Signals and Gates

Steven Costiou
Stéphane Ducasse
Oleksandr Zaitsev
Sebastian Jordan Montano
Inria

September 24, 2024

This exercise is adapted from the lectures and exercises of Walter Rudametkin.

In this exercise, you will model a logical circuit composed of logical gates. You will model a logical signal and propagate this signal through the logical circuit. During this exercise, you must write tests and you must avoid coding with conditionals (= no `"if"`).

1 Logical signals

You will design and implement a way to represent logical signals (for example, HighSignal and LowSignal) and their logical aspects. To talk about signal classes in a generic way, we refer in the text to a class LogicalSignal. The implementation should not use conditionals.

1.1 Protocols

Here is the protocol of the signals:

- value → returns the logical value of the signal (1 if high and 0 if low).
- not → returns the inverse signal.
  Example: HighSignal new not → returns an instance of LowSignal.
- and: aLogicalSignal → implements the logical operator "*and*".
  Example : new HighSignal new and: LowSignal new → returns an instance of LowSignal.
- or: aLogicalSignal → implements the logical operator "*or*".
  Example : HighSignal new or: LowSignal new → returns an instance of LowHigh.

1.2 Definition and Tests

1. Define tests for the protocol.
2. Define the corresponding classes: your tests should pass as you go.

2 A typical assembly

We consider circuits built out of logical gates *and*, *or* ou *not* connected to signal sources and which output to devices. Figure 2 shows an example of such circuit, with three sources, three logical gates and a device. The device is powered when it receives a high logical signal, and is off when it receive a low logical signal.

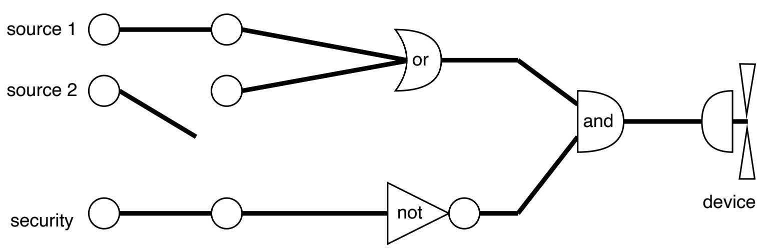

Figure 1: An example of a logical circuit.

The following scripts show how the circuit from Figure 2 can be created programmatically:

Initializing and connecting components.

```
source1 := Source new.
source2 := Source new.
sourceSecurity := Source new.
or := Or new.
not := Not new.
and := And new.
fan := Fan new.

or firstInput: source1.
or secondInput: source2.
not input: sourceSecurity.
and input: or.
and input: not.
fan input: and
```

Powering the components.

```
source1 on.
source2 off.
sourceSecurity on.
```

Inspecting the components.

```
fan state printString
fan inspect
```

In the quest of the Circuit class.

While we do not have yet a Circuit we can mimic using a simple collection holding the components. In exercise below, you will define a Circuit which will encapsulate this behavior in a better fashion.

```
elements := {
        source1 .
        source2 .
        sourceSecurity .
        or .
        not .
        and .
        fan
}.

String streamContents: [:s |
   elements do: [ :each | each printOn: s ]]
```

