Share, a metaprogramming pattern

Francesco Coppola, Stefano Perniola

Created on : March, 2020

Last updated : May, 2020

Project realized for the University of Camerino during the academic year 2019/2020

Table of contents

"It's easy to play any musical instrument: all you have to do is touch the right key at the right time and the instrument will play itself."[1]

Within the following pages it will be possible to find the documentation generated for the **Share** project.

The development of this code is to be attributed to the students **Francesco Coppola** and **Stefano Perniola**, while the part of abstract description and idealization of the pattern to the teacher **Rosario Culmone**.

[1] Johann Sebastian Bach, german composer and musician

Chapter 1

Introduction

There are millions of home automation devices in the world and billions in the future. Many of these interact with difficulty due to the **absence of a standard**[1]. In this project, we suggest a vision of how we can overcome the problems of interaction in pear-to-pear way.

Instead of looking for a **network standard**, we propose a programming standard through a design pattern. The proposed pattern design does not travel on data network but code. This feature allows not only to exceed the limits of data-oriented standards but **to make up for services on the fly**.

Verification mechanisms that guarantee correctness control the dynamic composition. Finally, an example is presented using the **Lua**[2] programming language.

It's worth noting that the efforts of both the ISO/IEC and the IETF and IRTF have some limitations from a practical perspective. They aren't standards as some IT pros might understand them. They are not detailed blueprints that engineers can design to.
Lua is a lightweight, high-level, multi-paradigm programming language designed primarily for embedded use in applications.

Chapter 2

Pattern

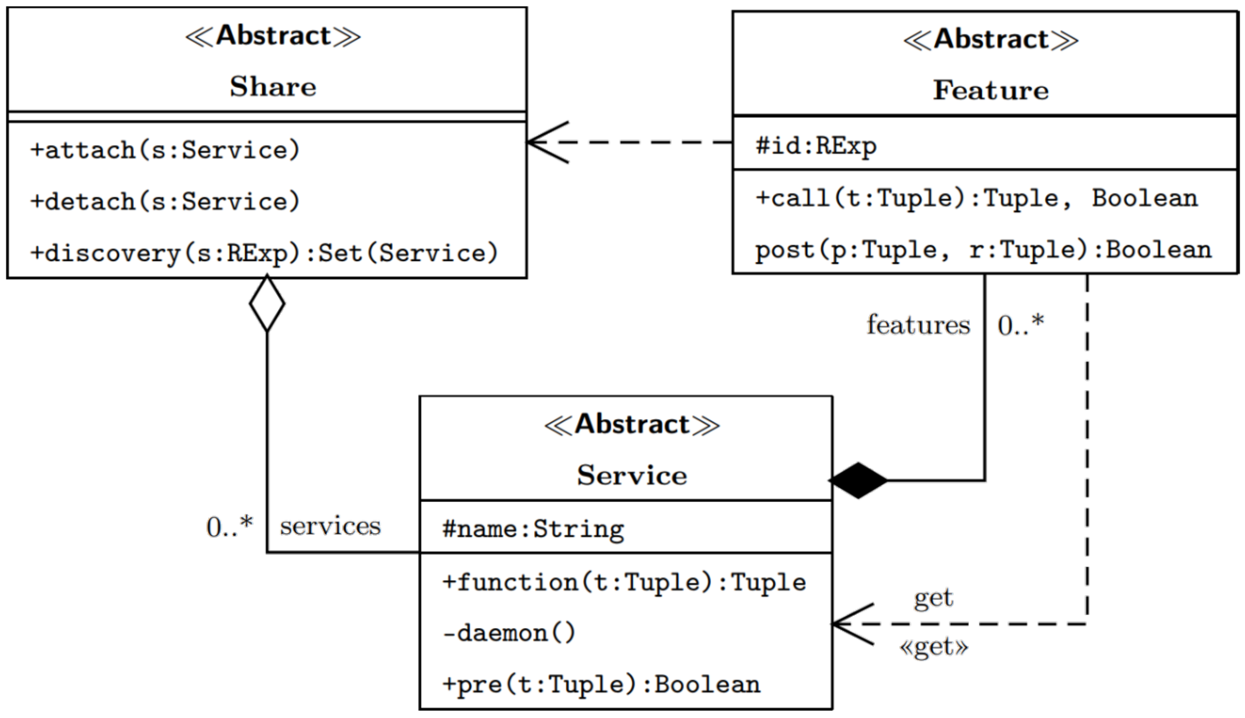
The design pattern **Share** is made up of three classes:

- Share
- Service
- Feature

Share manages the common space for sharing services and services with **Feature** the implementation of the service.

