Learning Object-Oriented Programming, Design and TDD with Pharo

Stéphane Ducasse

March 11, 2019

Layout and typography based on the sbabook LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

A Digital Animal: A gluttonous Tamagotchi

In this chapter we propose you to develop a small digital animal that has its own life and requires care from you. People were really fond of digital animals called tamagotchi. Therefore we will build one step by step. This project is a pretext for revisiting the basic actions to define a class, instance variables, messages and methods. We will also discuss encapsulation.

1.1 A gluttonous Tamagotchi

We have to decide the behavior that our digital monster should have. Here is the list of the behavior we propose you to implement for our tamagoshi.

- 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 nights and days. It goes to sleep at night 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 some colors to describe its state:

- blue when it is satisfied possibly sleeping,
- black when sleeping but hungry,
- green when wakened up,
- and red when it is hungry.

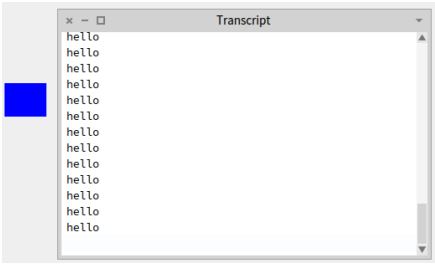

Figure 1-1 Our tamagotchi says hello in the Transcript.

1.2 Define Tamagotchi class

Define a new package named for example 'Tamagotchi' and define the class `Tamagoshi`. As we would like to be able to interact with it and have a graphical representation, we define the class `Tamagotchi` as a subclass of the class `Morph`. A morph is a graphical entity in Pharo. It knows how to draw itself on the screen.

```
Morph subclass: #Tamagotchi
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Tamagoshi'
```

We edit the class comment and write something in the vein of:

```
I represent a tamagotchi: A small virtual animal that has its own
    life.
```

Executing the following snippet should get you a blue square as shown in 1-1. Of course this is not really exciting but this is a start.

```
| t |
t := Tamagotchi new.
t openInHand
```

Open a Transcript

We will use the Transcript little window as a way to get some information from the tamagoshi. You can open a transcript using the tools menu or programmatically as follows:

```
Transcript open
```

The method `step` of a morph is called by the system at regular interval. We can redefine the method `step` of `Morph` as follows and we will obtain the situation depicted by Figure 1-1.

Figure 1-2 You can bring the halos around your tamagotchi by clicking on it while pressing option + command/control + shift

```
Tamagoshi >> step
  "Print some information on the output"
  self logCr: 'hello'
```

We should also define the interval between two steps, by defining the method `stepTime`.

```
Tamagotchi >> stepTime
  "Return the time interval between two outputs"
  ^ 500
```

1.3 Interacting with our live friend

Before continuing any further, we want to show you some ways to interact with the tamagotchi.

Halos

Pharo offers little graphical menus called halos. To bring them, click and press option, command/control and shift.

With the halos you can

- delete your tamagotchi. Pay attention you cannot bring it back.
- move it
- bring a menu and get an inspector as shown in Figure 1-3.

Note that it is easier to bring an inspector while creating the tamagotchi using the message `inspect` as follows:

```
| t |
t := Tamagotchi new.
t openInHand.
t inspect
```

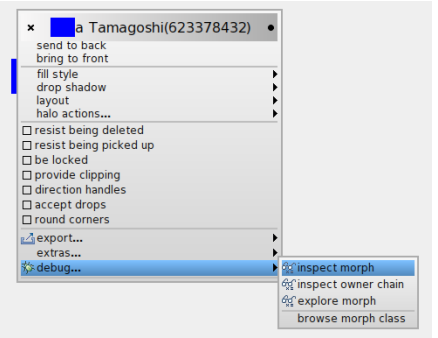

Figure 1-3 Bringing an inspector to interact with a tamgotchi.

