The Development of an Application in Smalltalk

Guy Lapalme
Département d'informatique et de recherche opérationnelle
Université de Montréal
Montréal, Québec
Canada H3C 3J7
e-mail:lapalme@iro.umontreal.ca

May 4, 1998

Abstract

This paper describes the development of a small application in Smalltalk. It illustrates the concepts of message passing, inheritance, class variables and methods, graphic animation and the Model-View-Controller approach to achieve a mouse and menu driven application.

1 Introduction

This paper shows the development of a small application in Smalltalk that has the advantage of being easily understood and visualized. The best way to learn Smalltalk is by experimenting with the system; it is very hard to learn "by the books". Ideally this article should be read after having loaded the corresponding classes (a machine readable version is available from the author), using the `System Browser` to inspect the code of the methods and experimenting sending messages to objects. But we try anyway to give a feeling of the development of an application for those who dot not have access to a Smalltalk environment. This application was developped using VisualWorks 2.0 from Park Place but it should be portable to other Smalltalk environments such as Visual Age.

Smalltalk is a very rich programming environment that allows a fast development of software prototypes because it is easy to reuse and to extend the system predefined software modules. It is an integrated environment, the system itself being defined in Smalltalk, and so there is only one paradigm: objects that receive messages.

We briefly review Smalltalk using the objects defined for our application. Goldberg and Robson [1] should be consulted for full details on the language and the system defined classes; Goldberg [2] describes the programming environment. Kahler and Patterson [4] also develop an application in Smalltalk and through it one learns how to use the environment. Hopkins and Horan [3] describe the use of the VisualWorks environment that we used to develop our application. Our paper takes for granted that one knows the programming environment and focuses on the structure of the classes involved.

2 Description of the application

This application describes the movements of graphical bodies that are displayed within a setting. Each body can "move" and display a sequence of pictures to simulate its movements. Graphical

applications in Smalltalk are defined using the Model-View-Controller paradigm which can be intuitively described as follows: the `model` is defined using a group of objects that represent the application; in our case, the setting will be represented by an object and each animal bodies will also be represented with distinct objects at a given position within the setting. The `view` defines how the model is displayed; in our case, we have both a textual view tracing each action of the bodies within the setting and a graphical view of a body drawn within the window corresponding to the setting. The `controller` deals with the input of the user (mouse clicks, menu selections, characters typed at the keyboard) and sends messages to the `model` to alter itself according to the input; the `model` then notifies the `view` that it has changed so that the display is updated appropriately. We will come back to these important concepts in section 6, but we will first give a short introduction to Smalltalk.

3 An introduction to Smalltalk

In an object oriented system, everything is described in terms of objects having an internal state and responding to messages. Objects are created from a mold (called a *class*). In our case, we have *animated bodies* moving within a *setting*. As most of the *animated bodies* will have the same behaviour, we define the class `AnimatedBody` from which we create objects, called instances that evolve within an instance of another class `Setting` which, not surprisingly, does not behave like an *animated body*. Given that we have two kinds of distinct animated bodies, *butterflies* and *flowers* that do not have exactly the same behaviour in certain cases, we define two subclasses of `AnimatedBody`, `Butterfly` and `Flower` in which we define their specific behaviour when they differ from the one defined in `AnimatedBody`.

Smalltalk uses the following convention for identifiers: global identifiers such as class names start with a capital letter while local ones are not capitalized. When an identifier is made up of more than one word, the words are not separated and each one (except possibly the first) is capitalized, for example, `theButterflyOverAFlower` is a local variable while `Setting` is a global one.

We first create a `Park`, an instance of `Setting`, in which instances of the classes `Butterfly` and `Flower` move. The `Setting` will eventually be represented on the screen by a window that delimits the visible area of activity of the animated bodies: if a body moves out of the `Park`, it is not displayed. An `AnimatedBody` is represented by a picture on the screen. Given that we have already created a `Flower` named `Daisy` and a `Butterfly` named `Peter`, Table 3 shows messages that can be sent to these objects.

Object	message	action
`Peter`	`turn`	change its direction
`Daisy`	`open`	show the flower as open
`Peter`	`moveTo:`	change its position
`Park`	`move:to:by:`	make a body change its position gradually

Table 1: Messages to instances of classes

In Smalltalk, messages can be unary, binary or nary. The syntax seems at first glance a bit unusual compared to other computer languages but, being systematic, it is very simple to learn and to get used to.

Unary messages are indicated by putting the name of the message **after** the receiver of the message (like in natural language ...), such as `Peter turn` or `Daisy close`.

Binary messages are indicated by special symbols such as `@`, `+`, `-`, `*` ... which are called operators in other languages; for example, in `500@600`, `@` denotes the binary message sent with the value `600` to the integer `500` and is used in Smalltalk to create an object of the class `Point` that indicates a position on the screen.

Nary messages are indicated by identifiers followed by a colon and a value to be sent with the message. For example, `Peter moveTo:600@500` sends the message `moveTo:` with the `Point` created by `600@500` to `Peter`. `Peter` should then disappear from its current position and reappear at the position `600@500`. More than one value can be sent along in a message by preceding each of them by an identifier followed by a colon; for example,

```
Park move: Daisy to: 600@500 by: 20
```

sends the message `move:to:by:` to `Park` with the values `Daisy`,`600@500` and `20`. Daisy should then move from its current position to its new position (`600@500`) by steps of `20`.

Unary, binary and nary messages can be mixed. Unary messages are sent before binary ones which are sent before the nary ones. When they have the same priority, they are sent from left to right. Parentheses can be used to override this form of associating values and messages. For example,

```
Park move: Peter to: 100 @ 4 factorial by: 15+10*3
```

is equivalent to

```
(Park move: Peter to: ( 100 @ (4 factorial)) by:((15+10)*3))
```

where the messages are sent in this order:

- `factorial`
- `@ 24`, `+ 10`, `* 3`
- `move: Peter to: (Point x:100 y:24)` [1] `by:75`

Everything in Smalltalk is done using message passing even control structures. An alternative statement is constructed using a message sent to an instance of the `Boolean` class; for example:

```
Daisy open
    ifTrue:[Peter turn]
    ifFalse:[Peter moveTo: Daisy position + (0@100)].
```

this is the `ifTrue:IfFalse` message sent to the boolean value created by `open` sent to `Daisy`. A series of Smalltalk messages between square brackets is called a `block` which are not executed until the block receives the message `value`. This message is seldom sent explicitely but it is usually sent by the system predefined methods; here it is sent by `ifTrue:ifFalse`.

