In *Proc. 17th International Conference on Software and Systems Engineering and their Applications (ICSSEA 2004)*
Paris, France, November 2004

Patterns in SNMP-Based Network Management

Paul E. Sevinç
ETH Zurich, Switzerland
E-mail: paul.sevinc@inf.ethz.ch

Jean-Philippe Martin-Flatin
University of Quebec in Montreal, Canada
E-mail: jp.martin-flatin@ieee.org

Rachid Guerraoui
EPFL, Switzerland
E-mail: rachid.guerraoui@epfl.ch

Abstract: *The current Internet management architecture (SNMP) focuses on network device management and low-level instrumentation data. A lot of activity is under way to replace or complement it with a solution covering enterprise management at large, which also includes the management of systems, applications, and services. In this exercise, the management community runs the risk of throwing the baby out with the bath water, as too much emphasis is put on a few well-known problems in SNMP (e.g., its poor scalability), and too little on its other characteristics, including those that contributed to its success. One way to avoid this is to explicitly capture the experience gained in SNMP-based network management. In this paper, we make one step in this direction by studying the SNMP management architecture from a software engineering standpoint, and identifying in it some of the architectural and design patterns defined in the literature. By characterizing Internet network management in the lingua franca of patterns, we strive to help retain the strengths of SNMP in future management architectures and make it easier for new software engineers to move to Internet management.*

Keywords*: Patterns, Network Management, Internet, SNMP.*

1. INTRODUCTION

The management of IP networks (i.e., networks transporting traffic primarily based on the Internet Protocol) has been dominated for a decade by an open management architecture named after its communication protocol: the Simple Network Management Protocol (SNMP) [17, 20]. SNMP is based on the client-server architecture and the manager-agent paradigm. A manager is an application that regularly polls data from agents embedded in managed devices, to work out the states of these devices, detect faults, analyze performance, etc. Some SNMP agents can spontaneously send events (called *traps* or *notifications*) to the manager to inform it of special conditions. In large organizations, we often have a hierarchy of managers; each of them is in charge of a management *domain* [12, 19].

Despite its large success in network management, SNMP-based management has been seriously questioned in the past few years, primarily because of its inability to deal with all aspects of enterprise management. Today, in Internet management, the goal is not so much to manage individual network devices such as routers or switches, but rather to integrate the management of network devices, systems, end-to-end networks (e.g., with quality-of-service guarantees), distributed applications, and services (e.g., in e-business and telecommunications). To this end, many new approaches have been proposed [12]. To date, the most serious alternative to SNMP is Web-Based Enterprise Management (WBEM), a management architecture promoted by a large industrial consortium and well known for its Common Information Model (CIM) [3].

Although technically different, most of these approaches share a common standpoint: there is nothing special in Internet management that requires management applications to be designed and built with domain-specific technologies and tools—a characteristic of SNMP today, which has a domain-specific transfer protocol, its own way of representing management data, a peculiar way of devising information models, etc. Most alternatives to SNMP contend that management applications are just regular distributed applications, for which the software engineering and distributed systems communities have plenty of solutions and tools on offer. For instance, some people now use Web Services or CORBA for manager-to-agent and manager-to-manager communication; others leverage standard object-oriented modeling techniques such as UML [12].

Before discarding SNMP and specifying a new management architecture, it is important to learn lessons from it, identify its strengths and weaknesses, and characterize it in terms that are meaningful to the entire software engineering community (particularly newcomers to Internet management). We have experienced that patterns [6] are just the right tool for achieving these goals. Patterns are schematic, proven solutions to recurring problems. They enable software engineers to capture and pass on software-development experience without the need for code, and currently constitute one of the best tools for design reusability. And in the end, they allow for a better design [5]:

> *"Ideally, in real life, we should go through an analysis-design-implementation-use cycle, learn from our mistakes, and then do it right: redesign properly and reimplement. Patterns help design properly in the first place."*

Architectural patterns are coarser grained than design patterns. In general, design patterns are object oriented and describe proven solutions to recurring design problems at the class or object level. Architectural patterns are not paradigm specific; they capture proven solutions to recurring composition problems of software entities. These entities can range from groups of modules or packages to single procedures or functions—the former being more typical than the latter.

Our approach was to study the SNMP management architecture and protocol from a software engineering perspective, based on three pattern compendia: Gamma *et al.* [6], Buschmann *et al.* [4], and Schmidt *et al.* [18] and one antipattern compendium: Brown *et al.* [2]. We limited our scope to these compendia because they are well known to software engineers, and because a comprehensive study of the literature has become prohibitive (see the huge number of patterns listed by Rising [16]). Only a selection of the patterns that we identified are presented here. Note that we generalized some of the original patterns because a number of implementations of SNMP-compliant managers and agents are not object oriented, and neither the SNMP management architecture nor the SNMP protocol are.

The rest of this paper is organized as follows. In the next 8 sections, we study one pattern at a time, presenting its main occurrences in SNMP-based network management. In Section 10, we investigate related work. Last, in Section 11, we make concluding remarks and give directions for future work.

2. FACADE AND WRAPPER FACADE

The *Facade* pattern [6] provides a unified interface to a set of interfaces in a subsystem. An example is depicted in Fig. 1.

