Applying the Extension Object Pattern to the Software Communications Architecture

Dominick Paniscotti
Bruce Trask

The Visitor Family of Design Patterns

In his book Agile Software Development, Pattern Principles and Practices, Robert C. Martin talks about the Visitor Family of Design Patterns. He begins by stating the problem:

> *Your need to add a new method to a hierarchy of classes, but the act of adding it will be painful or damaging to the design.*[1]

He goes on to state that the Visitor family of Design Patterns deals with this by

> *Allowing new methods to be added to existing hierarchies without modifying the hiearchies.*[1]

We have found this particular problem and its concomitant solutions to be a fertile source of didactic and pedagogical material for students and employees learning about design patterns.

Robert Martin enumerates the 4 members of the family as

1. Visitor
2. Acyclic Visitor
3. Decorator
4. Extension Object

His book goes into great detail as to what these patterns are all about and when to use one over the other.

Student and Employee Experience

Most students new to Design Patterns are familiar with the concept of hierarchies and the mechanism of inheritance. Most however, are locked into a mechanical use of that facility and don't know how to branch out and use that feature in new ways. Some students have trouble seeing when to use inheritance to solve a problem and when not to. Or, if they are using inheritance to solve a problem, they are not sure where to use it in the solution and where not to. Ask most students new to object-orientation or design patterns how to add new methods to an existing hierarchy without modifying the hierarchy and they might look at you sideways.

Student reactions to this question and to the problem and solution provided by the Visitor family of design patterns above, are very similar to Alan Shalloway's response to the Bridge Pattern intent in the Design Patterns, Gang of Four book which is to "De-couple an abstraction from its implementation so that the two can vary independently". Alan states, "I remember exactly what my first thoughts were when I read this: huh? How come I understand every word in this sentence but I have no idea what it means"[4]

We have seen similar reactions from design patterns students when faced with the problem of adding to existing hierarchies without modifying those hierarchies.

Their first question usually is "what exactly does it mean to add to hierarchies without modifying them?" Once they understand the answer to that question, they usually wonder what might motivate someone to want to do such a thing. Once they understand that, they wonder how one might implement it in a sound fashion applying the principles of object-oriented design and design patterns. Later they start to branch out on the various tradeoffs between different variants in the possible solutions.

The Software Communications Architecture

For us in industry, these students are new employees or experienced software engineers that are new to Object-Orientation and Design Patterns. In our company specifically, these practitioners are faced with understanding the Software Communications Architecture (http://jtrs.army.mil/sections/technicalinformation/fset_technical_sca.html), which is a component-based object model used for the architecting software-defined radios. At the heart of the design of this architecture is a need to add or extend existing hierarchies without modifying the hierarchies. The architecture loosely provides the facilities to use a member of the Visitor family of design patterns called the Extension Object Pattern[2], or the Extension Interface Pattern.[3]

We have found that designers struggling with understanding OO, Design Patterns, and the SCA, increase their skills in these areas when they understand the applicability of the Extension Object Pattern to this domain and its use case. Perhaps more importantly, they come to see the importance of using Design Patterns in solving serious real-world problems. In particular, the problem being solved in the case of the SCA is a heavyweight problem (the context of which we will explain later), and as such requires a somewhat heavyweight and flexible solution. There are, of course, tradeoffs and prices to be paid for this flexibility, but when implemented correctly these can be mitigated significantly. We feel this problem and the use of the Extension Object/Interface Pattern is a killer example of how Design Patterns can be effectively employed in the real world.