An iterative statement is constructed using blocks evaluated repeatedly. For example.

```
1 to: 4 do:
    [:i | Peter direction: i].
```

[1] this instance of the `Point` class is created by the `@` operator

makes Peter go successively in direction 1, 2, 3 and 4. This might look like an "ordinary" do-loop construct found in all computer languages but look closely: it is the `to:do:` message sent to the integer 1! The `Integer` class defines this method by sending the `value:` message to the block given as second argument (non evaluated). The `:i` before the vertical bar in the block is the parameter that gets bound to the argument of the `value:` message. Smalltalk has a very large set of iterators that can be sent to objects designating groups of objects; for example:

```
allButterflies do:
    [:aButterfly | aButterfly moveTo: aButterfly position + (50 @ 50)]
allFlowers collect:
    [:aFlower | aFlower isOpen]
```

the first statement making all elements of the list `allButterflies` move by `(50@50)` and the second giving a list of all open flowers. The scheme provides a high level of abstraction for iterating over groups of objects and makes it easy for a user to define new control structures. A conditional loop is constructed by sending a message to a block which can thus be evaluated as often as needed. For example:

```
[delta < distance]
    whileTrue:[distance := distance - 10].
```

the second block is executed while the first block evaluates to `true`. The assignment sign `:=` is not a message, it only saves a reference on the object created on the right of the assignment.

4 The `AnimatedBody` class

This class describes the general behavior of all the bodies that move around a setting. It defines instance variables representing the state of each body and it implements messages describing the actions performed by all bodies.

4.1 Instance variables

A body is defined by a name, a position in a setting. As we want to simulate movement, each body is defined by an array of bitmaps pictures (`Images` in the Smalltalk terminology) giving different aspects displayed cyclicly. A body can be represented by different pictures to indicate where it is heading; for example, it can be seen from the front, back or side. Figure 1 shows in a window the four directions (along the horizontal axis) and the two aspects for each (on the vertical axis) for a butterfly. A flower has only two "directions" and one aspect. As every `AnimatedBody` has this behavior, it is defined in this class but the exact picture to display is left to each kind of `AnimatedBody`. We will see in the next section how the pictures are kept but each `AnimatedBody` has its own direction and aspect.

`name`	identifies a body
`position`	where it is in the setting
`direction`	the current direction
`aspect`	the current aspect

Table 2: Instance variables of an `AnimatedBody` instance

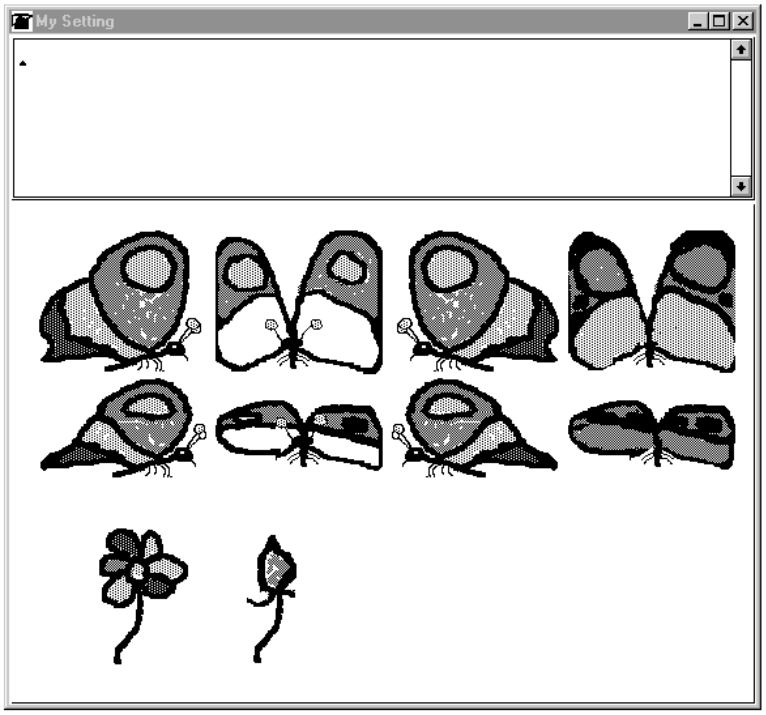

Figure 1: Directions and aspects of `AnimatedBody` instances. There are four directions and two aspects for a `Butterfly`. There are two directions and one aspect for a `Flower`.

4.2 Instance Methods

In Smaltalk, the internal state of an object is only accessible via messages so that eventually the structure of the objects can change without disturbing the objects that use it provided that the name of the message and its semantics stays the same. So we define "trivial" methods to access and modify these values. By convention, the name of the method, given here as the first line of the method, is the same at the one of the instance variable. The code of the method is indented from the name. The caret `^` signals that the following expression is the value returned from the call of the method. If there is no such arrow in a method, it returns `self`, i.e. a reference on the object that answered the message, after executing the code. These simple methods are essential because an instance value is not directly accessible from other objects; in this way, other objects can know the value but cannot change it.

```
aspect
    ^aspect

direction
    ^direction

name
    ^name

position
```

```
    ^position
```

For changing the values of the instance variables, we define methods that, by convention, have the same name as the instance variable but followed by a colon. They are thus nary messages, which indicate that a parameter follows the name of the method. This parameter is the new value of the parameter. For changing the position of a body this is quite simple.

```
position: p
    position := p
```

But changing the `aspect` and `direction` is a bit more complicated because the value has to lie within an acceptable number of possible values returned by the methods `numberOfAspects` and `numberOfDirections` (to be defined in section 4.3) which are sent to the object itself via the keyword `self`. `\\` is the modulo operator. Comments are indicated between double quotes `(")`. As a body should not care about the aspect and direction numbers, we build on these methods two other methods `nextAspect` and `nextDirection` that simply change to the next one.

```
aspect:a
    "new aspect of the body. aspect is a number between 1
     and numberOfAspects"
    aspect := ((a-1) \\ self numberOfAspects)+1

direction: d
    "new direction of the body; direction is a number between 1
     and numberOfDirections"
    direction := ((d - 1) \\ self numberOfDirections) + 1

nextAspect
    "change to the next aspect of the body"
    self aspect: aspect+1

nextDirection
    "change to the next direction of the body"
    self direction: direction+1
```

