ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Design Patterns for the Management of IP Networks

M.S. Thesis - Travail de Diplôme - Diplomarbeit

Paul E. Sevinç

Department of Electrical Engineering
Swiss Federal Institute of Technology Zurich

Supervisors:
Jean-Philippe Martin-Flatin
Dr. Marcus Brunner
Prof. Dr. Rachid Guerraoui
Prof. Dr. Bernhard Plattner

February 24, 2000

Abstract

The Simple Network Management Protocol (SNMP) has been the most widely used protocol for the management of computer networks based on the Internet Protocol (IP) for about ten years. Yet, because of shortcomings of both technical and commercial nature, new approaches for the management of IP networks have been proposed. The most popular is known as Web-based management and promotes the use of Web technologies in the management of IP networks.

In this thesis, we look at SNMP-based network management from the point of view of software engineering. We also study one variant of Web-based management proposed by Martin-Flatin known as the Java Management Platform (JAMAP). Specifically, we propose design patterns for object-oriented implementations of SNMP agents, SNMP managers, and the protocol itself, and we discuss the architecture of Martin-Flatin's platform with the help of design patterns. Finally, we study a prototype of JAMAP and make some suggestions for improvement.

Résumé

SNMP (Simple Network Management Protocol) est depuis plus de dix ans le protocole le plus répandu pour la gestion de réseaux informatiques basés sur IP (Internet Protocol). Mais à cause d'insuffisances aussi bien techniques que commerciales, de nouvelles méthodes pour la gestion de réseaux IP ont été proposées ces dernières années. L'une d'entre elles, la gestion basée sur les technologies Web, connaît actuellement une grande popularité.

Dans ce travail, nous considérons du point de vue génie logiciel la gestion de réseaux basée sur SNMP et une des méthodes basées sur le Web, proposée par Martin-Flatin. Concrètement, nous proposons des patternes conceptuels qui se prêtent à l'implémentation orientée objet d'agents SNMP, de gestionnaires SNMP et du protocole lui-même. Ensuite, nous discutons l'architecture de la plate-forme de Martin-Flatin à l'aide de patternes conceptuels. Finalement, nous étudions un prototype (JAMAP) mettant en oeuvre cette architecture, et nous proposons plusieurs améliorations.

Kurzfassung

SNMP, das Simple Network Management Protocol, ist seit mehr als zehn Jahren das verbreitetste Protokoll zur Verwaltung von Computernetzwerken basierend auf IP, dem Internet Protocol. Doch aufgrund von Unzulänglichkeiten sowohl technischer als auch kommerzieller Natur wurden neue Ansätze zur Verwaltung von IP-Netzwerken vorgeschlagen, wobei sich insbesondere die Web-basierten einer grossen Popularität erfreuen.

In dieser Arbeit betrachten wir SNMP-basierte sowie Martin-Flatin's Ansatz für Web-basierte Netzwerkverwaltung aus der Sicht des Software-Engineerings. Konkret schlagen wir Entwurfsmuster für objekt-orientierte Implementationen von SNMP-Agenten, SNMP-Manager und dem Protokoll selbst vor. Des weiteren besprechen wir die Architektur von Martin-Flatin's Plattform (JAMAP) mit Hilfe von Entwurfsmustern. Und schliesslich nehmen wir uns eines Prototypen an, welcher diese Plattform realisiert und machen einige Verbesserungsvorschläge.

This M.S. thesis (*travail de diplôme*, *Diplomarbeit*) was proposed by *Jean-Philippe Martin-Flatin*, who also acted as its prime advisor. An electrical engineering student at the *Swiss Federal Institute of Technology Zurich* (ETHZ), I worked on this thesis at the computer science and the communication systems departments of the *Swiss Federal Institute of Technology Lausanne* (EPFL) in the winter semester of 1999 / 2000.

Acknowledgments

First of all, I would like to thank my supervisors *Prof. Dr. Bernhard Plattner*, *Prof. Dr. Rachid Guerraoui*, *Dr. Marcus Brunner* and especially *Jean-Philippe Martin-Flatin* for entrusting me with this project. Prof. Plattner and Dr. Brunner are from the *Computer Engineering and Networks Laboratory* (TIK) at ETHZ. Prof. Guerraoui is from the *Operating Systems Laboratory* (LSE) at EPFL, and Mr. Martin-Flatin is from the *Institute for computer Communications and Applications* (ICA) at EPFL.

Furthermore, I thank *Dirk Riehle* for his valuable pointers to pattern-related papers and for giving me insight into the patterns community, and *Luc Girardin*, *Jean-Michel Reghem* and *Dani Seelhofer* for their proof-reading and comments. Remaining mistakes are mine.

Lausanne, February 24, 2000

Paul E. Sevinç

Table of Contents

1. Introduction

1.1. Background

Within a decade, computer networks have become an integral part of everyday life. They are more and more the basis for private, academic, commercial, and governmental information exchange and service provision. Yet merely setting up a network and hoping for the best will not do. In order to continuously provide the services their users can realistically expect, computer networks need to be managed.

The difficulty of managing a network not only stems from the size of the network. Other reasons include its heterogeneity, the scalability of the management platform, and security considerations. Without adequate tools, the network management staff cannot fulfill its tasks.

The current standard for the mangement of IP networks, the *Simple Network Management Protocol* (SNMP), has serious drawbacks of both technical and commercial nature [19, 27]. Suggestions for new approaches exist and seem to favor the use of Web technologies [10, 28]. The Java platform is gaining momentum in this arena as well [49].

Just as the Internet was becoming pervasive, design patterns became an essential tool in software engineering. In addition to previously unpublished design patterns, the patterns community is very interested in variants of and in new uses for published design patterns.

In particular, there is a trend towards the development and documentation of computer-network protocols and distributed systems based on design patterns [17, 18, 24, 42] that should not be ascribed to fashion or hype. We support this trend by demonstrating that design patterns can be put to good use in network-management software.

1.2. Objective

The first goal of this thesis is to propose design patterns suitable for object-oriented implementations of SNMP-compliant network-management software, including the protocol itself. In accordance with the original project description (see Appendix A), these propositions are based on Gamma *et al.*'s *Design Patterns* [16]. Additionally, we consulted Buschmann *et al.*'s *A System of Patterns* [6] as well as a few other publications.

The second goal is to identify antipatterns based on Brown *et al.*'s *AntiPatterns* [5]. The idea is not to study actual implementations, but to find antipatterns by taking a high-level look at the models underlying SNMP-based and Web-based network management.

The third goal is to improve the architecture of JAMAP (short for JAva MAnagement Platform) by (i) documenting it in terms of design patterns, and (ii) making suggestions for changes. JAMAP is a proposal for a Web-based network management platform. A prototype has been developed in the course of an earlier M.S. thesis [4, 30]. It exchanges data based on the push model [13, 29] and makes use of Web and Java technologies.

1.3. Outline

Chapter 2, **Software Engineering**, first introduces *patterns* and *antipatterns* and presents two examples, the *Model-View-Controller* pattern and the *Blob* antipattern. It then defines the term *framework* as used in software engineering.

Chapter 3, **Network Management**, discusses the general concepts and terminology of *network management*. It also gives an overview of SNMP and of an instance of Web-based network management.

Chapter 4, **Patterns and AntiPatterns in SNMP-based Network Management**, presents ten design patterns and four antipatterns, and explains how they relate to SNMP.

Chapter 5, **A Patterns View of JAMAP**, characterizes the architecture, that is the high-level design, of JAMAP in terms of design patterns and antipatterns.

Chapter 6, **Analysis of the JAMAP Prototype**, reviews the current JAMAP prototype and makes some suggestions for improvements. Whenever possible, the suggestions are justified with the help of design patterns or antipatterns.

Chapter 7, **Conclusion**, briefly summarizes this thesis and its contributions, lists the benefits for the student, and proposes follow-up projects for the future.

2. Software Engineering

Reuse is a key objective of software engineering. Software developers learn to reuse code (i.e., implementation) from the very beginning. Beyond implementation, they can profit from the experience that other developers gained during analysis and design phases of software projects. Indeed, part of this experience can be captured and passed on by *patterns*. In object-oriented programming, the desire to have both design and code reuse leads to the notion of *frameworks*.

In the remainder of this chapter, we give definitions of pattern, antipattern, and a few other pattern-related terms in the context of software engineering. We also define the terms framework and application framework, compare frameworks to procedure and class libraries, and discuss the relationship between patterns and frameworks.

2.1. Patterns and AntiPatterns

Based on his Ph.D. thesis, Erich Gamma in 1994 published the book *Design Patterns* [16] together with Richard Helm, Ralph Johnsson and John Vlissides[1]. This publication triggered intensive research in patterns for software engineering and other computer science & engineering disciplines that continues unabated today.

Gamma *et al.* were not the first to publish software-engineering patterns, though. Ward Cunningham and Kent Beck in 1987 presented their paper *Using Pattern Languages for Object-Oriented Programs* [8] at OOPSLA '87[2].

Cunningham and Beck were inspired by Christopher Alexander. In the 1970s, Alexander developed a pattern language for architecture [1]—the "house-building" discipline, that is, not software or computer architecture!

For more on the history of patterns, see Appleton [3]. For somewhat different definitions than the following, see Riehle and Züllighoven [39].

2.1.1. Pattern

Patterns are schematic, proven solutions to recurring problems. Basically, patterns are characterized by a name, a problem description, and a problem solution [36]. A well-known *name* allows us to concisely refer to a specific pattern. It is certainly easier to talk about "the Model-View-Controller pattern" (see Section 2.1.3.) than about "the pattern that consists of three classes which...". The *problem description* tells us in what situations the respective pattern is applicable. It includes conditions that must be met before and risks taken when applying the pattern. The *problem solution* explains how we can solve the problem. It typically does so by abstracting from an example.

"schematic" refers to the fact that—especially in books—patterns are usually documented based on some template. For example, Gamma *et al.* [16] always start with the name of a pattern, followed by its intent, an alias (possibly none or more than one), a motivational example, the structure of the pattern, etc. "recurring" refers to the

1. These four authors are also know as the *Gang of Four*.
2. OOPSLA is the annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications.

condition that a pattern must have been observed more than twice, otherwise it cannot be called a pattern yet.

2.1.2. Software Patterns

If we think of software development as simply an iteration of analysis, design, and implementation, we can define *design pattern* as *a schematic, proven solution to a recurring software design problem*. Even though this very intuitive definition is the one we adopted in this thesis, we do not want to conceal that Buschmann *et al.* (and other authors) make a finer distinction [6, pp. 12-14]:

> *"An* architectural pattern *expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them."*
>
> *"A* design pattern *provides a scheme for refining the subsystem or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context [16]."*
>
> *"An* idiom *is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language."*

These *pattern categories* are orthogonal to *problem categories* (e.g., patterns for access control). Gamma *et al.* [16] and Mössenböck [36], on the other hand, subdivide patterns[3] dependent on their *purpose*[4] (e.g., such for object creation) and their *scope* (class vs. object).

Different templates exist to document software patterns [5, 6, 16, 39]. Usually, the structure of a design pattern is depicted by a complete diagram while the implementation only shows the most important interfaces and some code snippets (possibly in pseudo-code). The diagram is not necessarily a class diagram, even though the term design pattern is often implicitly associated with object-oriented programming. We do not approve of this implication. Design patterns address architectural issues independent of a specific paradigm. (The computational issues are addressed by algorithms and data structures [3, 6].)

2.1.3. Example: Model-View-Controller

Before we give more definitions, we discuss an example. In Section 2.2.2., we will consider the Model-View-Controller framework. Here, we consider the Model-View-Controller pattern.

---

3. They use the terms pattern and design pattern interchangeably. Many authors do that when their article, paper, or book is about one kind of pattern only, and so will we.
4. in principle the same as, but coarser-grained than, Buschmann *et al.*'s [6] pattern categories

The *Model-View-Controller* (MVC) is often used for graphical user interface (GUI) elements (see Section 2.2.2.) and documents. Models *represent* information, views *present* information, and controllers *interpret* user manipulation [50] (see Figure 2-1).

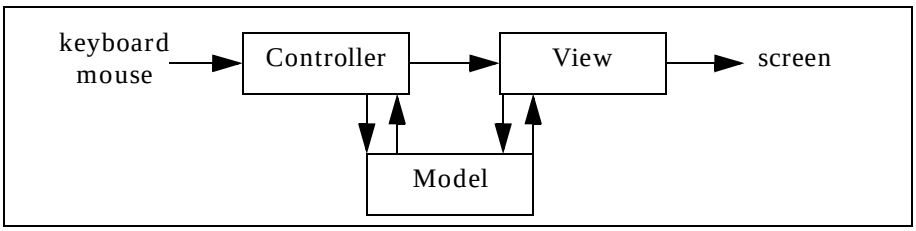

Figure 2-1: Components of the MVC [36]

Consider an HTML document. The model is the hierarchy of tags with the embedded content. If we open the HTML file in a browser, hyperlinks appear underlined (view) and can be activated (controller). If we open the same file in a text editor, we see the actual <a> tag (view), but we cannot activate the hyperlink even though we can edit it (controller).

The general *problem* is allowing several consistent views with different behavior on the same information. (In the HTML example, the views are not consistent until the user manually saves the file and reloads it.) The *solution* is to have models manage the information. Controller-view pairs subscribe to the model and are notified when it changed. For simplicity, they are even notified if they changed it themselves. How exactly they are paired and whether the controller or the view updates the model are implementational details.

As in the HTML example, one model can have more than one controller-view pair associated with it. But a controller-view pair belongs to exactly one model (see Section 5.2. for variants).

2.1.4. More on Patterns

A *pattern catalog* is a collection of patterns. The patterns in a catalog are usually documented based on the same template and organized in broad categories.

A *pattern system* is a pattern catalog in which the interrelationships between the patterns are made explicit. This goes beyond the mere mention of related patterns.

A *pattern language* is a pattern system with rules to combine the patterns into a larger whole. One can think of the patterns as the terminal symbols and of the combination rules as the syntax of the language.

2.1.5. AntiPattern

If patterns show how to do things right, then *antipatterns* show how to do them wrong. Antipatterns document common mistakes made by software engineers. Basically, antipatterns are characterized by a name, a problem description, a bad problem solution, symptoms caused by the bad solution, and a refactored problem solution (see Figure 2-2 right). The *name* and the *problem description* have the same purpose as for patterns. The *bad problem solution* may compile and run correctly, but it is not elegant, difficult to document and understand, or hard to maintain. A list of *symptoms* helps us spot an antipattern. The *refactored problem solution* shows how to

better solve the problem. Ideally, putting together the problem description and the refactored solution under a *different* name should yield a pattern.

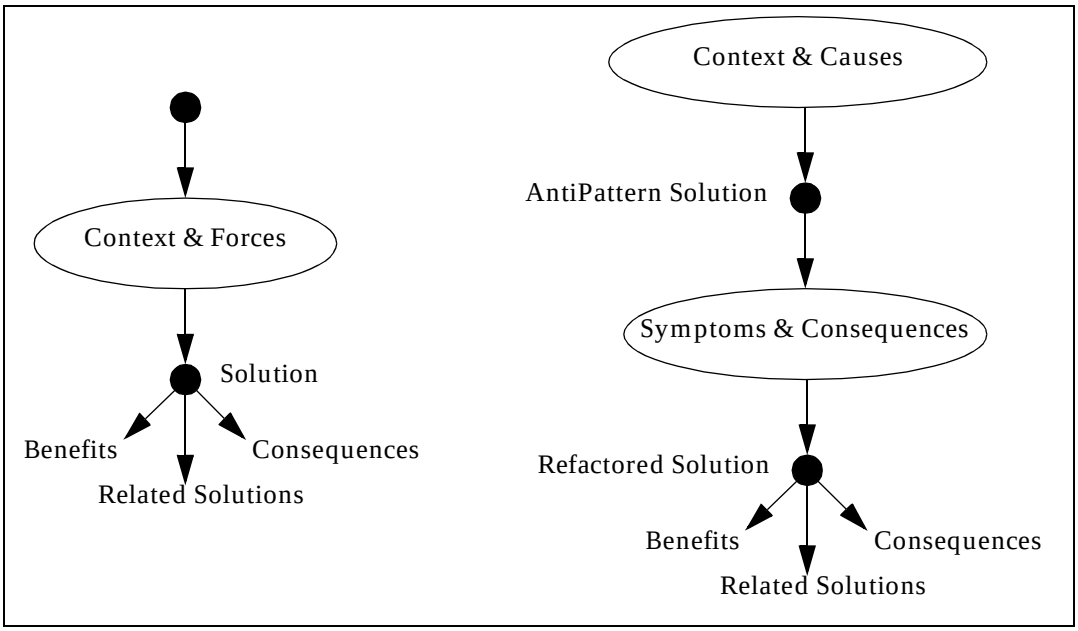

Figure 2-2: Pattern vs. AntiPattern [5]

In software development, antipatterns are useful both when developing a new software system and when extending an existing one. In the former case, we learn from our and other people's mistakes in order to avoid them. In the latter case, we identify those parts of the software system that have to be refactored before adding new functionality.