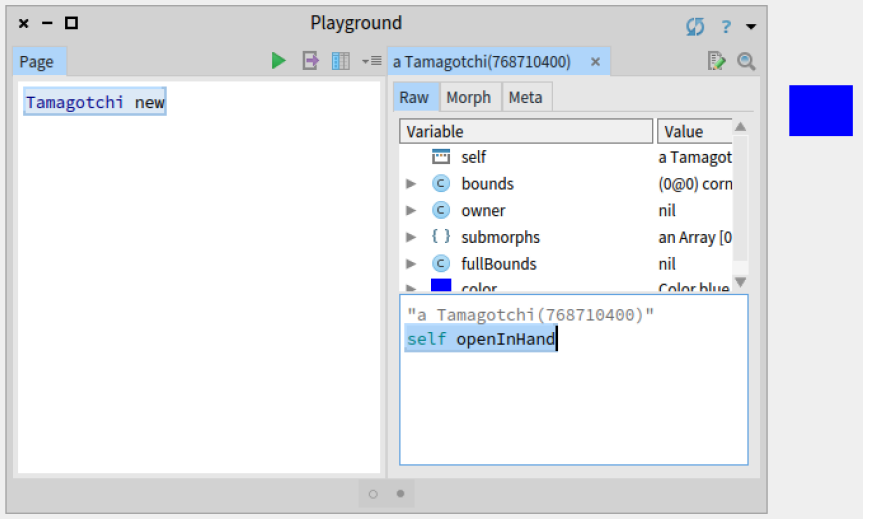

Figure 1-4 Using Playground to get an inspector on a tamagotchi and sending it the message `openInHand`.

Better use the playground

You can also use the menu item do it and go of the playground to obtain an integrated inspector. In Figure 1-4. We created a tamagotchi and we selected **do it and go** and we ask the tamagotchi to display itself.

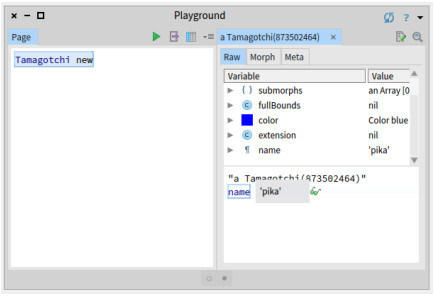

Figure 1-5 TamaWithName

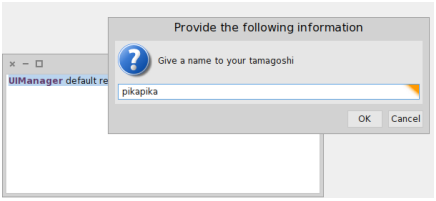

Figure 1-6 withPopUp

1.4 Adding a Name

```
Morph subclass: #Tamagoshi
  instanceVariableNames: 'name'
  classVariableNames: ''
  category: 'Tamagoshi'
```

Make sure that if we do not give a name it is named `'pika'`. When we create a new object by sending the `new` message, the message `initialize` is sent. So we redefine it to put a new value with the string `'pika'`. We use the expression `super initialize` to make sure that the Morph is well initialized. This concept will be explained later in the book.

```
Tamagotchi >> initialize
  super initialize.
  name := 'pika'
```

When we add a new instance variable to a class, all the existing instances get this instance variable but it will not be initialized correctly. So the best is to kill the tamagotchi and recreate one. You can kill a tamagotchi either using the halos or sending the message `delete` using the inspector.

```
UIManager default request:  'Give a name to your tamagoshi'
```

```
Tamagotchi >> nameMe
  name := UIManager default request: 'Give a name please!'
```

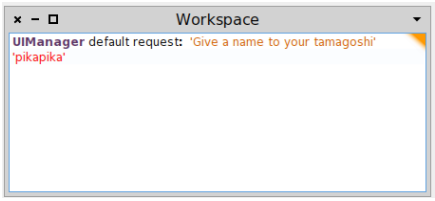

Figure 1-7 requestInTranscript

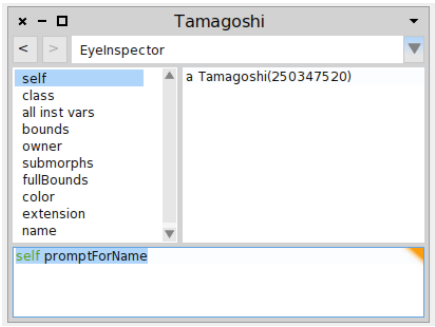

Figure 1-8 inspectorWithFifiPrompt

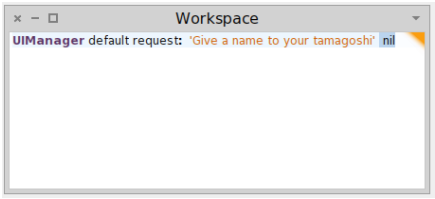

Figure 1-9 promptWithNil

```
Tamagotchi >> step
  self logCr: 'hello, my name is ', name
```

