

## Complex Interface Pattern

Louis-Pierre Gagnaux

### Example

Most applications are required to notify alarms and to provide management information, so that a responsible operator can track and control the behaviour of the applications [X733]. For example the operator wants to take immediate corrective actions when a critical application error occurs, e.g. the application or parts of the application are not operating in a required way. Like in figure 1 an application is reporting an alarm, e.g. the billing system cannot process incoming chargeable events in a required time-slot and that a defined threshold of pending chargeable event has been exceeded. An operator should then be able to ask for the state of the application, e.g. he could ask for the current amount of pending unprocessed chargeable events or for the current threshold value.

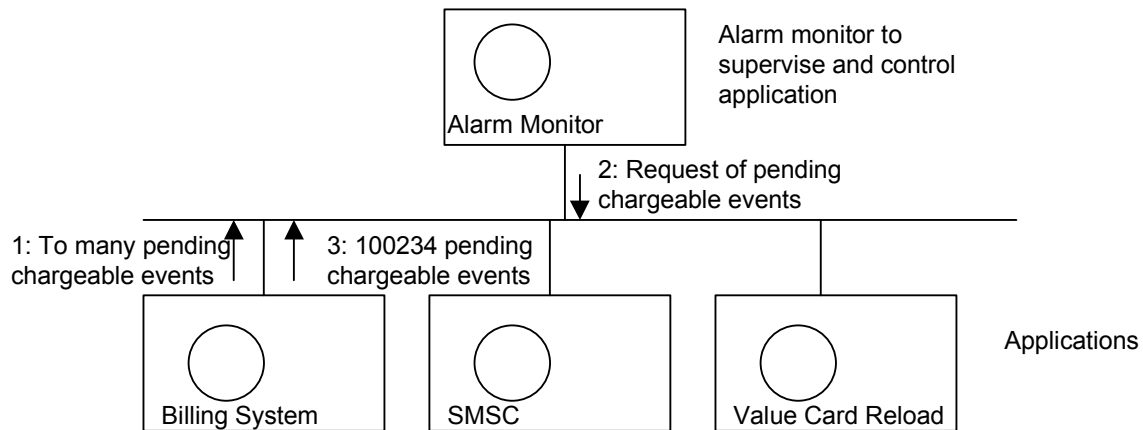


Figure 1 Example of an application alarm and monitoring of application alarms

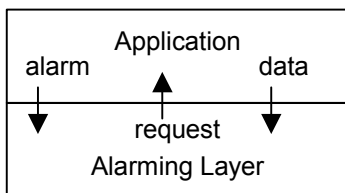


Figure 2 Encapsulation of the alarming protocol

Applications can use a proprietary solution to create alarms and monitor the alarms or can use a standard protocol like SNMP (Simple Network Management Protocol). Which ever protocol you decide to use, it is a good technique like in figure 2 to encapsulate the used protocol in a separate layer 'Alarming-Layer' and that the applications use a defined interface for alarming. This will allow to change the protocol layer at any later time and will prevent that the application programmers need to understand, how SNMP or any proprietary alarming protocol is working, they just needs to understand the interface.

Providing an alarming interface like in figure 2 is however not that simple and can lead to a rather complex interface. The application will have to provide different types of data like numbers, strings, time, date and may be even structured types like tables or lists.

## Context

Provide a complex interface with a set of classes with common behaviour and common encapsulated implementation.

## Problem

Often in telecommunication the software architecture is designed in layer, where each layer is responsible for defined tasks (protocol handling, marshalling, binary transport) and thus each underlying layer has to expose a more or less complex interface to the superior layer, where you have to consider the following requirements.

- The underlying layer should be adaptable without changing the interface. For example if the alarming layer should provide a more reliable transfer to the alarm monitor, it should be possible to adapt the alarming layer without having to adapt all applications using this interface.
- Often it is also required that the underlying layer is exchangeable at run-time, so that every application can be configured, which layer it wants to use, if different implementation of the underlying layer are available. The alarming layer could for example be implemented using once a proprietary protocol and later be implemented to use SNMP. So the customer can configure which protocol he wants to use for alarming.
- Implementing an interface where many objects have to be passed or provided from the interface will require to define a set of classes. Often these classes have a common interface. It is obvious that if an application in the alarming example needs to send many different objects, all these different objects will have to provide a 'send()' -method and it makes sense to define this method in a base class. To put the common part of the interface in a base class will then force also to structure the implementation similar. This is actually the major problem, how in conjunction having to provide a complete abstract interface inheritance can be used in the interface as also in the encapsulated implementation of the interface.
- If a layer is used by many applications, this layer is best deployed as a component, so that in case of enhancements or bug fixes only the component needs to be shipped and exchanged at customer site and that not all applications have to be recompiled. This requires that the component's interface is bound at run-time. See also the Component Pattern in [POSA2].