As we do not expect that an object changes it name during its life, we define a method that will be called only at initialisation time and that also initialises other instance variables.

```
name: aName
    self position:0 @ 0.
    self direction: 1.
    self aspect:1.
    name := aName
```

Moving a body is merely changing its position and aspect by calling some of these methods. Note again that an object can use some of its own methods by sending the message to `self`. Although the object could change its position directly by assigning a new value to the instance variable, it is more appropriate to call the `position:` method.

```
moveTo: aPosition
    self nextAspect.
    self position: aPosition
```

For a body to appear on the screen, we use a collection of images corresponding to each direction and aspect. These images will be displayed on the screen. The access to the current image is a bit more complicated (and interesting...). As we would like all instances of the same subclass of an `AnimatedBody` to share the same array of images, they cannot be instance variables because in this case, they would be repeated in each class. So we make them instance variables of the class instead. Note that this is different from the class variables which are merely a sort of global variable shared by all instances of the class. We will show examples of class variables in sections 4.5.2 and 4.5.3. The images of each subclass have to be distinct: i.e. a `Butterfly` does not have the same images as a `Flower`.

4.3 Class instance variables and methods

The class `AnimatedBody` thus needs three class instance variables described in table 3.

`film`	collection of images of a `AnimatedBody`
`nbAspects`	number of aspects
`nbDirections`	number of directions

Table 3: Class instance variables of an `AnimatedBody` class

For getting access to these variables, we define three class methods, i.e. methods that are sent to the class instead of the more usual instance one. Another example of a class message and one of the most often used is the `new` message that is sent to a class for the creation of an instance of the class.

```
film
    ^film

numberOfAspects
    ^nbAspects

numberOfDirections
    ^nbDirections
```

The display of an image on the screen is done using `OpaqueImage`s and we will explain later how they are used. The creation of these images is built from a shape and mask created using the MaskEditor of Smalltalk which saves the shapes and masks as methods in the classes. The shapes are saved under conventional names of the form *Classname* followed by a and d where a is an aspect number and d is a direction number. The corresponding "mask" is the name of the shape followed by M.

The film instance variable of the class is initialised with an array of dimension `nbAspects` each element being an array of `nbDirections` images. This method is called from the subclasses of `AnimatedBody` but as it is the same for all subclasses, it is defined in the class.

```
initFilm: nbA by: nbD
    "Initialize the Film from the resources"
    |f nomFig nomSh |
    nbAspects:=nbA.
```

```
    nbDirections:=nbD.
    film := Array new: nbAspects.
    1 to: nbAspects do:
        [:a |film at: a put: (Array new:nbDirections)].
    1 to: nbAspects do:
        [:a |f:=film at:a.
            1 to: nbDirections do:
                [:d  |
                    nomFig:=self printString, a printString, d printString.
                    nomSh :=nomFig,'M'.
                    f at: d put: (OpaqueImage
                                    figure:(self perform:(nomFig asSymbol))asImage
                                    shape:(self perform:(nomSh asSymbol))asImage).
                ]].
```

The last (but not least) class method is the method for creating an `AnimatedBody` instance which creates the instance using `new` and then calls the appropriate initialisation instance methods.

```
name: aName at: aPosition
    ^(self new name: aName) initializeAt: aPosition
```

4.4 Instance methods (continued)

Now for an `AnimatedBody` instance to access the film of the image of its class, it needs to call the `film` method of its class. The unary `class` message sent to an object returns its class to which the `film` class method is then sent. From this we define a method to return the current image of a body. Using the array of arrays of images initialized in `initFilm`.

```
image
    ^(self class film at:self aspect) at: self direction
```

It is also useful to have a method for printing an object i.e. the values of the instance variables with appropriate labels. This is conventionaly done by defining the `printOn:` method which is called by the system for showing the value of an object. The default definition merely prints the name of the class preceded by `a` such as `a Flower` or `an AnimatedBody`. The following methods make use of messages that add information at the end of an output stream: `nextPut:` adds a character, `nextPutAll` adds a string indicated here between single quotes (') and `space` adds a space character. The method uses the `printString` method which returns a character representation of an object; internally this method is defined using `printOn:` method defined on other objects. The semi-colon is used to indicate a cascade of messages sent to the same receiver i.e. message following the semi-colon is sent to the same receiver as the one preceding it. `printOn:`is called by the system whenever it needs to print an object.

```
printOn:stream
    "print information (name,direction, aspect, position)
      about the current object"
     stream
        nextPutAll:self class printString;
        space;
```

```
        nextPutAll:name;
        nextPutAll:'[Dir=';
        nextPutAll:direction printString;
        nextPutAll:',Asp=';
        nextPutAll:aspect printString;
        nextPutAll:',Pos=';
        nextPutAll:position rounded printString;
        nextPut:$].
```

Now that we have defined all the necessary methods of a general `AnimatedBody`, we can look at the specific behaviour of a subclass.

4.5 Subclasses of `AnimatedBody`

4.5.1 The `Butterfly` class

We define a `Butterfly` as a special `AnimatedBody` that can do everything an `AnimatedBody` can do and more. A `Butterfly` can appear to be fast or slow depending on the value of an instance variable `step` accessed and updated by the following methods:

```
step
    ^step

step: newStep
    step := newStep
```

We define a new movement method `turn`.

```
turn
    "the Butterfly turns itself"
    self nextDirection
```

We redefine `moveTo:` so that a `Butterfly` turns in the right direction before starting to go at a new position. This behavior is a "superset" of the `moveTo:` of an `AnimatedBody`.

```
moveTo: destination
    "turns in the right direction and then moves to the destination"
    self position x < destination x
        ifTrue: [self direction:1]
        ifFalse: [self direction: 3].
    super moveTo: destination.
```

This method uses `moveTo:` defined in `AnimatedBody` using `super` as receiver of the message. `super` designates the current object but the search for the methods starts at the superclass of the class where the `super` appears.