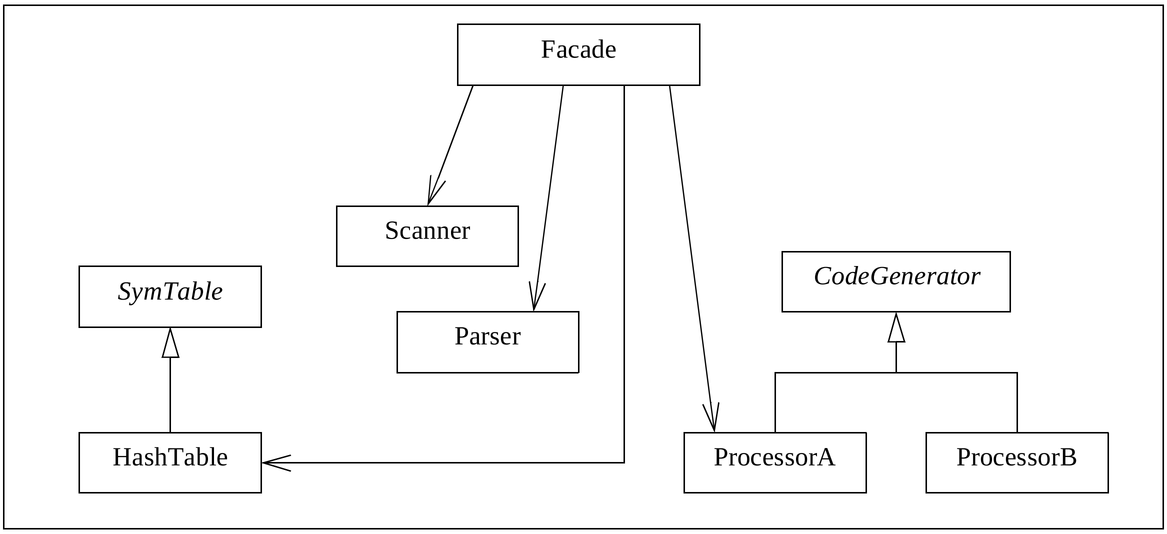

Fig. 1: Facade (adapted from [6])

A `Facade` class can shield the client of a subsystem from the subsystem's internals. As long as the `Facade` interface remains stable, the subsystem can be reorganized without breaking its clients. Another use of a `Facade` class is to offer a less complex, but also less powerful, interface as an alternative to working directly with the constituent classes. Consider for example a development subsystem consisting of scanners, parsers, code generators, etc. Many of its clients probably only want to translate from high-level language X to machine code Y. The `Facade` class can offer a method `compileFromXtoY()` that accepts a handle to the source code, takes care of all intermediate compilation steps, and returns a handle to the binary.

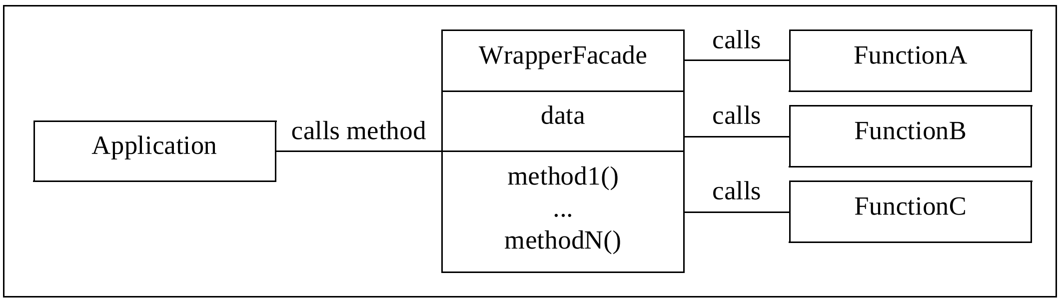

Fig. 2: Wrapper Facade [18]

The *Wrapper Facade* design pattern [18] provides concise, robust, portable, maintainable, and cohesive class interfaces (note the plural) that encapsulate low-level functions and data structures. A `WrapperFacade` class is typically intended to provide an object-oriented interface to a subsystem that is not object oriented (see Fig. 2).

In the context of Internet management, an occurrence of the *Wrapper Facade* is the interface between an object-oriented manager and a procedural application layer. For efficiency or legacy reasons, many protocols of the TCP/IP stack[1] are implemented in C. And even though Java (through the Java Native Interface) allows a programmer to directly invoke C functions, an SNMP manager should refrain from doing so. Instead, for conciseness, robustness, and all the reasons mentioned in the previous paragraph, we should introduce one or several classes in order to separate the protocol from the manager.

1. The so-called *TCP/IP stack* does not only include the Internet Protocol (IP) and the Transmission Control Protocol (TCP), but also other protocols such as the User Datagram Protocol (UDP), the Internet Control Message Protocol (ICMP), the Simple Network Management Protocol (SNMP), etc.

The *Facade* and *Wrapper Facade* patterns are also found in layered architectures in which the layers feature a service access point. The SNMP layer (all the layers in the TCP/IP stack, as a matter of fact) is no exception. It has to provide its clients with a well-defined interface, regardless of how many classes, functions, etc. were used to implement it. It is the task of the `Facade` class to support this interface and shield the clients from implementation details, if the implementation of the layer is object oriented. If it is not, the *Wrapper Facade* is the pattern of choice.

3. LAYERS

The *Layers* architectural pattern [4] helps structure applications that can be broken down into groups of subtasks, whereby each group of subtasks operates at a specific level of abstraction. This pattern is depicted in Fig. 3.

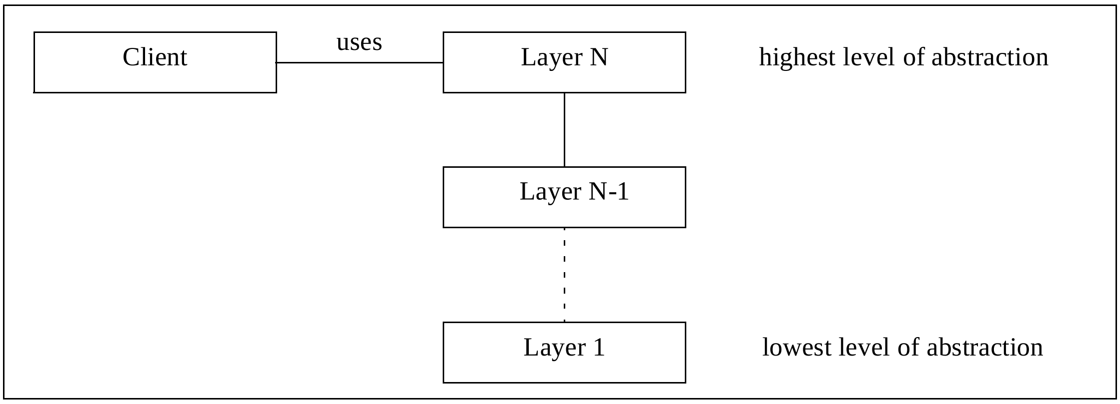

Fig. 3: Layers [4]

A well-known example of the *Layers* pattern is the Open Systems Interconnection (OSI) reference model [8], a seven-layer model defined by the International Organization for Standardization (ISO) for realizing heterogeneous networks and distributed systems. Together, the application, presentation, session, transport, network, data-link, and physical layers provide a rich set of communication facilities. Yet, each layer depends solely on the one below it and provides services only to the one above it through its service access point. The communication facilities can be changed by replacing one or more layers (e.g., a connection-oriented transport layer instead of a connectionless).