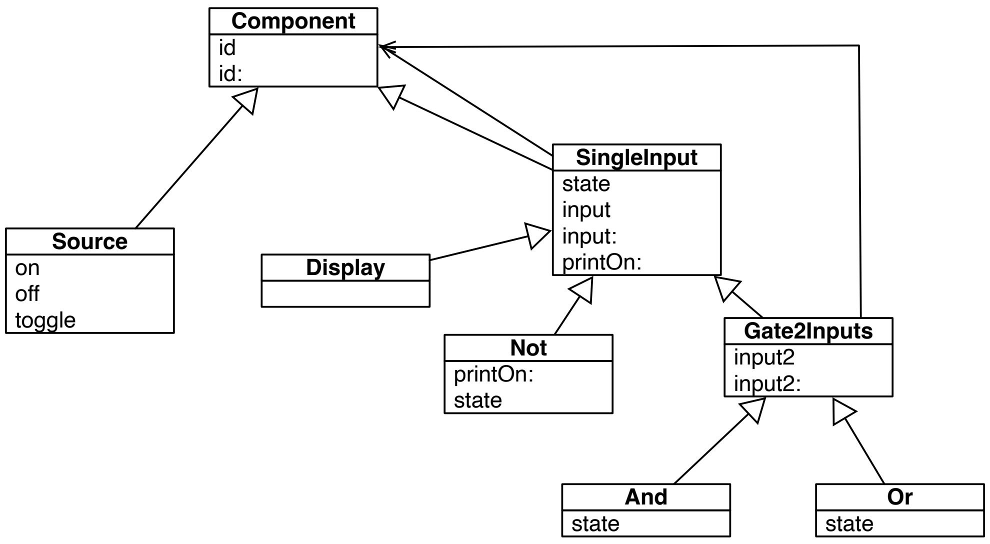

Figure 2: Component object model.

3 Component description

We will model our component following the class hierarchy depicted in Figure 2:

- Component is the root class of the hierarchy. It is abstract and provides methods id and id:. If the id has not been set using the method id:, the method id will set a default value by using the method hash of the receiver.
- A Source has one output and a state (on or off). It represents a power source or some binary input from the real world and it can be on or off like a switch.
- SingleInput is the abstract class root of the logical gates. It has as state and a single input.
- Display (device in Figure 2) is a concrete single input at the end of a circuit.
- Not is a single input that applies the not logical operation to the signal coming from its input.
- Gate2Input is abstract and factorizes logical gates with 2 inputs (And, Or, ...) connected in input on 2 components.

In the end, we are interested in knowing the state of display components, such as the device in Figure 2. To do so, we ask the device its state. The device then asks its input its states, which in turn asks its potential inputs their state, and so on. When arrived at a source, the source's state returns a logical signal that propagates back trough the logical circuit to the device.

4 Source implementation

The logical state of a component is provided by its method state which returns a signal. A Source can be changed using its respective methods on and off. The state of the other components can be calculated according to the state of the components connected to their inputs. Write a test for each of the following cases:

1. the methods on and off of the Source class.

2. make sure that a source is on at creation time.
3. implement the method state that returns the state of the source.
4. the method toggle that changes off to on and the inverse.

Here is a possible test:

```
testSourceIsOnByDefault

  | sourceSecurity  |
  sourceSecurity := Source new.
  self assert: sourceSecurity state value equals: 1.
```

4.1 SingleInput, Gate, And, Not, and Or implementation

Gate2Input is abstract and factorizes the logical gates with 2 inputs (And, Or, ...) connected in input on 2 components (input2 and input2:). Define tests for all the behavior.

1. Define classes Gate2Input, And, Or,
2. the corresponding state methods in each of the classes with tests,
3. a method printOn: that returns the following information:
    - The name of their class
    - their identifier (given by id)
    - components with input(s), the identifiers (via the method id) method) of the corresponding components or the character string 'not connected' if the input is not connected.

   For example, for an And (And (48d6c16c))) not connected at input 1, connected to input 2 on a not (Not (5abb7465)) : And (48d6c16c) in1: not connected in2: Not (5abb7465)

Here is a typical test:

```
testNotOnOn

  | sourceSecurity  not |
  sourceSecurity := Source new.
  not := Not new.
  not input: sourceSecurity.
  sourceSecurity on.
  self assert: not state value equals: 0.
```

5 Display implementation

A Display is connected with one input on a component:

- Define a couple of tests,
- implement the class Display and its state method if needed,
- implement a specific printOn: method.

6 More tests

- If you have not already done so, instantiate and link together the components necessary to model the circuit described in Figure 2,
- write tests that check the final state of the circuit connection (1 or 0) according to the state of the three sources at the beginning of the circuit.

7 Modeling a Circuit

So far, circuits have no existence of their own and are handled as individual components. We will model circuits as objects defined by the class Circuit.

The class Circuit.

The Circuit class has the following properties: it has an id, and it includes a collection of components that are connected to its single output.

To populate the components of a circuit, the circuit expects the component that will produce its final output. Based on that component, it will navigate up to the sources and set component connected to its sources as internal components.

```
c := Circuit new.
c fromOutput: fan.

String streamContents: [:s |
  c components do: [ :each | each printOn: s ]]
```

8 Bonus: Extensions

- Circuit as a component composite.
- Visitor that visits the Circuit.
- Signal logic operators create all the time new instances while they could just simply use a single instance of each class. Using class variables implement such behavior. You can verify that the behavior of the program and all the tests are unchanged.