With all these tools, we define a new method `visit: aFlower` where the Butterfly goes to a `Flower` and depending on whether it is open or not then it turns or else it stops a bit farther.

```
visit: aFlower
    self moveTo: aFlower position.
    aFlower isOpen
        ifTrue: [self turn]
        ifFalse: [self moveTo: position + (0 @ 100)]! !
```

The display of a `Butterfly` is obtained by redefining the `printOn:` method which starts by calling the one defined in `AnimatedBody`.

```
printOn: stream
    super printOn:stream.
    stream
        nextPutAll:'[Step=';
        nextPutAll:step printString;
        nextPut:$].
```

The initialization of a Butterfly is done as follows:

```
initializeAt: aPosition
    NbButterflies := NbButterflies+1.
    step := 20.
    self position: aPosition.
```

so that a `Butterfly` flies to its position when it is created. But how is a `Butterfly` created? by the `name:at:` class method of `AnimatedBody` described in section 4.3: Let us look the messages that are sent in the following call:

```
Butterfly name: 'Peter'  at: 100@100
```

`new` is sent to `self` that designates the `Butterfly` class (although `name:at:` is defined in `AnimatedBody`) and this creates a new Butterfly with all its instances variables set to `nil`. Then the `name: 'Peter'` message is sent to this `Butterfly`; this method, found in `AnimatedBody`, initializes the variables defined in it and returns `self` (by default) which finally receives the `initializeAt: 100@100` message that initializes `step` and makes the `Butterfly` go to its new position.

4.5.2 `Butterfly` class variables and methods

As we will need to keep track of the number of instances of `Butterfly`, we need some kind of global variable which is accessible from all butterflies. This can be obtained using a *class variable* that we call `NbButterflies`. In this application, we could have got the same effect by using a class instance variable like we did for `film`, but we want to illustrate a use of a class variable. This variable can used as a local variable from any `Butterfly` instance but we define also the `number` class method to give access to this value class variable. It will be used in the `newButterfly` and `newFlower` methods to suggest a unique name for a new instance of an `AnimatedBody`.

For initializing the class, we define the following class method which initialises the class variable and the `film` class instance variable.

```
initialize
    "Butterfly initialize"
    NbButterflies:=1.
    self initFilm: 2 by: 4.

number
    ^NbButterflies
```

4.5.3 The `Flower` class

The structure of the `Flower` class is basically the same as the `Butterfly` one except that it is much simpler because it can only `open` or `close` itself. The fact that it is open or closed depends on its current `direction` (defined in `AnimatedBody`). It also has only one `aspect` so it needs only two bitmaps. Like `Butterfly`, it inherits the `moveTo:`, `changeAspect`, etc. from `AnimatedBody`. It defines:

```
isOpen
    ^self direction = 1


close
    self direction:2

open
    self direction:1
```

The initialization of an instance and of the `Flower` class are done along the same lines as for the `Butterfly`.

```
initializeAt: aPosition
    NbFlowers := NbFlowers+1.
    self moveTo: aPosition
```

`Flower` has a class variable `NbFlowers` for counting its instances and the class methods are

```
initialize
    "Flower initialize"
    NbFlowers:=1.
    self initFilm: 1 by: 2.

number
    ^NbFlowers
```

5 The `Setting` class

The setting model is conceptually only an object that knows about *AnimatedBodies* that evolve within its borders. But at the moment there are no borders until we wish to display this model. This will be done by the *view* (see section 7) on this model.

A `Setting` has two instance variables: `bodies` is a collection of `AnimatedBody` that evolve within the setting and `traceStream` an output stream for keeping a written trace of the actions done by the bodies. For the moment, the trace is written on the `System Transcript` and so it is possible to test the Setting and imagine the motions of the bodies before going into the specifics of a graphical window. These variables are initialized by the following instance method:

```
initialize
    traceStream :=Transcript.
    bodies := Dictionary new.
```

which is called by the following definition of the `new` class method:

```
new
    ^super new initialize
```

The instance variables are accessed via the following methods:

```
bodies
    ^bodies


trace
    ^traceStream contents
```

For tracing we define the following method which first prints a description of the `AnimatedBody` and then writes the name of the action that will be given as a string. The calls to the `cr` and `flush` methods are used to end the current line and force the output to appear in the output stream (this is not strictly necessary for the `Transcript` but will be useful later).

```
trace: body action: action
    "writes a line of trace starting with the info from the body"
    body printOn:traceStream.
    traceStream nextPutAll:action;cr;flush.
    self changed: #trace
```

`changed:` method is used to indicate that an aspect of the model has been modified and that the view should be updated to reflect the change. In this case, this means that the trace has changed and thus the view corresponding to the trace will be redrawn in some way. It is up to the underlying system to signal the update and in section 7, we will see how the view gets notified of the change.

The setting is our basic entry point and it is the "dispatcher" of messages for all bodies that evolve within it. The setting creates the bodies and then ask them to move.

The creation of bodies are defined by the following methods which create an `AnimatedBody`, add it to the `bodies` instance variable and indicate the creation on the trace. Because, a new body has been created, it should be drawn on the screen by the view which is notified via the `changed:with:` method which is given the `#body` aspect and a reference on the created body.

```
butterflyNamed: name at: position
    "creates a new Butterfly"
    |b|
    b:=Butterfly name:name at:position.
    bodies at:name put:b.
    self changed:#body with:b.
    self trace: b action:'created'.
    ^b


flowerNamed: name at: position
    "creates a new Flower"
    |f|
    f:=Flower name:name at:position.
    bodies at:name put:f.
    self changed:#body with:f.
    self trace: f action:'created'.
    ^f
```