## Solution

The layer architecture pattern [POSA1] is a good starting point where the underlying layer exposes his interface with only abstract classes. This results that the interface is bound only at run-time.

In figure 3 the abstract classes 'I1' and 'I2' provides the interface. But how can a superior layer use these abstract classes? Obviously these classes cannot be instantiated, because they are abstract. So you will have to use the Abstract Factory Pattern [GHJV95], where the factory object creates objects of the concrete classes derived from the interface classes as figure 3 shows:

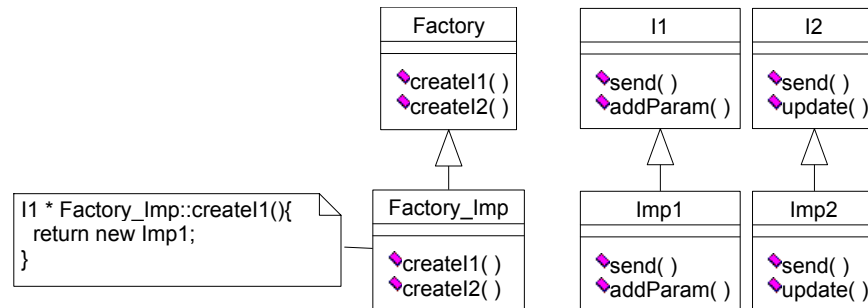


Figure 3 Abstract Factory Pattern

To use the interface class 'I1', you have to create a factory object and use the create methods to create the concrete objects, in this example an object of type 'Imp1', which implements the interface, defined by the abstract classes 'I1'.

Of course you could have just created an object of type 'Imp1', but then the application code will have to bind at compile-time to the underlying layer, which is not desired in case you want to install a new version of the underlying layer without having to recompile the superior layer.

The same principle applies to the factory object. Also here you cannot create an object of 'Factory\_Imp' directly. Normally this object is provided as a static object in a dynamic library and before it can be used the library is loaded and the factory object is found by a symbol lookup.

If the interface classes have a common interface, inheritance seems then to be the appropriate way to define the common interface once, as shown in figure 4. For example if the interfaces I1 and I2 have a both a 'send()' method it is most likely to define this method in the base interface 'ICommon':

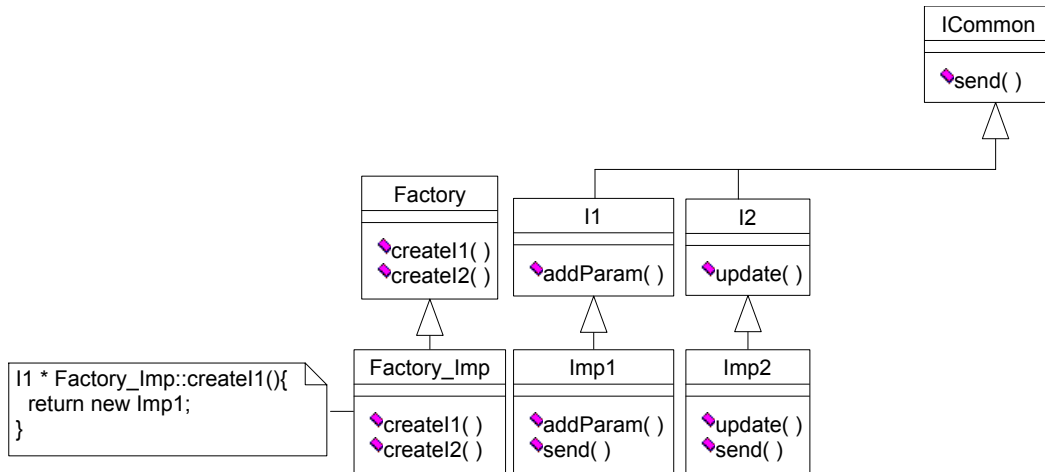


Figure 4 Abstract Factory Pattern with a common interface class

If common behaviour for the implementation classes exists, you will also try to implement this behaviour in a common base class, which is outlined in Figure 5. For example the semantic for sending objects could be the same for every object, if every object defines an appropriate encoding method, which will be called by the 'send()' method.

