1

A Digital Animal: A gluttonous Tomagoshi

In this chapter we propose you to develop a small digital animal that has its own life require care for you. People were really found of digital animals called tomagoshi. Therefore we will build one step by step. This project is a pretext for revisiting the basic actions to define a class, instance variables and methods.

1 A gluttonous Tomagoshi

We have to decide the behavior that our digital beats should have. Here is the list of the behavior we propose you to implement for our tomagoshi. The Figure **??** represents how the tomagoshi changes its state.

- It can eat and digest food. It can be hungry when its tummy is empty and satisfied when it eats enough food.
- It has its own life cycle with its own isNights and days. It goes to sleep at isNight and wake up the morning.
- It is gluttonous and selfish so falls asleep as soon as it eats enough.
- Its change color depending on its mood. We choose to have blue when it is satisfied possibly sleeping, black when sleeping but hungry, green when waking up, and red when it is hungry.

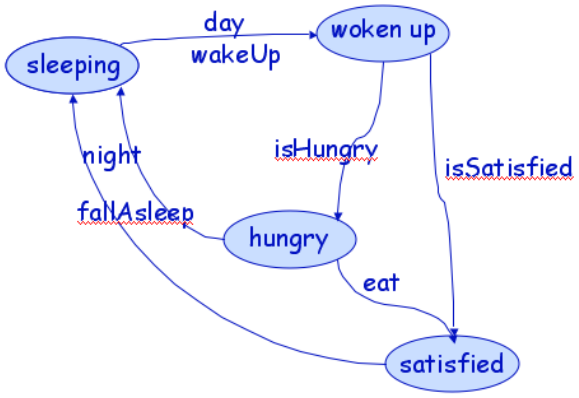

Figure 1.1:

2 Representing Day and isNight Passing

The first thing we want to represent is the time passing and the alternation of isNights and days. To represent the time passing, the length of isNights and days and the change between days and isNights we propose you to use two instance variables: `dayCount` and `isNight` defined in the class `Tomagoshi`.

To model isNight and day passing we have the idea that `dayCount` will hold a number represents the number of hours (or tick) left in the isNight or day. This number will decrease regurlarly and when arriving at zero the isNight or day will end and a new will start. We will use `isNight` as a boolean representing the fact that this is the isNight.

Define a new class category named for example 'TOMA' and define the class `Tomagoshi`. As we would like to be able to interact with it and have a graphical representation, we define the class `Tomagoshi` as a subclass of the class `Morph`.

Class 1.1

```
Morph subclass: #Tomagoshi
   instanceVariableNames: ' dayCount isNight'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'TOMA'
```

We edit the class comment by pressing the ? button and write something in the vein of:

```
I represent a tomagoshi. A small virtual animal that have its own life.

dayCount <Number> represents the number of hour (or tick) in my day and night.
isNight <Boolean> represents the fact that this is the night.
```

As done previously in chapter @@, we define the method `initializeToStandAlone` to invoke the ones of the superclasses, this way we will be able to create a graphical object. We also initialize the instance variables `dayCount` and `isNight` by calling a newly created method called `dayStart`. We choose the number 10 as the length of a day or isNight.

Method 1.1

```
In category initialize
Tomagoshi>>initializeToStandAlone
   "Initialize the internal state of a newly created tomagoshi"

   super initializeToStandAlone.
   self dayStart.
```

Method 1.2

```
In category initialize
Tomagoshi>>dayStart

   isNight := false.
   dayCount := 10
```

Now we need a way to regularly makes the time passing by decreasing the `dayCount` instance variables. The method `step` that is offered by the class `Morph`, is called by the system at regular time interval (specified in milliseconds by the method `steppingTime`). Therefore we specialize this method (see 'mthreftomastepone) so that it invokes a newly created method called `timePass` (see method **??**).