1.5 Making nameMe more robust

```
Tamagotchi >> nameMe

  | newName |
  newName := UIManager default request: 'Give a name please'.
  newName isNil
    ifTrue: [ ]
    ifFalse: [ name := newName ]
```

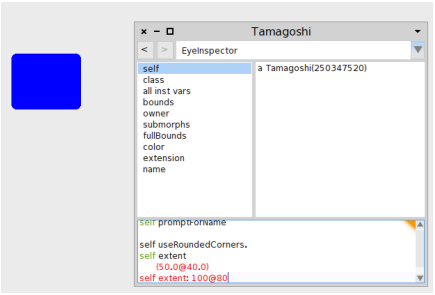

Figure 1-10 LargerRounder

```
Tamagotchi >> nameMe

  | newName |
  newName := UIManager default request: 'Give a name please'.
  newName isNotNil
    ifTrue: [ name := newName ]
```

1.6 Freeze

```
Tamagotchi >> freeze
  "Freeze the behavior of the receiver until it receives the message
    unfreeze."
  self stopStepping
```

```
Tamagotchi >> unfreeze
  "Get the receiver acting again."
  self startStepping
```

`drawOn:` controls the drawing of the Morph. We can for example add an oval.

```
drawOn: aCanvas

  super drawOn: aCanvas.
  aCanvas fillOval: self bounds
      fillStyle: self fillStyle
      borderWidth: 3
      borderColor: Color white.
```

```
eyesOn: aCanvas

  | b m y |
  b := self bounds.
  m := b width / 2.
  y := b width / 4.
  aCanvas
```

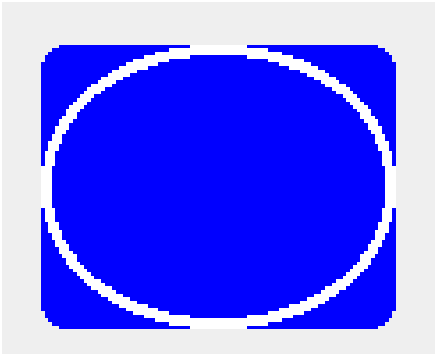

Figure 1-11 WithEllipse

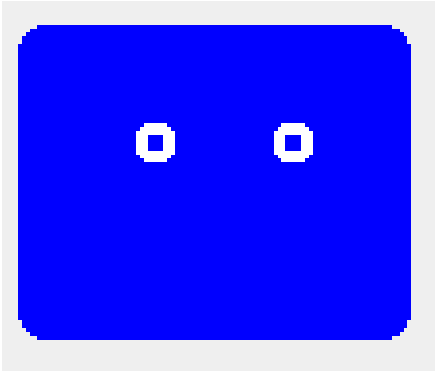

Figure 1-12 justEyes

```
      fillOval: ((b origin   + (m - 20 @ y)) extent: 10@10)
      fillStyle: self fillStyle
      borderWidth: 3
      borderColor: Color white.
  aCanvas
      fillOval: ((b origin   + (m + 15 @ y)) extent: 10@10)
      fillStyle: self fillStyle
      borderWidth: 3
      borderColor: Color white.
```

```
drawOn: aCanvas

  super drawOn: aCanvas.
  self eyesOn: aCanvas
```

Now we can define the mouth of our digital animal by drawing two lines like in the following and we obtain Figure~ref{fig:Smiling}.

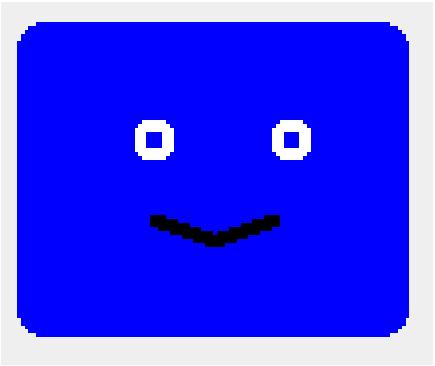

Figure 1-13 A smiling tamagoshi.

```
mouthOn: aCanvas

  | b m y middlePoint |
  b := self bounds.
  m := b width / 2.
  y := b width / 1.8.
  middlePoint := b origin +  ( m@y ).
  aCanvas
    line: b origin + ((m-15) @ (y -5)) to:  middlePoint width: 3
    color: Color black.
  aCanvas
    line: b origin + ((m+15) @ (y -5)) to: middlePoint width: 3
    color: Color black.
```

```
drawOn: aCanvas

  super drawOn: aCanvas.
  self eyesOn: aCanvas.
  self mouthOn: aCanvas.
```

