting the development environmnent strenghtens the liveness of the objects. Objects are not only entities created at run-time. They are present during all the phases of development. Being able to create an object from a nearly empty class, to specify new behavior and interact with the objects as soon as these methods are defined reinforce the anthromorphic metaphor behind object-oriented programming.

In Smalltalk a debugger just pops up when there is a problem, and lets the code to be changed on the fly and the execution continued as if the bug would not have existed. This behavior by letting the programmer freely changing method code without having to switch between programming modes, *i.e.*, from editing to compiling to debugging, reduces the mental gap between objects as acting entities and objects as description of computation.

4 Conclusion

We presented why Smalltalk and its environment are perfectly adapted to promote object thinking because of the omnipresent anthropomorphism used. We would like to stress that Self [US87] which is the result of the same design principles than Smalltalk to prototype-based languages offers the same advantages. Even if some aspects of Self are interesting like the fact that instance variables are declared using accessor methods or the absence of metaclasses, we do not have experienced teaching with Self so we cannot really evaluate the problems we may encounter.

By using the terms “Teaching Objects” in the title of this article and not object-oriented programming our intention was to stress that nearly 30 years after the invention of object-oriented programming it would be interesting that languages used for teaching could support the understanding of objects and not writing programs where the main goals are to fight with a badly designed type system. Althought this is a sweet dream to think that such an argument could be used to choose which language to use for teaching and not be only urged to teach what marketing departments of big companies have imposed to the world, we encourage researchers and teachers to scientifically evaluate the reasons that force them not to use adequate pedagogical tools.

References

[Ali] Squeak, http://www.alice.org/.

[DD00a] Stéphane Ducasse and Florence Ducasse. Caro, dis-moi c'est quoi programmer?, 2000. Support de cours de Technologie, 150 pages, http://www.iam.unibe.ch/ ducasse/.

[DD00b] Stéphane Ducasse and Florence Ducasse. De l'enseignement de concepts informatiques. *Journal de l'association EPI Enseignement Public et Informatiques*, (97), September 2000.

[Duc02] Stéphane Ducasse. *From Logo to OO: Learning Programming in Squeak (temporary title)*. Morgan Kaufman, 2002. http://www.iam.unibe.ch/∼ducasse/WebPages/Turtle.html.

[Gol95] Adele Goldberg. What should we teach? In *ACM SIGPLAN OOPS Messenger, Addendum to the proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications (Addendum)*, volume 6, 1995.

[GS02] Mark Guzdial and Elliot Soloway. Teaching the nintendo generation to program. *Communication of the ACM*, 2002.

[Guz01] Mark Guzdial. *Squeak - Object Oriented Design with Multimedia Applications*. Prentice-Hall, 2001.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, volume 21, November 1997.

[KG77] Alan Kay and Adele Goldberg. Personal dynamic media. *IEEE Computer*, 10(3):31–41, March 1977.

[Kue01] Thomas Kuehne. A smalltalk for students – a giant leap for studentkind. *Journal of Object-Oriented Programming*, may 2001.

[MG01] Kim Rose Mark Guzdial. *Squeak - Open Personal Computing and Multimedia*. Prentice-Hall, 2001.

[Mor] Morphic Wrapper, http://mathmorphs.swiki.net/2.

[SMU95] Randall B. Smith, John Maloney, and David Ungar. The self-4.0 user interface: manifesting a system-wide vision of concreteness, uniformity, and flexibility. In *Proceedings OOPSLA '95, ACM SIGPLAN Notices*, volume 6, 1995.

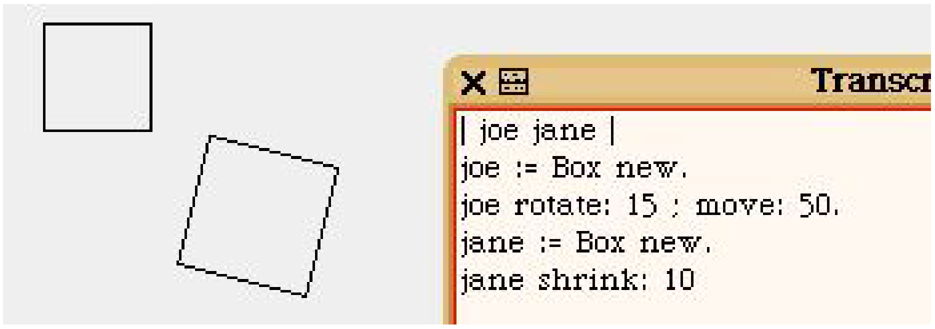
[Squ] Squeak, http://www.squeak.org/.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87*, *ACM SIGPLAN Notices*, volume 22, pages 227–242, December 1987.

A A Typical Programming Session

This section is a part of a forthcoming book whose goal is to teach object-oriented programming of all ages and *a priori* no background. The next chapter is a chapter that uses the Morphic system developed for Self [SMU95] and ported to Squeak [IKM$^+$97] [MG01].

1 Implementing Joe the Box



```
| joe jane |
joe := Box new.
joe rotate: 15 ; move: 50.
jane := Box new.
jane shrink: 10
```

@@Should we change the class definition template@@

In this chapter you will program your first class, the class `Box`. Doing so you will learn how to describe the behavior and the state of objects. We start with a first implementation that we later refine. This first class is one of the first example used to teach object-oriented concepts to novices by researchers that invented object-oriented programming.

1 Box's Behavior and State

A box has the following behavior, it knows how to draw itself, move to a given distance, move to a given point, rotate, grow and shrink. A typical scenario is described by the script 1.1. A graphical result is shown by the first figure of the chapter above.