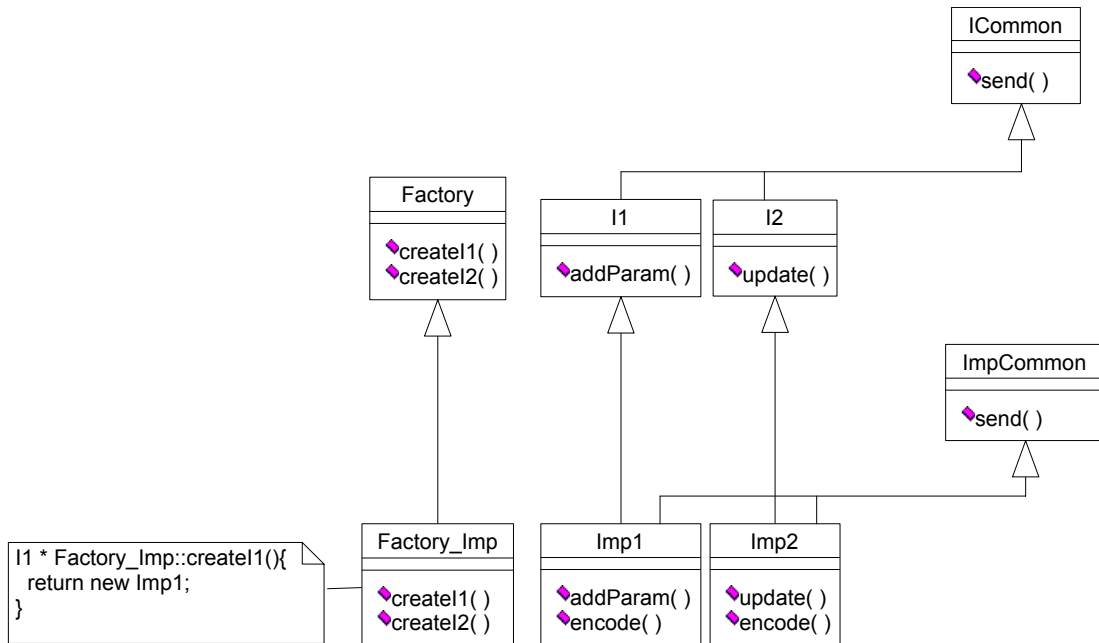


Figure 5 Abstract Factory Pattern with a common interface and a common implementation class

But this doesn't work, because the compiler will assume that the classes 'Imp1' and 'Imp2' are abstract because these classes do not implement the 'send()' method directly but through the inherited method from 'ImpCommon'.

The solution is drawn in figure 6 and is to define the class 'ImpCommon' as a derived class from 'ICommon'.