Each service **must** subscribe to at least one sharing service provider to allow others to use the service.

2.1 Description



The share pattern is a **metaprogramming pattern**[1] since some parts of the service are known only at the time of execution and depend on the state of the system

Metaprogramming is a programming technique in which computer programs have the ability to treat other programs as their data. It means that a program can be designed to read, generate, analyze or transform other programs, and even modify itself while running.

The definition of a service requires the coding of a function `function` and its `daemon` resident component, if any. A `pre` predicate specifies the preconditions for function.

A string attribute identifies service in a unique world. For example an **SNMP MIB** can be used to identify a service.

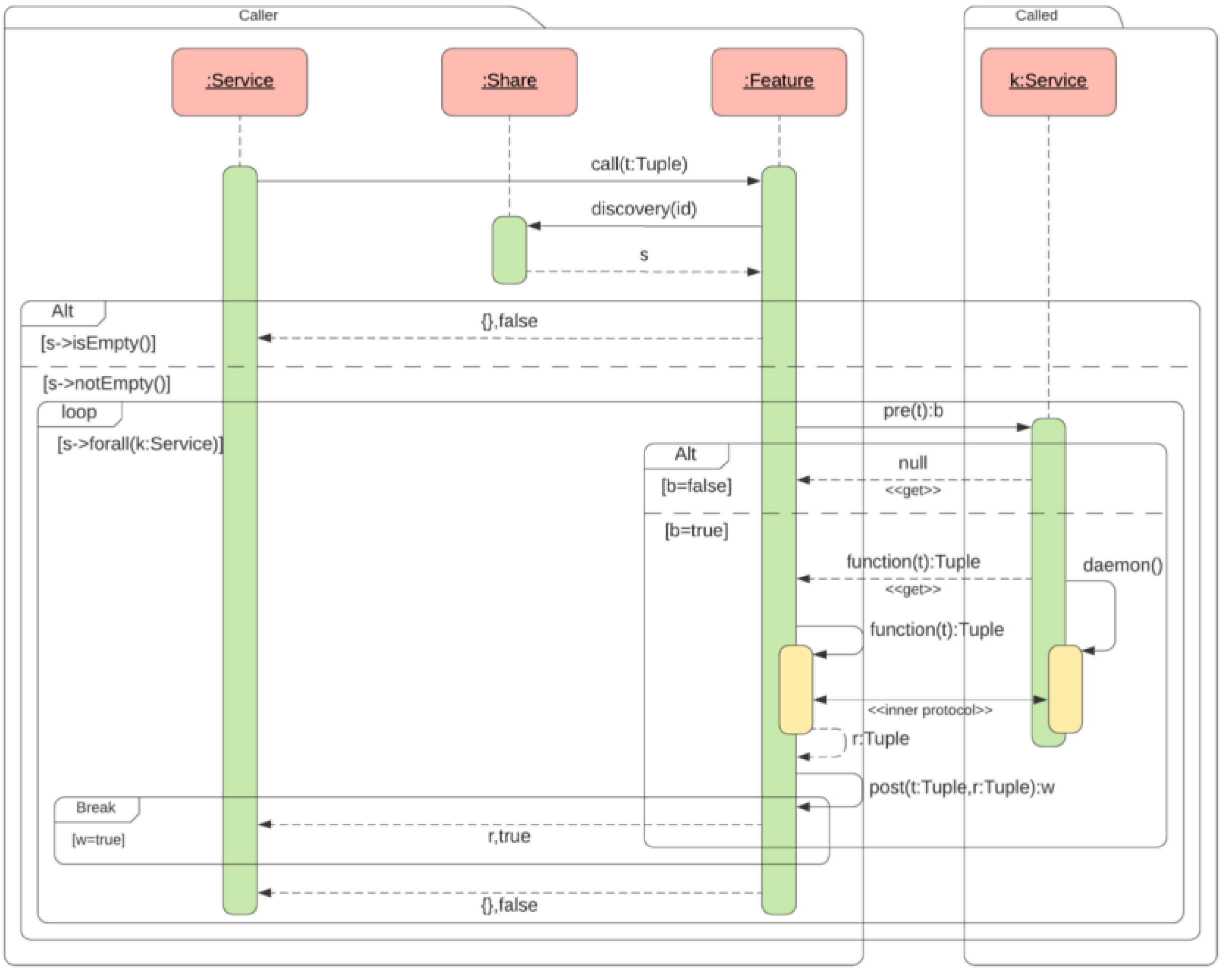
Note: A management information base **(MIB)** is a database used for managing the entities in a communication network. Most often associated with the Simple Network Management Protocol **(SNMP)**, the term is also used more generically in contexts such as in OSI/ISO Network management model.