Script 1.1 (*A typical Box scenario)*

```
| joe jane |
joe := Box new.
joe rotate: 15.
joe grow: 100.
joe move: 10@10.
joe moveTo: 150@200.
jane := Box new.
jane move: 30@-30.
jane shrink: 40.
jane rotate: 45
```

It is worth to spend some time looking at 1.1. It shows that while at the first glance the scripts may look similar they are not. From an object point of view, there is difference between asking a turtle to draw a square as shown in the chapter **??** and asking a box to draw itself. The methods and the arguments that they require are different. Here a box knows how to grow, shrink and rotate.

Before starting programming, we have to analyze the behavior of a box to imagine a possible way to program it. Here is the behavior a box should have. A box should know how to:

- ○ draw itself at a given location. When a new box is created it automatically displays itself.
- ○ move to a given location (method `moveTo: aPoint`).
- ○ rotate from a given angle (method `rotate: anInteger`).
- ○ translate from a certain distance (method `move: aPoint`).

It is fundamental to start by looking at objects from their *behavior*. An object is a behavioral entity, *i.e.*, an entity reacting to messages. A similar behavior can be implemented by different manners so it is crucial not to start to think in terms of the internal structures that may represent the object but in terms of the essence of the object, its *behavior*.

Teacher's Corner

Note the way we phrase the sentences describing box actions: we do not say the box is displayed but it displays itself. We always use the active form where the subject is the object itself. Considering the object as a living being is a good way to think in an object-oriented manner. Imagine talking about an animal or a person you will say that the person acts and not is acted by others.

Teacher's Corner

From behavior to state. Now from this description of the box's behavior, we should imagine a possible state for a box that could be used to implement the wished behavior. As this example and the concept of box are familiar, we propose that boxe state is represented by a size, a position and a tilt.

In fact any box will be represented by such a triplet (size, position and tilt) but each given object will have its own triplet values. For example, the box referenced by the variable `joe` in the script 1.1 has its *own* state, *i.e.*, its own size, position, and tilt. In the same way the box `jane` has a *similar* state because it is also a box created from the class `Box` too but it has its own state which may or not equal to the one of `joe`. When the state of one given box changes it does not change the state of the other boxes. This situation is illustrated by the first figure of this chapter. If this aspect is not clear we suggest you to (a) read the chapter **??** and (b) create the class `Box`, inspect two instances and modify their states.

2 Defining the class `Box`

To create a class we use a dedicated browser called the system browser or class browser . To open such a browser, bring up the default menu and chose the menu item open... and the item browser (or use the b).

To create a class, create first a new category (which represents a folder for all the classes we will create related to this small project) by selecting the item addItem of the menu associated with the leftmost pane of the browser (Figure 1.1). Name it for example `JoeTheBox`. When you select the newly created category, the system displays a template to help you defining a new class (see the class 1.1 and the figure 1.2).

Class 1.1

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

Modify the proposed template to obtain the class 1.2 and in the bottom pane bring the menu and choose the menu item accept. Now the class exists. The system shows you that the class is defined by displaying it in the second pane as shown in figure 1.3. Using the terminology used in other programming languages we