Sometimes, a layer does not provide any functionality of its own. Its sole purpose can be to abstract from lower layers, to make the entire system more stable or portable (e.g., a hardware abstraction layer). When a layer adapts the one below it, it acts as an *Adapter*[1] (see Section 4).

Note that the layers do not have to be shielded by incorporating a unified interface, as long as layer (N+1) does not depend on layer (N-1) or lower (see Fig. 4). A layer is shielded if its clients perceive it as an atomic unit; it is unshielded if its clients can see inside.

In the context of SNMP, the TCP/IP stack is an incarnation of the *Layers* pattern. SNMP is located at the application layer, where it provides services to its clients (SNMP managers and SNMP agents) and uses services provided by UDP at the transport layer. UDP uses services provided by IP at the network layer (often called *internet layer* in the IP world), etc.

Managed nodes (in particular, managed network devices) are another occurrence of the *Layers* pattern in the SNMP world. According to Rose [17], any managed node can be conceptualized as containing three components: “useful stuff”, which performs the functions desired by the user; management instrumentation, which interacts with the implementation of the managed node; and a management protocol, which permits the monitoring and control of the managed node.

1. Patterns having similar intents or structures are no exception. There are situations where multiple patterns apply, depending on the viewpoint taken. When the differences become philosophical rather than technical, they usually do not matter in practice.

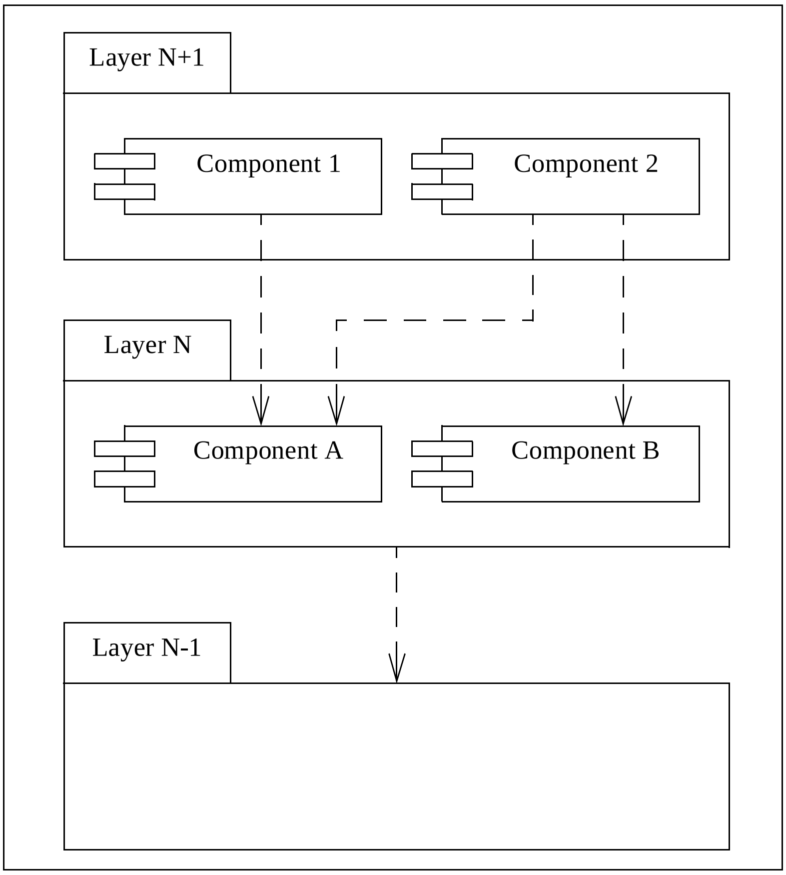

Fig. 4: Shielded and Unshielded Layers

Layers vs. Wrapper Facade

Unlike the *Wrapper Facade* pattern, which we can choose to apply or not to apply in the context of SNMP, the *Layers* pattern is implicit in SNMP. But we are still free to choose whether to apply the *Layers* pattern within the application layer, the manager, or the agent.

The *Layers* pattern does not specify what the different layers consist of, whereas the *Wrapper Facade* pattern would have no *raison d'être* without the object-oriented and procedural parts. Note also that the *Wrapper Facade* pattern can be considered a special case of the *Layers* pattern, with an intermediate layer shielding a higher, object-oriented layer from a lower, procedural layer.

4. ADAPTER

The *Adapter* pattern [6] converts the interface of a pre-existing class into another interface that the clients expect[1]. It enables the implementation (and thus the functionality) of the class to be reused, even if the interface of the class is not known by the potential clients.

For instance, a class providing encryption and decryption may feature a method with the following signature:

```
crypt(bool flag, int[] plainText, int[] cipherText)
```

whereas its clients may expect it to have methods such as:

```
encrypt(int[] plainText, int[] cipherText)
decrypt(int[] cipherText, int[] plainText)
```

1. In a strongly typed language such as Java, the *Adapter* pattern is even necessary in case the interface expected by the clients is contained in the interface of the pre-existing class, but the types differ.

Instead of reimplementing the functionality, a new class can simply forward `encrypt()` and `decrypt()` requests by invoking `crypt()`, setting the flag, and ordering the arguments accordingly. Methods returning a value also need to transform the replies whenever necessary.

