EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE LAUSANNE
POLITECNICO FEDERALE DI LOSANNA
SWISS FEDERAL INSTITUTE OF TECHNOLOGY LAUSANNE

COMMUNICATION SYSTEMS DEPARTMENT (DSC)
CH-1015 LAUSANNE, SWITZERLAND
http://dscwww.epfl.ch

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Patterns in SNMP-Based Management

Jean-Philippe Martin-Flatin, Paul E. Sevinç, and Rachid Guerraoui

September 2000

Technical Report DSC/2000/033

Patterns in SNMP-Based Management

Jean-Philippe Martin-Flatin[1], Paul E. Sevinç[1], and Rachid Guerraoui[2]

([1]) Swiss Federal Institute of Technology, Lausanne (EPFL)
Institute for computer Communications and Applications (ICA)
1015 Lausanne, Switzerland

([2]) Swiss Federal Institute of Technology, Lausanne (EPFL)
Communication Systems Department
1015 Lausanne, Switzerland

Contact person: Email: jp.martin-flatin@ieee.org Fax: +41-21-693-6610
Web: http://icawww.epfl.ch/~jpmf/

Abstract

A lot of activity is currently going on to replace the SNMP management architecture with a solution better suited to managing modern IP networks and systems. New candidates include Management by Delegation, active networks, and Web-based management. In this exercise, the management community runs the risk of throwing the baby out with the bath water by focusing too much on a few well-known problems exhibited by SNMP (e.g., its poor scalability) and neglecting most of its other characteristics, including those that contributed to its success (e.g., the reasons why it is simple). One way to avoid this is to explicitly capture the experience gained in the management of IP networks and systems with SNMP. In this paper, we make one step in this direction by studying the SNMP management architecture through a software engineer's eyes: we identify in SNMP some of the fundamental architectural and design patterns defined in the literature. Patterns are schematic, proven solutions to recurring problems. By characterizing the current management architecture in terms of patterns, we help retain the strengths of SNMP-based management in future management architectures. We also make it easier for new software engineers to move to network and systems management by characterizing this application domain in standard pattern terms, as opposed to using the jargon understood solely by the SNMP community.

Keywords: Patterns, Network Management, Systems Management, SNMP.

1. Introduction

For a decade, the management of IP networks and systems has been based on the Simple Network Management Protocol (SNMP) and variants of the SNMP management architecture (v1, v2c, and v3). Yet, because of SNMP's technical shortcomings and because of the way the SNMP market has historically evolved, SNMP-based management has recently been seriously questioned. New approaches have been proposed, including Management by Delegation, active networks, and Web-based management [7]. Although technically different, most of these approaches share a common standpoint: management applications should no longer use domain-specific technologies such as the SNMP protocol. Instead, they should be considered as standard distributed applications, for which the software-engineering community has many solutions and tools to offer.

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 comprehensible to the entire software-engineering community (particularly new engineers). Our contribution to this objective was to study SNMP-based management with a patterns perspective [4]. Why use patterns for that purpose? In our view, they are one of the best tools currently offered by the software-engineering community. In the late 1990s, they became a popular and successful means of providing reusability in software engineering. They enable software engineers to capture and pass on software-development experience without the need for code (unlike object-oriented frameworks, for instance); and consequently, they allow for a better design:

> "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 [3]."

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.

By studying the SNMP management architecture and protocol with a software engineer's perspective, we identified ten patterns in the context of SNMP: the *Facade* and the *Wrapper Facade*, the *Layers*, the *Adapter*, the *Proxy*, the *Bridge*, the *Whole-Part* and the *Composite*, the *Iterator*, and the *Mediator* [13]. To do so, we studied the architectural and design patterns published by Buschmann *et al.* [2], Gamma *et al.* [4], and Schmidt *et al.* [12]. We limited our scope to these three compendia because they are well known in the software-engineering community, particularly Gamma *et al.*, and because a comprehensive study of the literature has become prohibitive (see the huge number of patterns listed by Rising [10]). Note that we generalized some of the identified patterns, because implementations of SNMP-compliant managers and agents are generally not object-oriented, and neither the SNMP management architecture nor the SNMP protocol are.

2. Facade and Wrapper Facade

The *Facade* pattern [4] provides a unified interface to a set of interfaces in a subsystem. An example is depicted in Fig. 1.