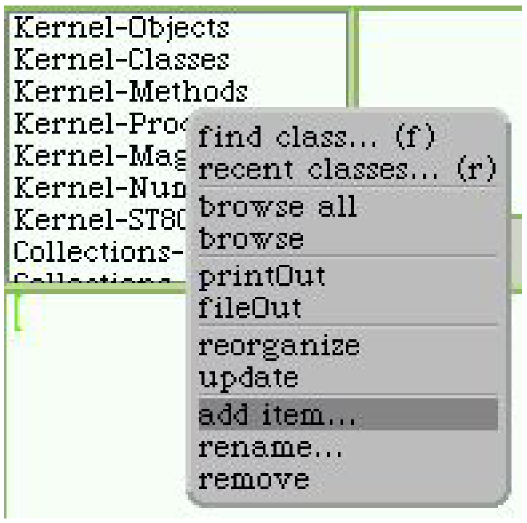
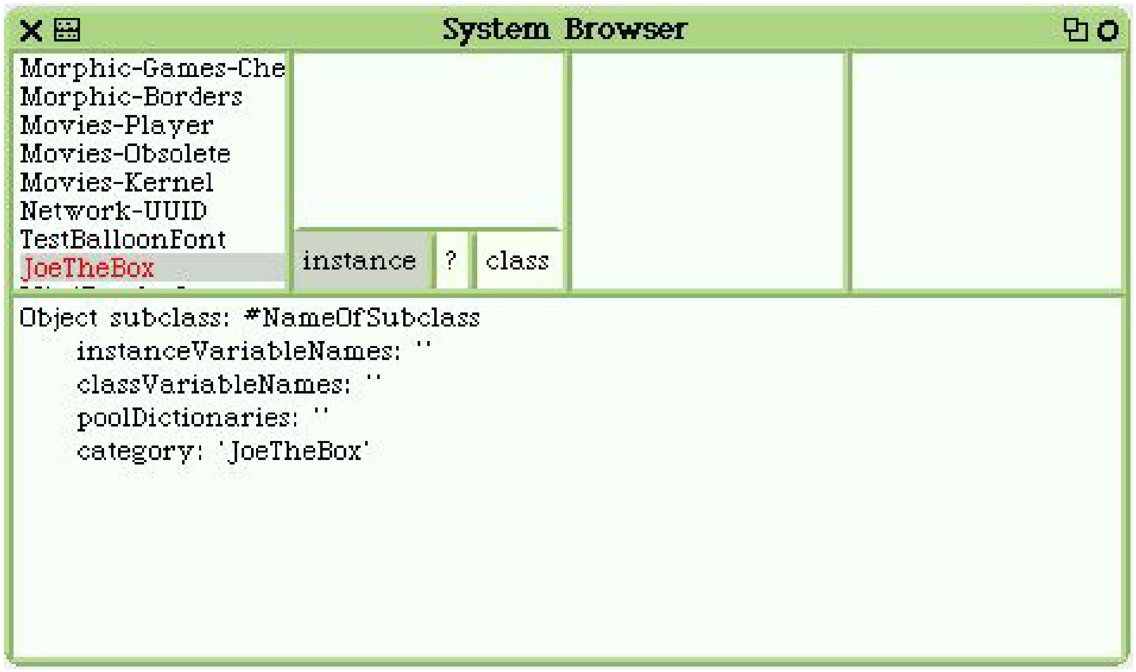
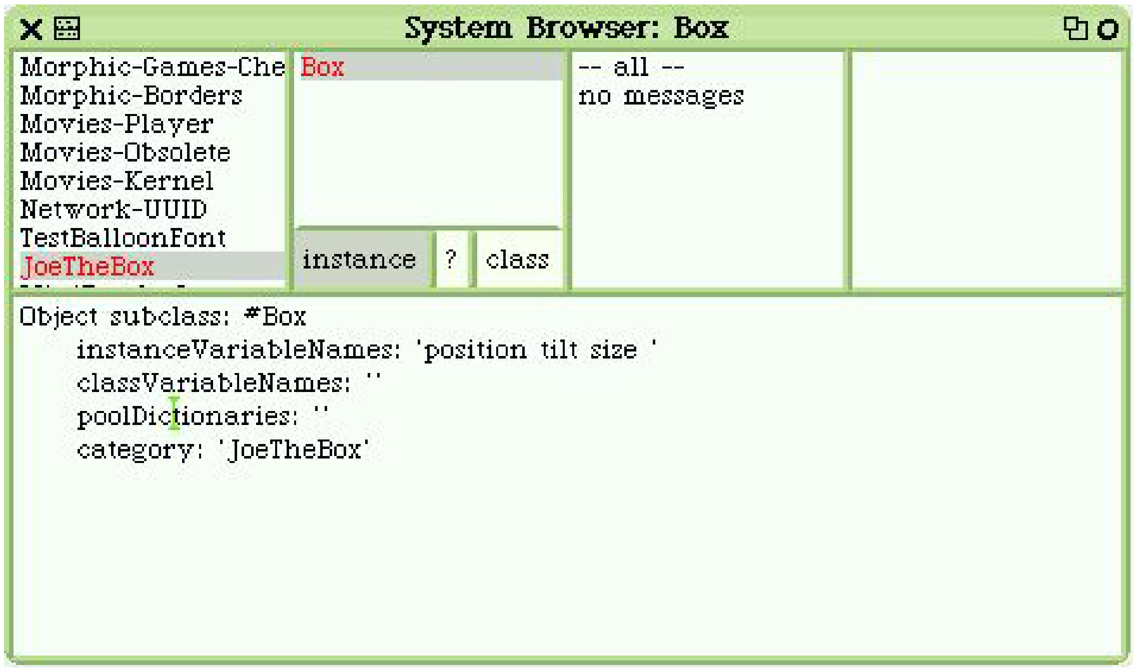


Figure 1.1: Bring up the menu over the leftmost pane of the browser and select the add item choice to create a new class category.



```
Object subclass: #NameOfSubclass
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'JoeTheBox'
```

Figure 1.2: The browser shows you that a new class has been created by displaying it in the second pane from the left.



```
Object subclass: #Box
    instanceVariableNames: 'position tilt size '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'JoeTheBox'
```

Figure 1.3: The browser shows you that a new class has been created by displaying it in the second pane from the left.

can say that the class has been *compiled*. This means that we could already create instances of this class, even if now this is not really useful since they do not have any specific behavior.

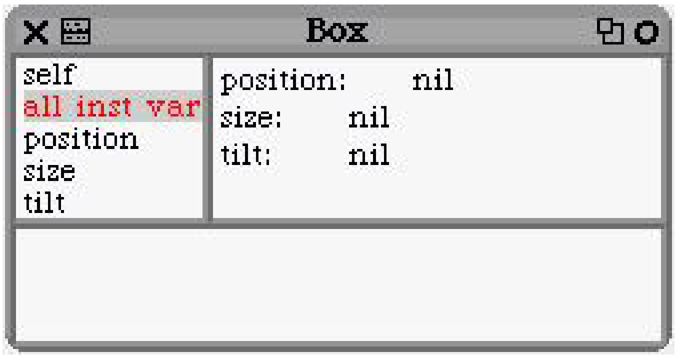
Class 1.2

```
Object subclass: #Box
   instanceVariableNames: 'position size tilt '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Joe The Box'
```

Here are some explanations about 1.2: A box is a simple object. Hence, it is a subclass `Object` The internal state of box instances, such as the boxes `joe` and `jane`, is represented by instance variables of the class `Box`. So line 2 we specify that the class `Box` has three instance variables by given their respective names. Here the class `Box` has the instance variables `position`, `size`, and `tilt`. This indicates to the class `Box` to create instances having three values representing the wished state. As shown by the class 1.2 we empty the other parts of the templates because they are irrelevant for now.

Important!

> A class acts as an object factory, an object model or an @@moule@@. The instance variables describe the state of the instances created by the classes. Each instance of the class will have the structure described by the class but filled with its own values. The factory metaphor is really useful to explain the difference between classes and instances. Classes are the description of instances.



```
position:    nil
size:    nil
tilt:    nil
```

Figure 1.4: Inspecting a *non initialized box*: all its instance variables are empty, *i.e.*, having `nil` as value.