Method 1.3

```
In category stepping
Tomagoshi>>step

   self timePass
```

Method 1.4

```
In category stepping
Tomagoshi>>steppingTime

   ^ 500
```

The method `timePass` makes a beep, then decrease the value of the instance variable `dayCount`, check whether the time period is over, if this is the case it reinitialize it and update the instance variable `isNight` to represent the new period. This last action is done by the method `isNightOrDayEnd` (see method **??**).

Method 1.5

```
In category day and isNight
Tomagoshi>>timePass
   "Manage the isNight and day alternance"

   Smalltalk beep.
   dayCount := dayCount -1.
   dayCount isZero
      ifTrue:
          [ self isNightOrDayEnd.
          dayCount := 10]
```

Method 1.6

```
In category day and isNight
Tomagoshi>>isNightOrDayEnd
  "alternate isNight and day"

   isNight := isNight not
```

3 Adding Behavior

Now we should represent the state of our animal, if it is sleeping, hungry, and how it will changes its emotions. Our digital animal is mainly occupied to satisfied its hunger. We add two instance variables to represent the appetite of our small beast. The instance variable `tummy` represents the amount of food that the tomagoshi ate. The instance variable `hunger` represents its appetite. It will only be satisfied when it tummy is higher than its hunger.

Class 1.2

```
Morph subclass: #Tomagoshi
   instanceVariableNames: ' dayCount isNight tummy hunger'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'TOMA'
```

We update the class comments with the following information.

```
tummy <Number> represents the number of times you feed me by clicking on me.
hunger <Number> represents my appetite power.
I will be hungry if you do not feed me enough, but I'm selfish so as soon as I' satisfie
```

We change the `initializeToStandAlone` method to initialize the tummy to zero to state that it did not ate, and we choose to set its hunger randomly between 1 and 3. (the method `Integer»atRandom` returns a random number between the 1 and the receiver).

Method 1.7

```
In category initialize
Tomagoshi>>initializeToStandAlone
    "Initialize the internal state of a newly created tomagoshi"

   super initializeToStandAlone.
   tummy := 0.
   hunger := 3 atRandom.
   self dayStart.
```

Then we define the method `eat` as simply incrementing the number of item eaten. The method `digest` as decrementing this number by one. However, we consider digesting a slower process than eating so we one decrement the contents of the `tummy` instance variable when the clock represented by `dayCount` value is even (here we use `isDivisibleBy: 2` because this way you can model different animal easily, the method `even` would have worked) (see method **??**).

Method 1.8

```
In category behavior
Tomagoshi>>eat
   tummy := tummy + 1
```

Method 1.9

```
In category behavior
Tomagoshi>>digest
   "Digest slowly: every two cycles, remove one from the tummy"

   (dayCount isDivisibleBy: 2)
      ifTrue: [ tummy := tummy -1]
```

As the digestion is a process that is linked with time, we call the method `digest` from the method `timePass` as shown in method **??**.

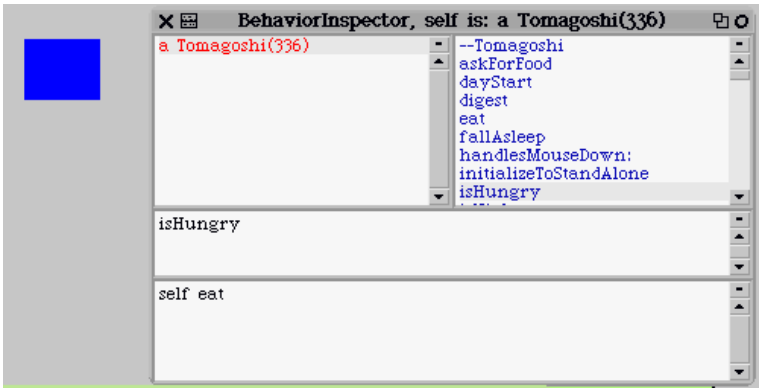

Figure 1.2: The tomagoshi inspected using the behavioral inspector. The right pane shows all the methods the object selected in the right pan understand. The bottom pane allows one to define methods or execute expression. The middle pane show the message definition and its comment.