2.1.6. Example: Blob

The Blob antipattern [5] is also known as the *God Class* problem [38]. Riehl [38] distinguishes the behavioral form and the data form.

In the *behavioral form*, one class performs most of the work while the remaining classes are rather trivial (see Figure 2-3 left). A better design would reassign the responsibilities (see Figure 2-3 right).

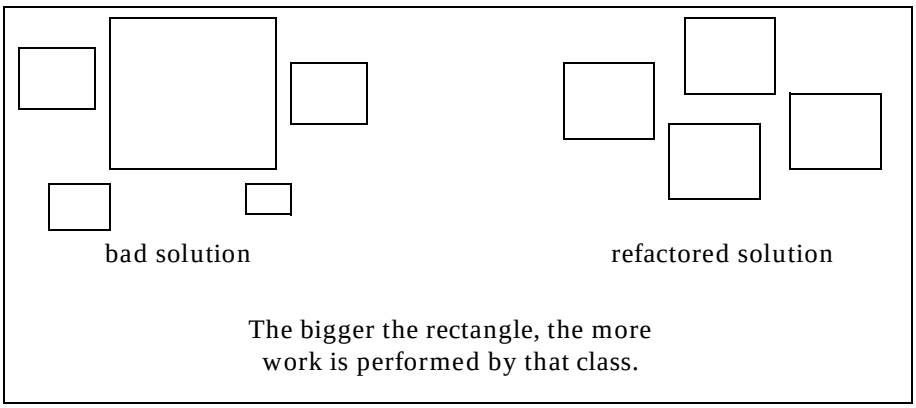

Figure 2-3: Behavioral Form

In the *data form*, one class encapsulates most of the data and provides accessor (i.e., `get`) and mutator (i.e., `set`) methods while the other classes perform the computations

on these data (see Figure 2-4 left). A better design would group related data and methods (see Figure 2-4 right).

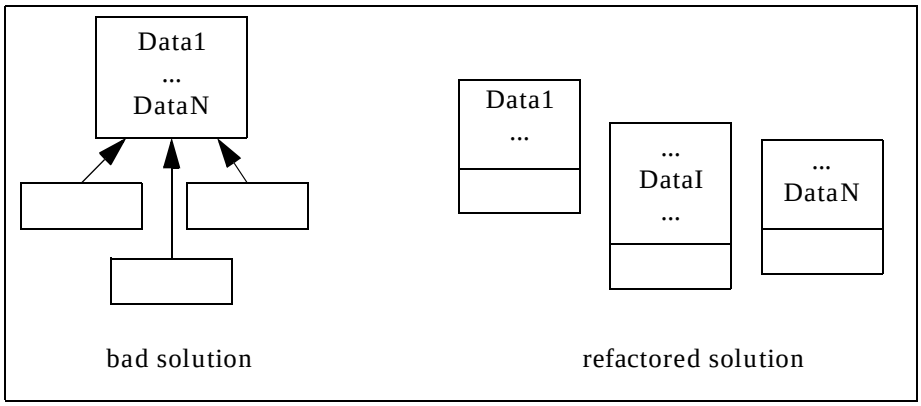

Figure 2-4: Data Form

2.2. Frameworks

The term framework can be confusing because it has a different meaning in network management and in software engineering. In this thesis, we define and use it solely in the latter context. Issues such as composition vs. inheritance, black-box vs. white-box reuse, or component frameworks [50, 51] are beyond the scope of this thesis.

2.2.1. Framework

A *framework* defines a customizable, object-oriented architecture and provides its implementation. It consists of cooperating classes and features *hot spots* [36] or *plug points* [9] that make customization possible in the first place (see Figure 2-5). Hot spots

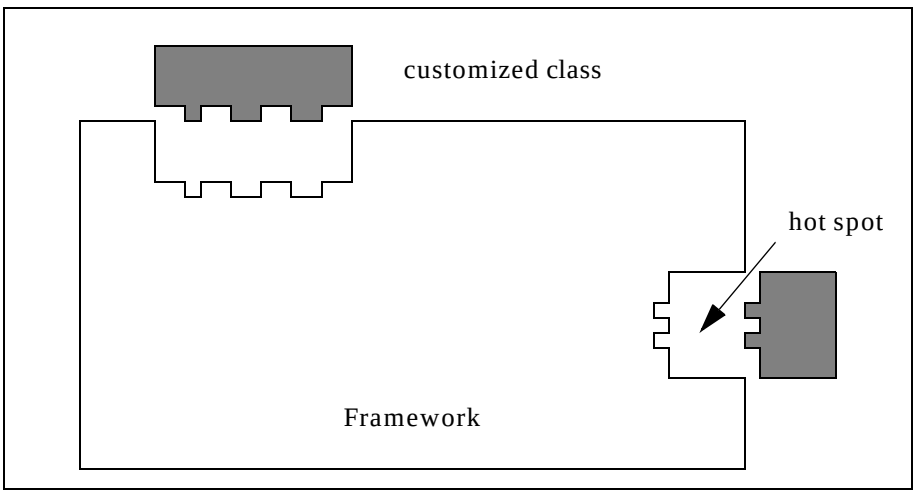

Figure 2-5: Framework [36]

are abstract or at least non-final classes. Within factory, template and hook methods [16] and by means of polymorphism, the framework invokes the methods on the subclasses of the hot spots (see Figure 2-6[5]). But it maintains overall control and makes sure that certain bask tasks are fulfilled. Frameworks are usually geared toward a

5. We adopted the Unified Modeling Language (UML) [14, 37] for the class, interaction, and package diagrams in this thesis.

specific domain, e.g., multiobjective optimization with genetic algorithms [44] or reliable distributed systems [18].

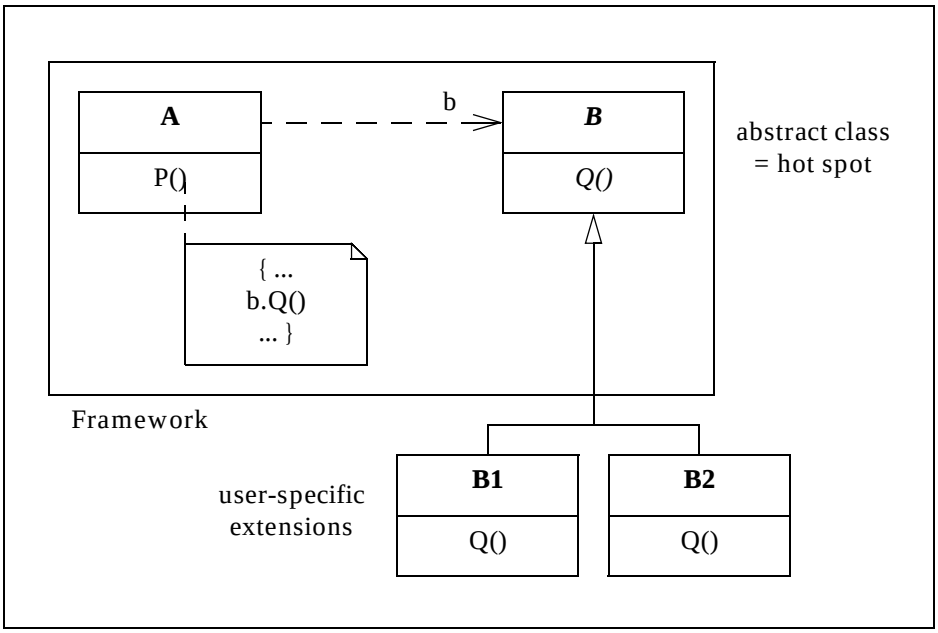

Figure 2-6: Hot Spot [36]

2.2.2. Example: Model-View-Controller

We already encountered the Model-View-Controller pattern in Section 2.1.3. Now we consider the Model-View-Controller framework.

The Model-View-Controller (MVC) underlies Java's GUI elements, the Swing set [22]. In the case of a text field, for instance, the model manages the actual text, the view displays the text on the screen, and the controller determines its editability. What makes the text field a framework is that we can not only use it *as is* but also replace one of its three constituent parts independent of the others by a subclass. Let us suppose that we want to allow ASCII characters only; in that case we replace the default controller by a subclass that ignores all other characters—the model and the view need not change.

2.2.3. Application Framework

Frameworks such as the MVC represent larger-grained units of reuse than single classes. Applications in a specific domain (e.g., productivity suites) have much more in common than simple GUI components, though: window management, menu handling, opening & saving documents, etc. An *application framework* is a standard program that can handle, for example, windows and menus, but that has to be extended by the programmer to give them content. Mössenböck [36] calls this *programming by difference.*

Popular examples of application frameworks are Microsoft's *MFC* for Windows [35] and Metrowerks' *PowerPlant* for the Mac OS [34].

2.2.4. Library vs. Framework

In procedure-oriented programming, the user of a procedure library develops the main procedure and calls library procedures at her own discretion (see Figure 2-7 left).

When using a class library, the user-defined classes are in control as well and determine the overall architecture of the system.

The opposite is the case when development is based on frameworks: the framework decides when the user's methods are invoked (see Figure 2-7 right). This is also called the Hollywood principle: *"Don't call us, we'll call you!"* [9, 36].

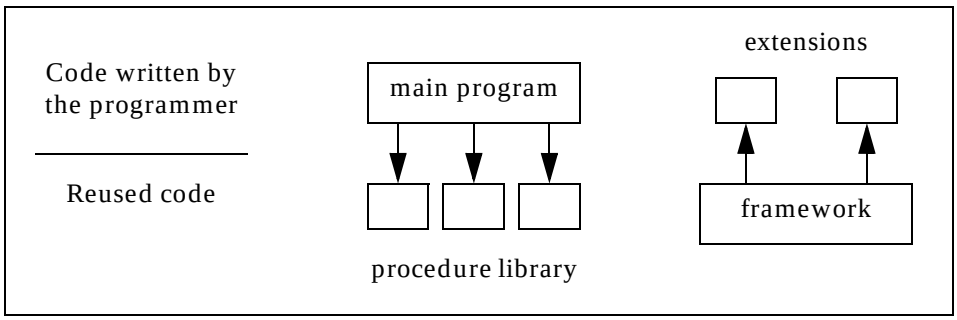

Figure 2-7: Library vs. Framework [36]

2.2.5. Patterns and Frameworks

According to Gamma *et al.*, design patterns and frameworks differ in three major ways [16, p. 28]:

> *"1.* Design patterns are more abstract than frameworks. *Frameworks can be embodied in code, but only examples of patterns can be embodied in code. A strength of frameworks is that they can be written down in programming languages and not only studied but executed and reused directly. [...]*
> *2.* Design patterns are smaller architectural elements than frameworks. *A typical framework contains several design patterns, but the reverse is never true.*
> *3.* Design patterns are less specialized than frameworks. *Frameworks always have a particular application domain. [...]"*

Thus, while classes promote code reuse and patterns promote design reuse, frameworks promote both code and design reuse. To quote Garbinato [18, p. 18], *"design patterns capture experience* about *software design, whereas frameworks* are *software."* As a matter of fact, design constraints imposed by a framework may render it difficult or impossible to combine several frameworks [31].

3. Network Management

Computer networks would be of little use if they did not work properly. Keeping a network up and running, maintaining its performance, and reducing the cost of ownership is what *network management* is about [33]. As computer networks grow in size and importance and at the same time become more heterogeneous, having management schemes that take into account scalability, heterogeneity, security, etc. is no longer optional but mandatory.

In the remainder of this chapter, we first introduce general network-management issues and then give an overview of SNMP and Web-based management.

3.1. General Issues

Management schemes differ in *how* they perform network management. What they have largely in common is *why* network management is performed and *who* does so.

3.1.1. Monitoring & Control

According to Stallings [46], network management encompasses network monitoring and network control.[1] *Monitoring* means to gather information about the network (e.g., the number of packets per minute a certain router handles). *Control* means to update the information the network depends on (e.g., change a routing table).

What, then, should be monitored or controlled? The International Organization for Standards (ISO) defines five key areas of network management [46]: fault management, accounting management, configuration and name management, performance management, and security management.

The purpose of *fault management* is to detect, isolate, and correct faults. A fault and an error are not the same thing. A *fault* needs to be dealt with by the management system (possibly automated). An *error* can be dealt with by a network node (e.g., by using channel coding to detect transmission errors).

The purpose of *accounting management* is to allocate ressources and charge for their services. Often the end users are not really charged (e.g., within a university). Nevertheless, if the administrators (see Section 3.1.2.) track the end-user activity, they can better plan for future changes of the network.

The purpose of *configuration and name management* is to identify the network nodes and to communicate with them in order to maintain or modify the services the network provides. This communication may consist of data collection, orders to enter a certain state, or even the command to gracefully shut down the node.

The purpose of *performance management* is to measure the performance level of the network. Central to performance management are the questions of what to measure and how to interpret the measurements. To answer the first question, Stallings [46] suggests and describes the following performance indicators: availability, response time, accuracy, throughput, and utilization.

---

1. Note that the term *network control* is sometimes used to refer to events occurring on a short time scale, e.g., when the reaction time is below 1s.

The purpose of *security management* is to make sure that only authorized entities gain access to network services (information, processor time, etc.). In particular, unauthorized users must not be able to control the network.

With fault, accounting and performance management, the emphasis is on monitoring. With configuration and security management, the emphasis is on control [46].

3.1.2. Entities

The network management staff consists of network administrators and network operators. The network administrators decide on the network policies and on the mechanisms to enforce them—loosely speaking, they are the boss. Network operators perform *regular management* if they monitor the network on a regular basis to anticipate problems, and *ad-hoc management* if they cope with a problem after it arised—and they are the first to be notified (possibly in the middle of the night) in case of emergency. That said, we will simply refer to both network administrators and network operators as *administrators*.

An administrator works at a Network Management Station (NMS). Unlike most of the network nodes, the NMS can be dedicated to network management only. The process that runs on the NMS on behalf of the administrator is commonly called the *manager* and should interact with her in a user-friendly manner.

The manager's communication peers on the managed nodes are the *agents*. The agents should at least be able to inform the manager about the state of their node and change it on request.

The manager and agent processes make use of the facilities provided by the network node on which they run (see Figure 3-1).

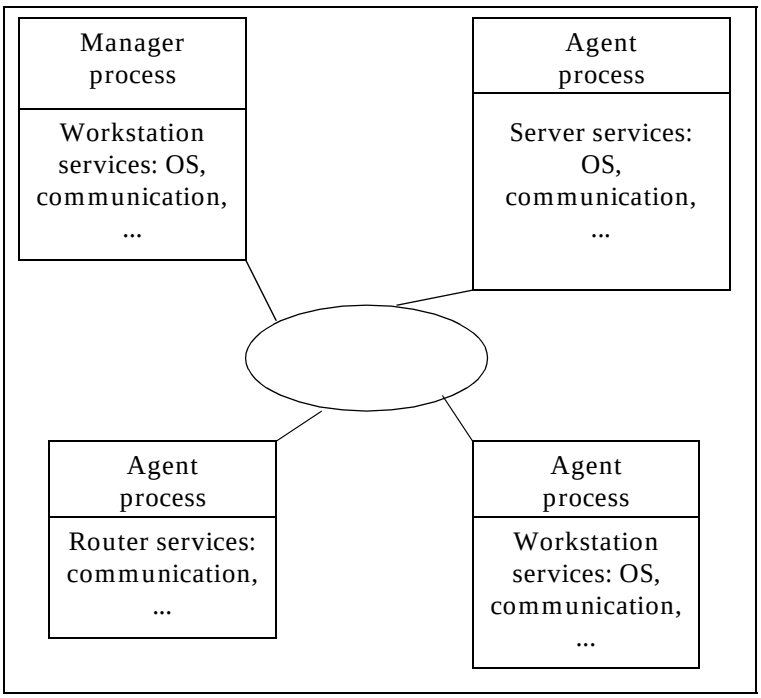

Figure 3-1: Network Management Processes (adapted from [46])

Nodes that cannot host an agent or do not support the management protocol need to be represented by another network node on which a *proxy agent* process runs. Very important is the principle of transparency [41]: when a real agent is managed via a proxy agent, it should appear to the manager as if it were communicating with the real agent directly.

3.1.3. Push vs. Pull

From the point of view of the manager, there are two ways of collecting data from the agent [29]: the manager keeps asking the agent to send data at each polling cycle (*pull* model), or the manager has the agent send data by subscribing to it once (*push* model).

Typically, notifications rely on the push model, whereas monitoring can rely on either the push or the pull model. The trade-off between the communication overhead in the pull model and the added responsability for the agents in the push model is an important design decision.

3.2. Simple Network Management Protocol

For about ten years, the *Simple Network Management Protocol* (SNMP) has been the standard for IP network management. It includes an application-level communication protocol running on top of the User Datagram Protocol (UDP) and defines the nature of the management information.

