APPLYING DESIGN PATTERNS TO SCA IMPLEMENTATIONS

Adem Zumbul (TUBITAK-UEKAE, Kocaeli, Turkey, ademz@uekae.tubitak.gov.tr);
Tuna Tugcu (Bogazici University, Istanbul, Turkey, tugcu@boun.edu.tr)

ABSTRACT

This paper provides insights into applying common design patterns while developing SCA compliant Core Frameworks and Waveforms. This paper also presents an approach to leverage SCA as a means of abstracting the Core Framework and Waveform implementations from the Operating System and Object Request Brokers. In addition to these, the experimental results of implementing a fully functional SCA Core Framework and some Waveforms in ACE/TAO and ORBExpress Object Request Brokers on Linux and VxWorks operating systems in terms of applying common design patterns are summarized.

1. INTRODUCTION

The sharp increase rate of technology behind the processors leads techniques to define radio behavior by using software and introduces Software Defined Radio (SDR) [1] concept. Software Defined Radios are flexible communication devices that can operate as different radios depending on the installed Waveforms. Since the hardware and software components of this evolving concept have not been fully standardized, the implemented SDR applications have lots of question marks on portability, reconfigurability and reusability issues. Software Communications Architecture (SCA) [2] is a standard that is intended to address these problems. It basically provides high level CORBA interfaces SDR applications. These interfaces contain common operations that every SDR application should implement. It also provides some guidelines to define behavioral details of these operations. Although the SCA standard tells developers “what to implement” it does not deal with “how to implement.” Most of the implementation details are intentionally left to the developers in order not to make the SCA harder to understand.

In order to derive the most benefit from SCA in terms of portability, reconfigurability and reusability, it is imperative to be familiar with the software engineering concepts such as Object Oriented Programming (OOP) [3] and design patterns [4]. The design patterns are optimum solutions to common software engineering problems. Applying design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems, and it also improves code readability for coders and architects who are familiar with the patterns.

The rest of the paper is organized as the following. Chapter 2 provides some background information on design pattern concept. Chapter 3 presents the usage of common design patterns in SCA compliant SDR implementations and then the paper concludes.

2. DESIGN PATTERNS

Design patterns as a whole can help people learn object-oriented thinking: how to leverage polymorphism, design for composition, delegation, balance responsibilities, and provide pluggable behavior. The importance of using suitable software design patterns has been understood better after the publication of the book *Design Patterns: Elements of Reusable Object-Oriented Software* [5]. The authors of it are often referred to as the GoF, or Gang of Four. In their book, they present 23 design patterns organized into three categories:

2.1. Creational Patterns

Creational patterns deal with object instantiation problems. There are 5 creational patterns including Abstract factory, Builder, Factory, Prototype and Singleton. They are intended to solve object creation problem.

2.2. Structural Patterns

Structural patterns consist of Adapter, Bridge, Composite, Decorator, Facade, Flyweight and Proxy. These patterns concern class and object composition. They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality. These patterns may be applied while designing a new system from scratch or while modifying existing codes to port from one system to another.

2.3. Behavioral Patterns

Behavioral patterns including Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method, and Visitor, solve object communication problems and depicts how objects act together. All of these patterns help developers to design their software better in terms of quality, reusability and understandability. Basically, all of the design patterns tell the following golden rules in common:

a- All client objects should always call the abstraction (interface) and not the exact implementation.

b- Future changes should not impact the existing system.

c- Change always what is changing.

d- Have loose coupling between objects.

3. APPLYING DESIGN PATTERNS TO SCA

In this section, we present some guidelines to apply common design patterns while developing SCA compliant SDR applications such as Core Framework and Waveforms and we provide some of the possible application areas.

3.1. Factory Method

Factory Method pattern defines an interface for creating an object, but let subclasses decide which class to instantiate. It lets a class defer instantiation to subclasses.