The actions of the bodies are defined by instance methods which merely dispatch the message to an animated body and traces the action. For each modification of a body, the model indicates that it has changed by sending the `changed:with:` message indicating which body has changed and so that the current display of a body is no longer valid, but it should be redrawn only after the new position, aspect or direction has been updated. Even though, in the next five messages, the position of the body is not changed, we must indicate that it has changed because all views do not have the same shape so the old area must be indicated as being invalid. This is why there are two calls to `changed:with:` method the first of which is called with `#nobody`.

```
ask:b inPlace: nb
    "Changes aspect nb times but always at the same place"
    (b isMemberOf: Butterfly) &(nb > 0) ifTrue:
        [nb timesRepeat: [self changed:#nobody with:b.
                          b nextAspect.
                          self changed:#body with:b].
        self trace:b action:'in place for ',nb printString].


ask:b visit:f
    (b isMemberOf: Butterfly)&(f isMemberOf:Flower) ifTrue:
        [self move:b to: f position by:b step.
        f isOpen
            ifTrue:[self turn:b. self ask:b inPlace:5]
            ifFalse:[self move:b to:f position+(0@100)].
        self trace:b action:'visited ',f name.]


close:body
    ((body isMemberOf: Flower) and: [body isOpen]) ifTrue:
        [self changed:#nobody with:body.
        body close.
        self changed:#body with:body.
        self trace:body action:'closed']


open:body
    ((body isMemberOf: Flower) and: [body isOpen not]) ifTrue:
        [self changed:#nobody with:body.
        body open.
        self changed:#body with:body.
        self trace:body action:'opened']


turn:body
    (body isMemberOf: Butterfly)ifTrue:
        [self changed:#nobody with:body.
        body turn.
        self changed:#body with:body.
        self trace:body action:'turned']
```

For moving, one must be more careful, because the view has to remove the body from the current position and to draw the new position. So there are two modifications: one at the old position but the view should not redraw it, and one at the new which in this case, it should redraw it. This is why again there are two calls to `changed:with:` method the first of which is called with `#nobody`.

```
move: body to:position
    "indicate change of position but not immediate repaint"
    self changed:#nobody with:body.
    "move now"
    body moveTo: position.
    "indicate new change of position and force repaint"
    self changed:#body with:body.
    self trace:body action:'moved'
```

For moving gradually, we repeatedly call the `moveTo:` using the step given as parameter.

```
move: body to: goal by:step
    |distance|
    distance := goal dist: body position.
    [step < distance] whileTrue:
    [self move:body to:body position
                      + (goal - body position * (step / distance)).
        distance := goal dist: body position].
    self move:body to: goal.
```

We have now created four classes: `AnimatedBody` which defines the general behavior of the bodies, two subclasses `Butterfly` and `Flower` having more particular actions and variables and one class `Setting` where the `Butterfly` and `Flower` instances are moving around. At this point, it is possible to create instances of these classes to get a written trace on the `System Transcript` of the actions of the bodies and also to inspect their instance variables.

Here is a small part of the interaction with this model that is typed in a workspace; the variable names are capitalized because the objects created will be simply kept in global variables:

```
Park :=Setting new.
Peter := Park butterflyNamed:'Peter' at:100@100.
Daisy:= Park flowerNamed:'Daisy' at:200@200.
Park ask: Peter visit:Daisy.
```

The following trace appears in the `System Transcript` where we see that the `Peter` and `Daisy` are first created and then `Peter` visits `Daisy` by moving gradually towards `Daisy`, turning and staying in place.

```
Butterfly Peter[Dir=1,Asp=1,Pos=100@100][Step=20]created
Flower Daisy[Dir=1,Asp=1,Pos=200@200]created
Butterfly Peter[Dir=1,Asp=2,Pos=114@114][Step=20]moved
Butterfly Peter[Dir=1,Asp=1,Pos=128@128][Step=20]moved
Butterfly Peter[Dir=1,Asp=2,Pos=142@142][Step=20]moved
Butterfly Peter[Dir=1,Asp=1,Pos=157@157][Step=20]moved
Butterfly Peter[Dir=1,Asp=2,Pos=171@171][Step=20]moved
```

```
Butterfly Peter[Dir=1,Asp=1,Pos=185@185][Step=20]moved
Butterfly Peter[Dir=1,Asp=2,Pos=199@199][Step=20]moved
Butterfly Peter[Dir=1,Asp=1,Pos=200@200][Step=20]moved
Butterfly Peter[Dir=2,Asp=1,Pos=200@200][Step=20]turned
Butterfly Peter[Dir=2,Asp=2,Pos=200@200][Step=20]in place for 5
Butterfly Peter[Dir=2,Asp=2,Pos=200@200][Step=20]visited Daisy
```

With this trace, we can be more confident that our application model works as intended. It is also possible to inspect each object to see the values of their instance variables.

We will now show how to view the behavior of the bodies within a window and how to send the messages defined in `Setting` using menu selections. In fact, `Setting`, `AnimatedBody` and its subclasses do not have to modified, we will define a view and a controller class on the model described by the `Setting` class.

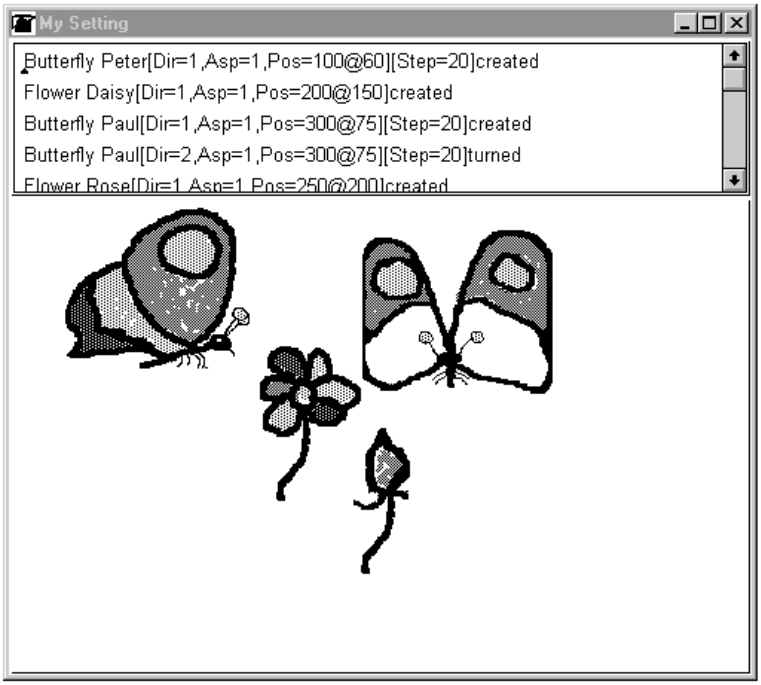

Figure 2: Snapshot of our application

6 The Model-View-Controller concept

As promised, we now embed our application in a standard Smalltalk window as shown in figure 2. We first briefly review the Model-View-Controller (MVC) concept which is at the heart of the user-interface of Smalltalk. Of course, we could bypass it completely and program an entirely new user interface but we choose not to because we want the user to keep the same kind of "reflexes" and actions in manipulating the `Setting` with `Butterflies` and `Flowers` as when manipulating other Smalltalk system objects. This choice enables the reuse of many predefined classes and thus eases and speeds up the development.