3 Initializing Instances

Once the class is defined, create and inspect one of its instances by executing the script 1.2 (see the chapter **??**). The figure 1.4 shows an inspector on a `Box` instance. All the instance variables have `nil` as value. Indeed, when an instance is created by invoking the method `new` on a class, the *default* behavior of the class is to return an *uninitialized* instance. Uninitialized means that all the instance variables of the newly created instances have no value. To represent the no value concept, Squeak uses the object `nil` [1]. That's why the instance variables of the inspected box have all as value `nil`.

Script 1.2 (*Inspecting a box)*

```
   Box new inspect
```

Having uninitialized values is not really good because methods may not work or have to test if the variables have been correctly initialized. But even then this is not satisfactory because if an instance variable is not initialized it is difficult to know the value to initialize it. In fact the best solution is to initialize the instance as soon as it is created.

For that purpose we specialize the method `initialize` that sets up a default state for a box. The method `Box»initialize` is automatically invoked by the method `new` on newly created instances. This method sets the instance variables values. Once this method defined, in the bottom pane of the inspector evaluate the expression `self initialize`. If you closed the inspector or want to convince you that the method `initialize` is invoked when a new instance is created, reuse the script 1.2 to check that the created instance is now well initialized. In both cases, you should obtain a situation similar to the one described by the figure 1.5.

Method 1.1

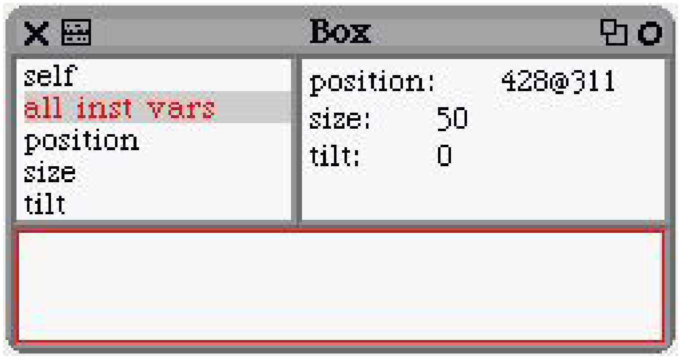
```
Box>>initialize
   "A box is initialized to be in the center of the screen, with
   50 pixels size and 0 tilt"

   size := 50.
   tilt := 0.
   position := World bounds center
```

Reader Note

The fact that the `new` method automatically call the `initialize` method is a little extension we added into Squeak. It may happen that such an extension will be included into Squeak at

[1]Nil comes from the latin nihil which means nothing.



```
self
all inst vars
position
size
tilt
```

```
position:    428@311
size:     50
tilt:     0
```

Figure 1.5: Inspecting an *initialized box*: all its instance variables contains some values coherent with their role.

the time you will read this book. In any case the plain Squeak solution to this problem is explained in the chapter **??** so that you can understand and program in Squeak without our little extension.

Reader Note End

4 About Returned Values

In Squeak, each method returns the receiver of the message (`self`) using the ˆ symbol if there is not another object returned explicitly. For example, the following method is strictly equivalent to the method 1.1.

Method 1.2

```
Box>>initialize
   "A box is initialized to be in the center of the screen, with
   50 pixels size and 0 tilt"

   size := 50.
   tilt := 0.
   position := World bounds center.
   ˆ self
```

In Squeak, any method returns per default the receiver of the message. To return a different value use the character ˆ followed by the expression to be returned.

5 Accessing Instance Variables

The method `initialize` above illustrates an important aspect of the object model of Squeak. The instance variables are accessible from the methods as if they were defined in the method body. For example, we are assigning 50 in the instance variable `size`. The instance variable `size` is accessible from any method of the class `Box`.

Contrary to the variables, called local, of a script which do not exist after the script execution, instance variables last the complete object life-time. We propose you some experiments to really understand this phenomena below. Note that this behavior is not new, we used it constantly with the turtle. For example, we changed the direction of the turtle using the method `north`, then later used the direction to perform some other actions.

We propose you to do some experiments to really understand this notion. Define the method `size` (the method 1.3) which returns the value of the instance variable `size` and `size: anInteger` (the method 1.4) which changes the value of the instance variable `size` to be the one specified as argument.

Method 1.3

```
Box>>size

   ^size
```

Method 1.4

```
Box>>size: anInteger

   size := anInteger
```

Now execute the script 1.3 and use the menu item print it to get the results we present using *returns*. If you have an inspector opened on a box instance, you can also execute the messages `self size`, `self size: 10` in the bottom left part of the inspector. Perform some other experiments to prove yourself you understand.

Script 1.3 (*Instance variables life-time)*

```
  | joe jane |
  joe := Box new.
  joe size. returns 50
  joe size: 10.
  joe size. returns 10
  joe size: 20.
  joe size. returns 20
  joe size: joe size + 5.
  joe size. returns 25
  jane := Box new.
  jane size. returns 50
```

In summary we have:

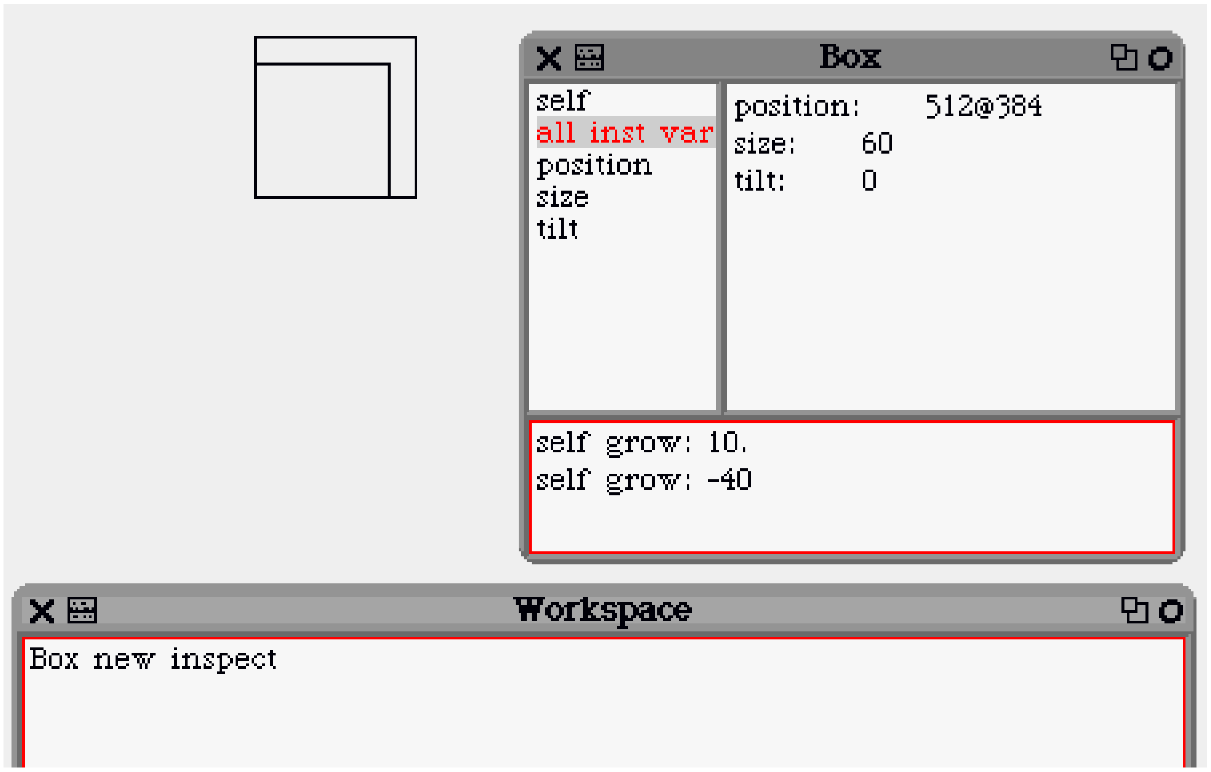
Instance variables are accessible to all the methods of a class. Instance variables last the same life-time than the object to which they belong to.

In Squeak, instance variables cannot be accessed from outside of an object. Instance variables are only accessible from the methods of the class that define them.

6 Drawing a Box and Other Operations

Now that we can initialize a newly created box using the method `initialize`, we are in position to define methods without been worried about the initialization of instance variables.

Method draw. We define the method `draw` that uses a turtle but we hide it as shown by the method **??**. We create a method, put it at the right position of the box, set the direction of the turtle to the tilt of the box, use the black color and then draw a square of the box size.



```
self
all inst var
position
size
tilt
```

```
position:    512@384
size:    60
tilt:    0
```

```
self grow: 10.
self grow: -40
```

```
Box new inspect
```

Figure 1.6: Using an inspector to send messages about newly created methods. Here the method `grow:` has been defined and we ask its execution via the inspector.

Method 1.5

```
Box>>draw
   "Draw the receiver position in black"
   "Box new initialize draw"

   | aTurtle |
   aTurtle := Turtle new hidden.
   aTurtle jumpAt: position.
   aTurtle turnRight: tilt.
   aTurtle penColor: Color black.
   4 timesRepeat: [aTurtle go: size.
                  aTurtle turnLeft: 90]
```

Test the method by executing the code `self draw` into the bottom pane of an inspector in a similar way than shown in figure 1.6, or by executing the script 1.4.

Script 1.4 (*sending the message draw to a box)*

```
| joe jane |
joe := Box new.
joe draw
```

During your experiments you may need to clear the screen. Use the the script 1.5 for that purpose.

Script 1.5 (*Clearing the screen)*

```
  World clearTurtleTrails
```

Experiment 1.1

Up until now, creating a new box did not displayed it. Change the method `initialize` so that any new box is automatically displayed.

Method grow: The method `grow: anInteger` makes the box growing of a certain size and redraw itself to reflect this size change. Use the inspector or dedicated scripts to tests your method. Try the script 1.6 to see that we have a problem.

Method 1.6

```
Box>>grow: increment
   "grow the receiver’s size from increment"

    size := size + increment.
    self draw
```

Script 1.6 (*Problem with the first `grow:` method.)*

```
  | joe |
  joe := Box new initialize.
  joe grow: 20.
```

The problem we have is that the turtle grows and redisplay itself well, but it does not remove the previous box shape. To solve that problem we propose you to define a method named `undraw` which is similar to the draw method except that it draw the box using a transparent color (the method 1.7).

Method 1.7

```
Box>>undraw
   "erase the receiver"

   | aTurtle |
   aTurtle := Turtle new.
   aTurtle jumpAt: position.
   aTurtle turnRight: tilt.
   aTurtle penColor: Color transparent.
   4 timesRepeat: [aTurtle go: size.
                  aTurtle turnLeft: 90]
```

Now that the method `undraw` is defined, the method `grow:` should call it before anything else as shown by the method 1.8.

Method 1.8

```
Box>>grow: increment
   "grow the receiver's size from increment"

   self undraw.
   size := size + increment.
   self draw
```

Experiment 1.2

Implement the methods

- `move: aPoint` which translate the box from a distance in x and y specified as a point.
- `moveAt: aPoint` which move the box to the specified point.
- `rotate: anInteger` which rotates the box of a given angle.
- `grow*: anInteger` and `shrink*: anInteger` that make grow and shrink the receiver by a given factor.