Some History of the SCA

Recently there has been a sea change in the way software is being designed in the Software-Defined Radio domain. This change was prompted by the US Government's mandate that all military radios and communications equipment must conform to a new software architecture called the Software Communications Architecture. In 1998, the Government assigned four companies the task of designing a software solution that would relieve the Government of having to pay over and over again for software as radio technologies change. This group of companies, the Modular Software Radio Consortium (MSRC), was tasked to analyze the radio communications and networking domain. They were to isolate and distill out of the domain a deployment and configuration framework that would work with all existing radio applications (called waveforms) as well as applications yet to be developed. They were also tasked to define the actual interfaces and the nature of the component object model that would allow software radio applications to be easily ported to new platforms with a minimum of expense. This

component object model and architecture would have to provide facilities and abstractions for cordoning off the commonality and variability in the domain, then define the interfaces to provide a consistent framework for all waveforms, while leaving a door open for open-ended extensions that would not rip apart the basic architecture. In other words, this architecture, design, and object model had to be constraining enough to promote portability and re-usability but flexible enough to allow for open ended extension. Not an easy task indeed.

The Problem and the Solution

The main problems facing the MSRC included these:

1. Provide portability of application software across diverse platforms
2. Support the dynamic deployment and configuration of radio applications across disparate physical hardware
3. Promote re-use of the radio applications as well as the software modules that make up the application
4. Allow for the support for all existing radio applications and future waveforms not yet developed

More specifically, the SCA developers had to account for the following when developing the basic object model and component abstraction (known as CF::Resource):

1. The designers of CF::Resource had no a priori knowledge of the interfaces component developers wanted to add to CF::Resource
2. Sub-classing is too static a solution. May require client side re-compilation
3. Risk of CF::Resource interface becoming bloated based upon demands of Application developers. Bloating ultimately leading to lack of interface cohesion
4. To avoid these problems, requires a component design that supports evolution, both anticipated and unanticipated

The broad strokes of their solution included these tenets:

1. Deliver a component-based framework
2. Abstract the physical hardware from the radio applications to increase portability/maintainability
3. Define standard mechanisms to describe, package, and deploy application components
4. Use middleware to promote language and hardware independence and provide location transparency
5. Use industry-standard design patterns to provide the necessary abstractions that would allow for scalable evolution of software defined radios.

These guiding principles were held during the design of the solution:

1. Favor object composition over class inheritance[5]

2. Interface Segregation Principle[1] – provide a mechanism for clients to retrieve only the interfaces they need and not depend on others provided by the component that they don't use.

3. Single Responsibility Principle[1] - keep related operations together by defining them in a separate interface.

4. Open Closed Principle[1] – extend the functionality of the component without impact to its existing implementation

A Review of the Extension Object Pattern

This next section will review the Extension Object/Interface Pattern, relating it fully to the components and abstractions of the Software Communications Architecture. Full descriptions of the Extension Object Pattern can be found online (http://163.152.216.87/selab/courses/adse-02s/supplement/gamma96.pdf) and in books. The Extension Interface Patterns is thoroughly described in books as well.

Erich Gamma says the intent of the Extension Object Pattern is this:

1. "Anticipate that an object's interface needs to be extended in the future. Additional interfaces are defined by extension objects[2]

2. Allow multiple interfaces to be exported by a component, to prevent bloating of interfaces and breaking of client code when developers extend or modify the functionality of the component"[2]