Anybody who has programmed an interactive application that uses menus and windows will appreciate that only a few pages of code are necessary in Smalltalk; much more has to be done in other environments where not only many system procedures calls with "strange" arguments have to be used but also one must often constructs ad-hoc "definition files" using arcane keywords and parameters.

Here everything is defined in Smalltalk and the same message sending paradigm is kept. Being a good metaphor for a user-interface (the user sends messages to the system which also answers via messages) it is no wonder that practically all approaches to windowing environments are object-oriented.

As its name indicates, model-view-controller is trilogy of classes:

model the application we want to show; here it is the `Setting`. The `AnimatedBody` class and its subclasses do not have to be modified to be used within a windowing environment.

view this is what is shown on the screen and is often divided into subviews each showing an aspect of the application. In our case, we decide to separate our view into two parts: one is the display of the bodies and the other is the written trace output.

controller this part deals with the user input and is responsible for dispatching appropriate messages to the model or the view depending on the mouse clicks and keyboard pressings.

The classes are related to one another because in the model-view-controller paradigm, a view instance is a *dependent*[2] of the model instance; the controller has direct access (via instance variables) to both the model and the view. Ideally the application is "pure" and should not be concerned with the fact that it is being displayed in a window; it only needs to signal that it has changed and so the view, its dependent, receives an `update:` message. When the system detects that a part of the window corresponding to the view should be redrawn, it calls the `displayOn:` method. We will show examples of implementation of these messages in section 7. The view and controller usually make great use of system predefined views and controllers either by subclassing, by sending them messages or by creating new instances.

As the view and the controller are intimately related, it is hard to describe them "sequentially". We have to make a few forward references or some approximations to be explained or detailled later. Figure 2 shows a `SettingView` as it appears on the screen.

7 The `SettingView` class

We want to use a Smalltalk view divided in two: one for the written trace and another for the display of the body images. It is the job of the view to reflect any changes done in the model. Instead of using the `System Transcript` like we did in section 5 for the written trace that we used in section 5, we use a `WriteStream` which is a Smalltalk class that implements a stream of characters whose contents can be displayed in a scrollable window by an instance of a `TextView`. As we want a separate `TextView` for each `Setting`, we redefine the `Setting` initialisation method as follows:

```
initialize
    traceStream :=WriteStream on:''.
    bodies := Dictionary new.
```

[2] in Smalltalk, if an object receives the `changed:` message, it sends the `update:` to all its dependents. `o1` is made dependent upon `o2` by sending `addDependent: o1` to `o2`

The setting should behave like any other Smalltalk window: its label at the top left gets "highlighted" when the view is active, roughly that means that the keyboard pressings are directed to it. This user interface protocol is implemented in a class called `ScheduledWindow`. So we create an instance of a `ScheduledWindow` controlled by an instance of a `StandardSystemController` in which we install two subviews: one for the display of animated bodies and one for the written trace. A `SettingView` is created and installed in a `ScheduledWindow` as the `open` class method which creates a `Setting` and then calls the `open:` class method on it.

```
open
    "SettingView open"
    ^self open: Setting new
```

The `open:` method creates the window composed of two views.

```
open: aSetting
    | topWindow cp textView  |
    topWindow := ScheduledWindow new.
    topWindow controller: StandardSystemController new.
    topWindow model: aSetting.
    topWindow label: 'My Setting'.
    topWindow minimumSize: 500@400.
    cp := CompositePart new.
    topWindow component: cp.

    cp add: (self new model: aSetting ; yourself )
        borderedIn: (LayoutFrame new leftFraction: 0;
                                     topFraction: 0.25;
                                     rightFraction: 1;
                                     bottomFraction: 1).
    textView := TextView on: aSetting
                         aspect: #trace
                         change: nil
                         menu: nil
                         initialSelection: nil.
    cp add: (LookPreferences edgeDecorator on: textView)
        borderedIn: (LayoutFrame new leftFraction: 0;
                                     topFraction: 0;
                                     rightFraction: 1;
                                     bottomFraction: 0.25).
    topWindow open.
    ^aSetting! !
```

We first create and initialize a `ScheduledWindow` and give its label. We then create a list of composite parts `cp` in which two subviews will be added. The window manager deals with the exact positionning of the views that will keep their proportion when the window is resized. Then we define a subview for the drawing whose window (i.e. the range of positions for the bodies to appear in the window); it takes the full width, as defined by `leftFraction` and `rightFraction` and the bottom 75% of the whole view as defined by `topFraction` and `bottomFraction`. For the trace subview taking the top 25% of the screen, we rely on the `TextView` which implements the

displaying and scrolling of text in a window. The whole window is finally opened. The drawing is managed by a combination of the Smalltalk objects that interact with the host computer operating system. Intuitively, redrawing is only done for the regions of the screen that have changed, either because the window has been partially put on top after being hidden or because some parts of screen may now be considered as *invalid*. As for the system cannot infer which parts of a view are now invalid because of modifications in the model, the model must indicate explicitly which part of the view are invalidated.

A simple example in our application is when a flower closes then the picture should be redrawn with a new image; so the model indicates to the view that the flower has changed by sending the `changed:with:` message as defined in section 5. As the view depends on the model, the view receives the `update:with:` message but the view does not manage directly the display, it merely indicates to the system that the region of the screen corresponding to the position of the flower is now invalid and that the view should now be updated. To do this, the system will send to the view the `displayOn:` method with an appropriate graphic context that deals with the clipping so that only the invalid parts of the screen will be changed. This reduces the flicker on the screen that would occur if all the window had to be redrawn.