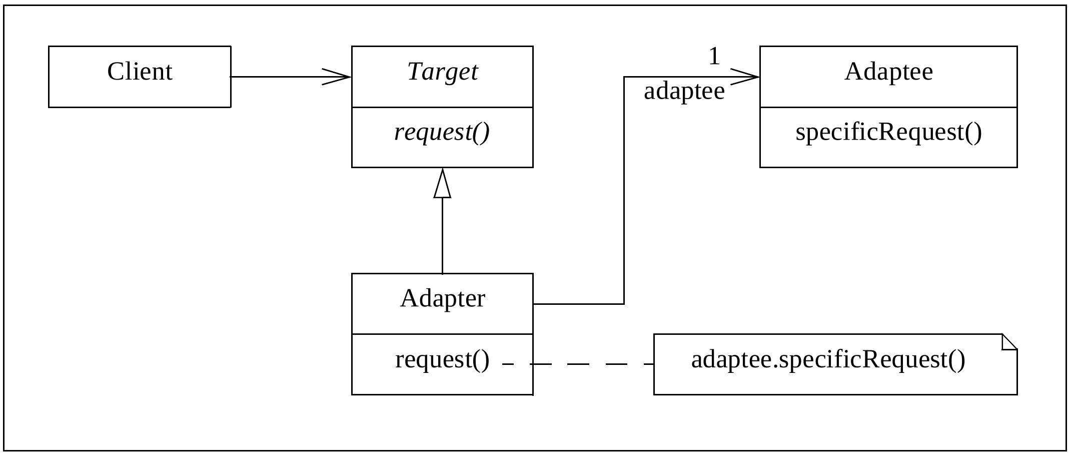

Fig. 5: Object Adapter (adapted from [6])

Gamma *et al*. [6] discuss two versions of the *Adapter* pattern: the *Object Adapter* (see Fig. 5) and the *Class Adapter* (see Fig. 6). The *Object Adapter* realizes the adaptation by using object composition; the *Class Adapter* achieves it by using multiple inheritance (e.g., in C++) or single implementation inheritance with multiple interface inheritance (e.g., in Java).

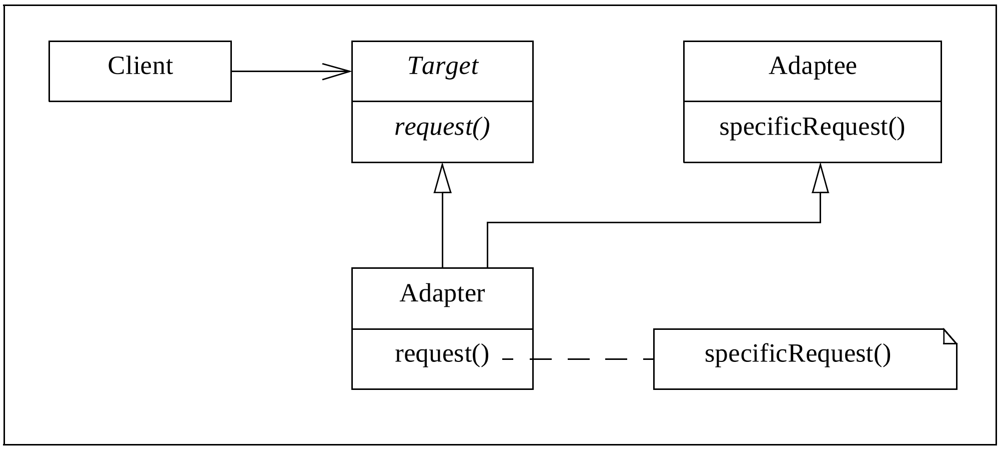

Fig. 6: Class Adapter (adapted from [6])

In SNMP-based network management, the *Object Adapter* can be found in networks with proxy agents. When a managed node hosts an agent that is not SNMP-compliant, a proxy agent needs to translate the manager's requests from SNMP to the protocol supported by the managed node, and *vice versa* for the replies. The proxy agent thus plays the role of the `Adapter` object, the manager corresponds to the `Client`, and the managed node is the `Adaptee`. As these three entities are located on different network nodes and run in different address spaces, information between them is not exchanged through direct method invocations but through the network.

When an agent issues an SNMPv1 trap or an SNMPv2 notification (different types of events in the SNMP world [20]), the proxy agent also needs to translate it before forwarding it to the manager. The proxy agent thus behaves as a two-way adapter [6], as depicted in Fig. 7.

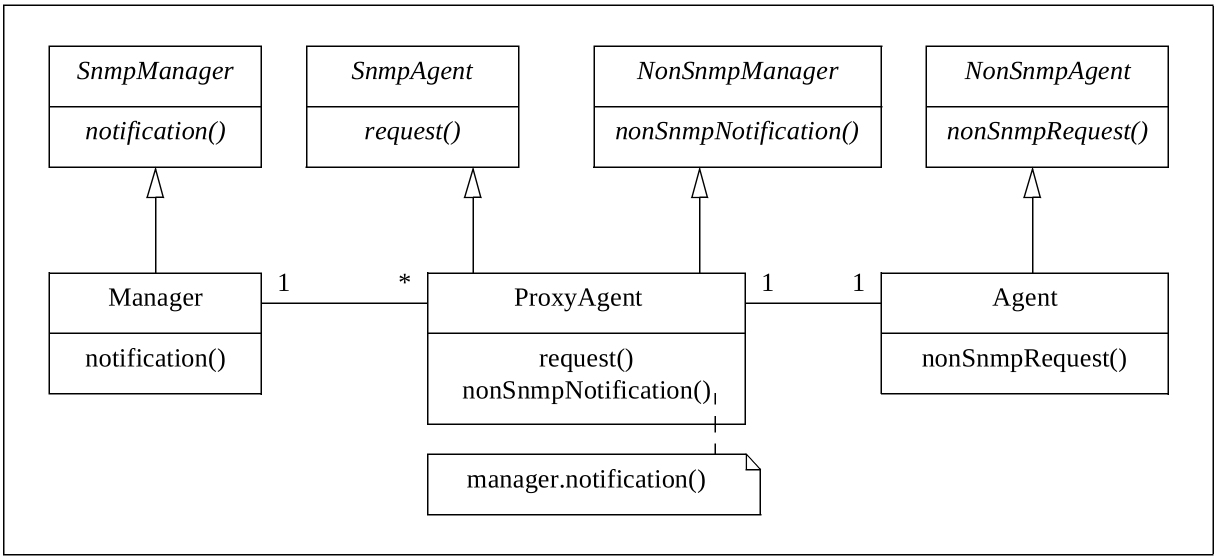

Fig. 7: SNMP Adapter

5. PROXY

The *Proxy* pattern [4, 6] makes the client of an object communicate with a representative of this object rather than the object itself. Such a representative can serve many purposes determined by its pre- and post-processing of requests. For transparency reasons, it is important that the `Proxy` and the `Original` classes have the same interface (see Fig. 8).