Method 1.10

```
In category day and isNight
Tomagoshi>>timePass
   "Manage the isNight and day alternance"

   Smalltalk beep.
   dayCount := dayCount -1.
   dayCount isZero
      ifTrue:
         [self isNightOrDayEnd.
         dayCount := 10].
   self digest
```

Now we define two methods `isHungry` and `isSatisfied` that returns whether the tomagoshi is hungry or not.

Method 1.11

```
In category behavior
Tomagoshi>>isHungry
   ^ hunger > tummy
```

Method 1.12

```
In category behavior
Tomagoshi>>isSatisfied
   ^self isHungry not
```

Now we need to be able to wake up our tomagoshi, so we define the method `wakeUp` (see method **??**). This method changes its color and its state. For the state representation, as we do not wanted to duplicate the way the state is represented we define two methods `wakeUpState` (see method **??**) which returns the internal representation associated to the state waking up. The method method **??** returns whether the tomagaoshi is sleeping and again we do not code directly the fact that waking up state is code as the symbol `#sleep` but use the method `wakeUpState` that defines this representation. Hence we will be able to

change the representation by just changing one method and not three.

Method 1.13

```
In category behavior
Tomagoshi>>wakeUp
   self color: Color green.
   state := self wakeUpState
```

Method 1.14

```
In category behavior
Tomagoshi>>wakeUpState
  "Return how we codify the fact that I sleep"
   ^ #sleep
```

Method 1.15

```
In category behavior
Tomagoshi>>isSleeping
   ^ state = self wakeUpState
```

Now we have nearly all the behavior defined, so we are ready to complete the method `step` as presented in method **??**.

Method 1.16

```
In category stepping
Tomagoshi>>step
  "This method is called by the system at regurlar time interval.
   It defines the tomagoshi behavior."

   self timePass.
   self isHungry
      ifTrue: [self color: Color red].
   self isSatisfied
      ifTrue:
         [self color: Color blue.
         self fallAsleep].
   self isNight
      ifTrue:
         [self color: Color black.
         self fallAsleep]
```

To work we still have to define the method `isNight` as shown in method **??**.Note that we could have use the instance variable `isNight` directly but this would have been breaking the rythm or abstraction level of the method body as we do not use direct access to instance variable at all in this method.

Method 1.17

```
In category day and isNight
Tomagoshi>>isNight
   ^ isNight
```

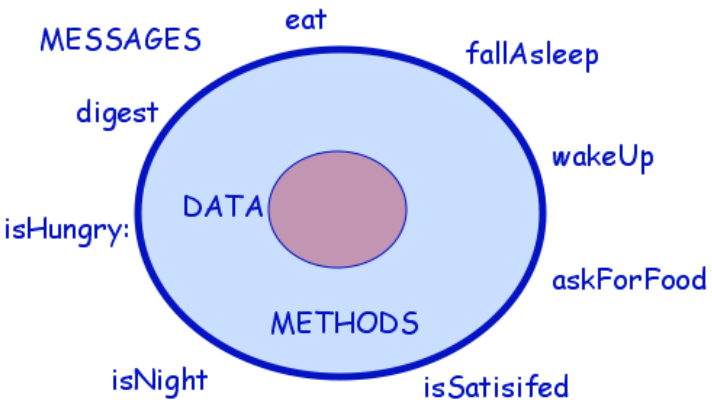

Figure 1.3: A tomagoshi. It proposes to the rest of the world a list of messages representing its behavior, *i.e.*, what he can do. From outside the tomagoshi its internal representation is not visible

Finally we just have to adapt slightly the `initializeToStandAlone` method so that a newly created tomagoshi is woken up.

Method 1.18

```
In category initialize
Tomagoshi>>initializeToStandAlone
    "Initialize the internal state of a newly created tomagoshi"

   super initializeToStandAlone.
   tummy := 0.
   hunger := 2 atRandom + 1.
   self dayStart.
   self wakeUp
```

Now we would like to be able to feed the tomagoshi just by clicking on it so we just define the two following methods (see method **??** and method **??**. These methods are fully explained in the chapter @@.

