

Share, a metaprogramming pattern



Francesco Coppola, Stefano Perniola

Created on : March, 2020

Last updated : May, 2020

Table of contents

Table of contents	i
1 Introduction	2
2 Pattern	3
2.1 Description	3
2.2 Sequence Diagram	4
3 Documentation	6
3.1 Tools with which it was made	6
3.2 Integration with Lua	6
4 Implementation	7
4.1 Share	7
4.2 Service	8
4.3 Feature	9

“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**.

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 peer-to-peer 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

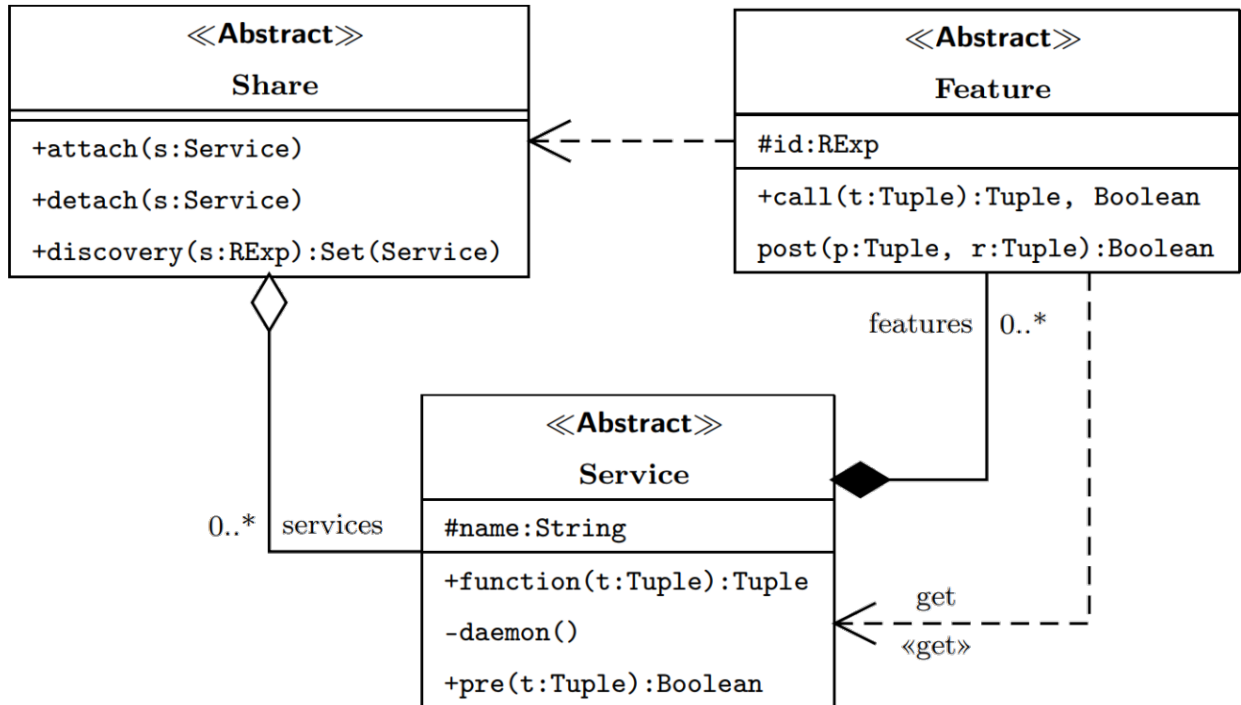
The design pattern **Share** is made up of three classes:

- 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

¹Metaprogramming 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.