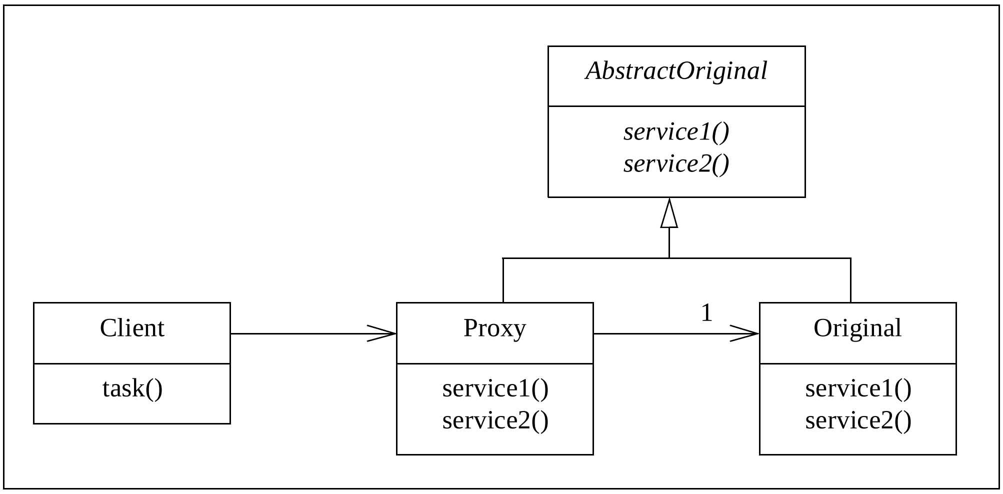

Fig. 8: Proxy (adapted from [4])

Because of its name and because it acts as an intermediary, a `Proxy` object may seem to correspond to a proxy agent in SNMP. In general, however, this is wrong! The `Proxy` class has the same interface as the `Original`, whereas a proxy agent may not have the same interface as the agent it represents.

The *Proxy* pattern can be broken down into more specific patterns, including the *Remote Proxy*, the *Virtual Proxy*, the *Protection Proxy*, the *Cache Proxy*, the *Synchronization Proxy*, the *Counting Proxy*, and the *Firewall Proxy* [4, 6]. The most relevant to SNMP-based network management are the *Protection Proxy* and the *Firewall Proxy*.

In the *Protection Proxy* pattern [4], a `Proxy` object controls access to the `Original`. It checks the access rights of a `Client` whenever a service is requested. A proxy agent can do the same for an agent that is not security aware, but is able to communicate in an SNMP-compliant way. For instance, an SNMP `set` request coming from an unauthorized manager would be discarded by the protecting agent, whereas a similar request from an authorized manager would be forwarded to the protected agent, possibly after removing the request's authentication tag.

In the *Firewall Proxy* pattern [4], a proxy process protects an internal trusted network from an external untrusted network. It represents server processes that communicate with a potentially hostile environment in order to protect against attacks—typically to avoid the disclosure of sensitive information or the misuse of network resources. In addition to supporting security management, firewalls are also relevant to SNMP-based network monitoring insofar as a managed node and its manager need not be on the same side with respect to the firewall (e.g., when managing a small branch office across a wide-area network link).

6. BRIDGE

The *Bridge* pattern [6] decouples an abstraction from its implementation so that the two can vary independently. It is depicted in Fig. 9. One of its benefits is that changes in the implementation of the abstraction have no impact on clients. The *Bridge* pattern unleashes its full power when there are several variants of the `RefinedAbstraction` and `ConcreteImp` classes.

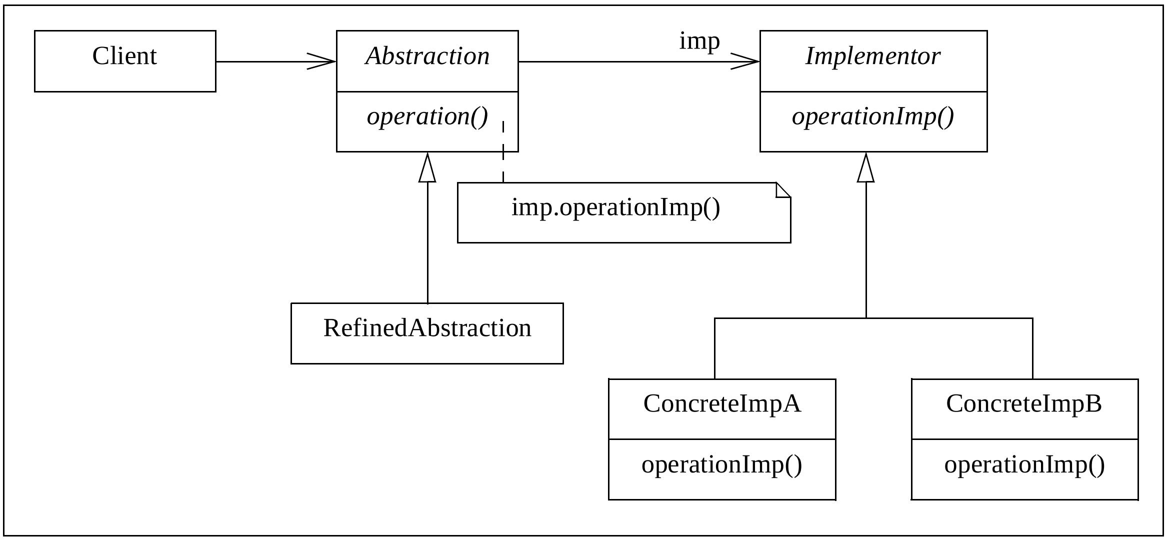

Fig. 9: Bridge (adapted from [6])

For example, let us assume that the `Abstraction` class provides the building blocks to draw different kinds of windows (document windows, dialog boxes, etc.). Every `RefinedAbstraction` corresponds to one such kind and is implemented in terms of abstraction services. A variant of `ConcreteImp` corresponds to a certain look-and-feel. By changing the `imp` reference, we can easily give a new look-and-feel to an existing kind of window. The *Bridge* pattern allows us to design only `(NumberOfRefinements + NumberOfImplementations)` classes, instead of having to design `(NumberOfRefinements * NumberOfImplementations)` classes.