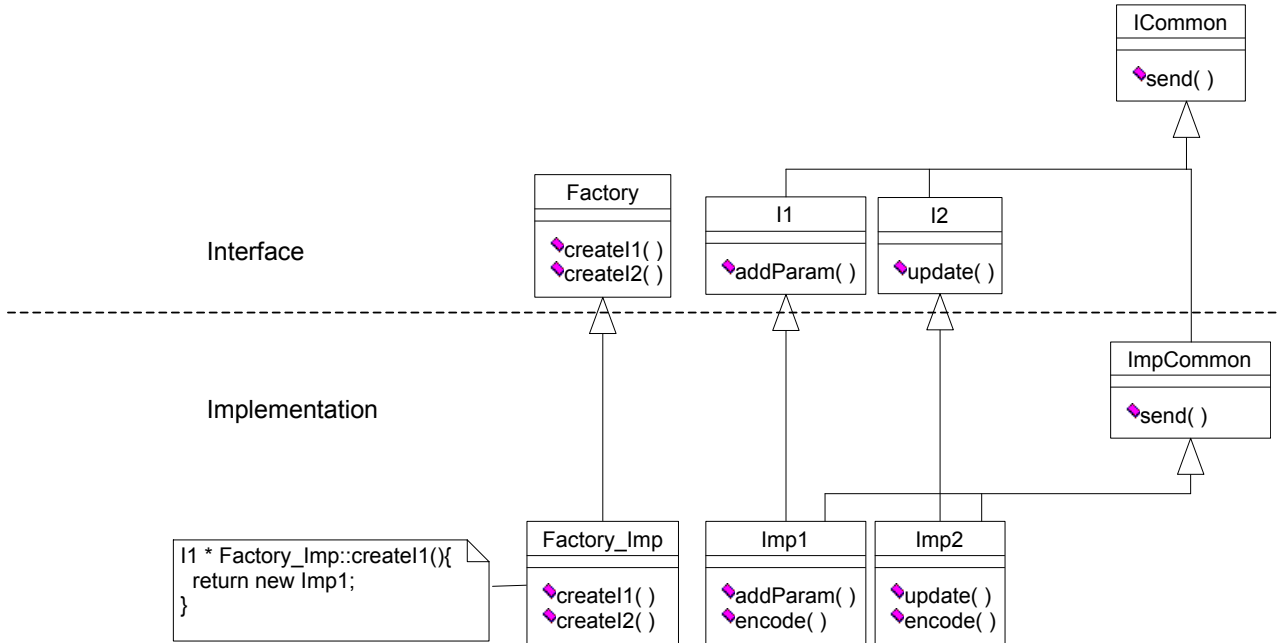


Figure 6 Complex Interface Pattern

Implementing a complex interface using the structure in figure 6 has the following strengths:

- It only reveals an interface while the implementation is encapsulated.
- The interface is abstract, thus the client using the interface binds at run-time. This gives the flexibility to exchange the component implementing the interface at run-time and that the client has not to be recompiled when another variant of the component is used. This late binding is the basic principle for deploying components, which mostly are implemented with dynamic libraries.
- Implementing an interface where many objects have to be passed or provided from the component, the common interface definition and also the common implementation behaviour is best implemented in separate base classes, where the implementation base class must also be derived from the interface base class. The concrete classes are then derived from the interface classes and also from the common implementation base class.
- If a component is used by many applications only the component needs to be shipped and exchanged at customer site in case of enhancements or bug fixes.

## Structure

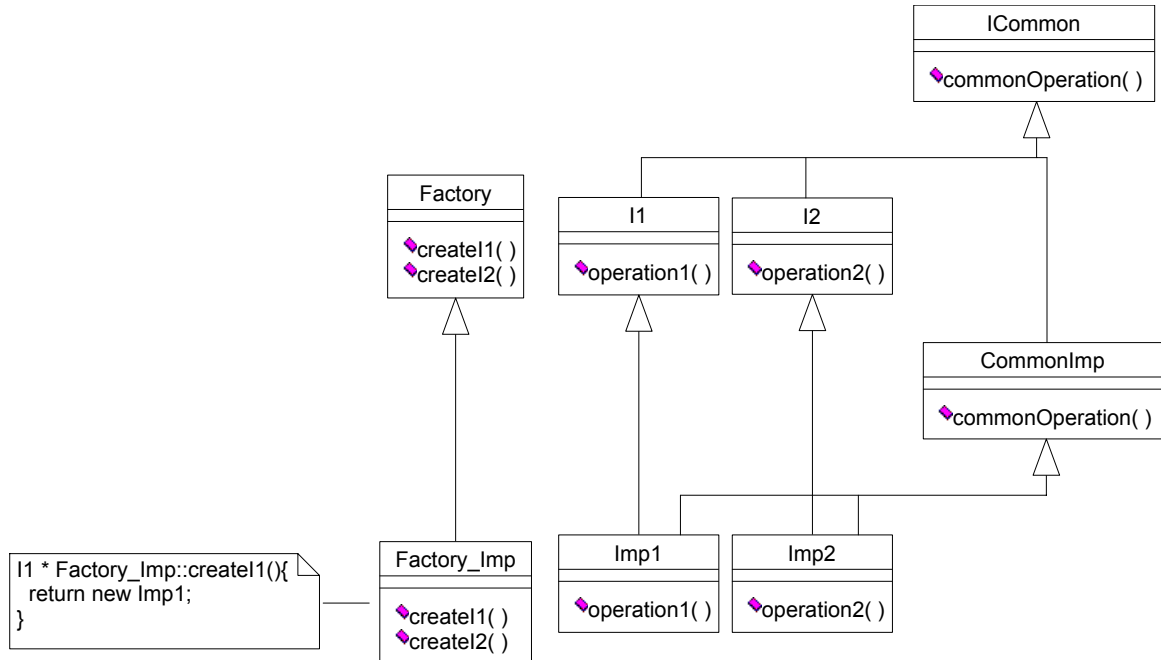


Figure 7 Complex Interface Pattern

The Complex Interface Pattern introduces the following participants:

### **Factory**

Implements the operations to create the concrete classes Impx.