Even though SNMP is an open standard (i.e., it is published as Requests for Comments [RFCs] by the Internet Engineering Task Force [IETF]), only a handful of companies actually compete in the manager-side software market. Martin-Flatin [27] explains how this situation arised.

Three versions of SNMP exist [46]: The original SNMP which is now also called SNMPv1 was replaced by SNMPv2. SNMPv3 defines a security capability and allows for future enhancements without the need for a new version. SNMPv3 is intended for use with SNMPv2.

3.2.1. Fundamental Axiom

The key to understanding SNMP is its fundamental axiom [41, p. 12]:

> *"The impact of adding network management to managed nodes must be minimal, reflecting the lowest common denominator."*

This translates into SNMP agents being very simple (what Wellens and Auerbach [54] call *dumb*) and SNMP managers being very smart (i.e., complex). Typically, there is only one management station.

While this distribution of network-management responsibilities reflected the capabilities of the managed nodes of ten years ago, today most are capable of doing more for network management (e.g., some preprocessing) without neglecting their main task (e.g., routing) today [19].

3.2.2. Management Information

SNMP management information is defined by the *Structure of Management Information* (SMI) and the *Management Information Base* (MIB).

SMI is a variant of the *Abstract Syntax Notation One* (ASN.1) [7, 46]. The types of managed variables are defined with SMI. In SNMP, every managed node is abstracted as a set of management variables. To monitor thus means to read variable values, to control means to write variable values. The transfer syntax used for exchanging variables between managers and agents are the *Basic Encoding Rules* (BER) [7, 46]. (For more on ASN.1 and BER and their applications in distributed systems in general, see Coulouris *et al.* [7]. For more on ASN.1 and BER and their application in SNMP in particular, see Stallings [46].)

The MIBs are collections of management variables. They are organized as trees with the variables as the leaves. Different types of network equipment support different types of MIBs. Both generic and vendor-specific MIBs exist. Generic MIBs are issued by the IETF, vendor-specific MIBs by the vendors of network equipment.

3.2.3. Information Exchange

When a manager wants to read or write a variable, it issues a *request*. The agent then returns the value of the variable or the write confirmation in a *reply* (see Figure 3-2). Sometimes the reply contains one or more exceptions, an error, or does not arrive at all. An *exception* indicates that a particular variable (see next paragraph) cannot be processed. An *error* indicates that the operation cannot be processed. If the reply does not arrive at all, the request times out.

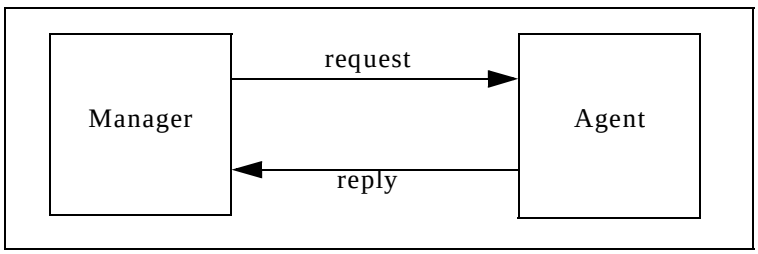

Figure 3-2: Request-Reply

There are four kinds of request of interest to us: get, get-next, get-bulk, and set. The *get request* asks for the value of one or more variables. The *get-next request* asks for the values of the variables that follow the variables given in the request. This request allows the manager to iterate over the MIB (which is a tree) and discover its structure. The *get-bulk request* asks for the values of one or more variables that follow the variables given in the request. This request can save the manager from using several get-next requests. The *set request* creates or updates one or more variables.

When an agent has to inform the manager about an important event (usually a problem), it sends an unsolicited *notification* (see Figure 3-3).

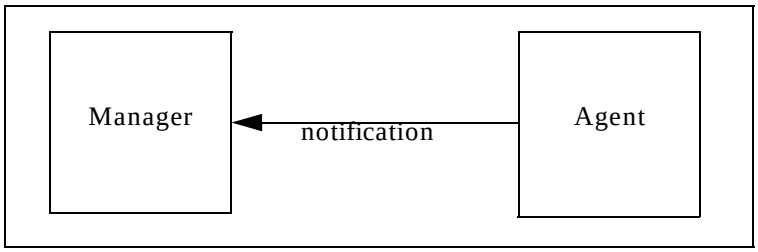

Figure 3-3: Notification

3.3. Web-based Network Management

Both the industry and academia are looking into managing networks by other means than SNMP [27]. Instead of replacing SNMP by yet another protocol, using Java-based or Web-based solutions seems to be more appealing [10, 49].

Several approaches have been suggested by Wellens and Auerbach [54], Sun microsystems [49], Deri [10], etc. Here, we will consider the approach of Martin-Flatin [28, 29], who makes use of both Java and Web technologies. We explain the model underlying the JAva MAnagement Platform (JAMAP) [4, 30].

3.3.1. Entities

In addition to the managed nodes represented by an agent and the administrators who work at NMSs, JAMAP also uses *management servers* as shown in Figure 3-4 and

Figure 3-5. Only one management server and one management station are depicted in these figures, but JAMAP supports several at the same time.

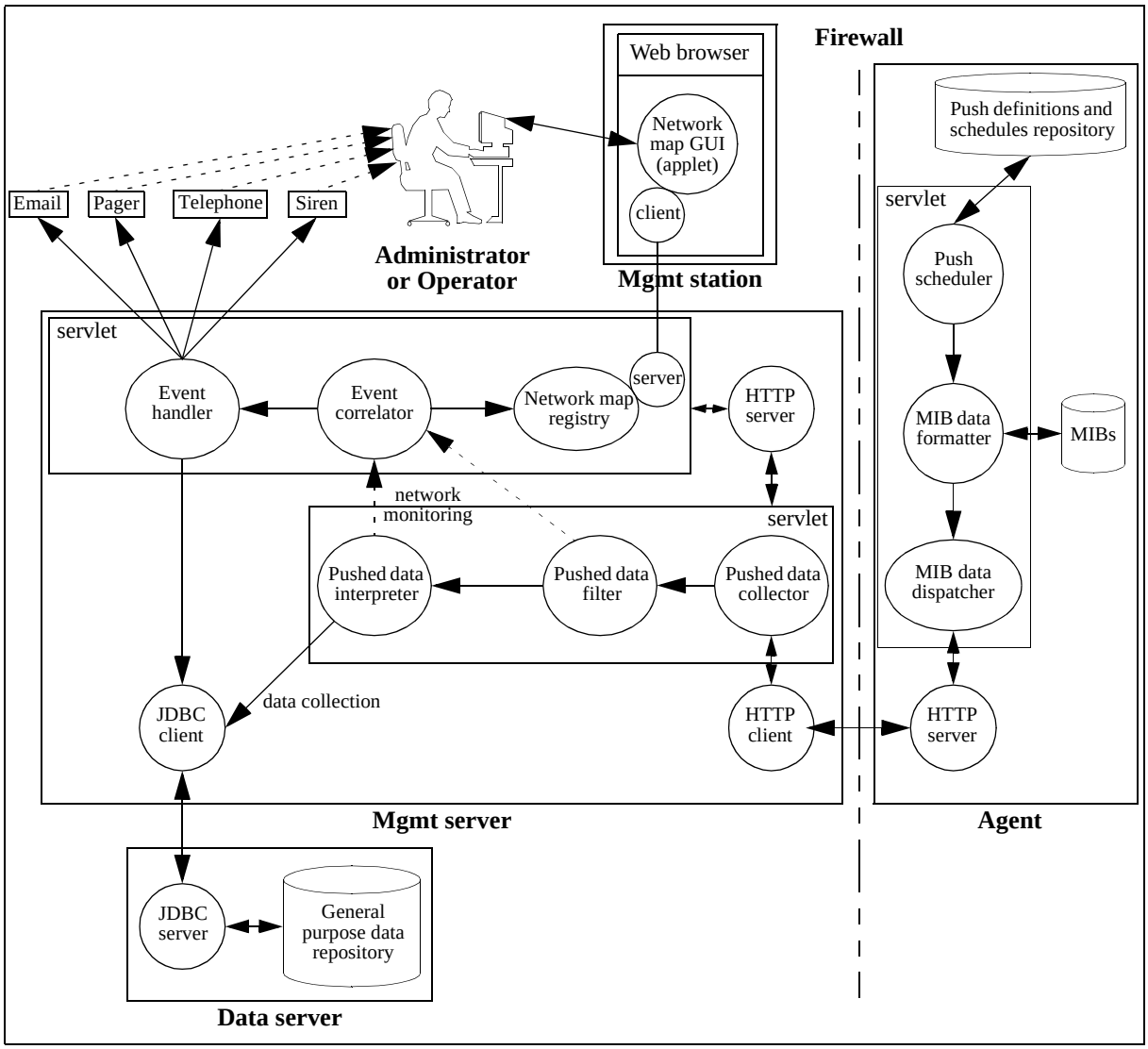

Figure 3-4: Monitoring and Data Collection [30]

A NMS must be able to run a Web browser that supports Java applets. Agents must be able to host an HTTP server with support for Java servlets. And the management

servers must be able to act both as an HTTP client and an HTTP server that supports Java servlets.

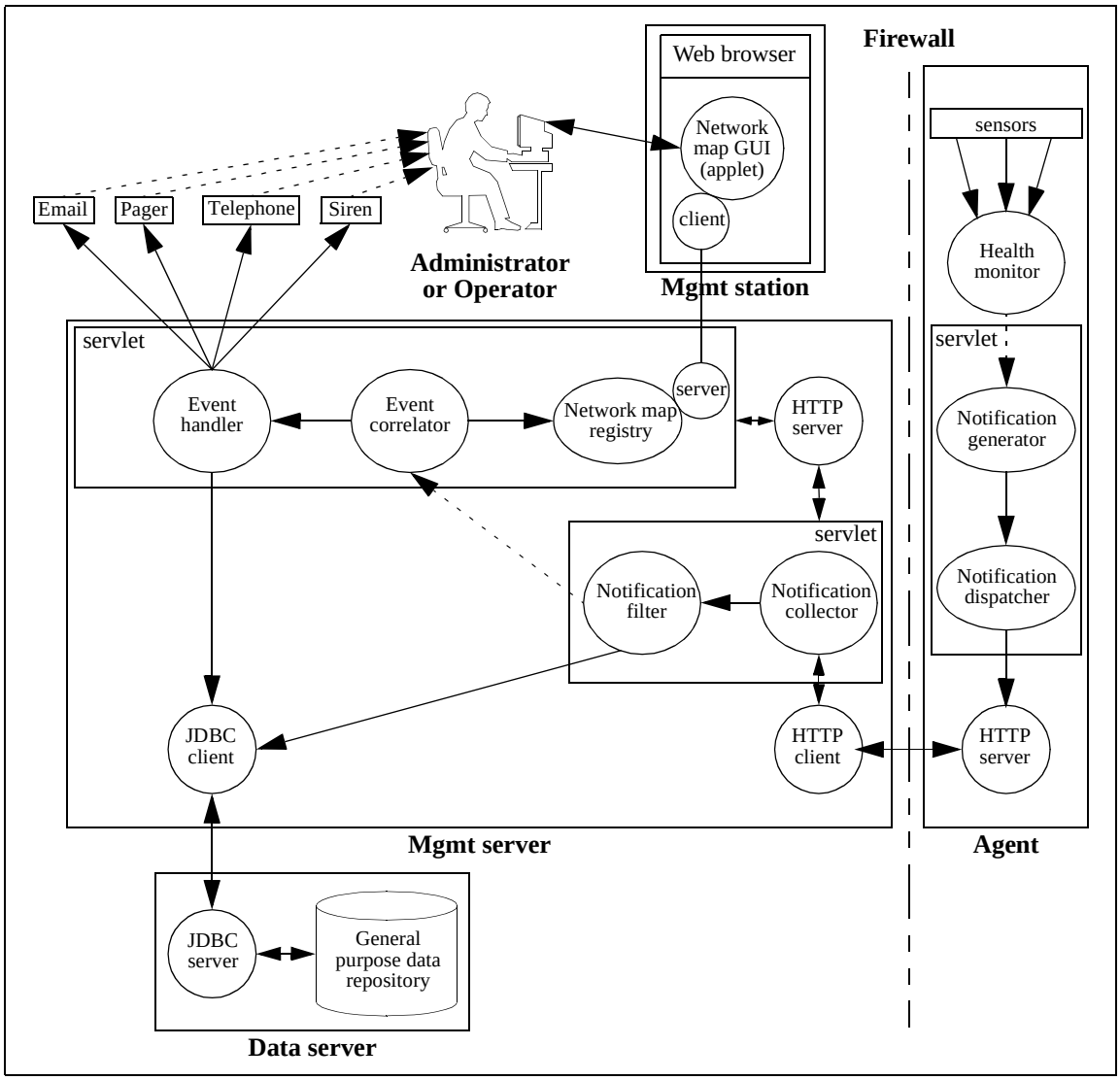

Figure 3-5: Notification Handling [30]

3.3.2. Subscription

The first phase of managing a network node is the *subscription*. Via well-known HTML pages, agents publish what MIBs they support and what notifications they can send. The administrators download applets to select the MIB variables and notifications they are interested in. They also define how often they want the agent to send the variables.

It should be possible to make the subscription data persistent. On the one hand, this allows to automatically resubscribe if the network node lost the subscription information (e.g., after a crash). On the other hand, for the same type of network equipment the applet needs to be downloaded only once, and the subscription data can then be sent to all agents through some scripting mechanism.

3.3.3. Monitoring and Data Collection

Variables are only polled during ad-hoc management. Regular management is performed by having the agents send the variables as often as defined during the subscription phase.

As shown in Figure 3-4, the servlet of the agent sends the variables to a management server where they are received by the pushed-data collector. The pushed-data collector connects to the agents upon startup and automatically reconnects to them if the connection is accidentally lost or if an administrator orders to do so after a deliberate disconnect (see next paragraph).

The role of the pushed-data filter is to detect misbehaving agents (e.g., a malicious agent bombing the management system with bogus data) and to make the pushed-data collector disconnect from them if this is necessary to protect the management system. If an agents misbehaves, the pushed-data filter generates an event for the event correlator.

Variables that have not been dropped by the filter are passed on to the pushed-data interpreter. Data that is meant to be collected (e.g., to later assess the performance of the system) is simply archived. Data for network monitoring is checked by the interpreter based on rules defined by the administrators. If the values of the variables are not ok (e.g., abnormal threshold passed), it generates an event for the event correlator. It also generates an event if an agent does not push data on time.

Also based on rules defined by the administrators, the event correlator analyses the data, and then updates the network map or calls the administrators' attention by some other means (via the event handler).

The network map is an image of the network state. Its graphical representation on the NMS is designed such that administrators can easily grasp this state.

3.3.4. Notification Handling

When the health monitor detects an abnormal condition, it contacts the notification generator which generates a standardized notification to be sent by the network dispatcher to a notification servlet (see Figure 3-5).

The notification collector and the notification filter have the same purpose as their pushed-data counterparts. There is no notification interpreter, however. Notifications are always logged *and* passed on to the event correlator.

4. Patterns and AntiPatterns in SNMP-based Network Management

Our goal in this chapter is to identify design patterns and antipatterns in the context of SNMP-based network management.

The sources for the work presented in this chapter and the next are the books by Gamma *et al.* [16], Buschmann *et al.* [6], and Brown *et al.* [5], as well as a preliminary version of Schmidt *et al.* [43].[1]

In the remainder of this chapter, we first discuss design patterns that we consider useful for object-oriented implementations of SNMP-compliant managers, agents, and the communication protocol itself. Then, we present software-development and software-architecture antipatterns.

4.1. Design Patterns

The power of design patterns stems from leading to a good software design—and eventually implementation—when they are well-chosen. Choosing the right design patterns in a given context can be more difficult than it may look, though. First, one needs a solid base of design patterns to choose from—not only in numbers, but also in understanding their similarities and differences, their benefits and liabilities.[2] Second, the design problem to be solved must be well-understood. And third, one has to overcome the mental barrier of thinking of design patterns as being about objects in a single address space only; design patterns describe responsibilities and collaborations of software entities in general.[3]

To cut a long story short, this section continuously evolved in the course of this thesis, with design patterns (and applications thereof) being included or excluded as we gained a better understanding of both design patterns and the problem at hand. However, we hope that this evolution is transparent; design patterns are most effective when the reader afterwards goes: *"Why didn't I think of this? It's so obvious now!"*

Some of the design patterns we discuss are inherent in SNMP. Some are found at the conceptual level, others at the specification level, yet others at the implementation level [14]. Even if some cases do not directly lead to a new design, we find that they are still valuable by providing insight into SNMP from a design pattern's perspective.