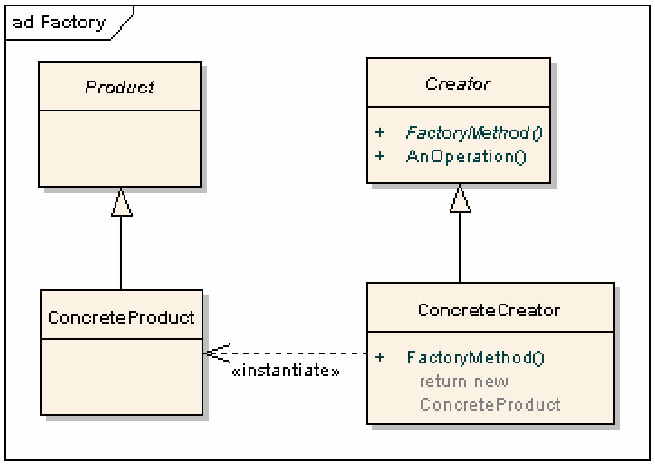
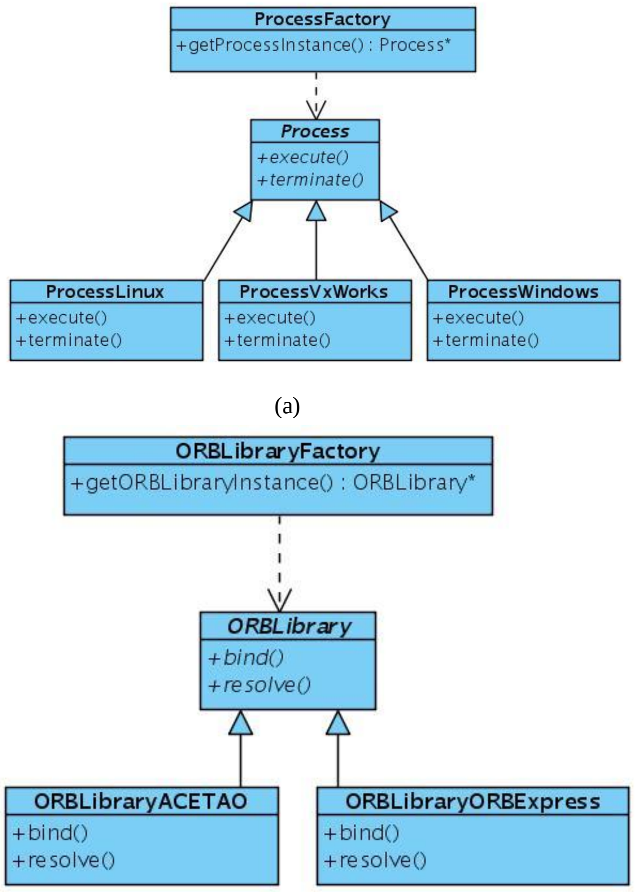


Figure 1: Factory Method Pattern.

As Figure 1 shows, the pattern uses two types of classes. The Product classes, which are the classes that make up the application and the Creator classes, which are responsible for defining the Factory methods used to create instances of Product objects. The Creator class defines the factory method, which returns an object of type Product. The Concrete classes provide the appropriate implementation for their respective base class.

Factory method is very useful and common design pattern to solve portability problems. It can be used to separate Operating System and Object Request Broker specific implementations with the rest of the system. Factory pattern uses a factory which decides which specific subclass to handle the request of the client.

This pattern is also suitable to manage configuration specific issues. Factory class may be used to check the configuration value when returning the possible concrete handler class. It is obvious that changing the configuration parameter will make the factory to return a new appropriate concrete handler. Also the factory class may keep a list of the created objects and can be used to kill or modify them by only traversing the managed list of instances.



(b)

Figure 2: Example Usage of Factory Method Pattern.

Figure 2 shows an example scenario where Factory Method pattern is used to separate OS specific codes from the rest of the system. In this diagram, Process is the interface class that defines mandatory functions that every derived class should implement. ProcessLinux and ProcessVxWorks classes concretely defines OS specific execute and terminate functions and overrides generic Process interface. In this scenario, ProcessFactory checks the OS configuration parameter of the Core Framework and returns appropriate Process class. Regardless of the returning concrete Process class, the client can call execute and terminate operations on the returned object.