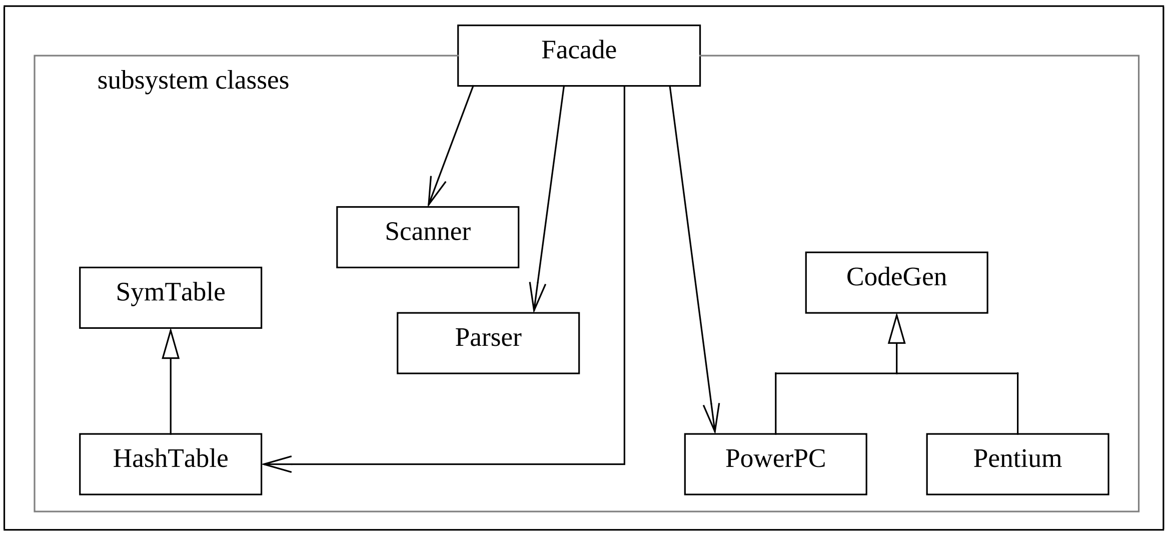

Fig. 1. Facade (adapted from [4])

A `Facade` class can shield the client of a subsystem from its 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.

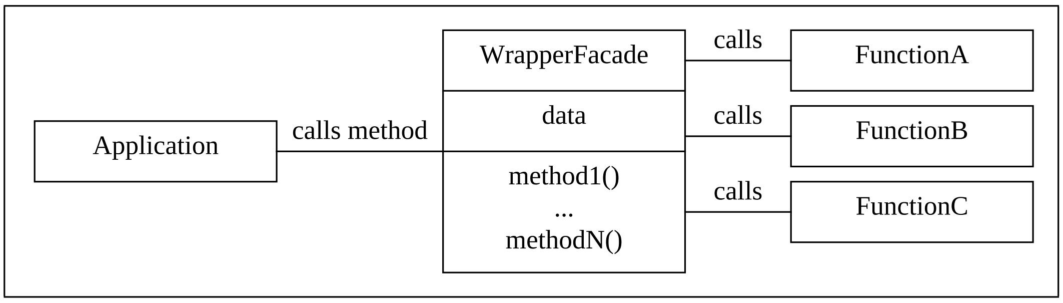

Fig. 2. Wrapper Facade [12]

The *Wrapper Facade* design pattern [12] 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 SNMP, a useful application of the *Wrapper Facade* is the interface between an object-oriented manager and a procedural application layer. For reasons of efficiency (or at least legacy), many protocols of the TCP/IP stack[1] are implemented in C. And even though Java (through the Java Native Interface) and C++ both allow a programmer to directly invoke C functions, an SNMP manager should not do so. Instead, for all the reasons given in the short definition above (conciseness, robustness, etc.), we should introduce one or several classes in order to separate the protocol from the manager.

The *Facade* and *Wrapper Facade* patterns can be very useful for 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 [2] 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.

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.

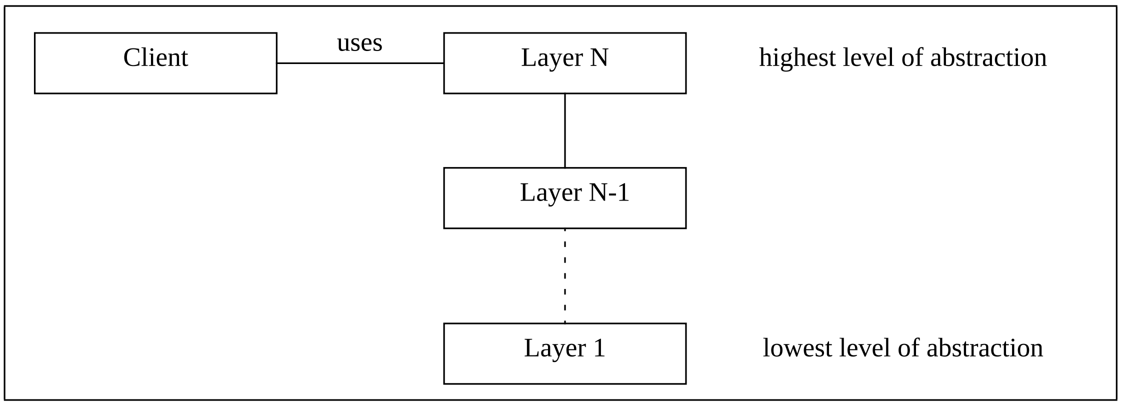

Fig. 3. Layers [2]

A well-known example of the *Layers* pattern is the Open Systems Interconnection (OSI) seven-layer model defined by the International Organization for Standardization (ISO). 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). This issue is also addressed by the *Facade* and *Wrapper Facade* patterns. Or the layer adapts the one below it, that is, it acts as an *Adapter*[1].