We left out design patterns that do not provide a solution to a typical network-management problem (in particular Buschmann *et al.*'s [6] idioms), even though they may prove useful in resolving general implementation issues of

1. We also skimmed through the books by Grand [20, 21], but did not find any design patterns of interest to us that are not discussed by Buschmann *et al.* [6] or Gamma *et al.* [16] as well.
2. Even though patterns authors go to great lengths to give intuitive examples, to list rules of applicability, to discuss variants, etc. in order to make up for the reader's possible lack of design experience, developing a feel for patterns is nevertheless essential.
3. A big help in this last regard was the *Building Software with Patterns* chapter in Schmidt *et al.* [43]. In addition to providing general guidelines that support a pattern-based software construction, that chapter includes examples showing design patterns such as the MVC in a distributed environment.

network-management software. For example, the Template Method pattern [16], which we do not discuss, is likely to be found in many object-oriented software system.

Similarly, just because our focus is on network-management related applications of a design pattern, this does not mean that the design pattern cannot be used anywhere else in the network-management software. On the contrary: the design patterns in this chapter are general enough to solve problems in different contexts and at different scales [6, 16].

4.1.1. Adapter

The *Adapter* pattern [16] converts the interface of a pre-existing class into another interface that clients expect.[4] It enables the implementation (and thus the functionality) of the class to be reused even if the interface of the class is not know by the potential client.

For instance, a class providing encryption and decryption may feature a method with signature
`crypt(bool flag, int[] plainText, int[] cipherText)`
whereas its client expects it to have methods such as
`encrypt(int[] plainText, int[] cipherText)` and
`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 need to transform the replies as well when necessary.

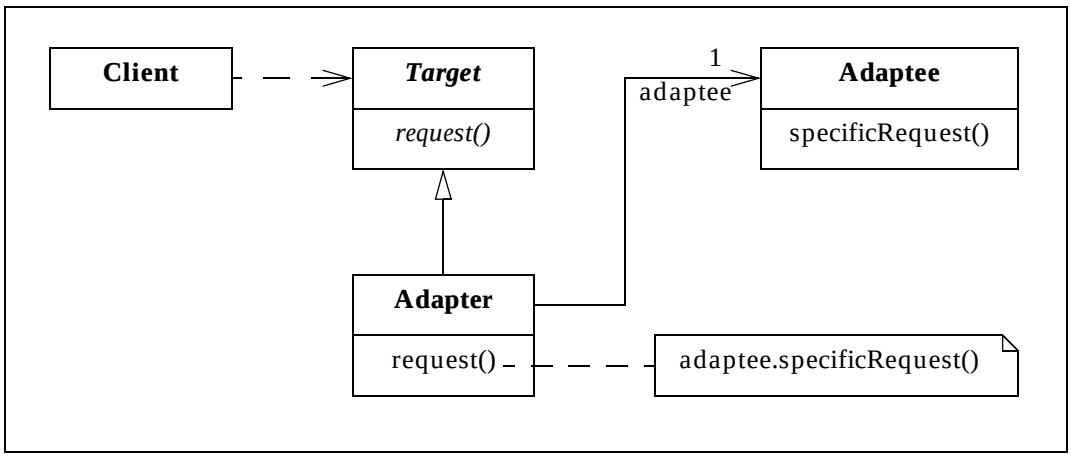

Figure 4-1: Object Adapter (adapted from [16])

Gamma *et al.* [16] discuss two versions of the Adapter pattern, the Object Adapter (see Figure 4-1) and the Class Adapter (see Figure 4-2). The Object Adapter realizes adaptation by using object composition, the Class Adapter by using multiple

4. In a strongly typed language, the Adapter pattern is even necessary in cases where the interface expected by the clients is contained in the interface of the pre-existing class, but where the types differ.

inheritance (e.g., in C++) or single implementation inheritance with multiple interface inheritance (e.g., in Java).

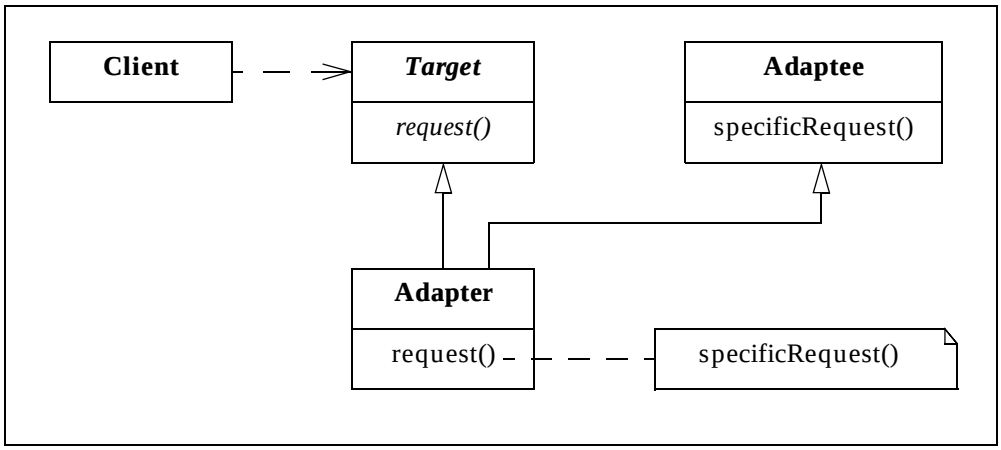

Figure 4-2: Class Adapter (adapted from [16])

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 4.1.2.).

Note further that when an agent issues a notification, the proxy agent needs to translate it as well before forwarding it to the manager. The proxy agent thus behaves like a two-way adapter [16] (see Figure 4-3).

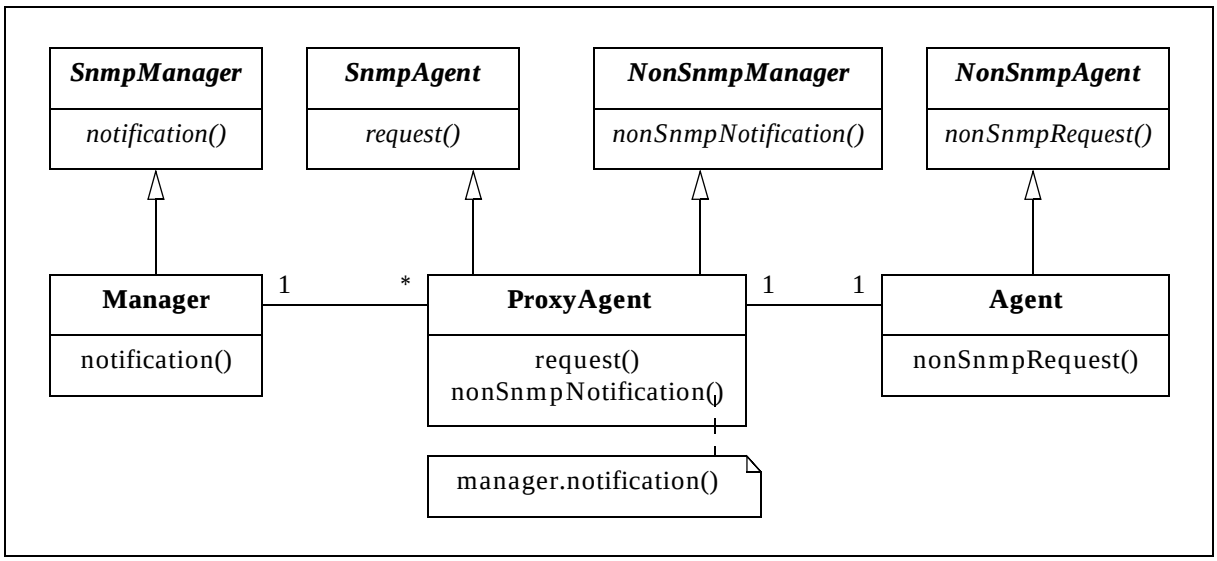

Figure 4-3: SNMP Adapter

4.1.2. Proxy

The *Proxy* pattern [6, 16] 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 Figure 4-4).

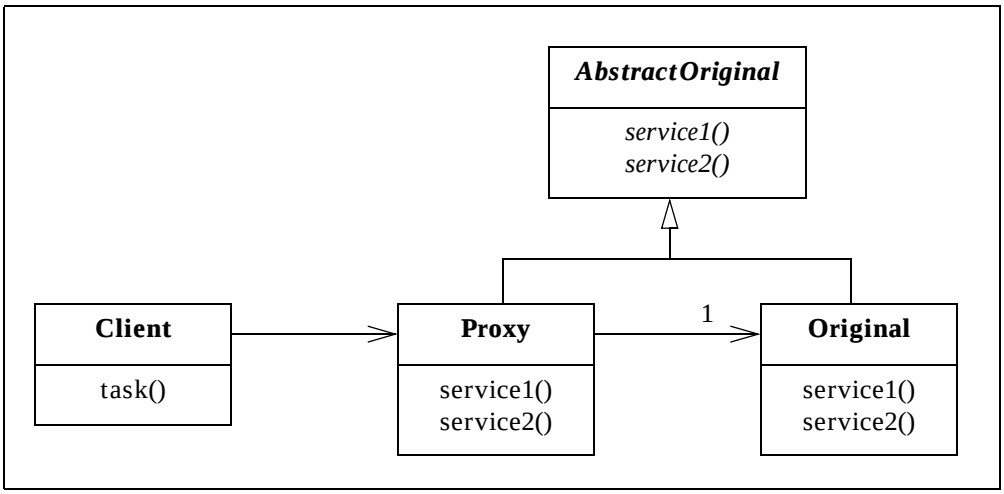

Figure 4-4: Proxy (adapted from [6])

Because of its name and because it acts as an intermediary, a `Proxy` object seems to correspond to a proxy agent. In general, 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 (see Section 4.1.1.).

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 [6, 16]. Most interesting to network management are the Protection Proxy and the Firewall Proxy.

In the *Protection Proxy* pattern [6], 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 which is not security-aware albeit being able to communicate in an SNMP-compliant way. For instance, a `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 stripping away the request authentication.

In the *Firewall Proxy* pattern [6], 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 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 WAN link).

4.1.3. Bridge

The *Bridge* pattern [16] decouples an abstraction from its implementation so that the two can vary independently. It is depicted in Figure 4-5. 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.

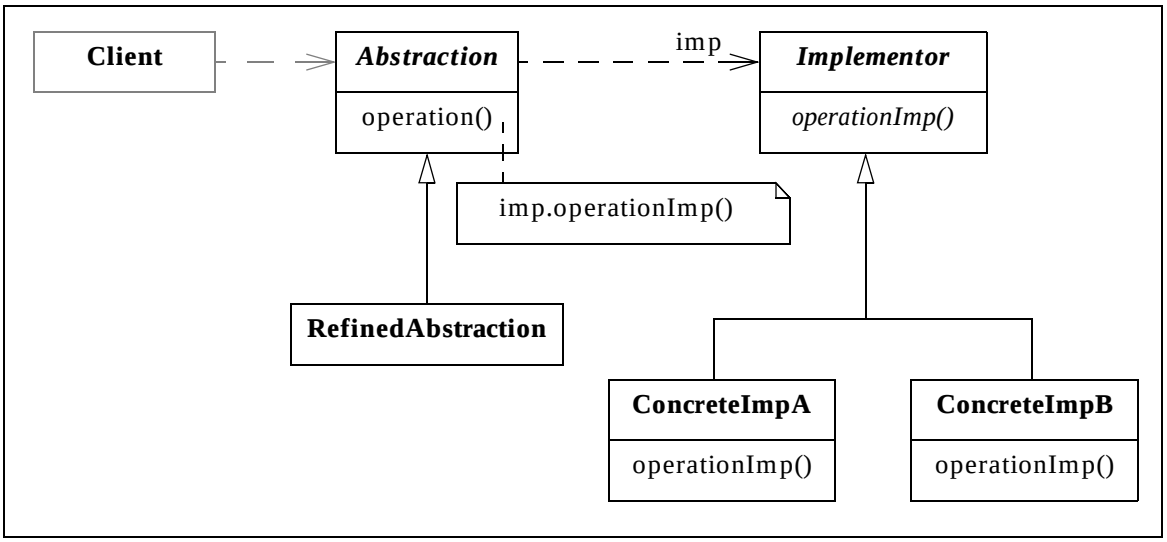

Figure 4-5: Bridge (adapted from [16])

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 what kind of persistent storage (database, spread-sheet, etc.) actually underlies their implementation. In particular, the management application becomes independent of a specific vendor's database system.

4.1.4. Whole-Part and Composite

The *Whole-Part* pattern [6] helps with the composition of objects that together form a semantic unit. A `Whole` class (see Figure 4-6) encapsulates its constituent `Parts`, organizes their collaboration, and provides a common interface to its functionality. The `Whole` prevents `Clients` from accessing these constituent `Parts` directly.

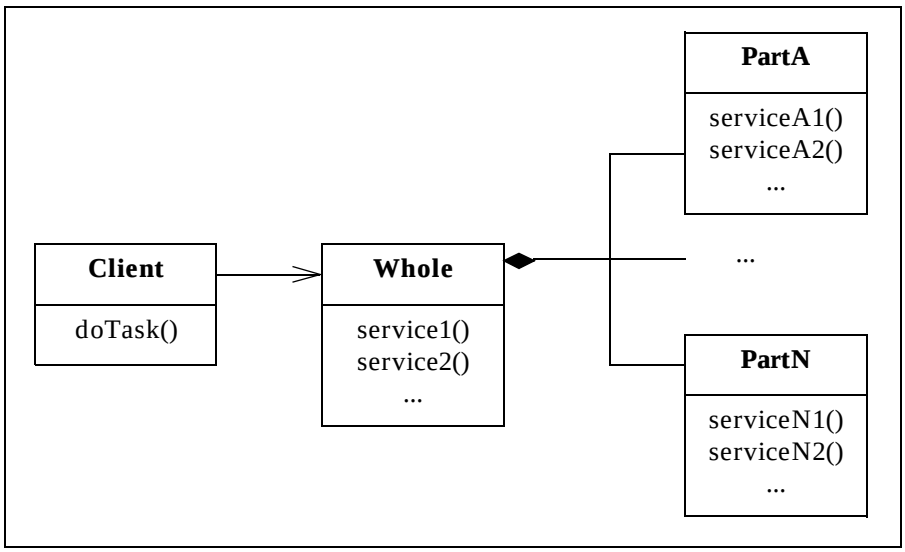

Figure 4-6: Whole-Part (adapted from [6])

In addition to simply managing homogeneous or heterogeneous `Parts`, the `Whole` may exhibit different behavior depending on its `Parts` (e.g., a molecule consisting of atoms in a simulation program). Buschmann *et al.* [6] call this *emergent behavior*.

The *Composite* pattern [16, 36] (see Figure 4-7) is only applicable to whole-part hierarchies in which the `Wholes` and the `Parts` can be treated uniformly. By using recursive composition, clients do not have to make a distinction between `Wholes` and `Parts`. 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.

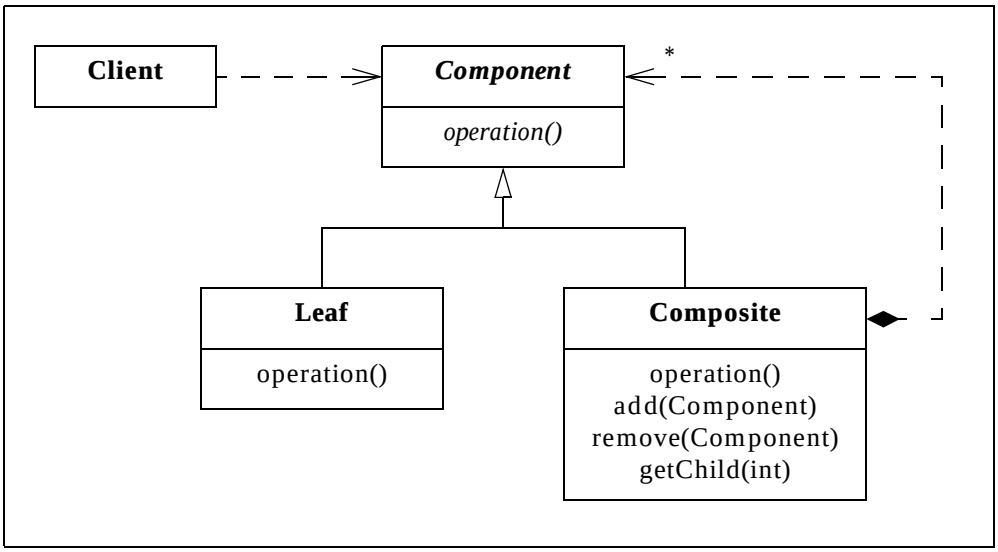

Figure 4-7: Composite (adapted from [16, 36])

SNMPv1 does not support distributed management at all, but SNMPv3 potentially allows for hierarchical management [26]. (SNMPv2's support was broken [26].) The

idea is to divide networks into subnetworks. Top-level managers manage these subnetworks by delegating management to mid-level managers (see Figure 4-8).