Figure 2-b shows an example usage of the Factory pattern to decouple the applications from ORB specific functions. ORBFactory class has a getORBLibrary method which checks the configuration parameter of the Core Framework and returns the concrete ORBLibrary class so that the clients that need CORBA functions can call ORB functions independently.

It is worth noting that Factory Method pattern is very helpful to deal with possible future changes. Changing the configuration parameter of the factory makes the system to behave according to the new situation without affecting the existing codes. This also allows the system to extend by defining new derived concrete classes.

3.2. Chain of Responsibility

Chain of Responsibility pattern is used to avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. It chains the receiving objects and passes the request along the chain until an object handles it.

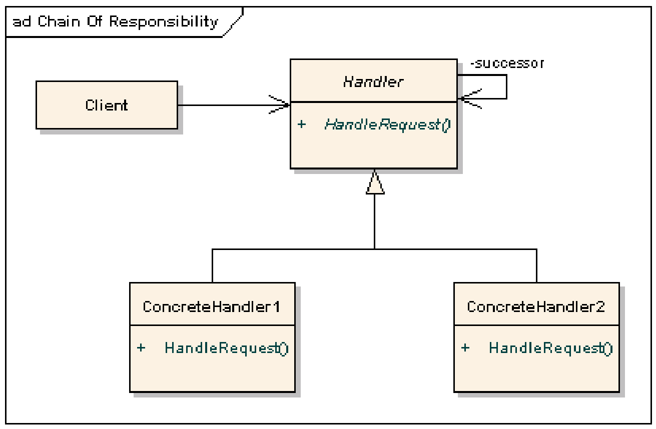


Figure 3: Chain of Responsibility Pattern.

This pattern can be applied to many cases while developing software for SDR systems. SCA standard tells developers to design their system in a hierarchical manner which means dividing the software architecture into collaborating components. Such a distributed system requires a strong management mechanism for the responsibilities of the components. From that point of view, this design pattern can be considered to be a useful blue print to handle responsibility related issues.

SCA interfaces defines port concept as a communication mechanism between components. A port represents a CORBA interface of which reference can be transferred between components so that distributed components can make CORBA calls on each other. Efficient use of port mechanism in conjunction with Chain of Responsibility pattern can let developers to manage object responsibilities even the objects are distributed.

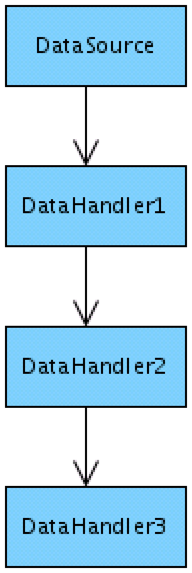


Figure 4: Example Usage of Chain of Responsibility Pattern.

Figure 4 illustrates a generic usage scenario which can be applied to similar situations. In this figure, four components of a Waveform are shown. They are all connected to each other by using port mechanism. In this layout, DataSource component is responsible to get some data to be processed by the Waveform and the other three components are chained each other and each of the component implements a different algorithm to process the data. In this example scenario, DataSource component is not aware of which component in the chain will handle the request and it does not have to. It only concentrates on the job of fetching the data to push to the chain. In this scenario, each DataHandler component checks some internal or external parameters to decide whether to handle the incoming data or not. The parameters that can be checked during deciding stage can be permissions, capacity values, priorities, dependencies, performance requirements and structural properties of the incoming data or so on.

It is obvious that this pattern lets insertion of new handlers into the chain without affecting the rest of the system. It is also valid for the removal case. The developer does not have to modify any component for any changes. This behavior can be life saving for the systems that requires frequent modification of the code according to changing conditions such as development phases. It can also be an interesting idea to chain the instantiations of the same component to balance the work load among different processors.