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 internet layer, etc.

Another occurrence of the *Layers* pattern in the SNMP world are the managed nodes. According to Rose [11], 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 not the 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.

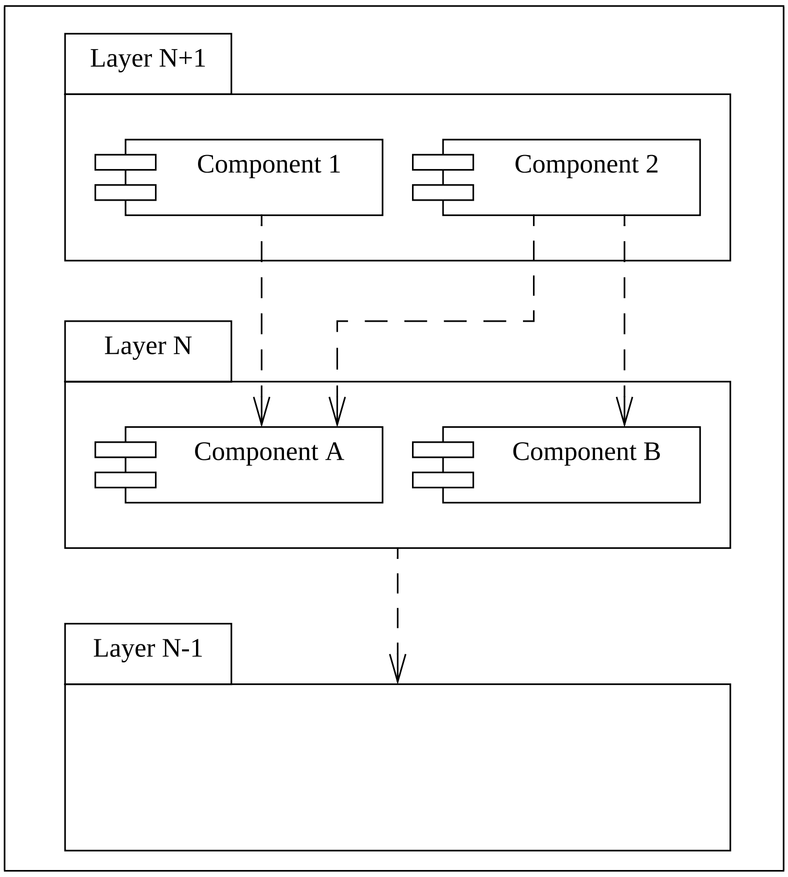

Fig. 4. Shielded and Unshielded Layers

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—although we are still free to choose whether to apply the *Layers* pattern *within* the application layer, the manager, or the agent.

Note that 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 as 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 [4] 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 client.

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

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

1. In a strongly typed language like 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.

whereas its client expects it to have methods such as:

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

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 when necessary.

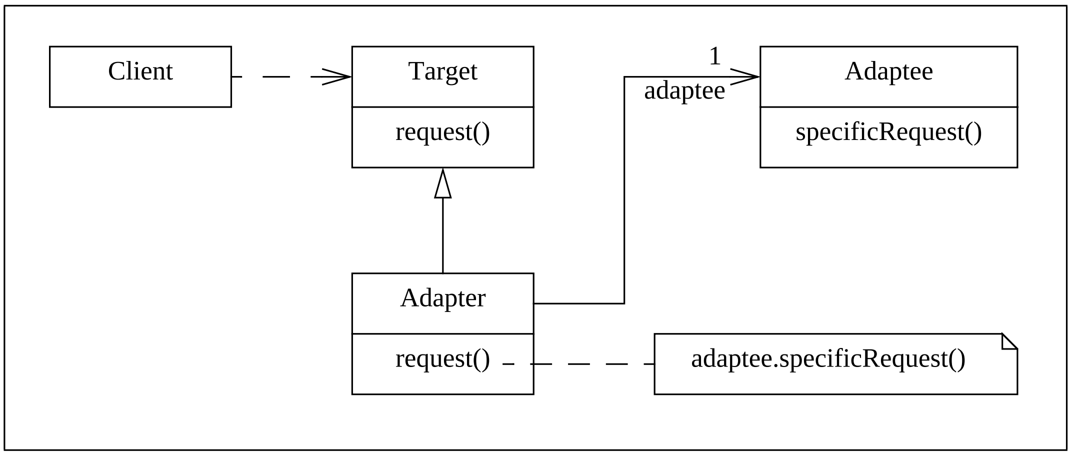

Fig. 5. Object Adapter (adapted from [4])

Gamma *et al.* [4] 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).