### **Interface (classes Ix and ICommon)**

Provides the interfaces to the superior layer or client which intends to use the interface

### **Implementation (classes Impx and ImpCommon)**

Implements the interfaces and implements the common behaviour of the interfaces in the class 'ImpCommon', which can be deployed as a component.

## Dynamics

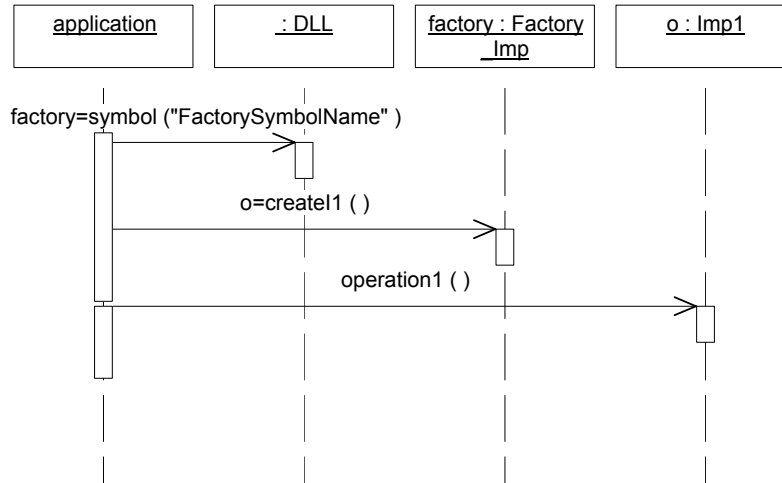


Figure 8 How to use the Complex Interface

The factory class is the access point for a client to access the interface. A client must first create an instance of the factory class to be able to create concrete objects with the behaviour of the defined interface classes.

To use the complex interface deployed as a component, requires to load the component, which is best deployed as a dynamic library. To load the component it is best practice to use the Wrapper Façade pattern [POSA2] and encapsulate the loading in the class 'DLL', which varies on every operating system. See also the Component Configurator pattern [POSA2], how to implement such a class on a Unix operating system. In my implementation I use just a simple method '*symbol(string symbol)*' which does loading as also finding the symbol in the same method.

The application has to look for a symbol using the '*symbol(string symbol)*'-method of the 'DLL' class. This symbol identifies the factory. If the library is successfully loaded the method returns a pointer to the factory object, upon which the application can create and use objects of the complex interface.

## Implementation

To implement the complex interface pattern, apply the following steps:

- 1) Identify the different objects you need to pass to the interface.
- 2) Create for every object an appropriate interface class as also an implementation class, where the interface is defined in the interface class and the behaviour is implemented in the implementation class.
- 3) For each implementation class provide a create method in the factory class.
- 4) If the interface classes provide a common interface, define this common interface in an interface base class and implement the common behaviour in the a implementation base class.
- 5) Derive the implementation base class from the interface base class.

## Variants

Concurrent access: If the interface has to allow concurrent access, meaning that different threads can access the interface, this is not a problem as long you do not have any state in the factory class.

Version management: The factory class could define a static read-only version attribute, which can be retrieved by the application, so that an application can decide at run-time if it wants to use newer functionality.

## Example Resolved

The Complex Interface Pattern was used for the implementation of a generic SNMP (Simple network management protocol) library, which can be used from any application, who wants to notify alarms or presents its states to a standard SNMP manager. SNMP has been defined in numerous RFCs using ASN1-Notation for defining and using Ber-encoding to transport messages. The goal of this library is to provide an interface for all applications in order that these applications can use a simple interface and that they don't have to deal with the cumbersome SNMP standard. Here a small SNMP overview is given, which of course is not complete:

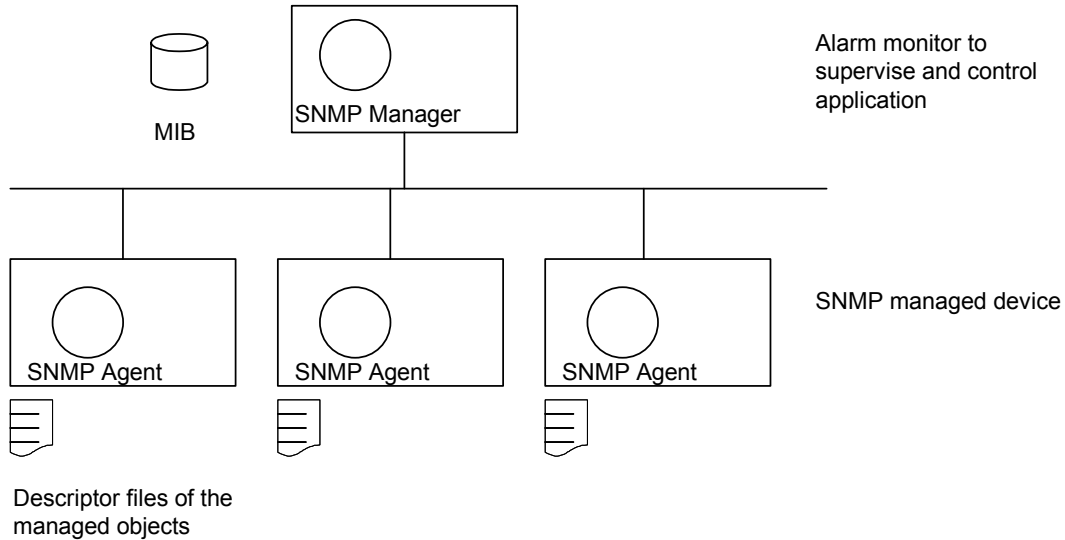


Figure 9 Overview of SNMP

SNMP is a network management protocol, which allows to monitor any network applications in a generic way from a standard SNMP Manager. In figure 8 all SNMP managed devices have normally an SNMP agent, which either responds to request (GetRequest) of the SNMP Manager or which can issue spontaneously trap messages to notify exceptional behaviour of the managed device, see figure 9. With SNMP it is also possible to control applications (SetRequest), which has not been addressed in this example.

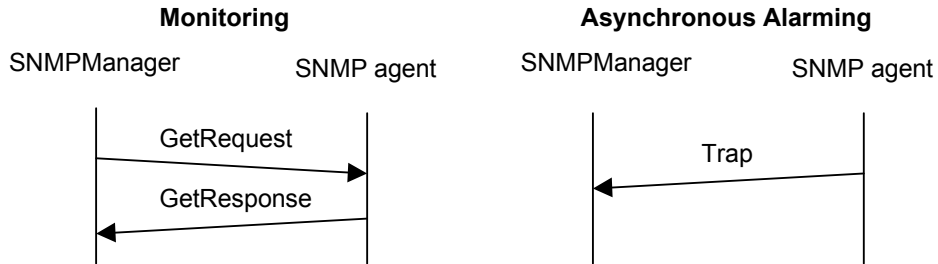


Figure 10 Monitoring and asynchronous alarming with SNMP

The information flow is performed with managed objects or so called simple objects, where the SNMP Manager can request values from simple objects or where an application can asynchronously send values of simple objects as alarms, so called traps. All managed objects are defined in descriptor files, where for every managed object a type and unique object identifier (oid) is assigned.

These descriptor must be compiled into the SNMP Management Information Base (MIB), so that the SNMP manager knows the types and object identifiers of the available simple objects. Normally you just have to copy the descriptor files (which need to follow a defined syntax when declaring managed objects) in a defined directory of the SNMP Manager.

To make it as simple as possible to use SNMP from applications running on a SNMP managed device the following architecture was chosen:

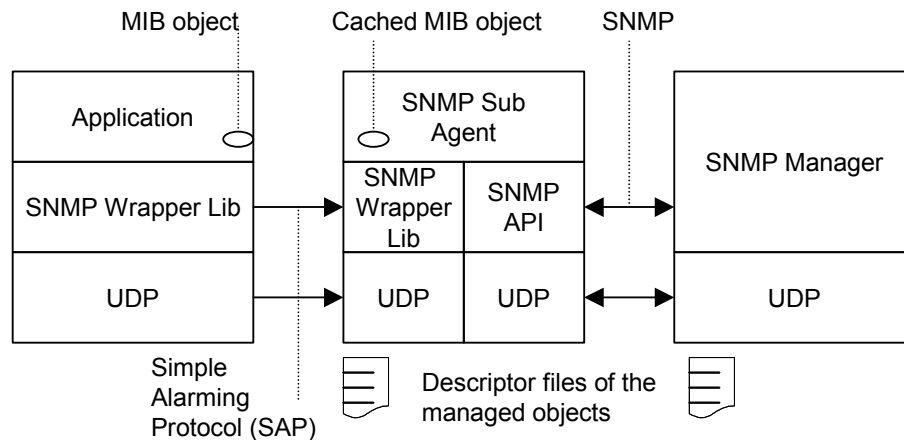


Figure 11 Encapsulating the SNMP semantics in a SNMP wrapper library and in a SNMP sub-agent adapter providing the protocol mapping from an own defined simple alarming protocol (SAP) to SNMP

All applications have to notify alarms and provide information for the SNMP manager through an easy to use SNMP wrapper library, which is using a simple alarming protocol (SAP) to a SNMP-sub-agent (SSA), which can act as either an SNMP adapter or as a sub-agent adapter to a Master-Agent/Sub-Agent system (in case there are other SNMP agent running on this system, which cannot be explained in more detail in this pattern).

The SSA defines for every MIB object an appropriate cached object, so that the application just has to update these cached objects, when they need to be updated. This simplifies the design of every application in such a manner, that the application does not have to handle request from the SNMP manager. The SSA is doing this, assuming that the cached MIB objects have always been updated appropriately from the applications. Hence the application has only to update its status to the cached objects and send traps when exceptional behaviour has to be notified to a SNMP manager.

The SNMP wrapper library is the only access point for the application. It is implementing a protocol stack which is used on the application side, where simple object can be created, encoded and send to the SNMP sub-agent and which is also used on the sub-agent side to decode the simple objects.

Finally we can summarise the SNMP wrapper library as a transparent reliable replication of simple objects from an application to the SNMP sub-agent. The SNMP wrapper library is an ideal candidate for the Complex Interface Pattern, where a family of classes with common interface and common implementation is required as you can discover on the next figure.