1.7 Eating Behavior

Let us add a tummy to our tamagoshi.

```
Morph subclass: #Tamagoshi
  instanceVariableNames: 'name tummy'
  classVariableNames: ''
  category: 'Tamagoshi'
```

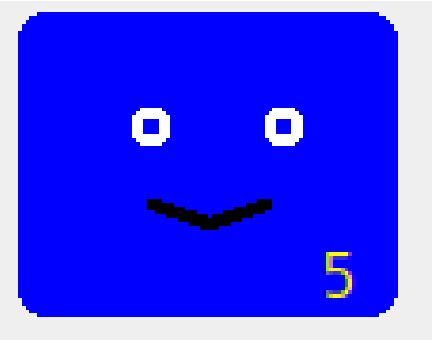

Figure 1-14 withbBelly

```
initialize

  super initialize.
  self useRoundedCorners.
  self extent: 100@80.
  name := 'pika'.
  tummy := 5.
```

Now we can also change the printing behavior to show its tummy.

```
step

  self logCr: 'hello, my name is ', name, ' and my tummy is: ',
     tummy printString.
```

```
tummyOn: aCanvas
  aCanvas
    drawString: tummy printString
    at: self bounds corner - (20 @ 20)
    font: nil
    color: Color yellow
```

do not forget to change the drawOn: method to invoke the tummyOn: method.

```
drawOn: aCanvas

  super drawOn: aCanvas.
  self eyesOn: aCanvas.
  self mouthOn: aCanvas.
  self tummyOn: aCanvas.
```

Now we would like to define the method eat so that we can feed our tamagoshi.

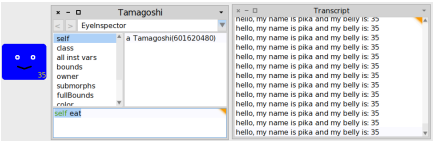

Figure 1-15 Now we show the food items in the tamagoshi' tummy

```
eat

  tummy := tummy + 1
```

```
eat

  tummy := tummy + 1.
  self changed.
```

Now we will tell the morph that it should react to mouse events.

```
handlesMouseDown: evt
  "Returning true means that the morph can react to mouse click"

  ^ true
```

And we will make sure that when we click on it, it gets fed.

```
mouseDown: evt

  self eat.
```

Now it may be wise to limit the stomach capability of our digital friend. We revise then the eat method to only increase it if we did not reach a limit.

```
eat

  tummy < 55
    ifTrue: [ tummy := tummy + 1
          self changed].
```

1.8 Digesting

Extract the code of the step method into a new method name talk.

```
talk

  self logCr: 'hello, my name is ', name, ' and my tummy is: ',
    tummy printString.
```

```
step

  self talk.
```

Now we are ready to make our tamagoshi digest its food.

```
digest

  tummy := tummy - 1.
  self changed.
```

Try with the inspector to send some digest messages to your tamagoshi.

Now this is annoying to add the changed message everywhere so we introduce a method named:

```
tummy: aNumber

  tummy := aNumber.
  self changed.
```

and we use it in eat and digest

```
digest

  self tummy: tummy - 1.
```

```
eat

  tummy < 55
    ifTrue: [ self tummy: tummy + 1 ].
```

```
step

  self talk.
  self digest.
```

You will quickly see that digest is wrong because we will get negative numbers really fast. So first we change the definition of digest to be

```
digest
  tummy = 0
    ifFalse: [ self tummy: tummy - 1]
```

Now since it is digesting too fast we should think how we could make our tamagoshi digest slower. What we need is a counter of the tick passed and based on that digest or not. So we add the hours instance variable to the class.

```
Morph subclass: #Tamagoshi
  instanceVariableNames: 'name tummy hours'
  classVariableNames: ''
  category: 'Tamagoshi'
```

```
initialize

  super initialize.
  self useRoundedCorners.
```

```
  self extent: 100@80.
  name := 'pika'.
  tummy := 5.
  hours := 0.
```

```
nextHour

  hours := hours + 1
```

```
step

  self talk.
  self nextHour.
  self digest
```

Now we can make our tamagoshi digest slower by simply making sure that several hours passed before it digests. Just checking for example if the number of hours is divisible b a given number will do the trick.

```
step

  self talk.
  self nextHour.
  (hours isDivisibleBy: 3)
    ifTrue: [self digest ]
```