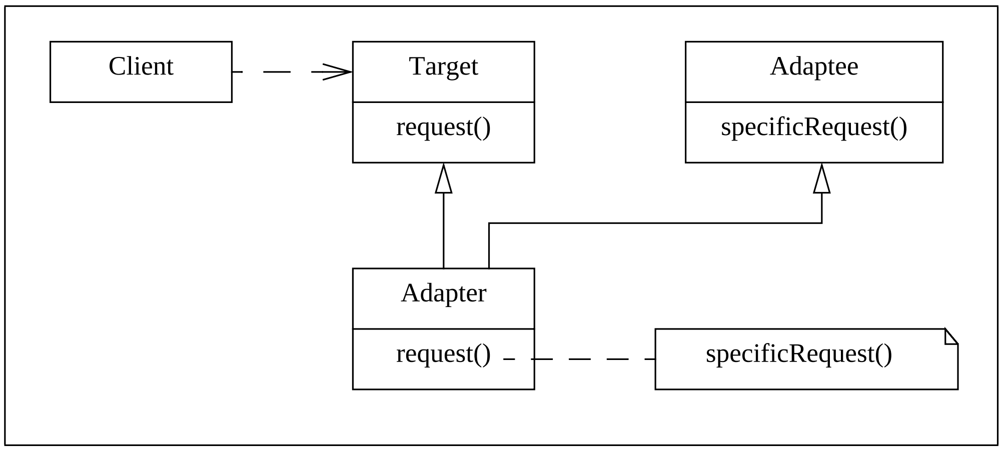

Fig. 6. Class Adapter (adapted from [4])

From a conceptual point of view, the *Object Adapter* can be seen in networks with proxy agents. When a managed node hosts an agent that is not SNMP-compliant, a proxy agent needs to translate manager requests. The proxy agent thus takes on the role of the `Adapter` object, and the manager corresponds to the `Client` and the agent to the `Adaptee`. As these three entities are located on different network nodes (not to mention in different address spaces), information between them is not exchanged through direct method invocations but through the network. Note that the *Proxy* pattern is not suitable for this particular situation (see Section 5).

When an agent issues a notification, the proxy agent also needs to translate it before forwarding it to the manager. The proxy agent thus behaves as a two-way adapter [4], as depicted in Fig. 7.

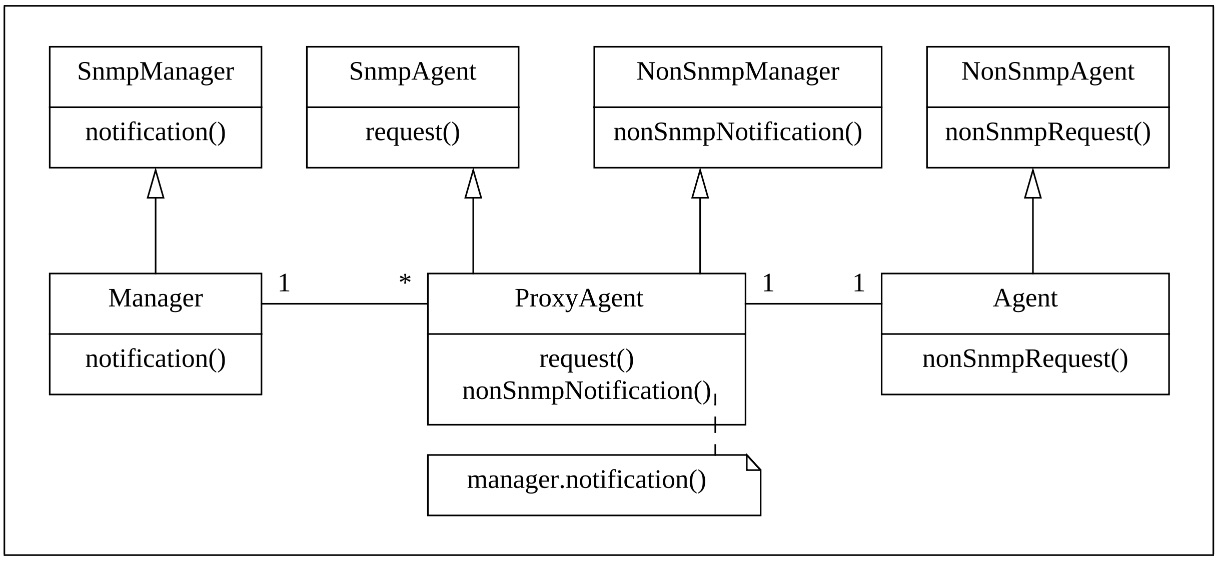

Fig. 7. SNMP Adapter

5. Proxy

The *Proxy* pattern [2, 4] makes the client of an object communicate with a representative rather than with 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).

