Notes on the "Service-oriented design" OOP domain modelling example

This notebook contains notes from studying the Service Oriented Design example in the Scala Book, part of the official Scala 3 documentation.

The example is in turn based on the paper by Martin Odersky and Matthias Zenger, "Scalable Component Abstractions".

> Our goal is to define a software component with a *family of types* that can be refined later in implementations of the component. Concretely, the following code defines the component `SubjectObserver` as a trait with two abstract type members, `S` (for subjects) and `O` (for observers)

The `SubjectObserver` trait

In [1]:
```
trait SubjectObserver:
  type S <: Subject
  type O <: Observer

  trait Subject:
    self: S =>
      private var observers: List[O] = List()
      def subscribe(obs: O): Unit =
        observers = obs :: observers
      def publish() =
        for obs <- observers do obs.notify(this)

  trait Observer:
    def notify(sub: S): Unit
```

Out[1]:
```
defined trait SubjectObserver
```

Top-level definitions

In the `SubjectObserver` trait's top level, four members are defined:

Two abstract types:

- `S` (implying subject)
- `O` (implying observer)

Two other traits:

- `Subject`
- `Observer`

Type definitions

The syntax `type S <: Subject`, called an "**upper bound**", is a declaration for an abstract type `S` which must be a subtype of `Subject`.

Meaning that:

> All traits and classes extending `SubjectObserver` are free to choose any type for `S` as long as the chosen type is a subtype of `Subject`.

Similarly, a type `O` is also defined, which must be a subtype of `Observer`.

Trait definitions

Each minor trait has the following members:

```
trait Subject:
  private var observers: List[O] = List()
  def subscribe(obs: O): Unit = // ...
```

```
  def publish() = // ...
trait Observer:
  def notify(sub: S): Unit
```

All of the members of the `Subject` trait are concrete, while the single member of the `Observer` trait is abstract.

Subject

The `Subject` trait defines three concrete members:

- A private, mutable field `observers` of type `List[O]`, which is initialized with `List()`
- Two public, concrete methods whose behavior depend on this field:
  - `subscribe(obs: O): Unit`
  - `publish(): Unit`

These have concrete bodies defined as such:

```
def subscribe(obs: O): Unit = observers = obs :: observers
def publish() = for obs <- observers do obs.notify(this)
```

The `subscribe` method appears to simply add a given `O` observer to the list of subscribers set in the `observers` field.

The `publish` method in turn seems to enable a `Subject` to go over each of its subscribers, set in the `observers` field, and call their `notify` method with the subject itself as the argument.

Something that stands out is that, unlike for `Observer`, the `Subject` trait has `self: S =>` before its member definitions. This is also explained in the documentation:

> This is called a *self-type annotation*. It requires subtypes of `Subject` to also be subtypes of `S`. This is necessary to be able to call `obs.notify` with `this` as an argument, since it requires a value of type `S`. If `S` was a *concrete* type, the self-type annotation could be replaced by `trait Subject extends S`.

Observer

The `Observer` trait defines a single, abstract method named `notify`, which takes an argument of type `S` named `sub`.

My understanding at this point is that it's meant to be called by a subject when it needs to notify the subscribed observer of something.

A `SubjectObserver` implementation

In [2]:
```
object SensorReader extends SubjectObserver:
  type S = Sensor
  type O = Display

  class Sensor(val label: String) extends Subject:
    private var currentValue = 0.0
    def value = currentValue
    def changeValue(v: Double) =
      currentValue = v
      publish()

  class Display extends Observer:
    def notify(sub: Sensor) =
      println(s"${sub.label} has value ${sub.value}")
```

Out[2]:
```
defined object SensorReader
```

Here, a `SensorObject` singleton is defined extending the previously defined `SubjectObserver` trait.