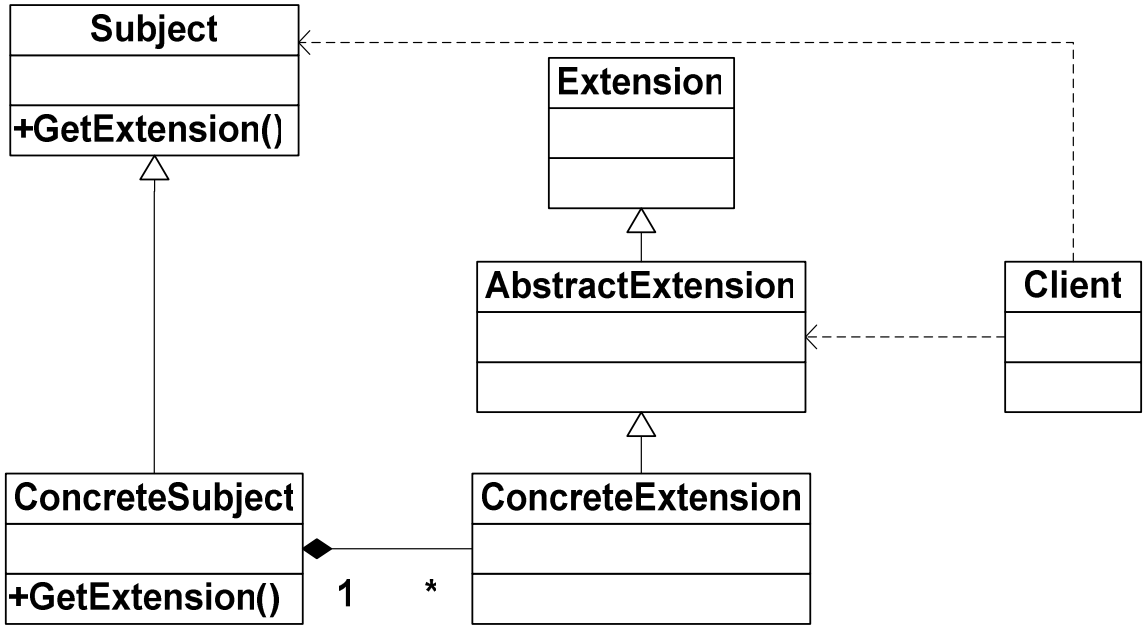
He goes on to say when one should use the Extension Object pattern:

1. "You need to support the addition of new or unforeseen interfaces to existing classes and you don't want to impact clients that don't need this new interface. Extension Objects lets you keep related operations together by defining them in a separate class" [2]

2. A class representing a key abstraction plays different roles for different clients. The number of roles the class can play should be open-ended. There is a need to preserve the key abstraction itself. For example a customer object is still a customer object even if different subsystems view it differently" [2]

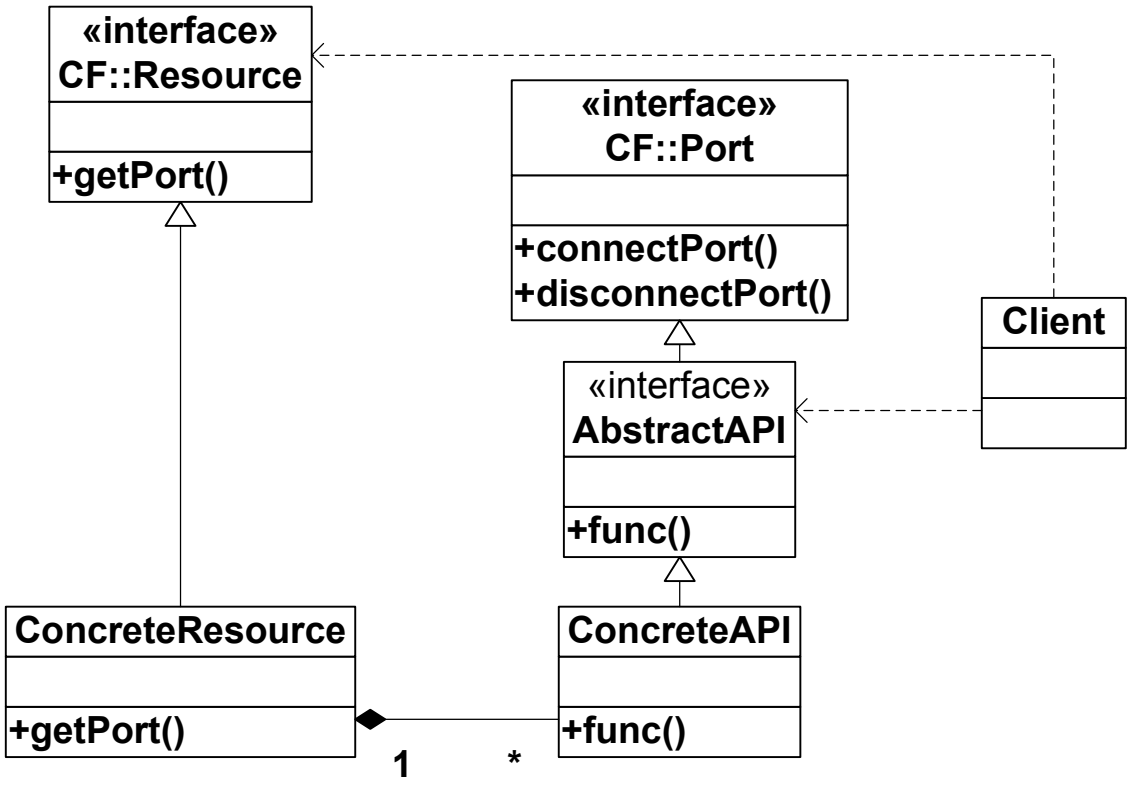
3. "A class should be extensible with new behavior without subclassing from it" [2]