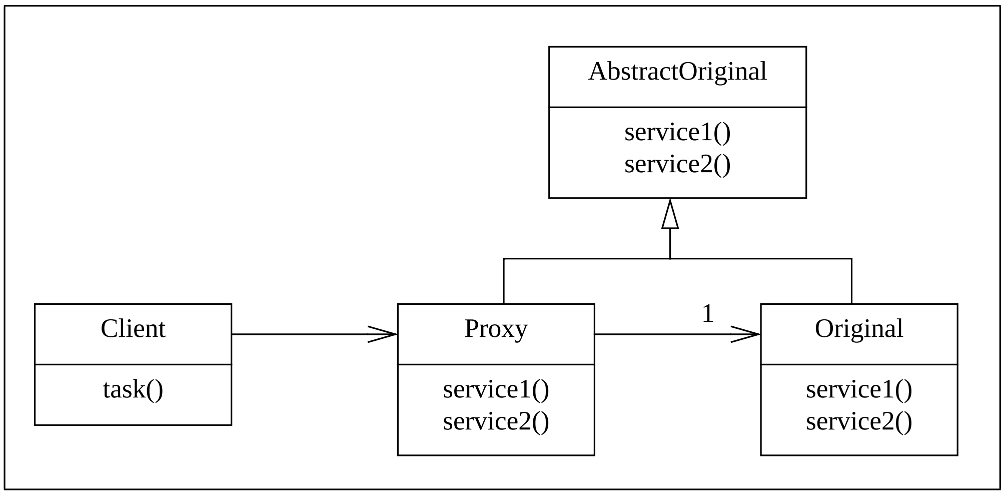

Fig. 8. Proxy (adapted from [2])

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, though, this is wrong! The `Proxy` class has the same interface as the `Original`, whereas a proxy agent and the agent it represents may not have the same interface.

Particular *Proxy* patterns are the *Remote Proxy*, the *Virtual Proxy*, the *Protection Proxy*, the *Cache Proxy*, the *Synchronization Proxy*, the *Counting Proxy*, and the *Firewall Proxy* [2, 4]. The most interesting to network and systems management are the *Protection Proxy* and the *Firewall Proxy*.

In the *Protection Proxy* pattern [2], 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 one from an authorized manager would be forwarded to the protected agent, possibly after removing the request's authentication tag.

In the *Firewall Proxy* pattern [2], 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. Firewalls are relevant to network and systems management insofar as the manager and an agent need not be on the same side with respect to the firewall (e.g., when managing a small subsidiary across a wide-area network link).

6. Bridge

The *Bridge* pattern [4] 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* unleashes its full power when there are several variants of the `RefinedAbstraction` and `ConcreteImp` classes.

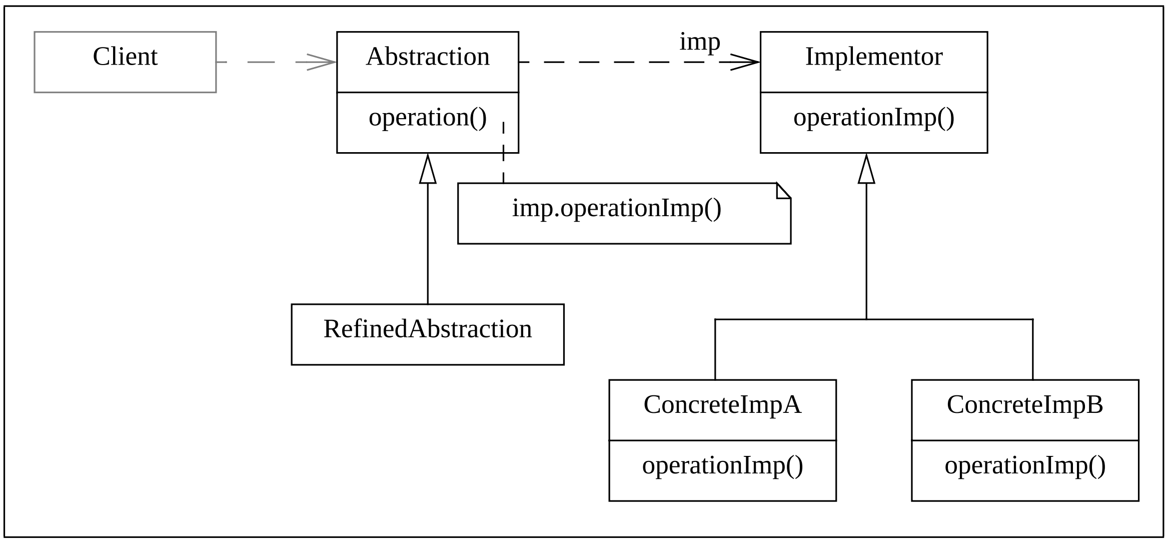

Fig. 9. Bridge (adapted from [4])

For example, let us assume that the `Abstraction` 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`'s 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* saves us from having to design `NumberOfRefinements` x `NumberOfImplementations` classes, i.e., we only have to design `NumberOfRefinements` + `NumberOfImplementations` classes.