1.9 Hungry

Now we get ready to make our tamagoshi changing face when he is hungry.

```
isHungry

  ^ tummy isZero
```

We can now rewrite digest using isHungry as follow:

```
digest
  self isHungry
    ifFalse: [ self tummy: tummy - 1]
```

```
hungryMouthOn: aCanvas

  | b m y middlePoint |
  b := self bounds.
  m := b width / 2.
  y := b width / 1.8.
  middlePoint := b origin +  ( m@y ).
  aCanvas
    line: b origin + ((m-15) @ (y + 5)) to:  middlePoint width: 3
    color: Color black.
  aCanvas
```

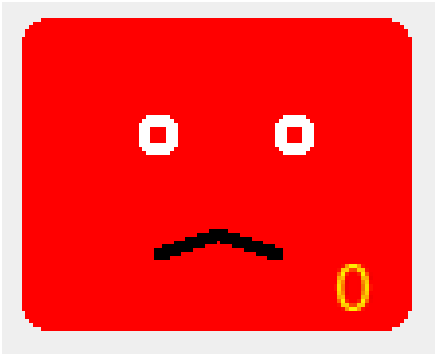

Figure 1-16 Our tamagoshi shows now that he is hungry

```
    line: b origin + ((m+15) @ (y + 5)) to: middlePoint width: 3
    color: Color black.
```

Now we can change a bit the messages send by our tamagoshi.

```
talk

  self isHungry
    ifTrue: [ self logCr: 'Please feed me' ]
    ifFalse: [ self logCr: 'hello, my name is ', name, ' and I''m
    ok'].
```

1.10 Nights and days

Now we want to manage the night and day period of our tamagoshi.

```
Morph subclass: #Tamagoshi
  instanceVariableNames: 'name tummy hours isNight'
  classVariableNames: ''
  category: 'Tamagoshi'
```

```
initialize

  super initialize.
  self useRoundedCorners.
  self extent: 100@80.
  name := 'pika'.
  tummy := 5.
  hours := 0.
  isNight := false
```

```
isNight

  ^ isNight
```

```
switchDayPeriod
  "Switch from night to day and day to night"
  isNight := isNight not.
```

```
checkAndNextDayPeriod
  "Switch from night to day and day to night when necessary"

  (hours isDivisibleBy: 12)
    ifTrue: [ self switchDayPeriod ]
```

We can change the messages to show that our animal is sleeping

```
talk
  self isNight
    ifTrue: [ self logCr: 'ZZZzzzzzzz' ]
    ifFalse: [
      self isHungry
        ifTrue: [ self logCr: 'Please feed me' ]
        ifFalse: [ self logCr: 'hello, my name is ' , name , ' and
    I''m ok' ] ]
```

As we will see later, these conditionals all over the place are not that nice, each time we add a new behavior we have to change talk and the drawOn: method logic becomes more and more complex.

```
sleepyEyesOn: aCanvas

  | b m y |
  b := self bounds.
  m := b width / 2.
  y := b width / 4.
  aCanvas
      fillOval: ((b origin   + (m - 20 @ y)) extent: 11@5)
      fillStyle: self fillStyle
      borderWidth: 3
      borderColor: Color white.
  aCanvas
      fillOval: ((b origin   + (m + 15 @ y)) extent: 11@5)
      fillStyle: self fillStyle
      borderWidth: 3
      borderColor: Color white.
```

```
drawOn: aCanvas
  super drawOn: aCanvas.
  self isNight
    ifTrue: [
        self sleepyEyesOn: aCanvas.
        self color: Color darkGray ]
    ifFalse: [
      self eyesOn: aCanvas.
      self isHungry
        ifTrue: [
```

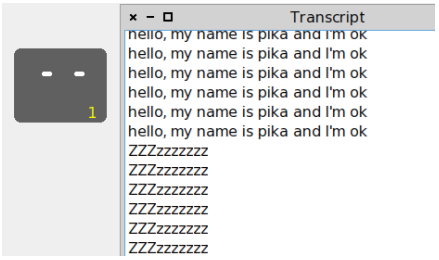

Figure 1-17 sleeping

```
        self color: Color red.
        self hungryMouthOn: aCanvas ]
      ifFalse: [
        self color: Color blue.
        self mouthOn: aCanvas ] ].
self bellyOn: aCanvas.
self changed.
```

1.11 Further Experiments

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

- 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, you could make it sings.
- Change its graphical representation.

Bibliography