The next method processes update messages received after the model has indicated that it changed. As `#trace` method are dealt with by the text view, we do not have to do anything else. But in the other cases (`#body` or `#nobody`), we have to indicate that the part of the display where the `body` is currently appearing is now invalid and should be redrawn (repaired in Smalltalk terminology). When the body only changes its appearance, we should redraw immediately but in the case of a movement, it is a bit more subtle: the current position is first invalidated but it should be redrawn immediately because this would make the body reappear at the same position. Once the body is at the new position, then that new position is invalidated and the redrawing made immediatly. To have the bitmap appear centered on their position, we shift their position by half their extent. This centering is also used in the `clickAnyBody` message to identify if a body is clicked or not.

```
update: anAspect with:body
    | ext |
    anAspect ~= #trace
        ifTrue:[ext:=body  image  extent.
                self invalidateRectangle:((body position-(ext/2)) extent:ext)
                     repairNow:anAspect=#body]
```

When a view changes size, is buried or moved, it calls `displayOn:` on all its subviews to refresh the screen. In this case, we redraw all bodies centered on their position. But the clipping done by the system will make it so that only the invalid parts will be effectively redrawn.

```
displayOn: gc
    model bodies do:
        [:b | gc  display: b image at: b position - (b image extent/2)]
```

A final method is

```
defaultControllerClass
    ^SettingController
```

used during initialisation to create a controller of the appropriate class, defined in the next section, which drives the view.

8 The `SettingController` class

We want the application to be mouse and menu driven. We choose the following use of the mouse buttons[3]:

`blue` for managing the window itself (bury, frame, collapse, move and close). It is the normal use in Smalltalk.

`yellow` for creating new bodies or for asking actions from an existing body if the mouse was clicked over a body that is already on the screen. If the mouse was not clicked over an existing body, an appropriate menu appears from which we choose to create either a `Butterfly` or a `Flower`. We are then asked to indicate using a prompting window where it should appear in the current setting. If the mouse was clicked over an `AnimatedBody`, then an appropriate menu appears for choosing a message to send to this body.

`red` is not used in this application.

As this class is mouse driven, it is defined as a subclass of `ControllerWithMenu` that defines appropriate methods to poll the mouse buttons; when one is pressed, a pop-up menu appears and the message corresponding to the selection is sent. Of course, this can be overridden by a subclass (and we do this with the yellow button) but this default behavior is often the one needed. Let us see now how each button activity for the drawing is programmed, we are not concerned with what happens in the trace window as it is a standard `TextView`.

8.1 Yellow button

The `ControllerWithMenu` class has a method `yellowButtonActivity` that is called if the yellow button is pressed in the view. We can redefine it in our application to the following:

```
yellowButtonActivity
    "if click is on an AnimatedBody then pop-up an appropriate action menu.
    Otherwise pop-up a creation menu."
    | selectedMess |
    selectedBody:= self clickAnyBody.
    selectedBody notNil
        ifTrue:[(selectedBody isMemberOf:Butterfly)
                    ifTrue:[selectedMess := ButterflyMenu startUp]
                    ifFalse:[selectedMess := FlowerMenu startUp]]
        ifFalse:[selectedMess:=CreationMenu startUp].
    selectedMess  ~= 0 ifTrue:[self perform:  selectedMess]
```

This defines the behaviour of the yellow button: the `selectedBody` is first determined and an appropriate menu appears and the choice is returned; this is done by the `startUp` message. `CreationMenu`,`CreationMessages`, `FlowerMenu`, `FlowerMessages`, `ButterflyMenu`, `ButterflyMessages` are class variables which are set when the class is initialized with:

```
initialize
    "SettingController initialize"
```

[3] on a system with a three button mouse, the blue, yellow and red are usually mapped onto the right, middle and left buttons.

```
    | animatedBodyMenuLabels animatedBodyMessages |
    animatedBodyMenuLabels := 'move\move gradually\'.
    animatedBodyMessages := #(#moveTo #moveToBy).
    ButterflyMessages := animatedBodyMessages,#(#turn #visit #inPlace).
    ButterflyMenu := Menu labels:(animatedBodyMenuLabels,'turn\visit\in place')withCRs
                         lines:(Array with:animatedBodyMessages size)
                         values: ButterflyMessages.
    FlowerMessages := animatedBodyMessages,#(#open #close).
    FlowerMenu := Menu labels:(animatedBodyMenuLabels,'open\close')withCRs
                      lines:(Array with:animatedBodyMessages size)
                      values: FlowerMessages.
    CreationMessages := #(#newButterfly #newFlower).
    CreationMenu := Menu labels:'new butterfly\new flower'withCRs
                        values: CreationMessages.
```

The text to display in the menu is defined in a string where each choice is separated by a carriage return. As writing a string with embedded carriage-returns may lead to bad program layout, it is possible to separate the choices with backslashes (\) and use the unary message `withCRs` that changes the backslashes to carriage-returns.

For finding if a click has been made over a body, we define the following method:

```
clickAnyBody
    | point |
    point := sensor waitClickButton.
    ^model bodies detect:
        [:b | b image containsPoint: point - (b position-(b image extent/2))]
        ifNone:[nil]
```

It loops over all bodies and checks that the `point` indicated with the mouse click is within the bounds of the bitmap displayed for the body. As the bodies are displayed shifted to appear centered on their position (see the `displayOn:` method), the same translation is used to determine is the mouse was clicked of the image of a body. The method returns the first body for which the block returns `True` and returns `Nil` if no body was found.

8.2 Menu Methods of `SettingController`

Now we define the message that we promised in our list of messages. The creation messages are:

```
newButterfly
    | name location |
    name:=self promptForString: 'Name of the butterfly ?'
                initialAnswer:'B',Butterfly number printString.
    name isNil ifFalse:[
        location:= self promptForPoint:'Where ?'.
        location isNil ifFalse:[model butterflyNamed:name  at:location]]

newFlower
    | name location |
    name:=self promptForString: 'Name of the flower ?'
```

```
                initialAnswer:'F',Flower number printString.
    name isNil ifFalse:[
        location:= self promptForPoint:'Where ?'.
        location isNil ifFalse:[model flowerNamed:name at:location]]
```

For each, we ask for the name and location of the body using methods described in the next section. The body is then created using the methods of section 5 and it is added to list of bodies of the view.