By applying the *Bridge*, the management application can use different logs (variants of `RefinedAbstraction`) without having to worry about the type of persistent storage (database, spread-sheet, etc.) that actually underlies their implementation. In particular, the management application becomes independent of a specific vendor's database system.

7. Whole-Part and Composite

The *Whole-Part* pattern [2] helps with the composition of objects that together form a semantic unit. A `Whole` class (see Fig. 10) encapsulates its constituent `Part`'s, organizes their collaboration, and provides a common interface to its functionality. The `Whole` prevents `Client`'s from accessing these constituent `Part`'s directly.

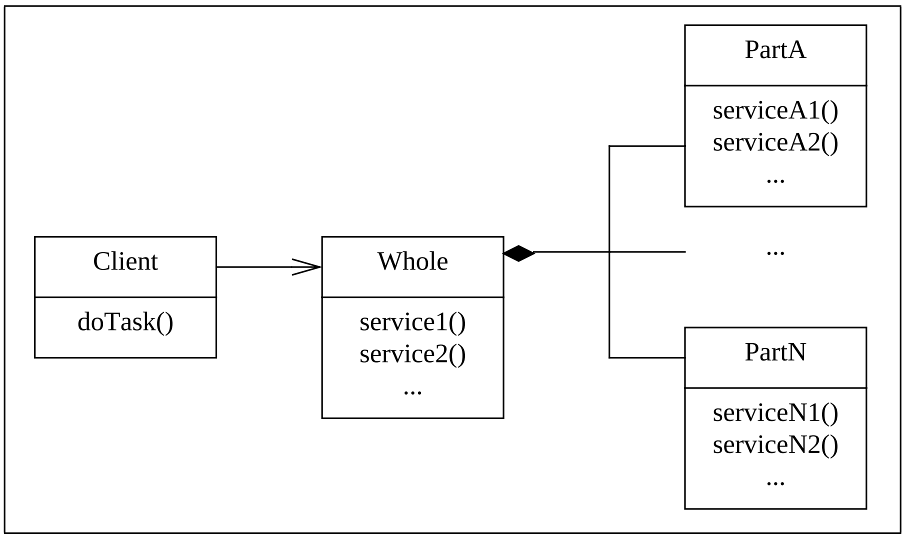

Fig. 10. Whole-Part (adapted from [2])

In addition to simply managing homogeneous or heterogeneous `Part`'s, the `Whole` class may exhibit different behaviors depending on its `Part`'s (e.g., a molecule consisting of atoms in a simulation program). Buschmann *et al.* [2] call this *emergent behavior*.

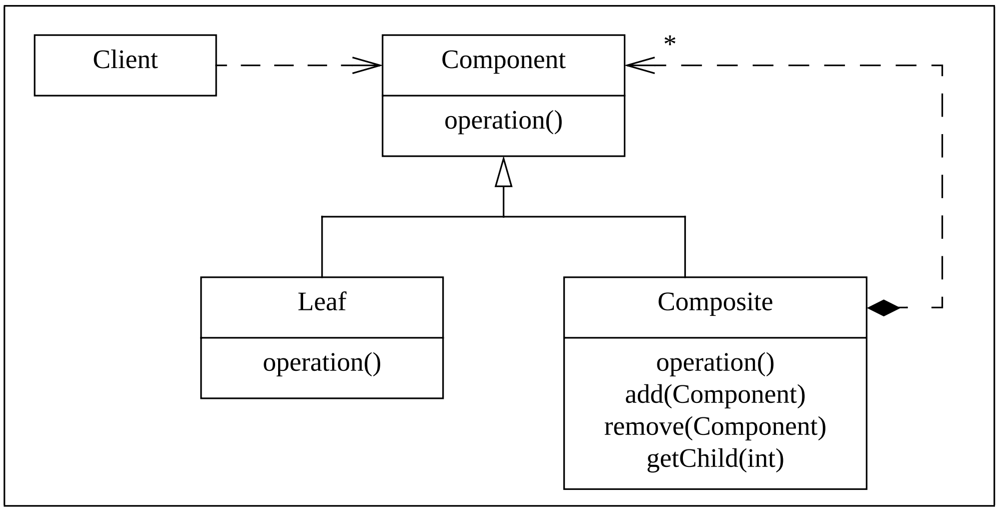

Fig. 11. Composite (adapted from [4])

The *Composite* pattern [4] (see Fig. 11) is only applicable to whole-part hierarchies in which the Whole's and the Part's can be treated uniformly. By using recursive composition, clients do not have to make a distinction between Whole's and Part's. As a matter of fact, as long as they do not compose objects themselves, they do not even have to know whether the object they interact with is a Composite or a Leaf, because they only depend on the Component interface.

SNMPv1 does not support distributed management, and SNMPv2's support was broken [7]; but SNMPv3 allows for some kind of hierarchical management [7]. The idea is to divide networks into subnetworks. Top-level managers manage these subnetworks by delegating management to mid-level managers (see Fig. 12).