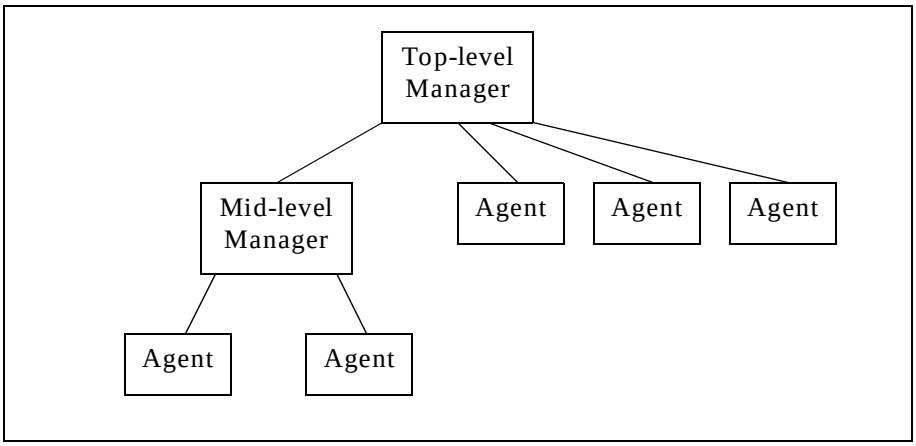

Figure 4-8: Hierarchical Network Management

Conceptually, the `Client` in the Whole-Part pattern therefore corresponds to a top-level manager, the `Whole` to a mid-level manager and the `Parts` to the agents.

Note that in practice, distributed network management today is usually realized by proprietary schemes because manager-to-manager interaction is still undefined in SNMP. This may change in the future.

4.1.5. Facade and Wrapper Facade

The *Facade* pattern [16] provides a unified interface to a set of interfaces in a subsystem. An example is depicted in Figure 4-9.

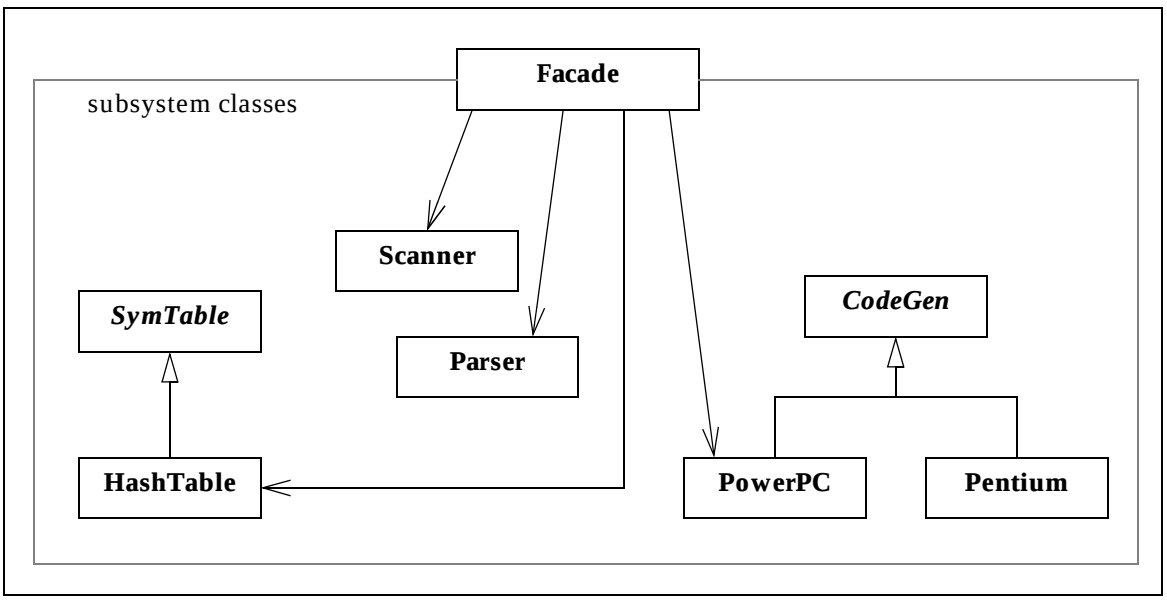

Figure 4-9: Facade (adapted from [16])

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. Or the `Facade` class can offer one, less complex but also less powerful interface as an alternative to working directly with the constituent classes. Consider a development subsystem consisting of scanners, parsers, code-generators, etc. Many of

its clients probably just want to translate from high-level language X to machine code Y. The `Facade` class could therefore supply a method `compileFromXtoY()` which accepts a handle to the source code, takes care of all intermediate compilation steps, and returns a handle to the binary.

The *Wrapper Facade* pattern [43] provides concise, robust, portable, maintainable, and cohesive class interfaces (note the plural) that encapsulate low-level functions and data structures. A `WrapperFacade` class is not meant to be an alternative. Its intent is clearly to provide an object-oriented interface to a non-object-oriented subsystem (see Figure 4-10).

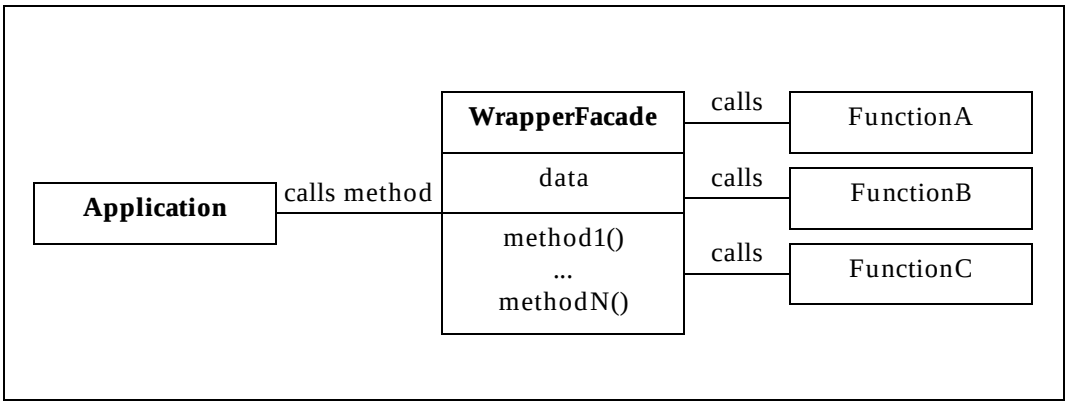

Figure 4-10: Wrapper Facade [43]

Facades and Wrapper Facades can be very useful for layered architectures (see Section 4.1.6.) in which the layers feature a service access point. The SNMP layer (all the layers in the TCP/IP protocol stack, as a matter of fact) is no exception. It has to provide its clients with a well-defined interface, independent of how many classes, functions, etc. were used to implement it. It is the task of the `Facade` class to provide this interface and shield the clients from implementational details if the implementation of the layer is object-oriented. If it is not, the Wrapper Facade is the pattern of choice.

4.1.6. Layers

The *Layers* pattern [6] helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction. It is depicted in Figure 4-11.

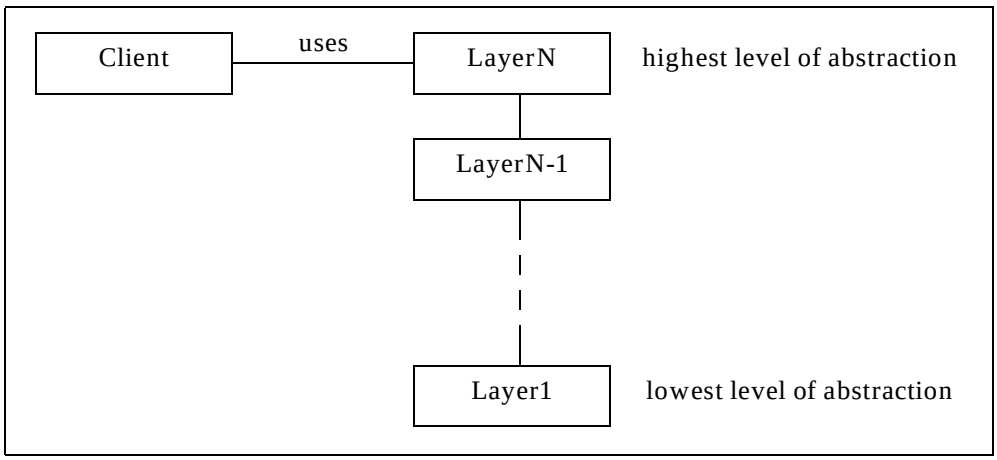

Figure 4-11: Layers [6]

A well-known example of Layers is the ISO Open Systems Interconnection (OSI) model [52]: Together the application, presentation, session, transport, network, data link, and physical layers provide 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 varied 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 whole system more stable or portable (e.g., a hardware abstraction layer). This issue is also addressed by the Facade and Wrapper Facade patterns (see Section 4.1.5.). Or the layer adapts the one below it, i.e., it acts as an Adapter (see Section 4.1.1.).[5]

Note that the layers do not have to be shielded by incorporating a unified interface as long as layer *i+1* does not depend on layer *i-1* or lower (see Figure 4-12). 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 protocol stack is one incarnation of the Layers pattern. SNMP is located at the application layer where it provides services to its clients (i.e., manager and agents) and uses services provided by UDP at the transport layer. UDP uses services provided by IP at the internet layer, etc.

Another incarnation are the managed nodes. According to Rose [41], 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 management protocol,

5. Patterns having similar intents or structures are not the exception. There are situations where multiple patterns apply, depending on the viewpoint taken. What is important is that when the differences become rather "philosophical" than technical, they probably do not matter in practice anyway.

which permits the monitoring and control of the managed node. To see the layering, simply replace the word "components" in the first sentence with "layers".

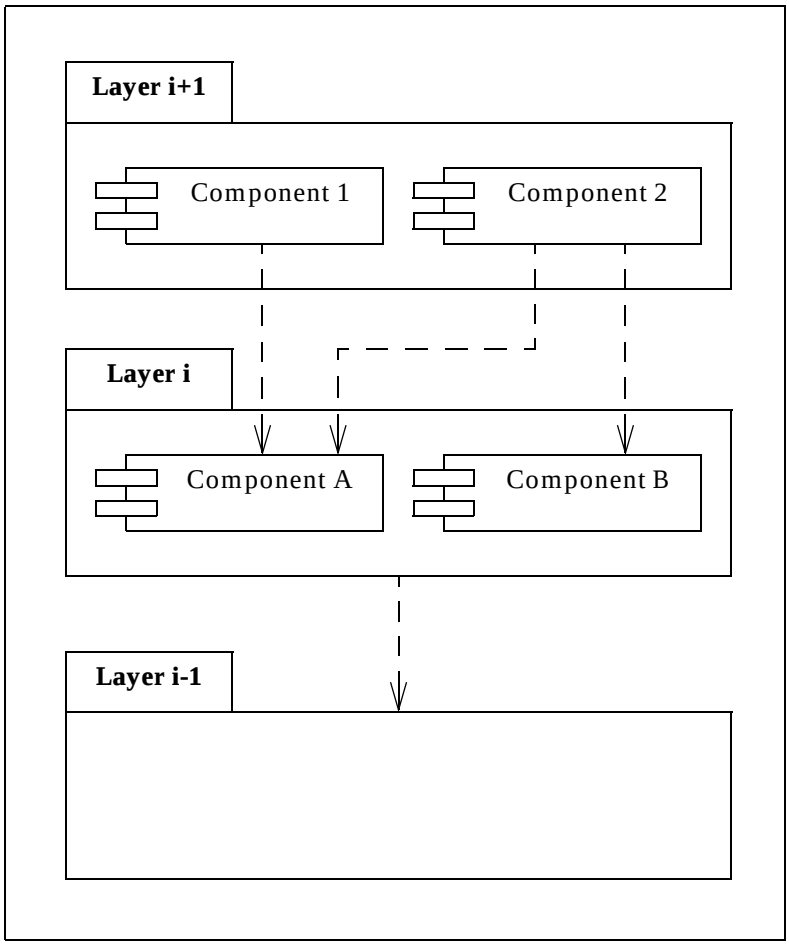

Figure 4-12: Shielded and Unshielded Layers

4.1.7. Iterator

The *Iterator* pattern [16] provides a way to access the elements of an aggregate object sequentially without exposing its underlying structure. It is depicted in Figure 4-13.

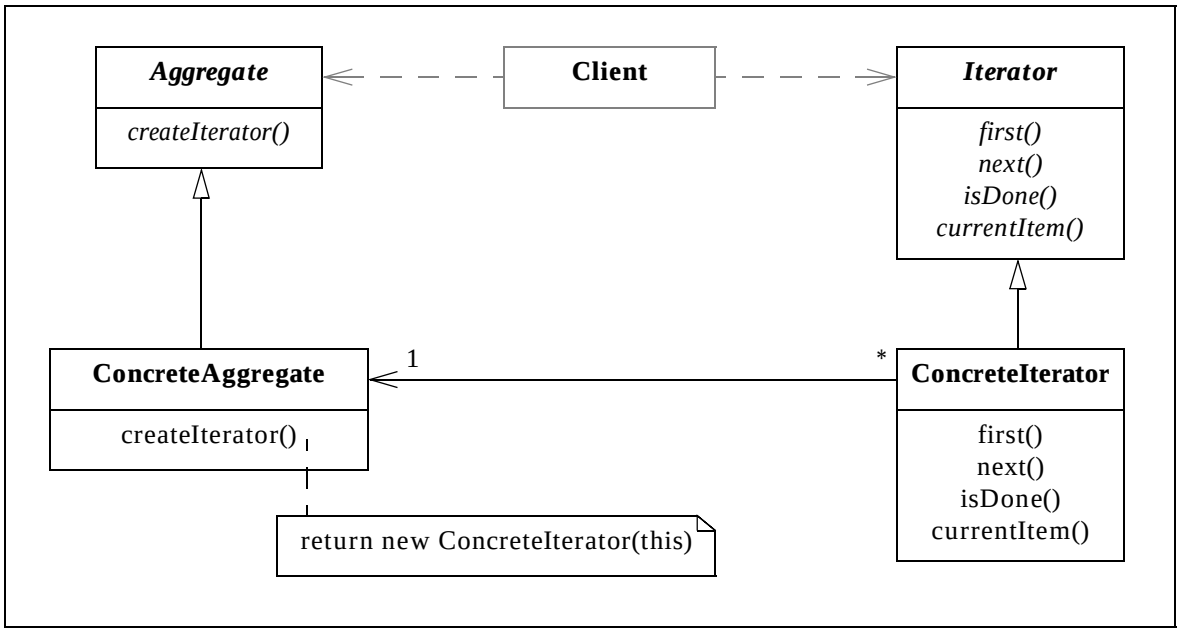

Figure 4-13: Iterator (adapted from [16])

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 bloating the container's interface, and more than one traversal can be pending on the same container (i.e., one traversal per iterator). Furthermore, by defining interfaces common to all containers and iterators, respectively, 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`, see Section 3.2.3.) 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 interfaces sub-group).

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 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. In the worst case, we just get a cleaner design by separating the core of the manager from the part (namely the `Iterator`) that knows how SNMP MIBs are represented. In the best case, we can use the manager with MIBs that provide `Iterator`s of their own without having to make major changes. One such change can consist of applying the Adapter pattern (see Section 4.1.1.) when the `Iterator` interface we designed and the one the new MIB provides do not match.

4.1.8. Mediator

The *Mediator* pattern [16] promotes loose coupling by keeping objects from referring to each other explicitly. It is depicted in Figure 4-14.

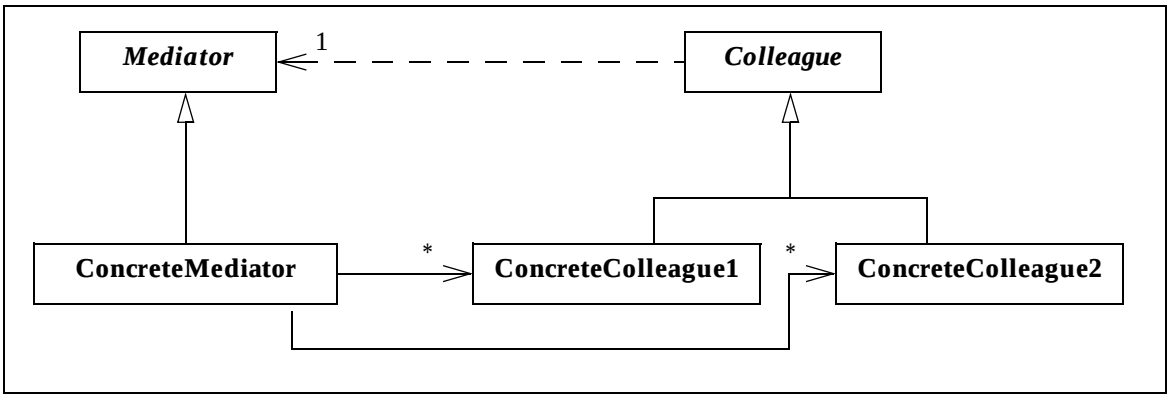

Figure 4-14: Mediator (adapted from [16])

The state of an object sometimes depends on the state of other objects (e.g., GUI elements within a dialog box). When one such object changes state (e.g., checking 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 state depends on each other) only need to inform the `Mediator` object when their state changes. The `Mediator` object then changes the state of other `ConcreteColleague`s as needed.