The movement messages are:

```
close
    model close: selectedBody

inPlace
    | nbTimes |
    nbTimes:=self promptForNumber:'How many times will ',selectedBody name,
                    '\fly at the same place ?'withCRs
                    initialAnswer:'10'.
    (nbTimes notNil & (nbTimes > 0)) ifTrue:
        [model ask: selectedBody inPlace:nbTimes]

moveTo
    | point |
    point := self promptForPoint:'Where does ',selectedBody name,' move ?'.
    point isNil ifFalse:[model move: selectedBody to:point].

moveToBy
    | point step  |
    point := self promptForPoint:'Where does ',selectedBody name,' move ?'.
    point notNil ifTrue:[
        step := self promptForNumber:'Length of each step?' initialAnswer:'10'.
        step ~=0 ifTrue:[
        model move: selectedBody to:point by:step]].
open
    model open:selectedBody

turn
    model turn: selectedBody
```

The methods are only a matter of prompting for missing information and then sending the appropriate message to the `selectedBody`. `visit:` is a bit more complicated because it first builds a menu of all flowers in the setting from which a selection is made. A warning dialog is displayed if there are no flowers yet.

```
visit
    |mb  fSel found |
    "Create a menu of flowers"
    found := false.
    mb:=MenuBuilder new.
```

```
    model bodies keysAndValuesDo:
            [:k :v| (v isMemberOf:Flower) and:
                    [found:=true.
                    mb addLabel: k  value: v]].
    "Was there at least one flower?"
    found ifTrue:
            [fSel:=mb startUp.
            (fSel ~= 0) ifTrue: [model ask:selectedBody visit:fSel]]
        ifFalse:[Dialog warn:'No flower yet to visit'].! !
```

8.3 Prompting methods

The messages in the previous section prompt the user with standard dialog boxes We can thus customize the user interface by reusing system classes such as `Dialog` but also the reading of an integer with `Integer readFrom:` and even the compiler itself to convert a string composed of two integers separated by `@` to an instance of `Point`.

```
promptForString:message initialAnswer:initAns
    ^Dialog request: message initialAnswer:initAns onCancel:[^nil].

promptForNumber:message initialAnswer:initAns
    ^Integer readFrom:
        (ReadStream on:
            (self promptForString:message initialAnswer:initAns))

promptForPoint:message

    | ans newPosition  |
    ans := Dialog   request: 'What is the new position'
                    initialAnswer: sensor cursorPoint printString.
    ans isNil ifTrue:[^nil].
    ((newPosition := Compiler evaluate: ans ) isKindOf: Point) ifFalse:[^nil].
    ^newPosition
```

We now have a setting in a view which can be started by simply executing:

```
SettingView open
```

which creates a new window in which yellow button clicks can be used to create and animate flowers and butterflies. Executing this statement more than one time creates independent settings.

9 Ideas for extending the system

It would be fun to add new kinds of bodies, for example: cows that eat flowers, crabs that only move in right angles, rabbits that jump, turtles that leave a path when they move, etc. We leave these as an exercise to the reader who would need to draw bitmaps and redefine the appropriate movement methods and controller menus and messages.

Another interesting extension would be to define groups of bodies that move in parallel using the quasi-parallelism available through the `Process` class. This might involve a bit more work to

get it working smoothly with a good user interface. It is relatively easy to have many bodies in parallel by using the `fork` message in the "script" and by inserting some code in the movement methods to stop temporarily the current object to give control to another one.

Another kind of extension that is really the "essence" of object oriented programming: subclassing would be to define a `StickyButterfly` so that when it `visit:`s a `Flower`, it follows it if the flower moves. Should the `StickyButterfly` decide to move elsewhere without `visit:`ing then the `Flower` can move without dragging the butterfly any more. The idea being that a `StickyButterfly` is now becomes dependent of a flower when it visits ones and so when the `Flower` moves, the `StickyButterfly` moves with it. "Stickiness" could also be defined for flowers.

10 Conclusion

We have shown how to develop an interactive application in Smalltalk. We first defined mouse-driven `AnimatedBodies` to move around in a window. This gave us the opportunity to see examples of instance and classes variables and methods, subclasses, dependency links and the model-view-controller approach to the user-interface. We finally gave many suggestions for extending the system.

References

[1] Goldberg A., Robson D., *Smalltalk: The Language and its Implementation*, Addison-Wesley, 1983.

[2] Goldberg A., *Smalltalk: The Interactive Programming Environment*, Addison-Wesley, 1984.

[3] Hopkins, T., Horan, B. *Smalltalk: an introduction to application development using VisualWorks*, Prentice-Hall, 1995.

[4] Kaehler T., Patterson J., *A Taste of Smalltalk*, Norton, 1986.

11 Acknowledgements

We thank Anne Bergeron who wrote the first version of this program after having been given only a few hours of introduction to Smalltalk. Her work was "spectacular" and used as a demo at the Université de Montréal. It prompted us to revise and extend the system. The bitmaps of the flowers and the butterflies were designed by her.

We thank Pierre Cointe whose idea it was to use video animals moving on the screen to illustrate object-oriented programming. We also had many fruitful discussions with him, Isabelle Borne, Jean-François Perrot and Jean Bézivin that gave us a better understanding of Smalltalk. We thank Jean Vignolle, Patrick Sallé and Henri Farreny of IRIT in Toulouse for providing a stimulating working environment where a Smalltalk-80 version of this program was first developped. The VisualWorks version was redesigned during a stay at the Ecole des Mines de Nantes in March 1998 where discussions with Frédéric Rivard were most fruitful. Frédéric was very patient with me for explaining the "new" way of using the Smalltalk environment.

Finally we acknowledge the contribution of Jean Vaucher who introduced Simula-67 to us in the early seventies. Since then, we have always considered that object oriented programming is the "right" way to go...

Page numbers of method names