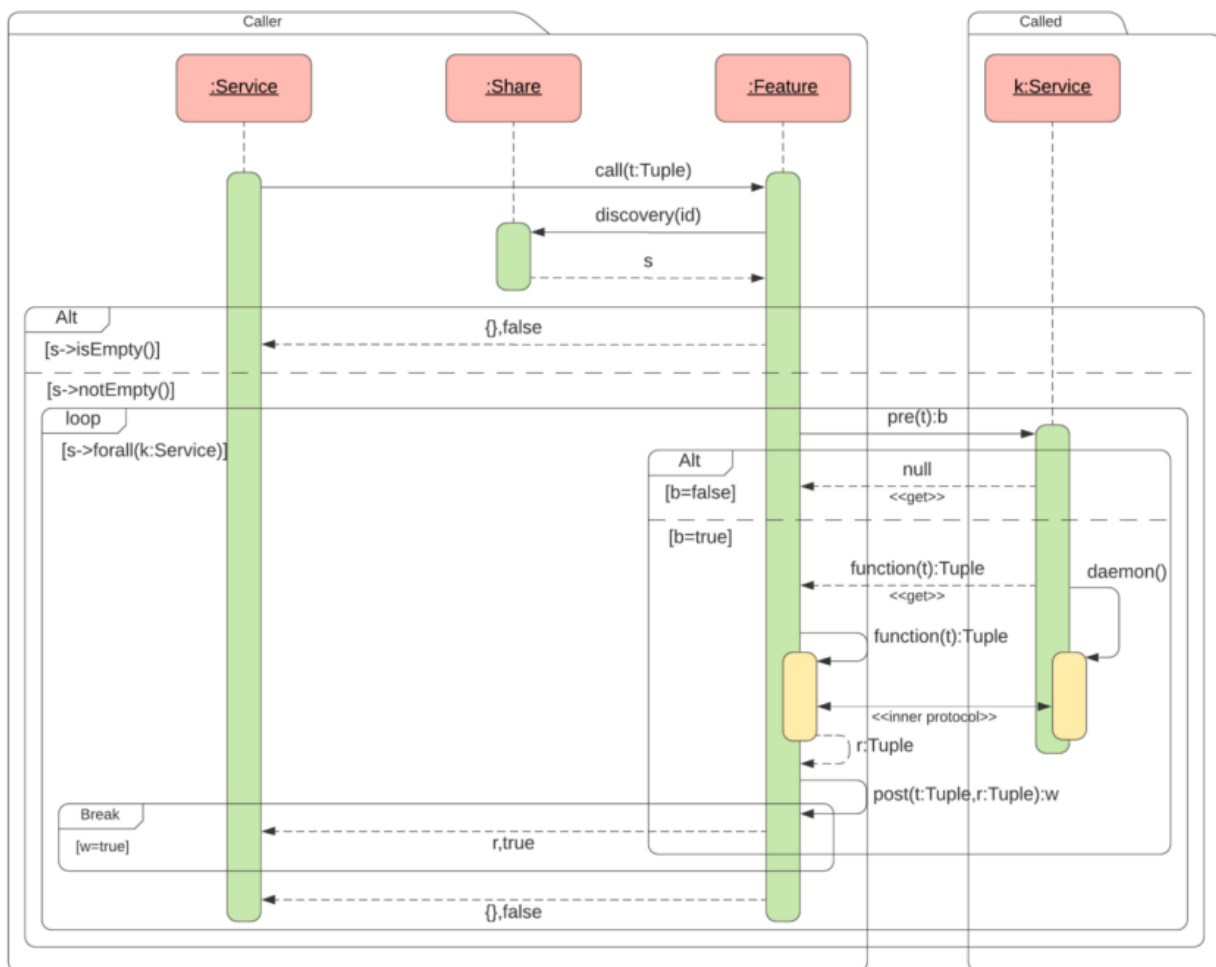
Note: A management information base (**MIB**) is a database used for managing the entities in a communication network. Most often associated with the Simple Network Management Protocol (**SNMP**), the term is also used more generically in contexts such as in OSI/ISO Network management model.

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 Sphinx² 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 you could simply start to document your class in this way:

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

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

¹Modified 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

²*Sphinx* is a documentation generator written and used by the Python community. It is written in Python, and also used in other environments.

³GitHub 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'être* 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 protocol
- `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

Index

A

`attach()` (*Share method*), 7

C

`call()` (*Feature method*), 9

D

`detach()` (*Share method*), 7

`discovery()` (*Share method*), 8

F

`Feature` (*built-in class*), 9

`find()` (*Share method*), 8

I

`is_present()` (*Share method*), 8

N

`new()` (*Feature method*), 9

`new()` (*Service method*), 8

`new()` (*Share method*), 7

O

`open_udp_socket()` (*Share method*), 8

S

`Service` (*built-in class*), 8

`Share` (*built-in class*), 7