In `function` coding there may be calls to external services that are specified by implementations of **Feature**. The `id` attribute defines a regular expression that describes semantically the service requested. The invocation of a service within `function` occurs through the call invocation relating to a specification present in features.

The `call` operation invokes discovery with an id attribute to identify everyone the services subscribed to **Share** to which the calling service is subscribed. If he comes produced a non-null set of services, the next phase of invocation of services.

The primitive `get` is used for this phase get code from a service

2.2 Sequence Diagram



The use of the pattern can be achieved by defining a **concrete** Share class and **two abstract** classes Service and Feature.

So while the Share class can be created directly, the Share pattern specifies the behavior the pattern requires basically the implementation of `function`, `daemon`, `pre` and `post` using unchanged the functionality of the remaining operations related to the structure of the pattern.

In the pattern structure, the subscription to the Share service is not places no constraints. So you can subscribe to multiple Share or only one. Another aspect is the interaction between daemon and function. The code associated with function is received by the get operator **immediately** after executed with daemon.

The interaction between `function` and `daemon` takes place through unforeseen protocols from the pattern. For example, the protocol can be used as **0MQ**. There correct interaction between function and daemon **is guaranteed by the fact that both they are developed by the same programmer**.

Note: ZeroMQ is a high-performance asynchronous messaging library, aimed at use in distributed or concurrent applications. It provides a **message queue**, but unlike message-oriented middleware, a **ZeroMQ system can run without a dedicated message broker**.

The attach operation is used to register with a `subscriber`. Any positive can be a subscriber however it is plausible that only a few provide this service. The search for a service is done through a descriptive string. Could be an ontological descriptor or simply a protocol **MIB** string SNMP. The match function was used in the specification and the pattern allows the nested call of services

Chapter 3

Documentation

One documentation to rule them all, one documentation to find them, One documentation to bring them all and in the darkness bind them.[1]

The entire generated documentation that is being used is the result of the union between **Sphinx** and `sphinx-lua`, two tools dedicated to the generation of texts starting from mere and pure code.

3.1 Tools with which it was made

Usually the part of the code documentation in Lua is somewhat **boring** and **stylistically questionable**.

From this premise comes the integration of Sphinx[2] within the project which has made the entire software park a *pleasant* and *clear* product in **understanding its API**.

3.2 Integration with Lua

For the documentation of the classes created, therefore, we relied on a tool that puts **Python** *(given the nature of Sphinx)* and **Lua** in symbiosis.

This tool is called `sphinx-lua`[3].

It can be easily installed using the following command:

```
pip install sphinx-lua
```

Once that you installed this on your machine your could simply start to document your class in this way:

```lua
--- Define a car.
--- @class MyOrg.Car
local cls = class()

--- @param foo number
function cls:test(foo)
end
```

Modified version of The One Ring, the central plot element in J. R. R. Tolkien's The Lord of the Rings (1954–55). It first appeared in the earlier story The Hobbit (1937) as a magic ring that grants the wearer invisibility

Sphinx is a documentation generator written and used by the Python community. It is written in Python, and also used in other environments.

GitHub page of the project sphinx-lua

Chapter 4

Implementation

The following page will show what has been **our implementation** of this pattern by providing the documentation of the APIs created.

4.1 Share

The **Share** class is the beating heart of the pattern of the same name, the *raison d'etre* of the same and spokesperson for the current of thought that characterized the project in its entirety: **elegance is everything**. The class has the list of services that a device proudly makes available to all, and offers features to add or remove others. But the main responsibility of this class is to perform the `discovery` function, a symbol of an endless adventure, an adventure that begins in the search for the services that best lend themselves to the arduous and meticulous work that the calling service requires.

The function uses the **MDNS protocol** to first calculate the ip address of each device on the network, then subsequently examine the table of services available from the same.

Note: In computer networking, the multicast DNS (mDNS) protocol resolves hostnames to IP addresses within small networks that do not include a local name server. It is a zero-configuration service, using essentially the same programming interfaces, packet formats and operating semantics as the unicast Domain Name System (DNS). Although Stuart Cheshire designed mDNS as a stand-alone protocol, it can work in concert with standard DNS servers.