7 Limiting duplication

The methods `draw` (the method 1.5) and `undraw` (the method 1.7) are nearly the same except for the color of the turtle. This is not really good, since every times we will change one method we will have to change the other and there is chance that we forgot.

Experiment 1.3

Propose a solution to this problem. The idea is that to avoid duplication, the methods `draw` and `undraw` can call a third method with the color of the pen as argument. Try to implement such a method before reading the solution.

The `drawWithColor: aColor` (the method 1.9) factors out the duplicated code. Change the methods `draw` and `undraw` to call this method with the right argument.

Method 1.9

```
Box>>drawWithColor: aColor
   "Draw the receiver using a given color"

   | aTurtle |
   aTurtle := Turtle new hidden.
   aTurtle jumpAt: position.
   aTurtle turnRight: tilt.
   aTurtle penColor: aColor.
   4 timesRepeat: [aTurtle go: size.
                   aTurtle turnLeft: 90]
```

In general we should avoid as much as possible to have duplicated code. This is not a problem to duplicate code for a small experiment. However, if you want to keep the code always think that you should create other method to share and reuse the duplicated code. Creating one or several methods to factor the duplicated code is a good trick to cure duplicated code.

Avoid duplicated code. Refactor the duplicated code by calling a method representing the duplicated code.

8 Looking at Alternate Designs

We said that the implementation we proposed is one of the multiple ways of implementing the behavior of the `Box` class. First let us analyze the current implementation. The class `Box` has its own state then gives a part of its state to a turtle. The `Box` class uses the `Turtle` class to realize its behavior. This is a common practice where a class do not repeat behavior but reuse the behavior of an existing class.

We used the class `Turtle` because it was familiar to us. However, another class, the class `Pen` could have been a possible candidate too. Look at the class `Pen` and change the method `drawWithColor:` to use it instead of `Turtle`. What is important is that the interface proposed by the class `Box` should not change. We are changing the implementation and this should not change the behavior of the `Box`.

Now if we look carefully we see that the turtle or the pen instance are created every time the box is draw and undraw. In addition the state of the box is systematically copied to the turtle state then lost because the turtle is recreated and the previous one is lost. One idea would be to use a turtle as representing part of the box state. Indeed a turtle has a position and a tilt. Define the class `BoxT` as shown below in the class 1.3 and reimplement some of the box's methods to convince you that this is possible. This solution has as advantages that less objects are created, less state is copied from the box to the turtle and as drawbacks that the class `Box` is tied with the class `Turtle`.

Class 1.3

```
Object subclass: #BoxT
   instanceVariableNames: ’size turtle’
   classVariableNames: ’’
   poolDictionaries: ’’
   category: ’Joe The Box’
```

To help you we show two methods, the method 1.10 and the method 1.11, that are important. Try to do it by yourself first. Implement all the other methods.

Method 1.10

```
BoxT>>initialize
   "A box is initialized to be in the center of the screen, with
   50 pixels size and 0 tilt"

   size := 50.
   turtle := Turtle new hidden.
   turtle jumpAt: World bounds center.
```

Method 1.11

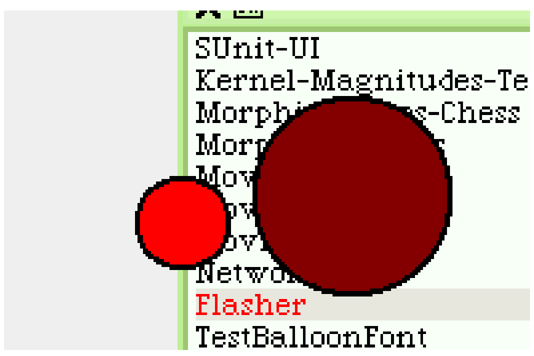
```
BoxT>>drawWithColor: aColor
   "Draw the receiver using a given color"

   turtle penColor: aColor.
   4 timesRepeat: [turtle go: size.
                   turtle turnLeft: 90]
```

B A First Morph

Squeak, the open-source Smalltalk developed by a part of the original Xerox team that invented Smalltalk [IKM^{+}97], [Squ], [MG01], is particularly interesting because Squeak interface is based the Morphic graphical interface inherited from Self [US87], [SMU95]. Morphs add another dimensions in the concretness and the anthropomorphism because they can be directly manupilated. The interested reader should read [MG01] and [Guz01] to get an overview of Morphic showing the possible direct interaction. In the following we illustrate how even with a traditional approach Morphs direct interaction promotes an anthopomorphic view of computation.

1
The Baby Clicking Game



In this chapter we propose you to build a small game for baby that learns how to use a mouse. The idea is to have a morph moving on the screen and to click on it to change its direction. This way we will show you how to change the behavior of a morph and how to add interaction with the morph. You will learn how you can define a class by refining another one and extending its behavior. This example will be then analyzed in the next chapter to explain you what is inheritance, i.e., how can we extend or refine the behavior of a class to obtain other classes with related behavior.

1 A Moving Morph

We do not want to create a Morph from scratch. For this purpose we will extend the `FlasherMorph` class. A FlasherMorph is a simple morph that flashes, *i.e.*, change color at constant rate. The the script **??** shows how to create such a simple morph.

Script 1.1 (*A flashing Morph)*

```
   FlasherMorph new openInWorld
```

We want to reuse this class but customize its behavior to our needs. First define the class `EscapingFlasher` as shown in the class **??**.

Class 1.1

```
FlasherMorph subclass: #EscapingFlasher
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Flasher'
```

The following step is to make the morph moves without intervention of the user. The Morphic system supports the animation of morphs and we will use it to make our morph moving. The `step` method is called by the system every amount of time determined by the `stepTime` method. The class `FlasherMorph` uses this mechanism to regularly change its color.