By applying the *Bridge* pattern, an SNMP-based management application can use different logs (variants of `RefinedAbstraction`) without having to worry about the type of persistent storage (relational database, LDAP directory, flat file, etc.) that actually underlies their implementation. In particular, the management application becomes independent of any vendors.

Another instance of the *Bridge* pattern is the way different encryption or compression schemes can be specified in SNMPv3. The rest of the management application uses them transparently: the abstraction is completely decoupled from the implementation.

7. WHOLE-PART

The *Whole-Part* pattern [4] helps with the composition of objects that together form a semantic unit. A `Whole` class (see Fig. 10) encapsulates its constituent `Part` classes, organizes their collaboration, and provides a common interface to its functionality. The `Whole` class prevents the `Client` classes from accessing its constituent `Part` classes directly.

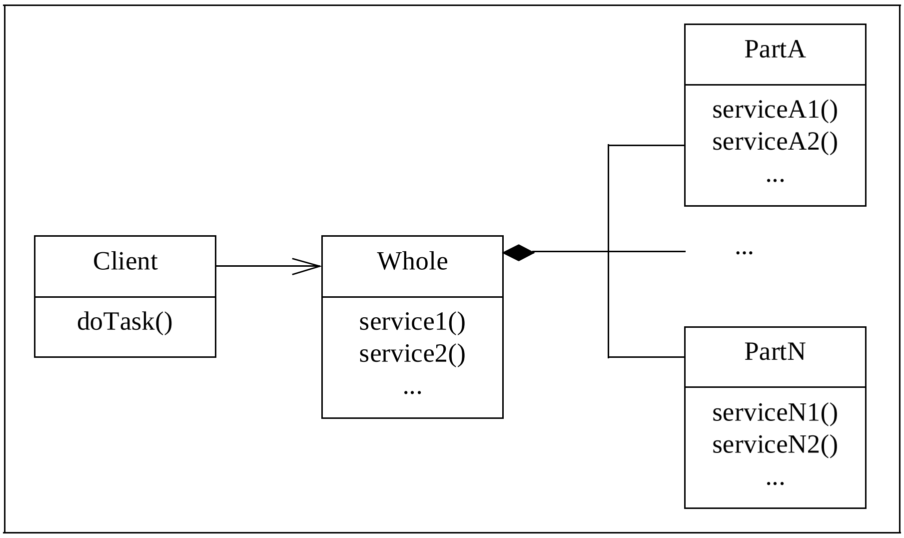

Fig. 10: Whole-Part (adapted from [4])

In addition to simply managing homogeneous or heterogeneous `Part` classes, the `Whole` class may exhibit different behaviors depending on its `Part` classes (e.g., a molecule consisting of atoms in a simulation program). Buschmann *et al.* [4] call this *emergent behavior*.

In SNMP-based management, the *Whole-Part* pattern shows up in distributed management. SNMPv1 only works in centralized mode and SNMPv2's support for distributed management is broken [12]; but SNMPv3 does allow for hierarchically distributed management (although its use is marginal in practice). The idea is to divide networks into multiple management domains—typically geographical domains (see Fig. 11). Each domain is managed by a mid-level manager, and the top-level manager is in charge of managing the boundary effects between these domains [12]. Conceptually, the `Client` in the *Whole-Part* pattern corresponds to the top-level manager in SNMPv3, the `Whole` to the mid-level manager, and each `Part` to an agent.

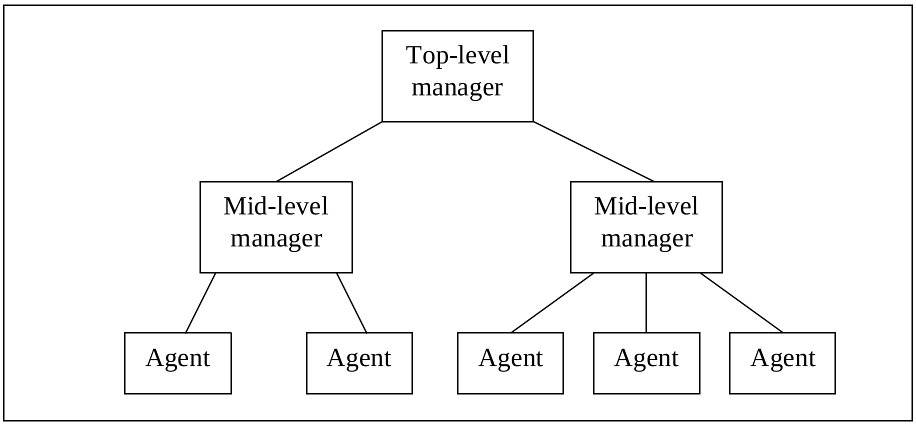

Fig. 11: Hierarchical Network Management

In practice, distributed network management today usually relies on proprietary schemes, because the semantics of manager-to-manager interactions are still not fully specified in SNMPv3. This may change in the future.

8. ITERATOR

The *Iterator* pattern [6] provides a way to access sequentially the elements of an aggregate object without exposing its underlying structure. This technique is depicted in Fig. 12.