The basic structure is shown in the figure below.



Descriptions of the responsibilities of the various components of this pattern are in the references.

We have found that when students and employees first start learning and trying to apply the principles of the SCA, they frequently get lost in the abstract nature of the specification and cannot see the forest for the trees. When presented with material as to how the Extension Object Pattern maps to the SCA, student and employee understanding and ability to apply the concepts of the SCA skyrockets. How does the pattern map to the SCA? Below is the basic component UML diagram from the SCA.

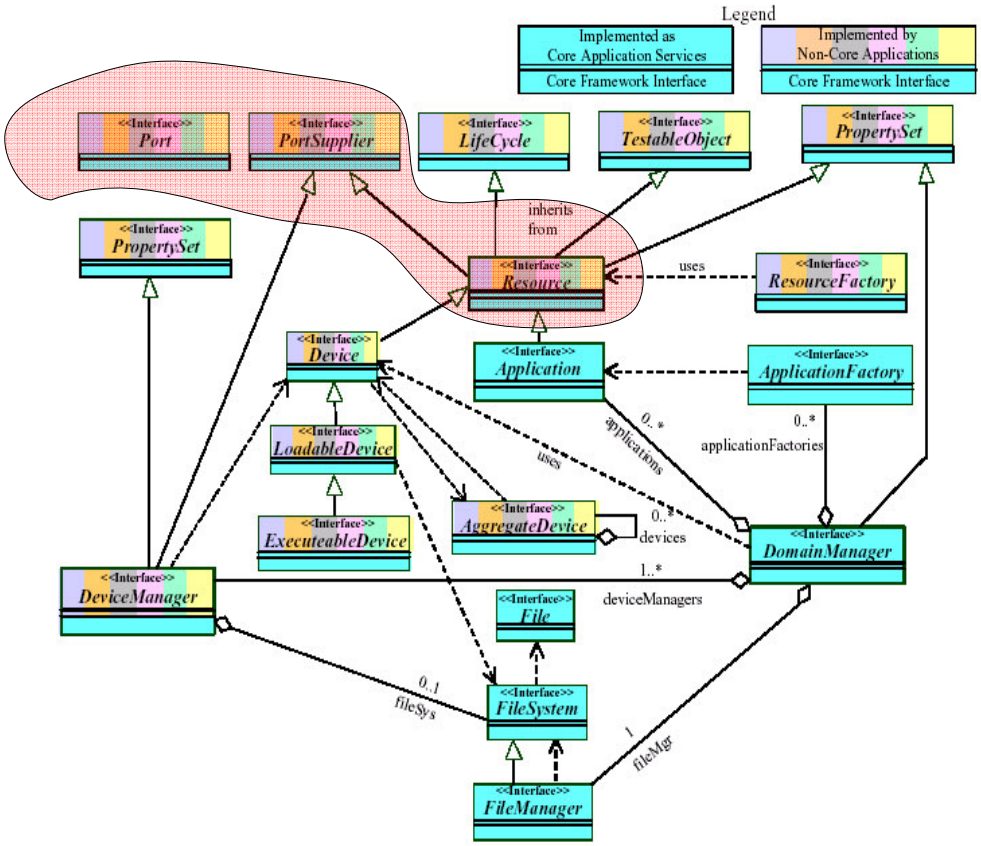


Even cursory examination of the above UML diagram and comparison of it to the canonical Extension Object structure diagram provided in the original pattern shows that the mapping is one to one. This direct correspondence runs deeper than just in the structure. The roles, responsibilities and dynamics map directly as well.

Using the Extension Object Pattern, these radio components truly move into the realm of being software ICs just like their hardware counterparts. Their "ports" correspond to the pins of an IC and the guts of the Resource correspond to the business logic of the integrated circuit.

The SCA's support for the Extension Object Pattern

It was mentioned earlier that the SCA "loosely" provides facilities to leverage the Extension Object Pattern or other members of the Visitor Family of Design Patterns. Below is the class diagram of the SCA with the areas of interest circled in red. The main component is the Resource class and it has a base interface called PortSupplier which provides a getPort() method. Notice that the SCA also provides a Port class but this class has no explicit relationships to the Resource hierarchy. As such this leaves the door wide open for (mis)applications of various techniques including particular design patterns such as the Extension Object Pattern. We have found that the introduction of the Extension Object Pattern—along with the structure, responsibilities, and dynamics that it requires—greatly assists students and engineers in understanding how the abstractions in the SCA best fit and play together



The Consequences and Benefits

Here are some of the consequences and benefits of using the pattern in this domain:

1. The key abstraction, CF::Resource, is not polluted with operations that are specific to a given client or waveform component. More flexible than extending the interface via subclassing. The getPort operation allows for open-ended extension.