So in addition to the color change performed each step, we will change the position of the Morph in diagonal. ThatÕs why the method `step` first does a `super step`. This means that we ask that the behavior of the `step` method defined on the class `FlasherMorph` to be executed then we change the position by adding 2 in x and 2 in y to the current position of the morph (by using adding to the position a point). Define the method `step` as shown in the the method **??**.

Method 1.1

```
EscapingMorph>>step
   "At each step change the position of the morph"

   super step.
   self position: self position + (2@2)
```

Now execute the script **??**, your morph should be moving.

Script 1.2 (*An escaping flasher)*

```
  EscapingFlasherMorph new openInWorld
```

2 Interactively Changing the Direction of Flasher

Now that the morph is moving we would like to be able to let a baby to click on it to change its direction. Morph allows one to specify a lot of different interactions. We do not want to go on the details. The basic idea is that certain methods should precise when the morph wants to be aware of certain events such as pressing or releasing the mouse button on it or entering or leaving it. Then the morph has to specify the behavior associated with the kind of event it is interested in. Let us look at a concrete example. For our game we want to know when the mouse button is pressed, for that we specify the method `handles-MouseDown:` to say that the morph wants event generated by pressing the mouse button (the method **??**) and the method `mouseDown:` to specify how it will behaves when it receives such event.

Method 1.2

```
EscapingMorph>>handlesMouseDown: evt

   ^ true
```

Note that this method overrides (it does not invoke the behavior of the method `handlesMouseDown:` defined in the superclass) the one defined in the superclass of `EscapingFlasher`, it just replaces this behavior for the class `EscapingFlasher`. The methods `step` and `handlesMouseDown:` are used in two different ways: the first extend the behavior while the second replace it. Now we specify the method `mouseDown:`. This method is invoked each time the mouse is pressed on a morph (if the method handlesMouseDown: specifies by returning true that the morph is interested into that kind of event). Right now we just print something in the `Transcript` to be sure that the morph reacts when we click on it. open a `Transcript` by dragging one from the left flap, create a new instance of `EscapingFlasher` and click on it.

Method 1.3

```
EscapingMorph>>mouseDown: evt


   Transcript show: ' mouse down' ; cr
```

Our goal is that when we click on the morph it should change its direction. So we should think about how can we represent this behavior. Different approaches exist, the first one that came in our mind is that we just have to be able to change the point that is added in the time method. To change the direction, we just have to negate the point. By adding a negated point the flasher will go up instead of down. What we need is a way to know if the last point added was positive or negative. When the morph is clicked we just have to revert this information. Even simpler if we store a point representing the last point added to the position we just have to negate it each time the morph is clicked. So we will add an instance variable to the `EscapingFlasher` class so that each instances will now know the last point added. Modify the class `EscapingFlasher` according to the class **??**.

Class 1.2

```
FlasherMorph subclass: #EscapingFlasher
   instanceVariableNames: 'point'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Flasher'
```

Define the method `initialize` that is called each time a new object is created. Here we want to let the morph initialized itself as it is done by its superclass then initialize the point. ThatÕs why we use `super initialize` as first line of the method **??**.

Method 1.4

```
EscapingMorph>>initialize

   super initialize.
   point := 2@2.
```

For a baby the default size of the flasher is too small so we propose you to take the advantage of initializing the flasher to change its size. Try the method **??**

Method 1.5

```
EscapingMorph>>initialize
   "self new openInWorld"


   super initialize.
   self bounds: (self bounds scaleBy: 3).
   point := 2@2.
```

Now changing direction is just negating the point each escaping flasher morph holds. Define the method `changeDirection` as described in the method **??**. Here this method adds a specific behavior to the class `EscapingMorph` and does not hide or override any behavior of the superclass.

Method 1.6

```
EscapingMorph>>changeDirection

   point := point negated
```

Redefine the method `mouseDown:` to call the method `changeDirection` as shown in the method **??**.

Method 1.7

```
EscapingMorph>>mouseDown: evt

   self changeDirection
```

3 Accelerating Flasher

Any good game should stress a bit its player else it quickly starts to get bored. So we would like our escaping flasher to accelerate each time we click on it. To still let a chance to the player to get it we would like the following behavior: clicking on it should accelerate it if the morph is not at its maximum speed. If this is the case it should slow down to its original speed. Hence, the game will be to click fast when the flasher is accelerating until it is getting slowly again.

Class 1.3

```
FlasherMorph subclass: #EscapingFlasher
instanceVariableNames: 'point increasing '
classVariableNames: ''
poolDictionaries: ''
category: 'Flasher'
```

Method 1.8

```
initialize

super initialize.
point := 2@2.
self bounds:  (self bounds scaleBy: 3).
increasing := true
```

Method 1.9

```
mouseEnter: evt

increasing
ifTrue: [self speedUp]
ifFalse: [self slowDown]
```

Method 1.10

```
slowDown

point < (5@5)
ifTrue: [increasing := true]
ifFalse: [ point := point - (2@2)]
```

Method 1.11

```
speedUp

point > (20 @20)
ifTrue: [increasing := false]
ifFalse: [point := point + (5@5)]
```

4 Saving the parents

Parents are usually less good than kids to play game but they are better at typing keyboard. So we ant to have a mode where nervous parents can stop the morph and reset its speed. Doing so you will learn how a morph can manage keyboard events. Define a subclass of the `EscapingFlasher` if you want to keep its behavior. The following assumes you did so. Handling keyboard is slightly more complex than mouse event. It requires two extra methods. First, we specify that the moprh wants to get keyboard events by defining the method `handlesKeyboard:` to return true. Then we define the method `keyDown:` to handle the key and perform associate action.