The `discovery` function is called by `call`, a function that finds an origin but perhaps does not find an end, a very long path that only the best services can undertake to the end. And so it is that from these services the transfer of **knowledge** takes place between the calling device and the called device, a knowledge obtained in a transversal, unorthodox way, based on the code and not on the network. The caller does not get knowledge directly, if he has to conquer it by executing the code that was provided to him by the caller.

Share doesn't give you fish, but it can help you fish. Elegance is everything.

`class Share`

`new()`
The constructor of the object Share

Returns The new Share just created with the table of available services
Return type *Share*

`attach(`*s*`)`
This method inserts a service into the table of available services

Parameters `s` (`Service`) – The service to add

`detach(`*s*`)`
This method removes a service from the table of available services

Parameters `s` (`Service`) – The service to remove

`is_present(`*s*, *t*`)`
This method search a service from the services table and returns true if it finds an occurrence

Parameters
- `s` (`Service`) – The service to search
- `t` (*`table`*) – The table on which doing the search

Returns True if the service is present, false otherwise
Return type boolean

`discovery(`*macro_mib*`)`

Parameters `macro_mib` (*`any`*) –

`find(`*macro_mib*`)`
Internal function that retrieve the set of services with the corresponding prefix

Parameters `macro_mib` (*`str`*) – The prefix of the mib to search
Returns The set of corresponding services
Return type table

`open_udp_socket(`*ip*, *macro_mib*, *result*`)`
Internal function used to establish a remote connection with udp socket

Parameters
- `ip` (*`str`*) – The ip of the remote device
- `macro_mib` (*`str`*) – MIB of the service owned by the remote service
- `result` (*`table`*) – The table used to save all results

4.2 Service

The **Service** class represents any functionality made available by the device that owns it.

The responsibility of this class, in addition to providing the result of the computation with the parameters requested by the caller through the `daemon` function, is to establish a safe and reliable communication protocol with the same.

Each Service has a personal **MIB** which makes it **unique** in the environment in which it operates.

`class Service`

`new(`*i*, *f*, *d*, *p*, *...*`)`
The constructor of the object Service

Parameters
- `i` (*`str`*) – The MIB of the current Service
- `f` (*`str`*) – The function wrapped in a string that allows the communication with the inner protcol
- `d` (*`function`*) – The defined daemon that share data with the function of the same Service
- `p` (*`function`*) – The pre-condition necessary to checking
- `vararg` (*`any`*) – The set of features

Returns The new Service just created or nil in case of any issues
Return type *Service*

4.3 Feature

The **Feature** class represents the entity that contains the complete set of services that perform the same function.

In fact, this class has a **MIB** that identifies a macro category within its environment. Each Feature has a feature called `post` which checks the postconditions of each result received.

The responsibility of this class is also to identify the list of desired services that share the same network through the `call` function.

`class Feature`

`new`(*i*, *p*)

The constructor of the object Feature

Parameters
- `i` (*str*) – The MIB of the current Feature
- `p` (*function*) – The post-condition necessary to checking

Returns The new Feature just created or nil in case of any issues

Return type *Feature*

`call`(*...*)

A stub that searches, verifies, executes and produces the results related to a remote service

Parameters `vararg` (*any*) – The parameters that are called are a regular expression and the parameters on which to perform the operation

Returns Produces a boolean indicating whether the operation is successful and a table with the values produced by the requested service

Return type table or boolean

Chapter 5

Example

In the following section some examples will be shown that have the purpose of making the substance of the pattern **more concretely** understood and what some uses of this pattern may be.

5.1 Abstract

The concept of service management is of significant importance, as is the dynamism of the same, which plays an important role in the **reliability**, **stability** and **correctness** of the services provided.

In this context, **Share** allows you to change the semantics of a service according to the devices available at a given moment, thus leaving **complete freedom** on the low-level interaction model, preferring the modularity and composition of the services.

In order to correctly verify the constraints defined by the communicating devices, we have relied on an approach based on the design *by contract*, which defines `pre-conditions` and post-conditions on the values provided respectively by the caller and the called party. In this way you have a guarantee on the validity and consistency of the data obtained.

5.2 Calculation of the square root

For simplicity, assume that the service requested is the mere calculation of the square root on a set parameter. First of all, the device requesting the service must know the `MIB`, that is a regular expression that uniquely identifies a class of semantically equivalent functionality.