Containers such as lists and trees often need to be traversed. By making an `Iterator` object responsible for access and traversal of the container, different kinds of traversal (e.g., forward and backward) can be supported without clogging up the container's interface, and several traversals can be pending on the same container (one traversal per iterator). Furthermore, by defining interfaces common to all containers and iterators, the dynamic type of the container can easily be changed at a later time and methods need not depend on it.

SNMP managers can iterate over agent MIBs (using `get-next` or `get-bulk` operations) to perform an SNMP-walk (i.e., to retrieve an entire MIB subtree), or to discover all the interfaces of a managed node (the `Interfaces` subgroup in MIB-II [13]). At first sight, this may seem to have nothing to do with the *Iterator* pattern: there is no `Iterator` object between the manager and an agent MIB, and we know in advance that SNMP MIBs have a tree structure. But if we apply the *Iterator* pattern at the manager, we make the manager more reusable as it does not depend on a specific MIB structure (a desirable feature when we consider replacing SNMP with another management architecture). At least, we get a cleaner design by separating the core of the manager from the part (namely the `Iterator`) that knows how SNMP MIBs are represented. But the manager could also use MIBs that provide `Iterator` objects of their own without having to make major changes. One such change can consist in applying the *Adapter* pattern when the `Iterator` interface we designed and the one the new MIB provides do not match.

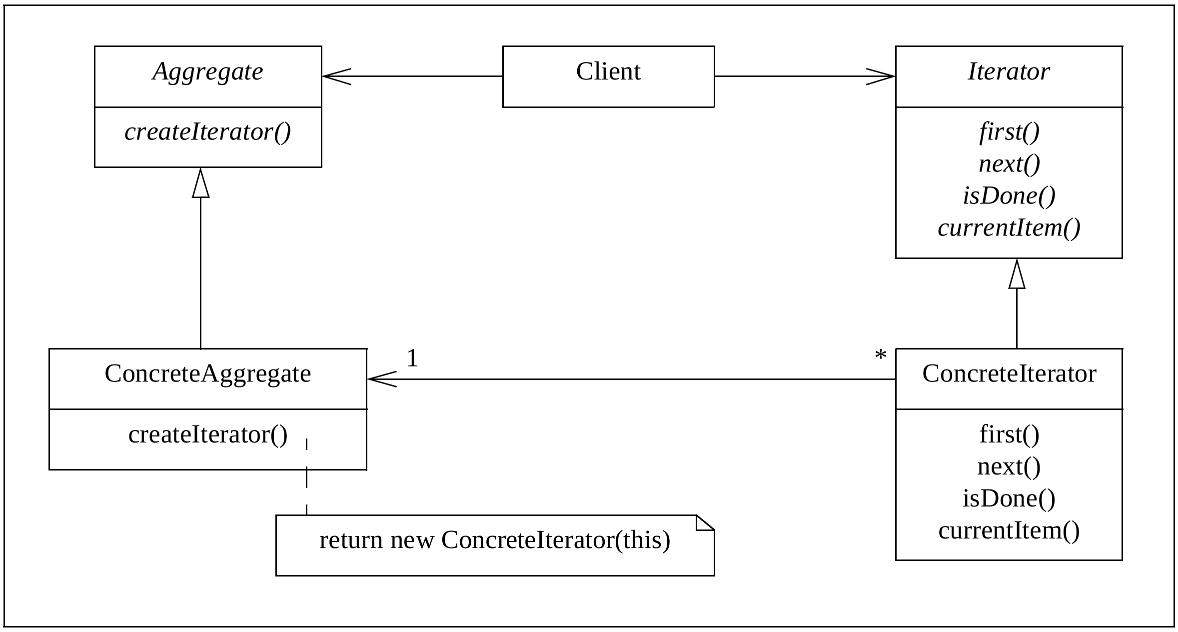

Fig. 12: Iterator (adapted from [6])

9. MEDIATOR

The *Mediator* pattern [6] promotes loose coupling by preventing objects from referring to one another explicitly. It is depicted in Fig. 13.

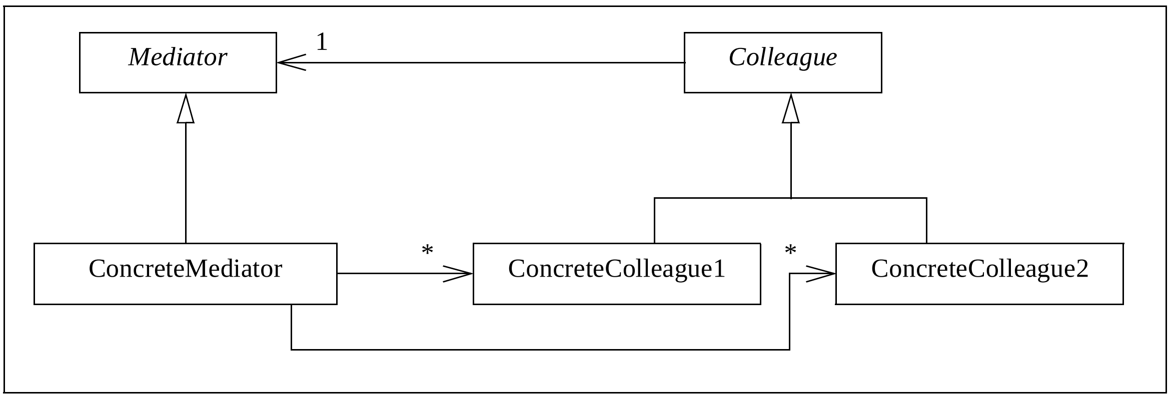

Fig. 13: Mediator (adapted from [6])

The state of an object sometimes depends on the state of other objects—e.g., GUI (Graphical User Interface) elements within a dialog box. When the state of such an object changes (e.g., when the user checks off a check box), dependent objects may have to change their states as a consequence (e.g., enabling a text field). By applying the *Mediator* pattern, `ConcreteColleague` objects (whose states depend on one another) only need to inform the `Mediator` object when their states change. The `Mediator` object then changes the states of other `ConcreteColleague` objects as needed.

In SNMP-based network management, from a conceptual viewpoint, the manager mediates between network nodes that depend on one another, as an agent may notify the manager about an event that causes the manager to change the states of other nodes. Nevertheless, some nodes can change their states in a coordinated fashion without the intervention of the manager. For example, routers exchange and update their routing tables independently of the SNMP managers.