Method 1.12

```
EscapingFlasherWithKeyboard>>handlesKeyboard: evt


   ^ true
```

Method 1.13

```
EscapingFlasherWithKeyboard>>keyDown: anEvent
   "Handle a key down event. The default response is to do nothing."


   | char |
   char := anEvent keyCharacter.
   char = $a
      ifTrue: [self resetSpeed].
   char = $q
      ifTrue: [self slowDown].
   char = $s
      ifTrue: [self stop].
   char = $l
      ifTrue: [self startStepping].
```

Now the problem is that the focus of the mouse is not given to the morph. If you type on the morph the letters will continue to appear in your browser or workspace. To assign correctly the focus, define the methods `mouseEnter:` and `mouseLeave:` as follow.

Method 1.14

```
EscapingFlasherWithKeyboard>>mouseEnter: evt

   evt hand keyboardFocus: self
```

Method 1.15

```
EscapingFlasherWithKeyboard>>mouseLeave: evt


   evt hand releaseKeyboardFocus: self
```

Now you can implement the functionality you want to associate with the keys. For example, the method `resetSpeed` can be defined as follow:

Method 1.16

```
EscapingFlasherWithKeyboard>>resetSpeed


   point := 2@2.
```

The methods `stop` and `startStepping` are defined on the class `Moprh`. stop stops the animation of the morph, i.e., the `step` method is not invoke anymore and `startStepping` does the opposite.

We could improve the code by defining a method named `defaultSpeed` returning the point `2@2` and to use it everywhere we would need to initialize the speed. So such a method would have to be defined into the class `EscapingFlasher`. Note that while a system evolves the code changes and it is frequent to have to adapt or refactor the code of classes to get a better system. A solution is only good for the problem it solves today, tomorrow we may have another slightly different problem that will require to modify our solution.

5 Exercises

Browse the class FlasherMorph and understands how it works. Try to implement the following variations and have fun discovering how to do it.

- ◦ The flasher should be flashing in green instead of one when it is hit for the first time or when is going up.
- ◦ Change the intensity of the color according to the speed of the flasher. The more you click it the brighter. Look into the class FlasherMorph and Color.
- ◦ Change the size of the flasher according to the number of times.
- ◦ Define the direction to which the morph should go randomly. 10 atRandom returns a number between 1 and 10.

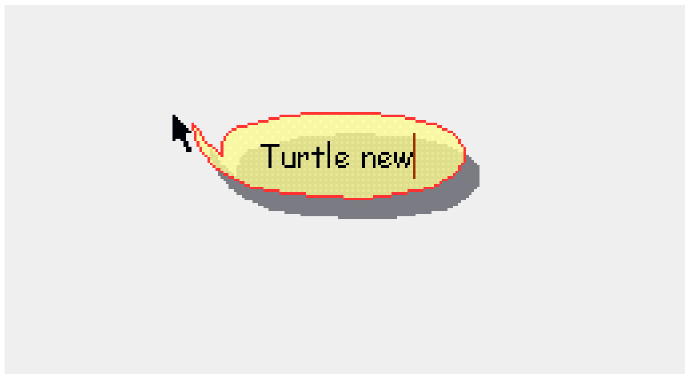
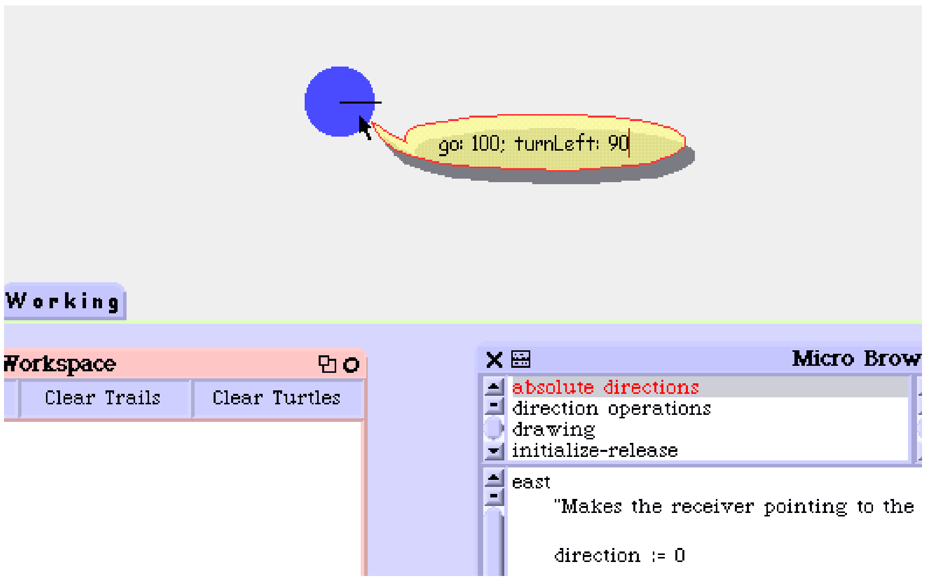


Figure 1: Creating a turtle by asking the class to create an instance.

C Direct interaction using Morphic Wrapper

Here we present only two screenshots showing the use of Morphic Wrapper [Mor] to support direct interaction. The figure 1 shows that we ask to create a turtle by typing in the air. In the future we plan to introduce classes as Morphs as proposed in Morphic Wrappers to support the idea that classes are factory of objects. The figure 2 shows how messages are sent to this turtle by asking it directly.



```
east
	"Makes the receiver pointing to the

	direction := 0
```

Figure 2: Sending a message to a turtle by directly interacting with it using the Morphic Wrapper framework.