From a conceptual point of view, 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 [52].

Another application of the Mediator pattern in network 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".

4.2. AntiPatterns

The purpose of antipatterns is not only to find problems, but also to solve or at least alleviate them. However, as our intent is not to revise SNMP, we content ourselves with identifying reasons that lead to some of its shortcomings. This will help us to avoid them in Web-based network management.

4.2.1. The Blob

The Blob antipattern [5] has been introduced in Section 2.1.6. In short, it is found in designs where one class monopolizes the processing, and other classes do little and primarily encapsulate data (see Figure 4-15).

The distribution of responsibilities in SNMP has a striking resemblance with this antipattern. The manager is powerful and complex while the agents basically only feature `get` and `set` functions. This is one consequence of the fundamental axiom (see Section 3.2.1.) and eventually led to the "myth of the dumb agent" [54]. The inaccuracy of this axiom has been highlighted for years by Yemini and Goldszmidt [56, 19].

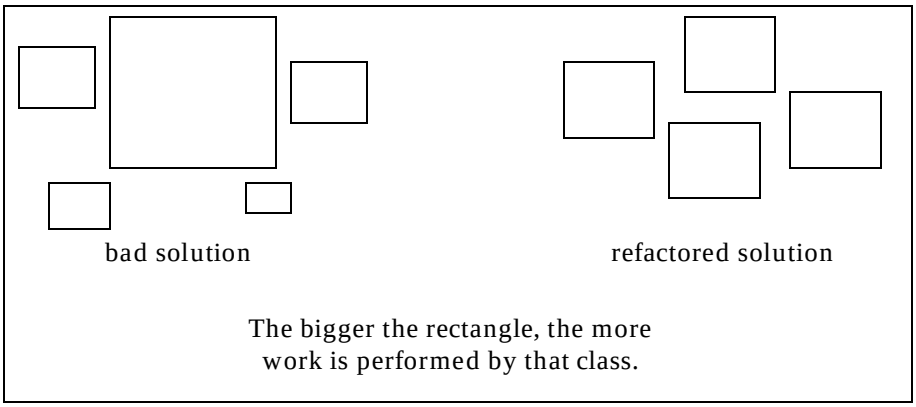

Figure 4-15: The Blob (behavioral form)

Especially in cases where the managed nodes are modern workstations, printers, and other devices with powerful CPUs, one should look into the possibility of using idle CPU cycles for network-management purposes.[6] In performance management, for example, some agents could compute usage statistics locally and send the results to the manager instead of sending the raw measurement data.

4.2.2. Golden Hammer

The *Golden Hammer* antipattern [5] typically occurs when a development team has gained a high level of competence in a particular technology and every new product or development effort is viewed as something that is best solved with it. Its main symptoms are that (i) solutions have inferior performance, scalability, etc. when compared to other solutions in the industry, and (ii) new developments rely heavily on a specific vendor product or technology.

The Java platform used to be a typical example [22, 23]. Originally, there was exactly one kind of Java run-time environment. Over the years, it became clear that the Java run-time environment was not powerful enough for large-scale enterprise applications and too demanding for embedded systems. This was acknowledged and addressed by Sun microsystems in 1999. Today, the Java run-time environment (as well as the software development kit) is available in three editions: J2ME (micro), J2SE (standard), and J2EE (enterprise) [48]. The micro edition is further customizable by defining profiles, that is, a specification of the minimum set of Application Programming Interfaces (APIs) useful for a particular kind of consumer device and a specification of the Java virtual machine functions required to support those APIs [47].

SNMP's fundamental axiom (see Section 3.2.1.) can be considered as an instance of this antipattern. By reading Rose [41], one gets the impression that many major design decisions were based on the fundamental axiom: minimizing the impact of

6. After all, CPUs are much more powerful today than they used to be in the late 1980s, when SNMPv1 was devised. To quote Wellens and Auerbach [54, p. 2]: *"Today's network devices often contain processors and memory exceeding that of our management platforms of a few years ago."*

management on the agents (and thus shifting the burden to the management station), for instance, or choosing UDP as the transport layer protocol. (SNMP as a whole could have turned into a Golden Hammer instance, too, if the people who suggested to use SNMP for any kind of management—such as systems management and policy management—had been listened to.)

Another example of the Golden Hammer was the attempt, in the early 1990s, to impose the OSI management stack (CMIP, CMIS, etc.) on all managed nodes in LANs, including bottom of the range devices. A stack that had been developed for telecommunications networks and that could not compete with the TCP/IP stack in LANs.

4.2.3. Design by Committee

The *Design by Committee* antipattern [5] refers to a complex software design which is the product of a committee process. Some of its symptoms are an overly complex design documentation, the requirements and the design lacking convergence and stability, and architects and developers having conflicting interpretation of the design.

McCloghrie [32], Rose [41], and Stallings [46] explain in detail the adverse effect that this antipattern had on the design of SNMPv2, resulting in multiple versions and their rejection by the market.

4.2.4. Reinvent the Wheel

The *Reinvent the Wheel* antipattern [5] appears when custom software systems are built from scratch instead of taking advantage of reuse. Typical symptoms are closed system architectures, replication of commercial software functions, and immature and unstable architectures—because engineers make mistakes. Furthermore, systems designed from the ground up usually take longer to develop, which makes them more expensive.[7]

It would not be fair to say that by developing SNMP, the IETF reinvented the wheel. Ten years ago, there was no application layer protocol suitable for network management. There was no distributed object technology—such as OMG's CORBA or Microsoft's DCOM [50]—either.

But today, would a developer new to network management (any kind of newcomer, not necessarily a recent graduate) not be right in considering SNMP-based solutions to be instances of Reinvent the Wheel? After all, SNMP keeps developers from falling back on middleware and protocols they master when developing network-management software.[8]—The answer is no. Because plain and simple, SNMP is just one case of the more general problem of legacy systems.

---

7. Another cost factor is training: Reinvent the Wheel often leads to domain-specific or proprietary solutions that people are not trained for.
8. At least this avoids the Golden Hammer, see Section 4.2.2.

5. A Patterns View of JAMAP

Several proposals for Web-based network management have been made so far, including WBEM [11, 53], Wellens and Auerbach [54], and Anerousis [2]. In this thesis, we refer to the proposal by Martin-Flatin [28, 29], the JAva MAnagement Platform (JAMAP) [30].

JAMAP is partially realized in a prototype that is also called JAMAP [4]. This prototype is the topic of the next chapter.

In the remainder of this chapter, we characterize JAMAP in terms of design patterns. We start with a coarse-grained view and refine it in the course of the text. In the last two sections, we discuss a design pattern that has no JAMAP counterpart yet as well as an antipattern.

5.1. The Big Picture

Three patterns convey an overview of the JAMAP architecture: Pipes and Filters, Layers (see Section 4.1.6.), and Model-View-Controller (see Section 2.1.3.).

The *Pipes and Filters* pattern [6] provides a structure for systems that process a stream of data. It processes a data stream coming from a data source and going to a data sink by enriching, refining, or transforming the data within filters connected by pipes as depicted in Figure 5-1.

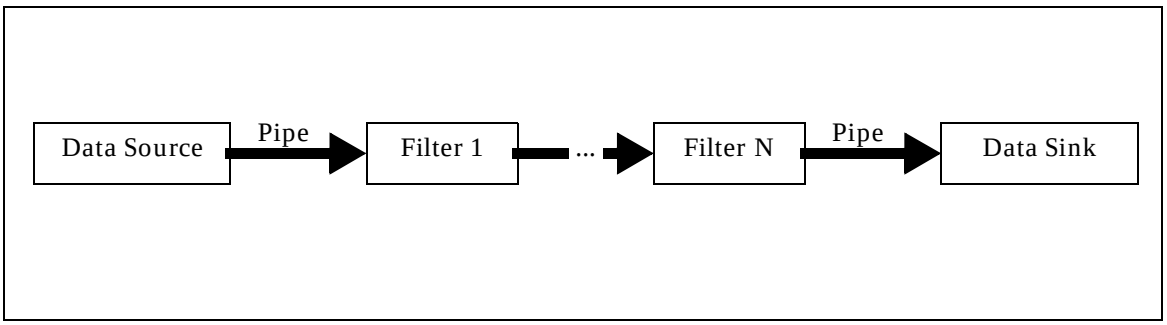

Figure 5-1: Processing Pipeline

The activity of a filter can be triggered by the subsequent pipe pulling output data from the filter or the previous pipeline pushing input data to the filter. Alternatively, the filter can be active in a loop pulling its input from and pushing its output down the pipeline.

Pipes can synchronize two active components. If only one of the adjacent filters is active, the pipe can be implemented by a direct call from the active to the passive component. (In the Decorator pattern [see Section 5.3.2.], the Decorators are filters that activate each other directly, without the intervention of pipes.)

The data source can either actively push data to the first processing stage or passively provide data when the first filter pulls. An active data sink pulls results out of the preceding processing stage, whereas a passive data sink allows the preceding filter to push the results into it.

Pipes and Filters. While there are many pipes and filter components in JAMAP (e.g., HTTP connections and pushed-data interpreters), their order remains static. For example, a pushed-data collector is always followed by a pushed-data filter. Thus, this pattern has primarily been applied for a clear separation of concerns, and not to allow for new kinds of information processing by reordering the filters.

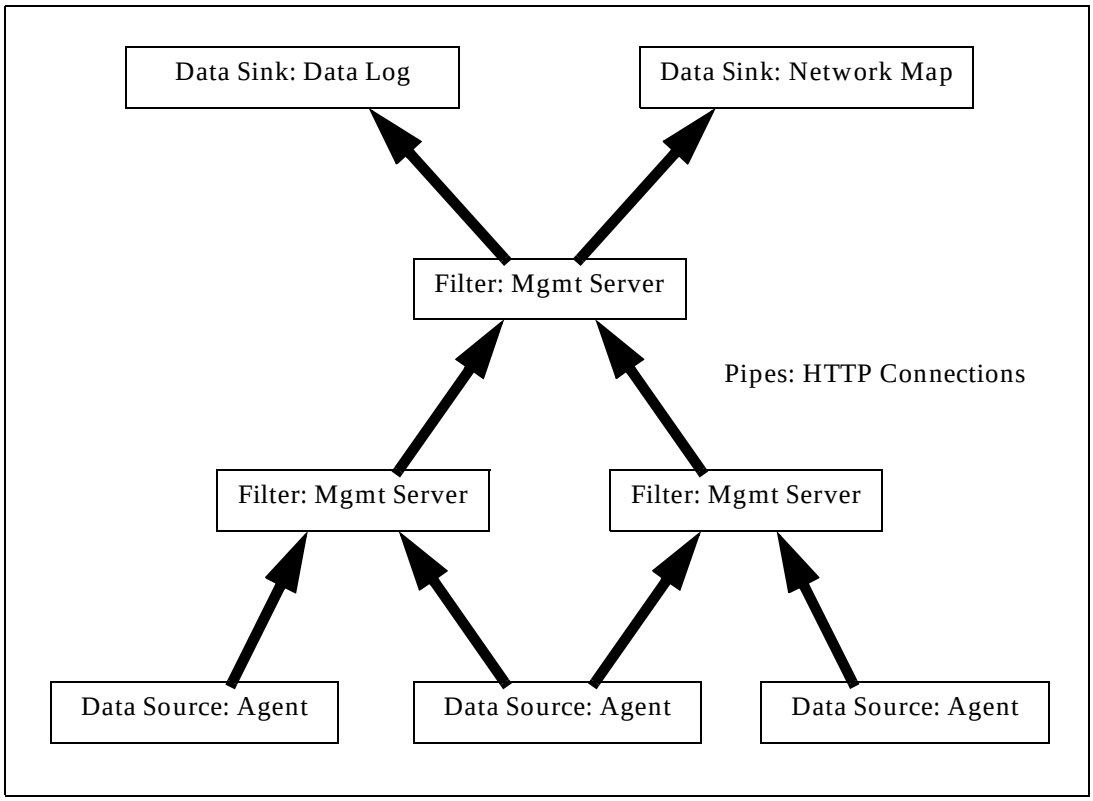

Figure 5-2: High-Level View

Unlike the simple processing pipeline depicted in Figure 5-1, both the high-level view of the management system (see Figure 5-2) and the mid-level view of the pushed-data servlet (see Figure 5-3) show that the JAMAP filters can have several input or output pipes.

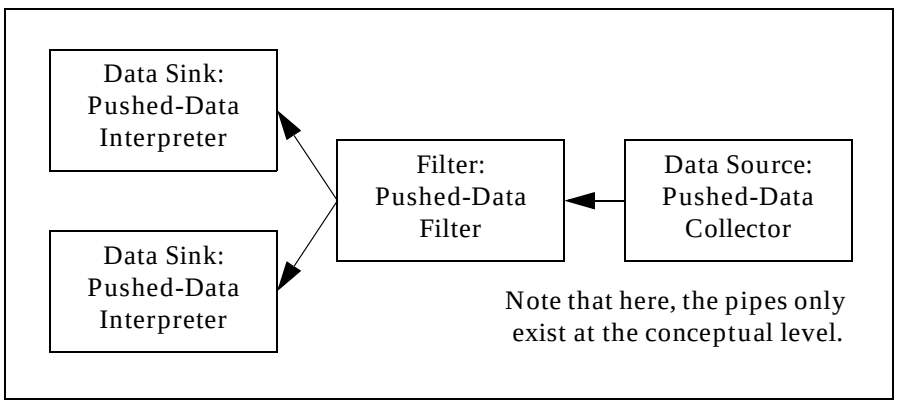

Figure 5-3: Mid-Level View

Layers. Both the pipes and the filters are instances of the Layers pattern (see Figure 5-4). The communication relies on HTTP at the application layer, TCP at the transport layer, and IP at the internet layer. The applets and the servlets are layered on

top of the Java Virtual Machine, which in turn is layered on top of the host's operating system.

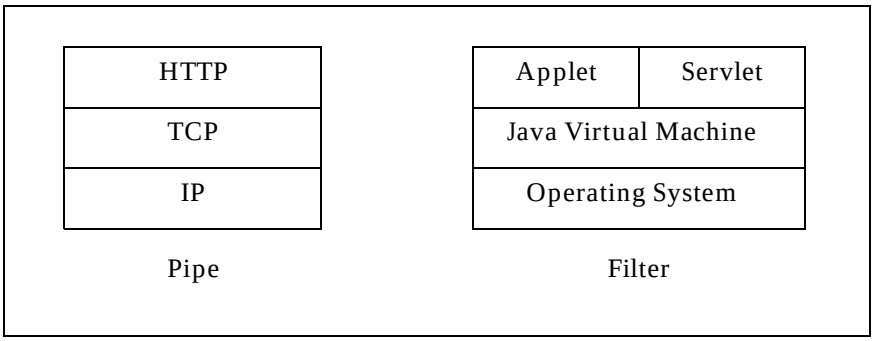

Figure 5-4: Layered Pipes and Filters

Model-View-Controller. An agent can be managed by one or more management stations, i.e., a manager monitors (view) and controls (controller) the agent (model). To help the administrators in their duties, monitoring can be supported by different views. And the controllers can depend on the administrator's privileges to control the network. Unlike the typical case discussed in Section 2.1.3., we have several models (i.e., agents) in JAMAP (see Figure 5-5).[1]

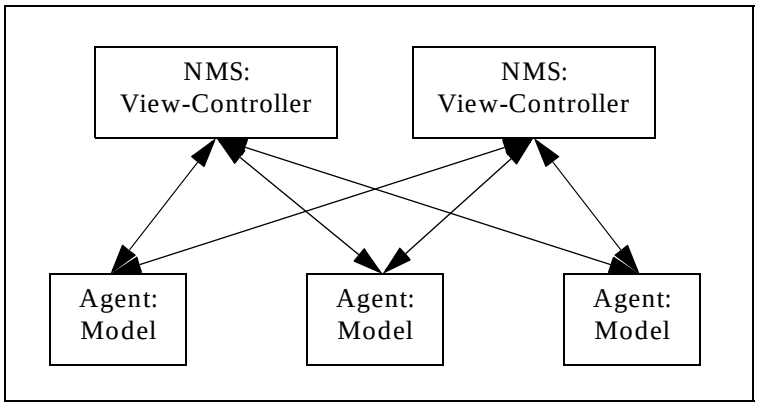

Figure 5-5: MVC in JAMAP

5.2. Information Flow

A core aspect of JAMAP is the push-based communication and the possibility to dynamically connect management servers with managed nodes and management stations with management servers. Both issues are addressed by the Observer pattern and one of its variants, the Strict Propagator.

To log information, JAMAP uses JDBC [23], an example of the Bridge pattern (see Section 4.1.3.), which keeps the management server(s) and the data server(s) independent. Changing one does not affect the other. In SNMP-based network management, buying a network management platform reduces the choice of database management systems because of peer-to-peer agreements between vendors [27].