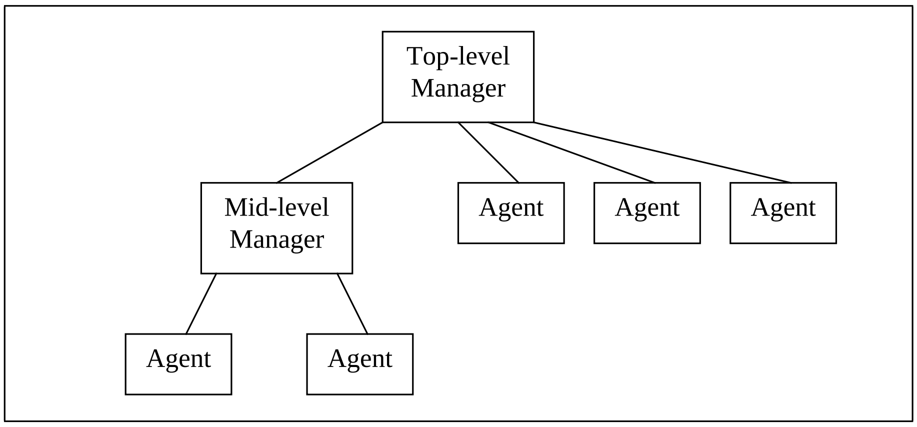

Fig. 12. Hierarchical Network and Systems Management

Conceptually, the Client in the *Whole-Part* pattern therefore corresponds to the top-level manager in SNMP, the Whole to the mid-level manager, and the Part's to the agents.

In practice, distributed network and systems management today usually relies on proprietary schemes, because manager-to-manager interactions are still not fully specified in SNMP. This may change in the future, thanks to the work of the IETF DISMAN Working Group.

8. Iterator

The *Iterator* pattern [4] provides a way to access sequentially the elements of an aggregate object without exposing its underlying structure. This technique is depicted in Fig. 13.

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) to perform an SNMP walk (that is, retrieving an entire MIB by starting at its root), or to discover all the interfaces of a node (MIB-II's Interfaces subgroup).

At first sight, this seems 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. 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`'s 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.

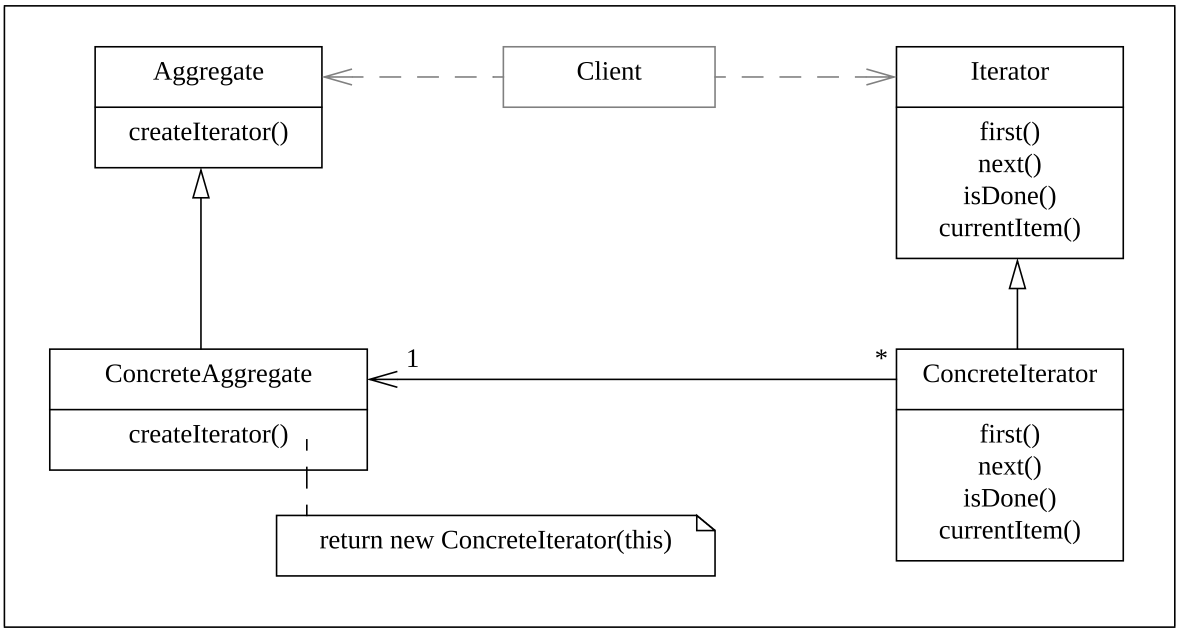

Fig. 13. Iterator (adapted from [4])

9. Mediator

The *Mediator* pattern [4] promotes loose coupling by keeping objects from referring to each other explicitly. It is depicted in Fig. 14.