Method 1.19

```
In category stepping
Tomagoshi>>handlesMouseDown: evt
   "true means that the morph can react when the mouse down over it"
   ^ true
```

Method 1.20

```
In category stepping
Tomagoshi>>mouseDown: evt
    self eat
```

4 Further Experiments

Now we propose you different modifications to change the behavior of your tomagoshi.

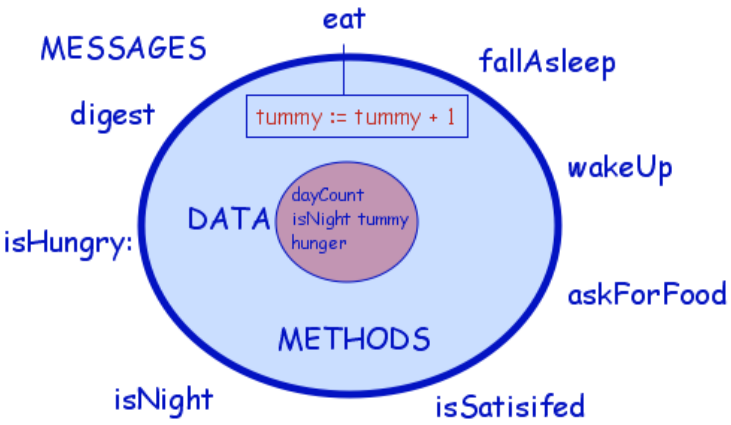

Figure 1.4: An object presents an interface composed by a set of messages defining *what* the object can do. This behavior is realized by methods that specify *how* the behavior is implemented. To realize the behavior, data is most of the time required. The data is only accessed by the methods.

- Make that we can only feed the tomagoshi the day.
- Make it die when it is starving from too long time (hints you can use the method `delete` or `stopStepping`).
- Make it happy after lunch and not only falling asleep. You could make it sings.
- Change its graphical representation.
- Using the method `PopUpMenu inform: aString` that pop up a menu, make the tomagoshi ask for food.
- Define the method `printOn: aStream` so that we see the state of the tomagoshi in inspector or when sending a `printString` message to a tomagoshi. The new displayed string could be `'a Tomagoshi (98987) with state: asleep'` instead of what we have now: `'a Tomagoshi (98987)'`

Improving the code. `dayCount := 10` is duplicated in several places. If we want to change the length of the day we will have to change this constant in lot of places. Change this situation so that we only have this logic in one place.

5 What the Example Shows

We hope that implementing a tomagoshi was certainly a fun experiment. However this is more than that. Now we will look at all the lessons we can learn from it.

Messages vs. Methods: What vs. How. The tomagoshi implementation illustrates the difference between messages and methods. A message represents *what* an object is capable to do. It is composed by a list of actions. In the figure**??**, the messages are all the behavior of the tomagoshi such as `eat`, `digest`, `wakeUp`... Methods define *how* the behavior described by the message is actually specified. The Figure **??** shows that the message `eat` is realized by the method `eat`. As we decided to represent the tummy of a tomagoshi using just one number representing the number of items the tomagoshi ate, the *method* `eat` increments the `tummy` instance variables.

Encapsulation. The example illustrates that the data hold in instance variable is private to the object. Only its methods can access them. In fact as someone interacting with the tomagoshi we do not have to know how it is internally defined. We are only interested by the messages that we can send to it. We say that the data is encapsulated. What encapsulation provides is that we can continue to interact with the tomagoshi using the same messages even if its internal representation changes. We can rely on its messages. For example, we used two instance variables one for the tummy and one for the hunger of the tomagoshi. Other implementation strategies can still provides the same interface (set of messages) while changing the number and roles of instance variables.

Factoring Logic. At the code level the example shows how the internal representation of certain state can be factored and only expressed in one place. Hence the method `wakeUpState` which returns a symbol (see method **??**) is the only one that defines how the state is represented. This means that we can change it freely. In our case this is not important but we could. This avoid to duplicate this knowledge all over the places. This follow the rule "Say once and only once".