To satisfy its contract, it defines:

```
type S = Sensor
type O = Display
```

- A type `S` — referring to the trait's "subject" concept — assigned the class `Sensor`, this class defined in the body of the object
- A type `O` — referring to the trait's "observer" concept — which is assigned to the class `Display`, also defined in the body of the object

It still must implement the minor traits `Subject` and `Observer` to fulfill the contract, and it does that in defining the previously assigned types:

```
class Sensor(val label: String) extends Subject:
  // ....

class Display extends Observer:
  def notify(sub: Sensor) =
    // ...
```

- The class `Sensor` takes a string `label` value for its constructor and extends `Subject`
- The class `Display` takes no arguments for its constructor and extends `Observer`

The `Sensor` class does not need to implement any methods since the `Subject` trait has all concrete methods.

However, it *does* have three members that provide a way to access and modify a `currentValue` private var, initialized to the double 0.0:

```
class Sensor(val label: String) extends Subject:
    private var currentValue = 0.0
    def value = currentValue
    def changeValue(v: Double) =
      currentValue = v
      publish()
```

This encapsulated logic ensures that whenever the `currentValue` is modified, `publish` is also called, therefore notifying all subscribed observers.

The `Display` class must yet implement a `notify(sub: S)` method that handles a notification event.

It does so by printing to STDOUT the interpolated string `"${sub.label} has value ${sub.value}"`:

```
class Display extends Observer:
    def notify(sub: Sensor) =
      println(s"${sub.label} has value ${sub.value}")
```

The interpolated string relies on members of the `Sensor` class defined in the implementation alone, rather than inherited from the trait `Subject` trait.

The documentation emphasizes how this design demonstrates an object-oriented paradigm:

> Besides, being an example of a service oriented design, this code also highlights many aspects of object-oriented programming:
>
> - The class `Sensor` introduces its own private state (`currentValue`) and encapsulates modification of the state behind the method `changeValue` .
> - The implementation of `changeValue` uses the method `publish` defined in the extended trait.
> - The class `Display` extends the trait `Observer` , and implements the missing method `notify` .

It also makes the following observation:

> It is important to point out that the implementation of `notify` can only safely access the label and value of `sub` , since we originally declared the parameter to be of type `S` .

It is alluding to the contract established in the nested `Observer` trait:

```
trait SubjectObserver:
  type S <: Subject
  // ...

  trait Subject:
    self: S =>
    // ...

trait Observer:
    def notify(sub: S): Unit
```

Considering that `S` had to be a subtype of `Subject` , as per its `type S <: Subject` definition, and given the self-type annotation `self: S =>` in the body of the `Subject` trait established that the subtypes of `Subject` (in this case, `Sensor`) have to also be subtypes of `S` , then:

```
Sensor <: Subject && Sensor <: S
```

Because of this double constraint, `Sensor` can pass itself as a subtype of `Subject` instead of as an instance of `Subject`

when `publish(this)` is called.

The implication of safety being, possibly, that through subtyping, the instance of `Sensor` can be used where an extended implementation of `Subject` is expected, but insofar as it is also the type assigned to `S`.

Invoking the `SensorReader` logic

In the example below, the output is two times an identical message printed for the same change in the `currentValue` of `sensor1`, since it has two different observers subscribed (`d1` and `d2`), and a single message printed for the change to the value of `s2` from its one observer `d1`.

In [5]:
```
import SensorReader.*

// setting up a network
val s1 = Sensor("sensor1")
val s2 = Sensor("sensor2")
val d1 = SensorReader.Display()
val d2 = SensorReader.Display()
s1.subscribe(d1)
s1.subscribe(d2)
s2.subscribe(d1)

// propagating updates through the network
s1.changeValue(2)
s2.changeValue(3)
```

```
sensor1 has value 2.0
sensor1 has value 2.0
sensor2 has value 3.0
```

Out[5]:
```
import SensorReader.*

// setting up a network

s1: Sensor = ammonite.$sess.cell2$Helper$SensorReader$Sensor@2a8e138f
s2: Sensor = ammonite.$sess.cell2$Helper$SensorReader$Sensor@10fc52e9
d1: Display = ammonite.$sess.cell2$Helper$SensorReader$Display@38234ace
d2: Display = ammonite.$sess.cell2$Helper$SensorReader$Display@182f632a
```