3.3. Adapter

Adapter pattern is used to convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

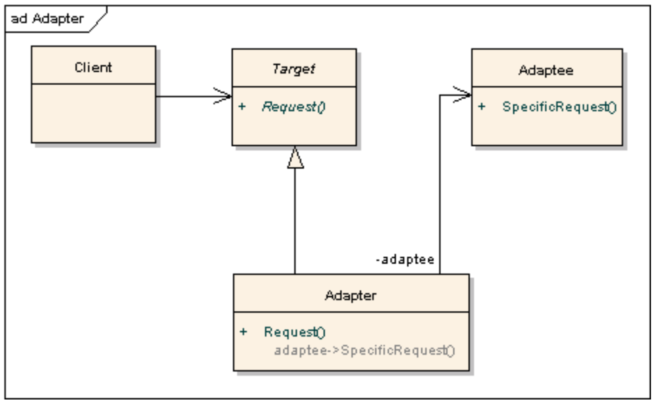


Figure 5: Adapter Pattern.

Adapter pattern has also very common usage in SCA implementations. It allows legacy codes that do not support SCA interfaces to work together with the SCA codes. It simply adapts the old interface to the new one.

Adapter pattern is typically not used when designing a new system from scratch but rather used to port existing codes from one interface to another.

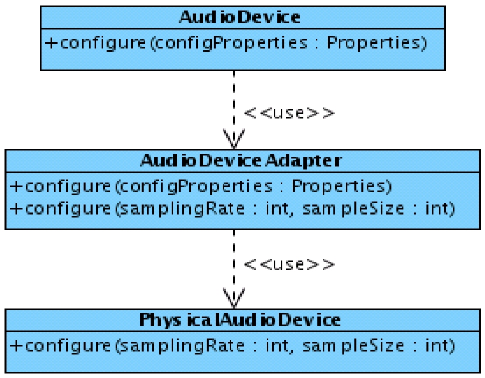

AudioDevice
+configure(configProperties : Properties)
<<use>>
AudioDeviceAdapter
+configure(configProperties : Properties)
+configure(samplingRate : int, sampleSize : int)
<<use>>
PhysicalAudioDevice
+configure(samplingRate : int, sampleSize : int)

Figure 6: Example Usage of Adapter Pattern.

Figure 6 shows an example usage of the Adapter pattern. In this scenario adapter class adapts configure methods of different audio device interfaces. As shown in the figure, legacy PhysicalAudioDevice class has a configure method which accepts integer configuration parameters, whereas SCA compliant AudioDevice class accepts Properties structure as input. AudioDeviceAdapter class translates parameters of these classes between each other so that they can work together.

3.4. Singleton

Singleton pattern is applied to ensure a class only has one instance, and provide a global point of access to it. It is a relatively simple pattern to apply.

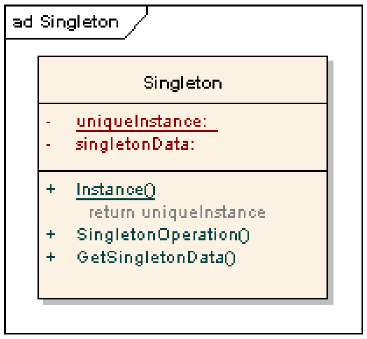


Figure 7: Singleton Pattern.

In the programming domain, it is a very common situation that a class is required to have only one instance. For the SCA point of view, Singleton pattern can be frequently used. For example, Device classes that wraps a specific hardware usually requires to have only one instance, because the managed device usually cannot be initialized more than once and the capacity values should be under control of a single capacity manager. Another example can be the ORBLibrary classes that initialize and manage POA (Portable Object Adapter) according to a specific ORB policy. Singleton pattern can be used together with the Factory Method pattern to ensure returned concrete classes to have only one instance. In this case, Factory class can check the instance count of the singleton objects and return the same object whenever it creates an instance.

3.5. State

State pattern allows an object to alter its behavior when it's internal state changes. The benefit of State pattern is that state specific code is localized in the class that represents that state.

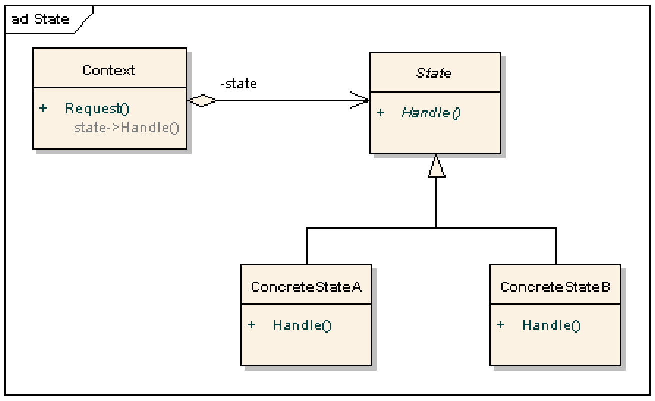


Figure 8: State Pattern.

