Learning Object-Oriented Programming, Design and TDD with Pharo

Stéphane Ducasse

March 11, 2019

Copyright 2017 by Stéphane Ducasse.

The contents of this book are protected under the Creative Commons Attribution-ShareAlike 3.0 Unported license.

You are **free**:

- to **Share**: to copy, distribute and transmit the work,
- to **Remix**: to adapt the work,

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page:
http://creativecommons.org/licenses/by-sa/3.0/

Any of the above conditions can be waived if you get permission from the copyright holder. Nothing in this license impairs or restricts the author's moral rights.

Your fair dealing and other rights are in no way affected by the above. This is a human-readable summary of the Legal Code (the full license):
http://creativecommons.org/licenses/by-sa/3.0/legalcode

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

Contents

Illustrations

CHAPTER 1

Beacons and Satellites

In this chapter you will build a simulator for beacons and satellites. Beacons are in the sea and collect data and from time to time they should synchronise with satelittes to send data.

In reality, satelittes broadcast signals and beacons are polling at regular interval for signals, then once they know that they are in range based on the first signal, a communication is established and data is exchanged.

In the simulator we will present how we can implement communication between losely coupled objects. You will build step by step different variations around the observer/observable idiom. This idiom is important since it is used in Model-View-Controller and Self addressed stamped enveloppe (S.A.S.E) patterns. Beacons will register to satellites and when the satelittes are in range they will notify the beacons interested in the notification.

1.1 Description

A beacon is inside the sea and it collects data. It is fully autonomous. After a certain period of time it migrates to the surface waiting to send the data it collected. To communicate with satelittes, a satelitte should be available, i.e., within the zone where the beacon is.

A satellite is moving around earth at a certain speed and ranging a portion of sea. It can only communicate with beacons within such range.

The system is fully dynamic in the sense that new beacons may be added or removed. Satelittes may be present or not.

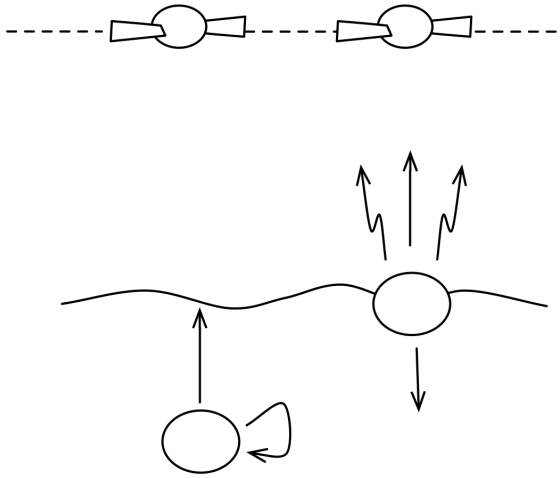

Figure 1-1 Beacons and Satelittes.

1.2 A simple model

```
Object subclass: #Satelitte
  instanceVariableNames: 'observers'
  classVariableNames: ''
  package: 'SatelitteAndBeacon'
```

```
Satelitte >> initialize
  observers := OrderedCollection new
```

```
Object subclass: #Beacon
  instanceVariableNames: 'data'
  classVariableNames: ''
  package: 'SatelitteAndBeacon'
```

1.3 V1: Simple observer / observable

We start with a simple schema where beacons

- register to satellites and
- when the satelittes are in range they notify the beacons that registered.

Registration

A beacon register to a satellite as follows:

```
Satelitte >> register: aBeacon
  self addObserver: aBeacon
```

Notification

```
Satelitte >> position: aPoint
  position := aPoint.
  self notify
```

```
Satelitte >> notify
  observers do: [ :aBeacon | aBeacon salelittePositionChanged: self ]
```

1.4 V1 Implementation

???

1.5 V2: Analysis

This first implementation has several drawbacks.

- One of the problem is that the message is hardcoded.
- Second Imagine that the satellite should emit different notification for its position, protocol to be used, frequency.... and each kind of beacon can register for the notification kinds that fits it. We must have a list of each kind of observed property.

1.6 V2: Introducing events

```
Satelitte >> register: aBeacon forEvent: aEventClass
  aSatelitte1 addObserver: aBeacon1 with: aEventClass
```

```
Satelitte >> addObserver: anObserver with: anEventClass
  observerDict at: anEventClass iAbsentPut: [OrderedCollection new].
  (observerDict at: anEventClass) add: anObserver
```

```
Satelitte >> position: aPoint
  position := aPoint.
  self notify: (PositionChanged with: self)
```

```
Satelitte >> notify: anEvent
  (observersDict at: anEvent class) ifPresent: [ :aBeaconList |
    aBeaconList do: [:aBeacon| anEvent fireOn: aBeacon ]
```

Implementation

```
Object subclass: #SBEvent
  instanceVariableNames: 'observable'
  classVariableNames: ''
  package: 'SatelitteAndBeacon'
```

```
SBEvent subclass: #SBPositionChanged
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'SatelitteAndBeacon'
```

```
SBEvent subclass: #SBProtocolChanged
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'SatelitteAndBeacon'
```

```
SBPositionChanged >> fireOn: anObserver
  anObserver salelittePositionChanged: observable
```

```
SBProtocolChanged >> fireOn: anObserver
  anObserver salelitteProtocolChanged: observable
```

V2 analysis

Advantages

- we reuse the same mechanism for different kind of observable properties.

Drawbacks

- One event means that the message is also hardcoded. There is tight dependencies between the event type and the kind of behavior that is available on the observer side.

1.7 V3 Specifying the message

Now the observer can specify the message that it wants to receive.

```
aSatelitte1 when: SBPositionChanged send: #readyForHandShakeWith:
    to: aBeacon1
```

```
aSatelitte1 when: SBProtocolChanged send: #useProtocol: to: aBeacon1
```

```
Satelitte >> when: anEventClass send: aSelector to: anObserver
  observerDict at: anEventClass iAbsentPut: [OrderedCollection new].
  (observerDict at: anEventClass) add: (aSelector -> anObserver)
```

```
Satelitte >> position: aPoint
  position := aPoint.
  self notify: (PositionChanged with: self)
```

```
Satelitte >> notify: anEvent
  (observersDict at: anEvent class) ifPresent: [ :aBeaconList |
    aBeaconList do: [ :aBeaconAssoc |
      aBeaconAssoc value perform: aBeaconAssoc key with: anEvent) ]
```

1.8 V5 Factoring out the announcer

The notification and management at notification should be packaged as a separate class so that we can reuse it by just delegating to it.

```
Object subclass: #BSAnnouncement
  instanceVariableNames: 'selector observer'
```

```
Object subclass: #BSAnnouncer
  instanceVariableNames: 'observerDict'
```

```
BSAnnouncer >> when: anEventClass send: aSelector to: anObserver
  observerDict at: anEventClass iAbsentPut: [ OrderedCollection new]
    .
  (observerDict at: anEventClass) add:
    (BSAnnouncement send: aSelector to: anObserver)
```

```
BSAnnouncer >> notify: anEvent
  (observersDict at: anEvent class) ifPresent: [ :aBeaconList |
    aBeaconList do: [ :anAnnouncement |
      anAnnouncement observer
        perform: anAnnouncement selector
        with: anEvent) ]
```

```
Satelitte >> notify: anEvent
  self announcer notify: anEvent
```

```
Satelitte >> when: anEventClass send: aSelector to: anObserver
  self announcer when: anEventClass send: aSelector to: anObserver
```

1.9 Discussion about lookup of events

Bibliography