The *Observer* pattern [16] defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated

1. And unlike SNMP-based management where typically only one NMS manages an agent, we have several views and controllers in JAMAP.

automatically. It is also known as the *Publisher-Subscriber* pattern [6] and is depicted in Figure 5-6.

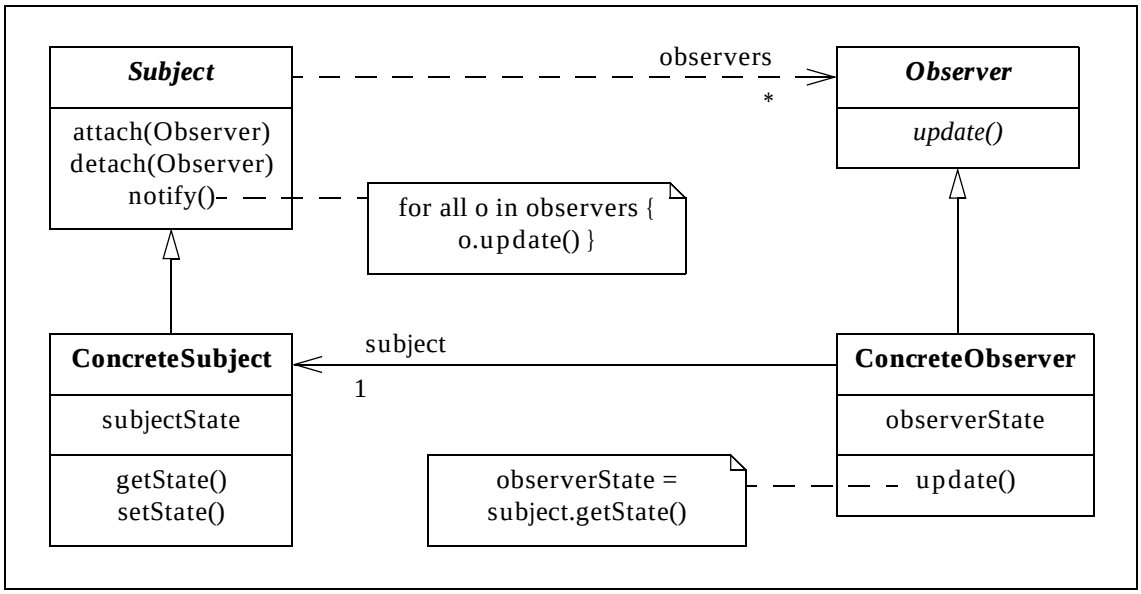

Figure 5-6: Observer (adapted from [16])

The Observer pattern is an ingredient of the MVC (see Section 2.1.3.). The `Model` corresponds to the `Subject` and a `View-Controller` pair to an `Observer` object. But the use of the Observer pattern is of course not limited to GUIs as we shall se shortly.

An interaction between a `Subject` and `Observer`s is depicted in Figure 5-7. After an `Observer` object changed the state of the `Subject`, the `Subject` notifies every `Observer` that then fetches the state.

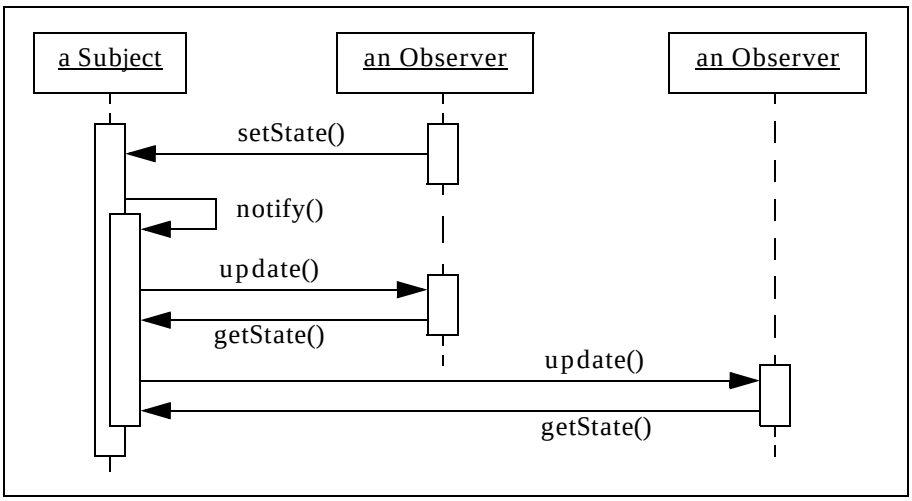

Figure 5-7: Subject-Observer Interaction (adapted from [16])

Gamma *et al.* discuss several points with regard to the Observer pattern, three of which are of particular interest to us [16, pp. 297-298]:

> *"2.* Observing more than one subject. *It might make sense in some situations for an observer to depend on more than one subject. [...] It's necessary to extend the*

Update interface in such cases to let the observer know which *subject is sending the notification. [...]*

6. Avoiding observer-specific update protocols: the push and the pull models. *Implementations of the Observer pattern often have the subject broadcast additional information about the change. The subject passes this information as an argument to Update. The amount of information may vary widely.*

At one extreme, which we call the ***push model****, the subject sends observers detailed information about the change, whether they want it or not. At the other extreme is the* ***pull model****; the subject sends nothing but the most minimal notification, and observers ask for details explicitly thereafter.*

[...]

7. Specifying modifications of interest explicitly. *You can improve update efficiency by extending the subject's registration interface to allow registering observers only for specific events of interest. When such an event occurs, the subject informs only those observers that have registered interest in that event."*

To perform regular management in JAMAP, management stations observe (i.e., monitor) more than one agent, state information is transmitted in a push fashion, and management stations explicitly subscribe to MIB data and notifications they are interested in.

Even though being correct, this view is too high-level for implementational purposes: Pushed data is first received by management servers, and management stations observe the network map registry.

The *Propagator* [12] is a family of patterns related to the Observer that works with an arbitrary dependency network (vs. Observer's single level of dependents). The *Strict Propagator* is the family member of interest to us and is depicted in Figure 5-8.

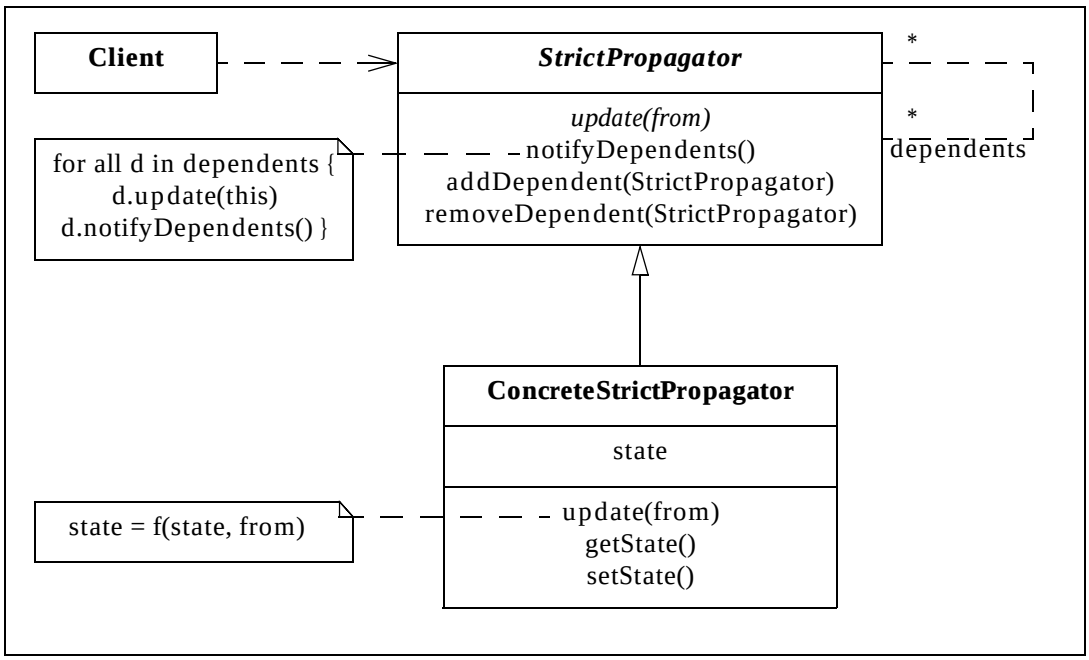

Figure 5-8: Strict Propagator (adapted from [12])

A `StrictPropagator` object can act as an `Observer` and a `Subject` at the same time. This makes is possible to both observe and be observed.

In JAMAP, agents are `Subject`s and are only observed. The management servlets running on the management server are `StrictPropagator`s: they both observe (the agents) and are observed (pushed-data servlets by the events servlet, the event servlet by the management stations). Management stations are `Observer`s and are not observed themselves.

Buschmann *et al.* briefly treat three variants of the Observer. One of them can be of interest to certain realizations of JAMAP [16, p. 341]:

> "Gatekeeper. *The Publisher-Subscriber pattern can be also applied to distributed systems. [...] In one process a component sends out messages, while in the receiving process a singleton 'gatekeeper' demultiplexes them by surveying the entry points for the process. The gatekeeper notifies event-handling subscribers [Observers] when events for which they registered occur."*

The Gatekeeper pattern, also known as the Reactor patter [43], should be considered in cases where threading is not or only poorly supported. In theory, this should not be too big an issue in Java. In practice, the performance of Java threading can be poor and necessitate a reimplementation based on the Gatekeppper, because either the thread support by the operating system or the thread implementation of the Java run-time system is poor.

5.3. Servlets Patterns

5.3.1. Singleton

The *Singleton* pattern [16] ensures that a class has one instance only and provides a global point of access to it. It is depicted in Figure 5-9.

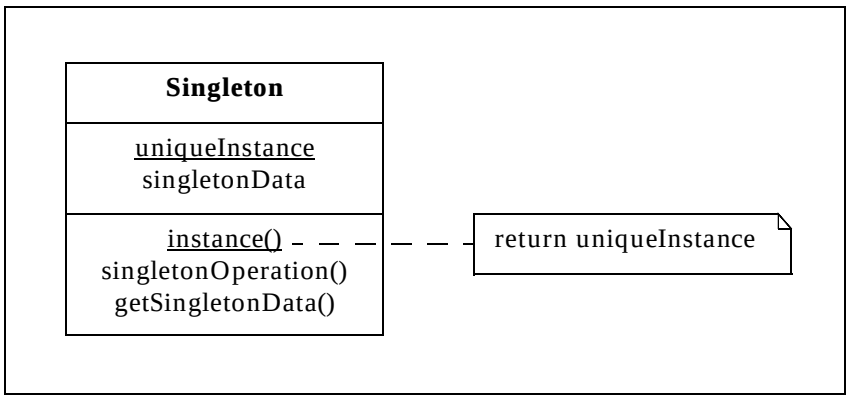

Figure 5-9: Singleton (adapted from [16])

A well-known example for `Singleton`s are abstract data structures. Unlike, for instance, a list defined as an abstract data type, a list defined as an abstract data structure must be shared by all its clients. Different programming languages provide different ways of realizing them, e.g., using modules or namespaces, hiding the constructor, or making the whole class static.

A server supporting servlets instantiates a servlet at most once, i.e., servlets are `Singleton`s. Note, however, that if a servlet thread is busy servicing a request when another one arrives, a new servlet thread will be forked to serve the arriving request. The crucial point is that all these threads share the same servlet state, at whatever point of execution they might be.

5.3.2. Decorator

The *Decorator* pattern [16] dynamically provides an object with additional responsibilities, without using inheritance. It is depicted in Figure 5-10.

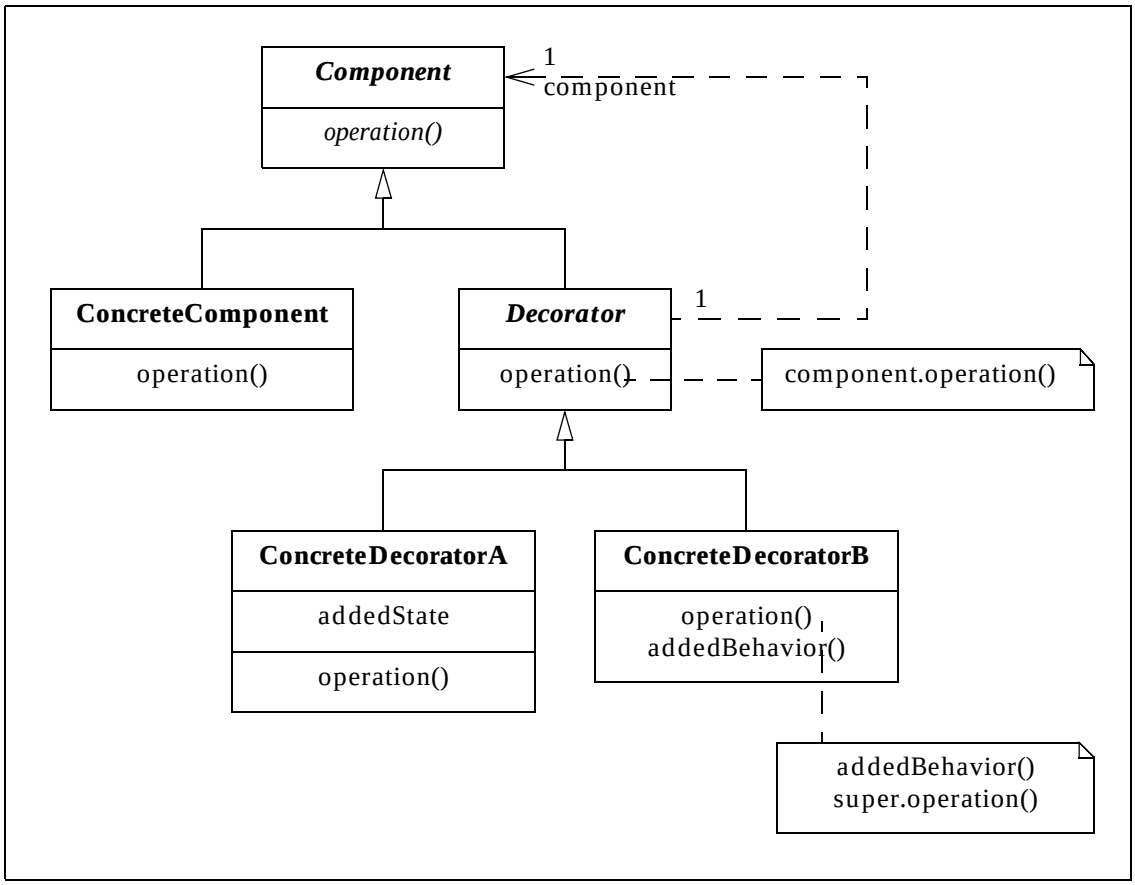

Figure 5-10: Decorator (adapted from [16])

For example, if we want to make a file stream class and a network stream class (that both inherit from the same abstract stream class) more secure, we can subclass each of them to define an encrypted file stream class and an encrypted network stream class. If, additionally, we want streams that compress data, we can equally well define a compressed file stream class and a compressed network stream class. What about file streams that are both compressed and encrypted? Supplying all the desired variants gets quickly out of control.

By applying the Decorator pattern, we define a compressed stream class and an encrypted stream class that can work with *any* kind of stream. A compressed and encrypted file stream can now easily be constructed at run time by cascading these three streams.

This is exactly how Java streams are realized [22]. An abstract class called `InputStream` (`Component`) is the superclass of all input streams. Concrete subclasses such as `FileInputStream` (`ConcreteComponent`) actually write data to a medium. The abstract subclass `FilterInputStream` (`Decorator`) is the superclass of all classes that transform data, but do not write it to a medium themselves. The `GZIPInputStream` (`ConcreteDecorator`) is a concrete subclass of `FileInputStream`. Output streams are organized analogously. JAMAP intensively uses Java streams and thus the Decorator pattern.

5.3.3. Strategy

The *Strategy* pattern [16] defines a family of algorithms, encapsulates each one, and makes them interchangeable. It is depicted in Figure 5-11.