2. Dealing with extension interfaces can be more complicated from a client's perspective, but this complication is cordoned off to the core interfaces of the Core Framework. These interfaces are the ApplicationFactory and the DomainManager. They deal with the querying of components for extension interfaces (getPort), "soldering" together the connections (connectPort and disconnectPort), and the necessary error/exception handling needed if the interface does not exist or the connection fails.
3. The ports themselves can verify that they are being connected to the correct type of port (via the _narrow call). Component developers need not concern themselves with this complication
4. Using the Extension Object Pattern, the SCA Application can export its different roles depending on what interfaces are requested and supplied by particular

clients. For example, certain clients are interested in the SCA app from a data throughput standpoint versus built-in test clients.

5. Security – both the extension object and proxy patterns lend themselves well as mechanisms to support security mechanisms. Requests for interfaces can be validated and smart proxies can be used to control traversal from usesport to providesport.
6. Allow for event sinks and sources and log sinks and sources to be easily added to Resources.

One additional benefit is that practitioners tasked with writing software according to this specification understand the underlying patterns upon which the heart of the architecture is built. They can understand the commonality and variability the SCA is designed both to constrain and to allow for extension. The SCA is a very unique example of a design and architecture that must scale over time and evolve with new applications and waveforms and do so in such a way so as to remain cohesive. The use of the Extension Object/Interface Pattern is an excellent example for students to see how one accomplishes such flexibility and evolution in a sound manner.

[1] Agile Software Development – Pattern Principles and Practices, Robert C. Martin, Prentice Hall, 2003
[2] Pattern Language of Pattern Design 3, R. C. Martin, D. Riehle, F. Buschmann (eds.)
[3] Schmidt, et al. *Pattern-Oriented Software Architecture*, Volume 2, Wiley, 2000
[4] Design Patterns Explained, Alan Shalloway, Addison Wesley, 2002
[5] Design Patterns, Erich Gamma et. al. Addison Wesley, 1995