SCA defines three types of state types for Device classes. They are OperationalState, AdminState and UsageState. OperationalState can be ENABLED or DISABLED and indicates whether the device is functioning or not. AdminState keeps track of the permission or prohibition against using the device and it can take values of LOCKED, SHUTTING_DOWN or UNLOCKED. Finally, UsageState defines Device's usage state and can be IDLE, ACTIVE or BUSY. IDLE means that Device is not in use, BUSY corresponds to Device is in use and no capacity is left for allocation and ACTIVE shows that Device is in use and it still has some capacity for allocation. In addition to built-in states, it is possible to add user defined states for different situations of the components.

Applying State pattern helps developers to separate state dependent operations from the rest of the functional code of the components and it reduces complexity.

3.6. Facade

Facade pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

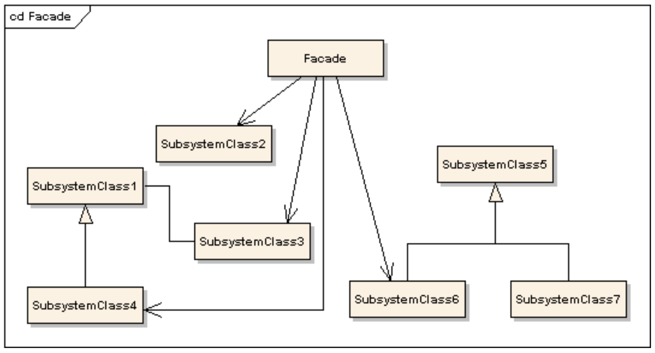


Figure 9: Facade Pattern.

It also simplifies and beautifies an existing cumbersome class by behaving as a door to its complex interface. It means it works as an intermediator between the client and the subsystem. Facade should not be the part of the subsystem, if this is the case it should move to the subsystem and a new Facade class should be generated.

From SCA point of view, Facade pattern can be applied while porting non-SCA legacy codes to SCA compliant wrapper codes. SCA wrapper codes may use Facade classes to access the legacy parts so that the developer may not spent time to re-implement already existing functional codes. Also Façade pattern may be applied to collect separate CORBA interfaces into a single CORBA interface which may reduce complexity.

4. CONCLUSION

In this paper, we summarize design pattern concept in general and provide example application areas for some of them to the SDR and SCA applications. We present the usage of Factory Method, Chain of Responsibility, Adapter, State, Singleton, and Facade patterns and provide some UML diagrams to illustrate our ideas. In addition to our work, application areas of the rest of the 23 design patterns may also be explored as a future work.

As part of a project in TUBITAK, we have developed SCA compliant Core Framework and some Waveforms to run on Linux and VxWorks operating systems on ACETAO and ORBExpress object request brokers. During development stage we have investigated suitable design patterns and applied them to our code in order to capitalize the benefits of SCA. Applying design patterns dramatically supported our development stages and the developed code has been tested against portability between different operating environments. By only changing some configuration parameters and recompiling the existing codes we have achieved to run our Core Framework on different platforms.

5. REFERENCES

[1] Software defined radio: architectures, systems, and functions. Dillinger, Madani, Alonistioti. Wiley, 2003. 454 pages. ISBN 0470851643 ISBN-13: 9780470851647.
[2] Joint Tactical Radio System, "Software communications architecture specification-Final", Version 2.2.2, Space and Naval Warfare System Center, San Diego CA, 15 May 2006.
[3] (1995) Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN 0-201-63361-2.
[4] Fowler, Martin (2006-08-01). Writing Software Patterns. Retrieved on 2007-03-06.
[5] Design Patterns: Elements of Reusable Object-Oriented Software. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides in 1995.