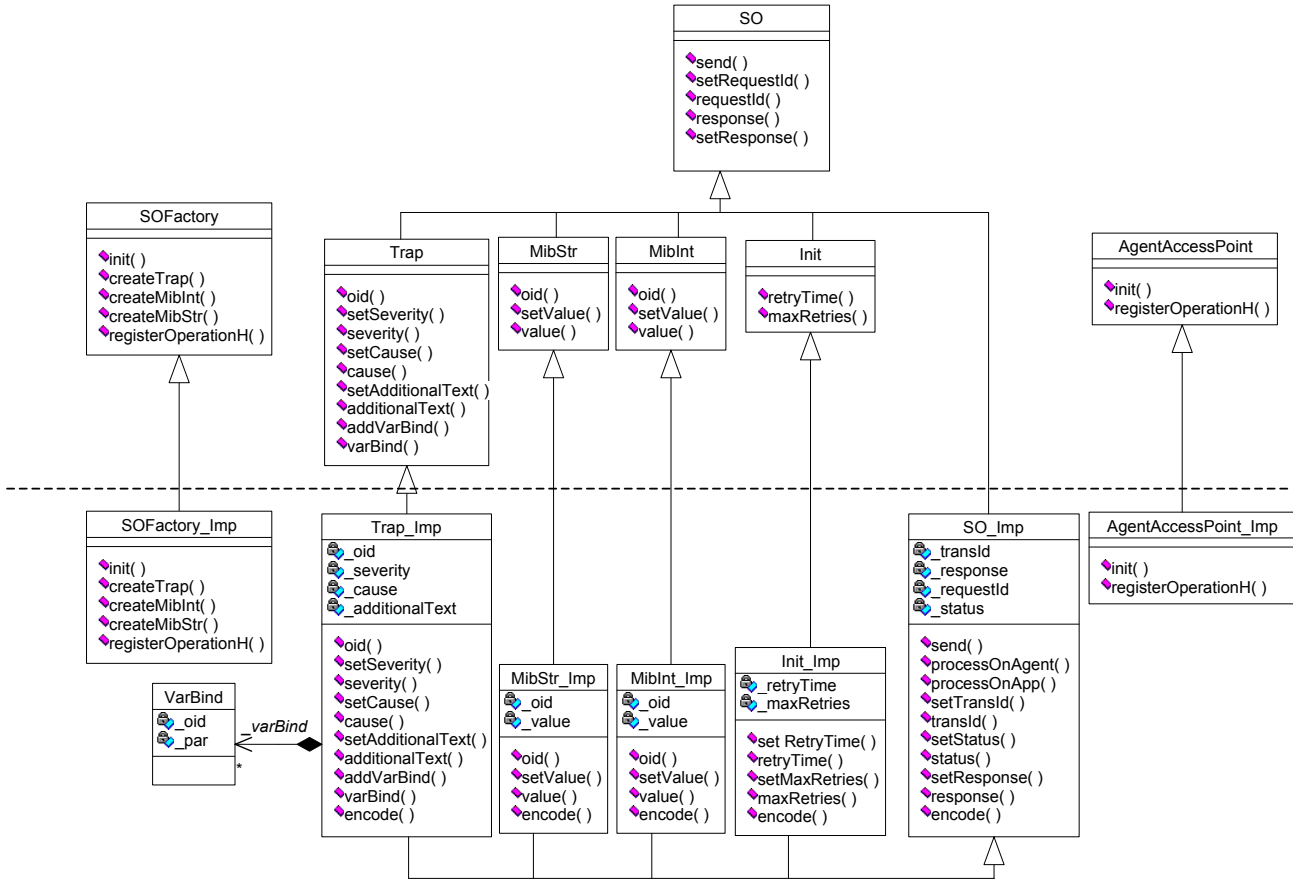


Figure 12 SNMP wrapper library using the Complex Interface Pattern

In figure 12 only the interface to the application has been drawn. All classes for protocol handling, encoding, decoding and binary transport have been omitted in this figure.

The interface to the application is provided with the classes 'Trap', 'MibStr', and 'MibInt'. With an object of type 'Trap' an application can send an alarm to the SNMP sub-agent. With an object of type 'MibStr' or 'MibInt' an application can send a new value of type 'int' or of type 'string' for a managed object, so that the SNMP sub-agent can update the cached value. The 'SOFactory' class is the access point for the application and allows to create the simple objects, while the 'AgentAccessPoint' class is the access point for the sub-agent, where it can register an operation handler, which will be called from the underlying decoding layer, when a simple object is received.

Because we have chosen to use UDP as transport layer, which is not a reliable transfer protocol, the reliable transfer of simple objects has to be handled by the application. Therefore the application can also register an operation handler, which will be called back after each simple object has been delivered successfully or without success to the sub-agent. This operation handler will receive the simple object with an aggregated response, which can be retrieved with the access method 'response()' from every simple object. Every simple object can also set a request-id before it is send to the sub-agent, to be able to track the correct delivery of a sequence of simple objects. Sending as also the handling of the response and the request-id is common for all simple objects so that the interface can be defined in the common base class 'SO' while the common implementation is programmed in 'SO\_Imp'.

Not common access methods of simple objects are declared in the interface class and implemented in the concrete class as for example the setting of the severity of a trap.

Encapsulated common behaviour for every simple object can be programmed in 'SO\_Imp' and is therefore invisible from the application. E.g. for every concrete simple object we use a transaction-id on the protocol layer (not shown here in this diagram), to track open transactions, which are sent objects from which no response from the sub-agent have arrived yet and which will be retransmitted after a defined time-out.

Last you can hide behaviour from the application in defining operations only in the concrete classes like the encoding of the simple objects, which will call the appropriate encoding interface of the marshalling layer.

## **See also**

Abstract Factory Pattern [GHJV95]

Layer Pattern [POSA1]

Component Configurator [POSA2]

## **References**

- [GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns Elements and Reuse of Object-Oriented Design, Addison Wesley, 1995
- [Per96] D. Perkins, E. McGinnis, Understanding SNMP MIBs, Prentice Hall, 1996
- [POSA1] F. Bushmann, R. Meunier, H. Rohnert, P. Sommerlad and Michael Stal, Pattern-Oriented Software Architecture - A System of Patterns, John Wiley & Sons, 1996
- [POSA2] D. Schmidt, M. Stal, H. Rohnert and F. Bushmann, Pattern-Oriented Software Architecture – Volume 2 – Patterns for Concurrent and Networked Objects, John Wiley & Sons, 2000
- [Ste98] W.R. Stevens: Unix Network Programming, Volume 1: Networking APIs: Sockets and XTI, 2<sup>nd</sup> edition, Prentice Hall 1998
- [X.733] CCITT X.733 Information Technology – Open Systems Interconnection – System Management: Alarm Reporting Function