First step For example, if the category **"Mathematics"** had the number `1` as specific `MIB` and the square root had the number `2`, the mib on which to make the call would be `1.2.*`. In this way the device is able to search, and eventually **discover**, all the devices that provide services that at that moment calculate the square root.
Client side

```
-- 2 is the number on which we calculate the square root
services["1.2.1.0"].features[1]:call(2)
```

Server side

```
udp_discovery:sendto(Utilities:table_to_string(disc:find(data_discovery)), ip_discovery, port_
↪discovery)
```

Second step Once the table of available services has been obtained, the calling device interrogates these services **one by one** by sending them the parameter on which to perform the calculation *(in this case the pre-conditions must verify that this parameter is greater than zero).*
Client side

```
check_param(mib, ..., udp_feature)
```

Server side

```
if (services[mib].pre(param)) then
    udp_call:sendto(services[mib].func, ip_call, port_call)
end
```

Third step In this phase, a **unicast communication** is initiated between the devices (managed by the function and daemon functions) **which guarantees confidentiality between the parties**, and in which the calling device receives the function to be performed locally to obtain the desired result.

Note: In computer networking, **unicast** refers to a one-to-one transmission from one point in the network to another point; that is, **one sender and one receiver**, each identified by a network address.

Client side

```
.
.
tcp:connect(host, port);
tcp:send(data.."\n");
.
.
local s, status, partial = tcp:receive()
```

Server side

```
services[mib].daemon()
```

Fourth step Once the result is obtained, the calling device can decide whether to validate this result *(checking it in the post-conditions)* or whether to move on to the next service. As soon as one of these services is able to meet the established `post-conditions`, the **workflow** will end.
Client side

```
if (res and self.post(..., res)) then
    log.info("[POST-CONDITION SUCCESSFUL]")
    return res, true
end
```

5.3 Temperature measurement

The pattern also provides for the possibility of requesting services that **do not provide parameters** from the calling device. For example, we admit that a device needs to know the atmospheric room temperature to perform a certain task. This request does not include any parameter as the calculation of the temperature is a procedure that **does not involve any operation on the data**, but simply makes a measurement and query a possible thermometer.

First step The calling device will therefore simply need to invoke the `MIB` that identifies any service that has a thermometer, in this case for purely demonstrative purposes it is assumed to be `2.1.*`
Client side

```
-- the call function has no parameter as you can see
services["2.1.1.0"].features[1]:call()
```

Server side

```
udp_discovery:sendto(Utilities:table_to_string(disc:find(data_discovery)), ip_discovery, port_
→discovery)
```

Second step Once the table of available services is obtained, the calling device **interrogates** these services one by one (in this case the `pre-conditions` are always exceeded as no parameters are received).
Client side

```
check_param(mib, ..., udp_feature)
```

Server side

```
if (services[mib].pre(param)) then
    udp_call:sendto(services[mib].func, ip_call, port_call)
end
```

The third and fourth steps correspond exactly to the example shown above.

5.4 Nested call

A `nested call` is defined as any context in which the device called, **to fulfill the result of the local computation**, it needs to in turn make a call to another service.

5.4.1 In-depth explanation

Home automation devices, smart industry, smart city, smart energy or any other type of device they are installed over time and vary in number and type **without** a pre-arranged installation plan.

Providing services that include **their collaboration** is difficult because interconnection and service interaction protocols are different and could vary over time.

In this context, the adaptive and very flexible nature of the pattern allows the possibility **to perform nested calls between devices**, so as to further enhance the concept of interoperability between the themselves and take full advantage of the dynamism that distinguishes them.

5.4.2 Concrete demonstration

Let's consider a sporting event, where racing bikes compete. Suppose it is necessary calculate the speed of the bikes and compare them to each other to understand which is the **best** rider.

The devices available are:

- A **big screen**, which shows a general ranking to the fans based on the average speed of the drivers
- **Speed detectors** scattered across the track, which could be of different brands and calculate speeds with different measurement systems *(km/h, m/s, mph)*
- A **classification device**, which sorts the set of all speeds detected in descending order and ensures uniformity between data *(converts m/s to km/ h or vice versa)*

In this context, the maxi-screen, in order to update the data shown in the table, **makes a call to the classification device, which in turn makes a call to the speed detectors to first receive any updates on the drivers' speeds**.

All these logics can be developed on the individual devices that have competence on the actions they undertake according to the different configurations that are detected allowing `flexibility`, `adaptation` and `fault tolerance` to the system.

Chapter 6

Future works

In the near future it would certainly be interesting to implement features like these:

- Creation of a domain language for **IoT** systems
- Static analysis of the system performances
- Integration with data cloud

Index