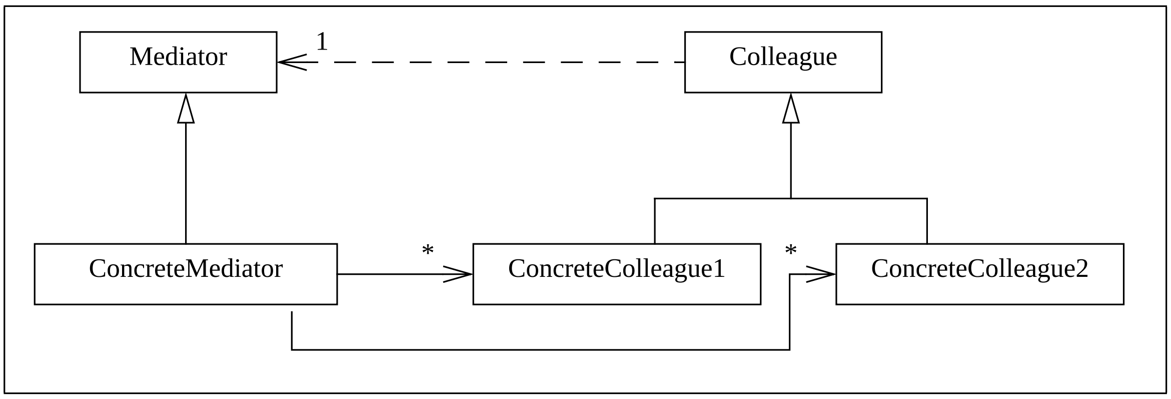

Fig. 14. Mediator (adapted from [4])

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 one such object changes state (e.g., when the user checks off a check box), dependent objects may have to change their state as a consequence (e.g., enabling a text field). By applying the *Mediator* pattern, `ConcreteColleague`'s (whose states depend on each other) only need to inform the `Mediator` object when their states change. The `Mediator` object then changes the states of other `ConcreteColleague`'s as needed.

From a conceptual viewpoint, the manager mediates between network nodes that depend on each other, as an agent may notify the manager about an event that causes the manager to change the state of other nodes. Nevertheless, note that some nodes change their state in a coordinated fashion without the intervention of the manager. For example, routers exchange and update their routing tables among themselves.

Another application of the *Mediator* pattern in SNMP-based management is within the network map GUI. For instance, when an icon representing a router changes its state (e.g., represented by a color) to “down”, the map `Mediator` object must change the state of all network nodes that can only be reached through that router to “unreachable” or “undetermined”.

10. Related Work

In the past, several papers already presented the application of patterns to networking technologies, e.g. protocols [5] and telecommunication systems [1, 8]. One defined new patterns related to the use of GUIs in network management [6]. But to the best of our knowledge, this article is the first attempt to identify patterns in SNMP-based management at large.

11. Conclusion

By characterizing SNMP-based management in terms of patterns, we met two goals. First, patterns allow us to stress and document the strengths and weaknesses of SNMP. By formalizing this know-how, we make it less likely that the same design mistakes remain in the next management architecture for the IP world, and that good design solutions in SNMP be replaced with poorer solutions. Second, patterns give software engineers a description of a domain (network and systems management) 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 network and systems management, and we favor reusability by considering a management application as a standard distributed application.

For future work, it would be interesting to study patterns in Web-based management and active networks, to compare them with those used in SNMP-based management, and to learn some lessons for future management architectures. Another direction that we intend to investigate is to define new patterns specific to integrated management—that is, the integration of network, systems, application, service, and policy management.

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

[3] 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.)

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

[5] 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.

[6] 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.

[7] J.P. Martin-Flatin. *Web-Based Management of IP Networks and Systems*. Ph.D. thesis, Swiss Federal Institute of Technology, Lausanne (EPFL), 2000.

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

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

[10] L. Rising. *The Pattern Almanac 2000*. Addison Wesley, Reading, MA, USA, 2000.

[11] M.T. Rose. *The Simple Book: an Introduction to Networking Management*. Revised 2nd edition. Prentice Hall, Upper Saddle River, NJ, USA, 1996.

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

[13] P.E. Sevinç. *Design Patterns for the Management of IP Networks*. M.S. thesis, Communication Systems Dept., EPFL, Lausanne, Switzerland, February 2000.

Biographies

J.P. Martin-Flatin will soon receive a Ph.D. degree in CS from the Swiss Federal Institute of Technology, Lausanne (EPFL). Prior to that, he worked in industry for 8 years in network and systems management, security, Web engineering, and software development.

P.E. Sevinç received an M.S. degree in EE from the Swiss Federal Institute of Technology Zurich (ETH Zurich) in 2000. Currently a freelance engineer in Switzerland, he will join Trilogy Software in January 2001.

R. Guerraoui is an assistant professor in CS at the Swiss Federal Institute of Technology, Lausanne (EPFL). In 1998-99, he worked as a faculty hire with HP Labs in Palo Alto, CA, USA. He received a Ph.D. degree in CS from Orsay, Paris, France in 1992. He was co-chair of ECOOP’99 and is the chair of Middleware 2001.