Another application of the *Mediator* pattern in SNMP-based network management lies in the network map GUI. For instance, when an icon representing a router changes its state (typically represented by a color) to "down", the map `Mediator` object must change to "undetermined" the states of all the network nodes that can no longer be reached via this router.

10. RELATED WORK

In the past, several authors recognized existing patterns in, or defined new patterns for, networking technologies, e.g. protocols [7, 9] and telecommunication systems [1, 14]. One article defined new patterns for the use of GUIs in network management [10]. Another proposed navigation patterns for scalability [11]. But to the best of our knowledge, this article is the first to identify and document the occurrence of well-known patterns throughout the scope of SNMP-based network management.

11. CONCLUSION

By characterizing SNMP-based network management in the *lingua franca* of patterns, we have met two goals. First, patterns allow us to capture and document the best practices of SNMP-based management. By formalizing this know-how, we make it less likely that good design solutions in SNMP be replaced with poorer solutions in future management architectures. Second, patterns give software engineers a description of a domain (network management in the IP world) they may not be familiar with, in a language (patterns) they feel comfortable with. By doing so, we reduce the learning phase for software engineers moving to Internet management, and we foster reusability by considering a management application as a standard distributed application.

For future work, it would be interesting to study patterns in different areas of enterprise management (e.g., CIM-based management), compare them with those used in SNMP-based management, and learn some lessons for future management architectures. Another interesting endeavor would be to characterize the entire enterprise management domain (i.e., network, systems, application, and service management) in terms of patterns, which may require the definition of new patterns specific to this application domain.

REFERENCES

[1] M. Adams, J. Coplien, R. Gamoke, R. Hanmer, F. Keeve, and K. Nicodemus. "Fault-Tolerant Telecommunication System Patterns". In *Proc. 2nd Conference on the Pattern Languages of Programs (PLoP'95)*, Monticello, IL, USA, September 1995. Available at <http://www.bell-labs.com/user/cope/Patterns/PLoP95_telecom.html>.

[2] W.H. Brown, R.C. Malveau, H.W. McCormick III, and T.J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998.

[3] W. Bumpus, J.W. Sweitzer, P. Thompson, A.R. Westerinen, and R.C. Williams. *Common Information Model: Implementing the Object Model for Enterprise Management*. Wiley, 2000.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture—A System of Patterns*. Wiley, 1996.

[5] J. Coplien. *Organization and Architecture*. Seminar given at the CHOOSE Forum '99 on Object-Oriented Software Architecture, Annual Conference of the Swiss Group for Object-Oriented Systems and Environments, University of Bern, Switzerland, March 1999. (Approximate transcript checked with the author.)

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[7] B. Garbinato, P. Felber, and R. Guerraoui. "Strategy Pattern for Composing Reliable Distributed Protocols". In *Proc. 3rd Conference on the Pattern Languages of Programs (PLoP'96)*, Monticello, IL, USA, September 1996.

[8] H.G. Hegering, S. Abeck, and B. Neumair. *Integrated Management of Networked Systems: Concepts, Architectures, and Their Operational Application.* Morgan Kaufmann, 1999.

[9] H. Hüni, R.E. Johnson, and R. Engel. "A Framework for Network Protocol Software". In *Proc. 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, Austin, TX, USA, October 1995.

[10] R.K. Keller, J. Tessier, and G. von Bochmann. "A Pattern System for Network Management Interfaces". *Communications of the ACM*, 41(9):86–93, 1998.

[11] K.S. Lim and R. Stadler. "A Navigation Pattern for Scalable Internet Management". In *Proc. 7th IEEE/IFIP International Symposium on Integrated Network Management (IM 2001)*, Seattle, USA, May 2001, pp. 405–420.

[12] J.P. Martin-Flatin. *Web-Based Management of IP Networks and Systems*. Wiley, 2002.

[13] K. McCloghrie and M. Rose (Eds.). *RFC 1213. Management Information Base for Network Management of TCP/IP-based internets: MIB-II*. IETF, March 1991.

[14] G. Meszaros. "Design Patterns in Telecommunications System Architecture". *IEEE Communications Magazine*, 37(4):40–45, 1999.

[15] L. Rising. "Patterns: A Way to Reuse Expertise". *IEEE Communications Magazine*, 37(4):34–36, 1999.

[16] L. Rising. *The Pattern Almanac 2000*. Addison-Wesley, 2000.

[17] M.T. Rose. *The Simple Book: an Introduction to Networking Management*. Revised 2nd edition. Prentice Hall, 1996.

[18] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects. Volume 2.* Wiley, 2000.

[19] M. Sloman (Ed.). *Network and Distributed Systems Management*. Addison-Wesley, Wokingham, UK, 1994.

[20] W. Stallings. *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*. Third edition. Addison-Wesley, 1999.

BIOGRAPHIES

Paul E. Sevinç is a doctoral student and assistant in the Information Security Group of the Swiss Federal Institute of Technology Zurich (ETHZ). He received an M.Sc. degree in electrical engineering from ETHZ and spent a couple of years in industry as a software engineer before returning to his Alma Mater. His research interests include security engineering, software engineering and systems software.

Jean-Philippe Martin-Flatin is an Adjunct Professor at University of Quebec in Montreal (UQAM) in Canada. Prior to that, he worked with CERN in Switzerland, AT&T Labs Research in New Jersey and ECMWF in UK. His research interests include distributed systems management, UML modeling, Web Services, self-organized systems and software architecture. He holds a Ph.D. degree in CS from the Swiss Federal Institute of Technology Lausanne (EPFL).

Rachid Guerraoui is a Professor in Computer Science at the Swiss Federal Institute of Technology Lausanne (EPFL), where he leads the Distributed Programming Laboratory. Prior to that, he worked with the Research Center of École des Mines de Paris, Commissariat à l'Energie Atomique in Saclay, and HP Labs in Palo Alto. Rachid is interested in distributed algorithms and programming languages.