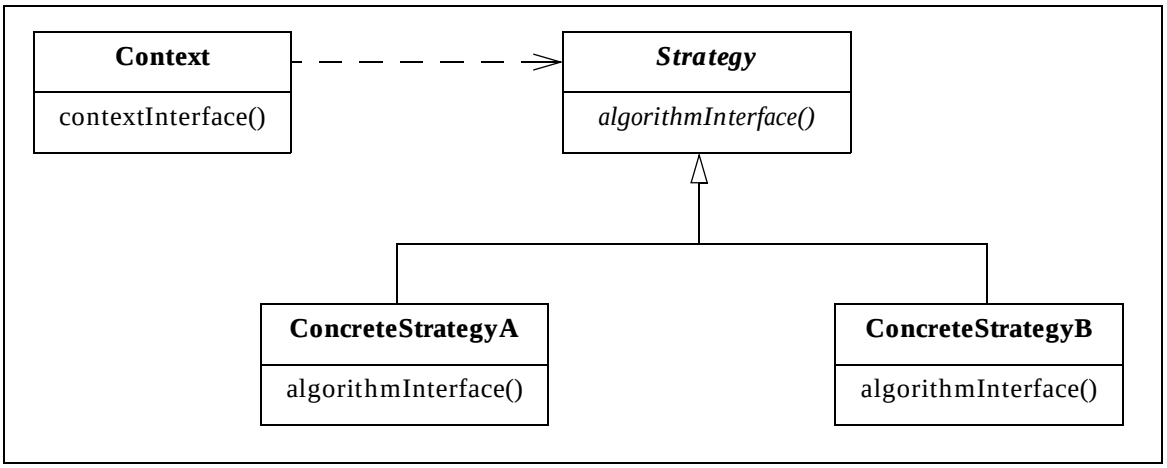

Figure 5-11: Strategy (adapted from [16])

The same task (e.g., sorting) can sometimes be carried out by different algorithms (e.g., bubble sort, quicksort, etc.). If it is desirable for a class to support several such algorithms, they can all be implemented in the same class (and, at a given time, selected through a conditional statement), or the class is subclassed for every algorithm. Alternatively, the Strategy pattern suggests to decouple the varying behavior from the class that exhibits this behavior and to define each algorithm in a class of its own.

Prior to sending data, an agent may compress, encrypt, or authenticate these data (see also Section 5.3.2.). By defining class hierarchies for the compression, encryption, and authentication schemes, we can easily configure a running system to use a new scheme. The receivers (i.e., the pushed-data servlets on the management servers) need to be configured accordingly prior to checking the authentication of, decrypting, or decompressing these data.

The flexibility offered by the Strategy pattern is useful in the pushed-data interpreter. A rule can be created or modified (with the rule editor applet) whenever an administrator deems it necessary. In particular, a rule can be deployed long after the pushed-data interpreter has taken up its work.

Note that in the Strategy pattern, all `ConcreteStrategy` classes can work with the same input data and only differ in how they compute on it (and possibly the output data they generate). In JAMAP, the interpreter rules may also differ in what input data they understand (SMI-based, XML-based, etc.).

An event correlator can provide a quite similar rule creation and modification mechanism that allows administrators to define and apply new correlation rules at run time.

5.3.4. Builder

The *Builder* pattern [16] separates the construction of a complex object from its representation so that the same construction process can create different representations. It is depicted in Figure 5-12.

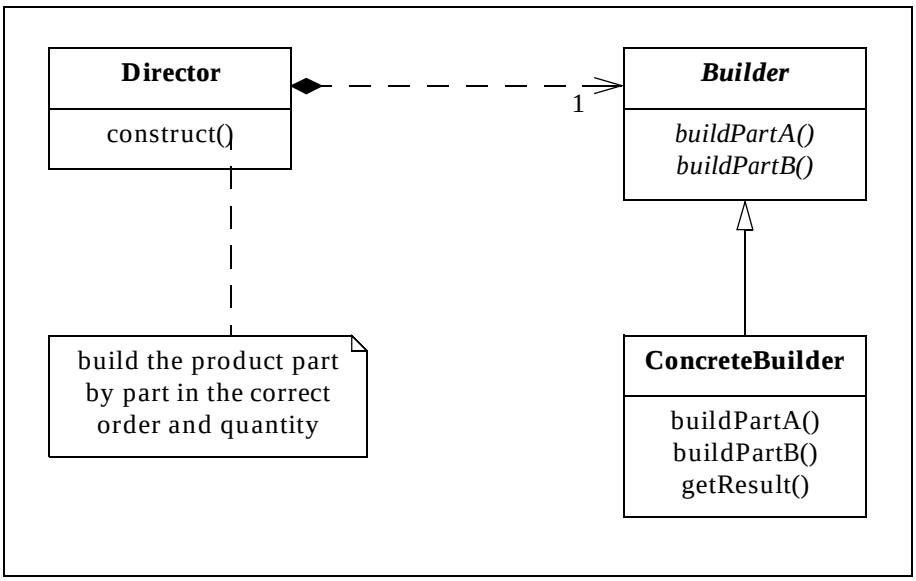

Figure 5-12: Builder (adapted from [16])

A `Director` acts on behalf of a client object (e.g., a development environment) that asks it to build a complex product (e.g., machine code) after providing it with the necessary data (e.g., a parse tree) and the respective `ConcreteBuilder` (e.g., for PowerPCs). The client knows the `ConcreteBuilder`, so it knows how to extract the end product (i.e., what method to call).

An application of the Builder pattern in JAMAP (see Figure 5-13) is the separation between the push scheduler and the MIB data formatter. The `PushScheduler` merely asks the `MIBFormatter` it is given (e.g., the first time an HTTP GET request is serviced by the servlet) to return an array of bytes representing the variable specified with the OID, so that the variable can be embedded in a MIME part. Note that the constructed object is not complex in this case, but the separation allows for different types of representation, including XML, strings, etc.

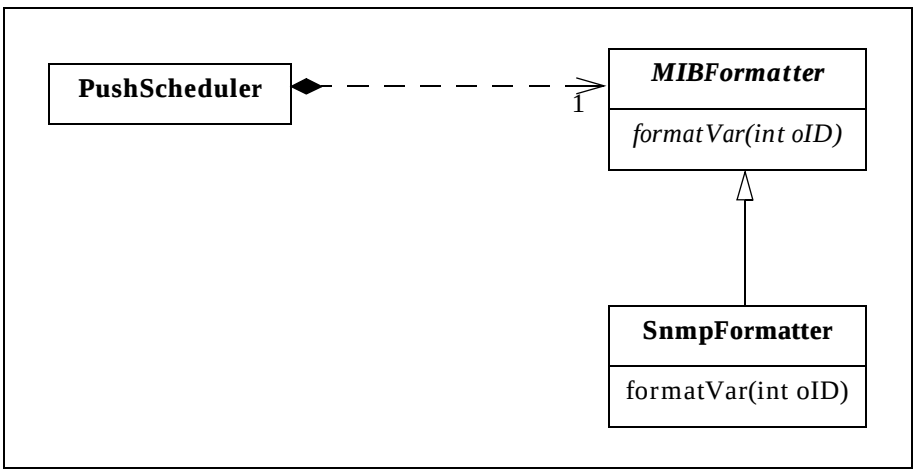

Figure 5-13: MIB Formatter

5.4. Command Pattern

The *Command* pattern [16] decouples the invoker of a request (command) from the receiver by encapsulating requests as objects. Different invokers can be parameterized with the same request (thus leading to the same result), and the same invoker can be parameterized with different requests at different times. Because requests are first-class objects, they can easily be queued or logged, e.g., to support undoable operations. The Command pattern is depicted in Figure 5-14.

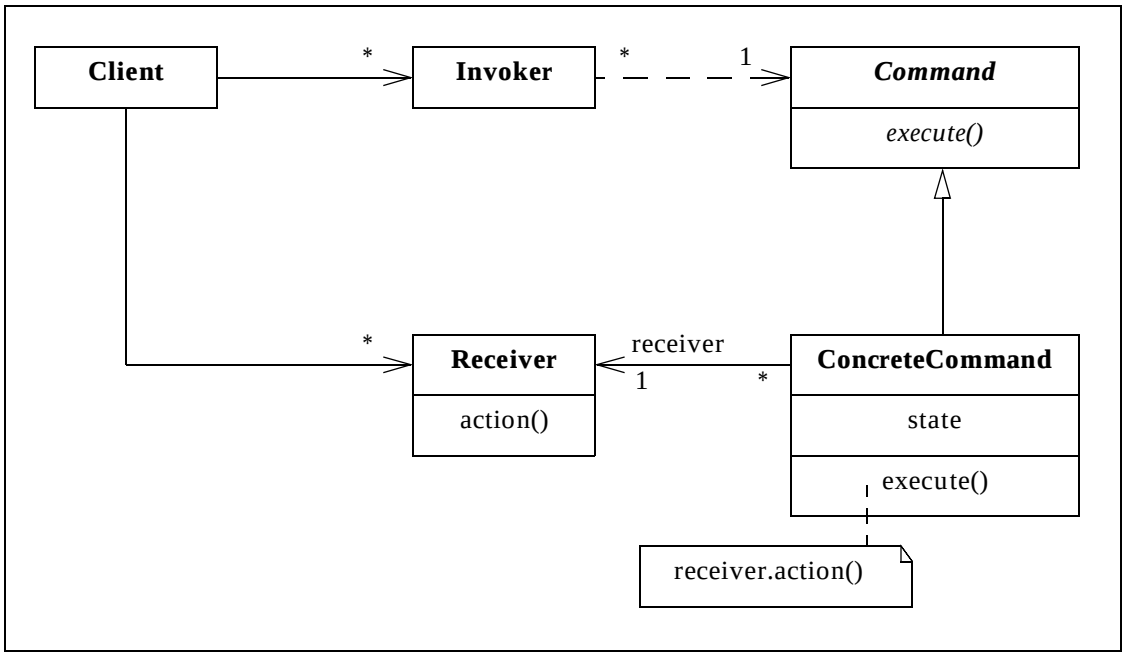

Figure 5-14: Command (adapted from [16])

For example, a word processor (`Client`) can parameterize both a menu item and a button in a toolbar (`Invokers`) with the same `ConcreteCommand` (e.g., "cut"). Beforehand, the word processor has given the `ConcreteCommand` the identity of the current document (`Receiver`). The `ConcreteCommand` also knows whether its widgets should be enabled or not and whether it has to put itself on an undo stack after execution. An interaction is depicted in Figure 5-15.

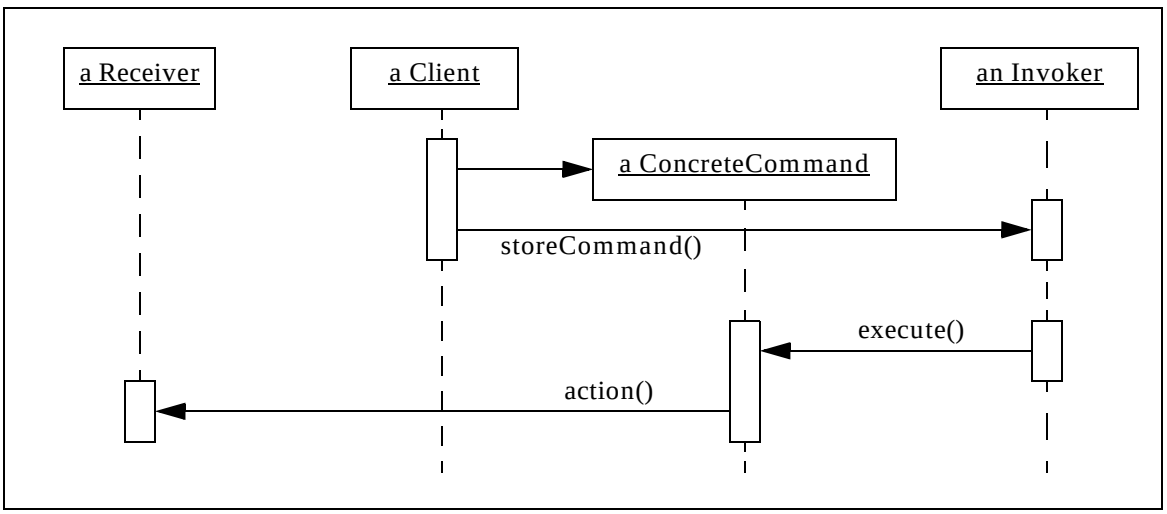

Figure 5-15: Command Interaction (adapted from [16])

The Command (along with the *Command Processor* pattern [6]) is certainly a viable pattern for the GUIs in JAMAP[2], but GUIs are—with the exception of the management-specific GUI discussed in Section 4.1.8.—not our concern in this thesis. Nevertheless, the Command pattern can inspire us to delegate management tasks to agents by sending them `Command`-like objects they execute locally.[3] Such a `Command` object can even be composed of other `Command` objects as suggested by the MacroCommand pattern (see Figure 5-16), a special case of the Composite pattern (see Section 4.1.4.).

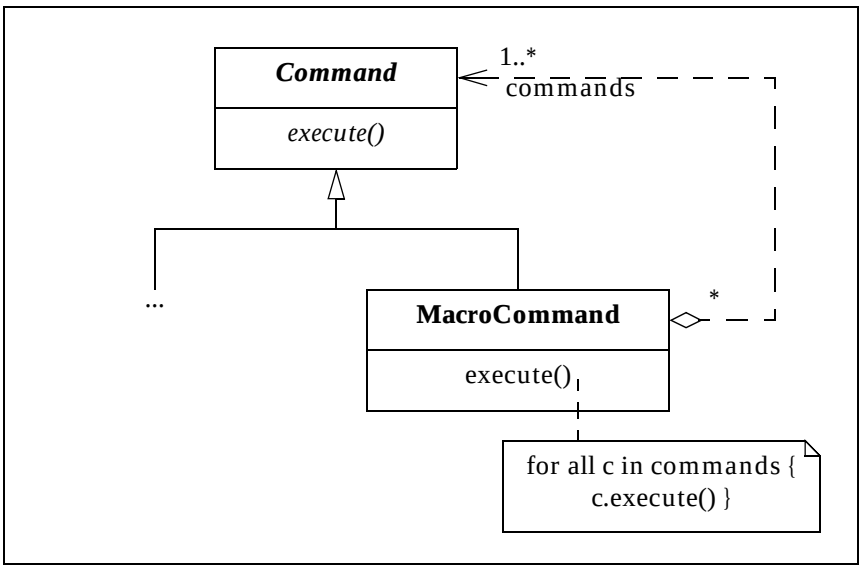

Figure 5-16: MacroCommand (adapted from [16])

5.5. Continuous Obsolescence AntiPattern

Continuous Obsolescence [5] stems from technology changing so rapidly that developers have trouble keeping up with the current version of software and finding combinations of product releases that work together.

As Brown *et al.* point out [5, p. 85]: *"Java is a well-known example of this phenomenon, with new versions coming out every few months."*

JAMAP relies heavily on Java and is thus indirectly prone to Continuous Obsolescence. For example, what used to be called Embedded Java has now evolved into the Java Platform Micro Edition (J2ME) [47]. This is of concern to developers developing agents, but the problem seems to be local. At first sight, it even seems like all the servlets (on both the managers and the agents) in a given network can be based on different releases of Java without causing interoperability problems.

Appearences are deceiving: wherever Java serialization is used, chances are that objects cannot be deserialized. Every class automatically has a (`static final long`) version number called `serialVersionUID` which is a function of the field declarations and the method signatures of the class [22]. If the interface changes, then so does the `serialVersionUID`. And if at the time of deserialization the `serialVersionUID` of the object and the class do not match, an exception is thrown. Nevertheless, the version number can manually be set to the old one if the new class implementation is compatible to the old.

2. The more so as the Java platform supports it through the `Action` interface [22].
3. This is an example for the *remote evaluation* paradigm of mobile code [15].

6. Analysis of the JAMAP Prototype

The JAva MAnagement Platform (JAMAP) is a prototype of a management platform for Web-based network management originally developed by Laurent Bovet [4][1]. A high-level overview of JAMAP is given by Martin-Flatin *et al.* [30].

JAMAP encompasses about 60 classes and 10 interfaces [4]. The agent-side classes realize what is necessary for data subscription and data dispatching. Notifications are not fully supported yet. The manager-side classes realize what is necessary for pushed-data and elementary event handling.[2]

The core characteristics of JAMAP are (i) the HTTP-based communication, (ii) the use of applets and servlets, and (iii) the use of serialized Java objects for the data exchange. Between managers and agents, the serialized objects are embedded in MIME messages.

In the remainder of this chapter, we discuss JAMAP's support for distributing the management platform, take a closer look at two of the servlets, and point out implementation and design limitations that do not affect the overall architecture of JAMAP. The last two sections are primarily destined for people working with the source code.

For the sake of simplicity, we assume here that the reader is familiar with the above-mentioned documents. If need be, JAMAP is explained in detail in [4, 30].

6.1. Distributed Network Management Platform

As shown in Figure 6-1, we will typically have several agents (dozens or hundreds), a few (or only one) management servers featuring the pushed-data servlet, one management server featuring the events servlet, and one or more management stations running the network-map applet and possibly data-subscription applets. The events servlet and one pushed-data servlet can of course run on the same management server.

Agents have the possibility to send data to more than one pushed-data servlet. This is primarily for reasons of redundancy, i.e., all pushed-data servlets connected to a given agent would typically receive the same data. But in order not to limit the application, we also want the possibility to send different data to the pushed-data servlets.

Similarly, pushed-data servlets have the possibility to send data to more than one events servlet. While allowing only one events servlet in a given network solves the problem of maintaining consistency between several network-map registries (because

1. *Personal remark:* I am very impressed by Laurent Bovet's work. In addition to designing and implementing a runnable prototype using classes from different origins, he also set up an HTTP server and had some remarkable ideas I would probably not have had. The shortcomings I discuss are in my opinion largely due to a lack of time and to the experimental nature of his work. Any critique is meant to be purely technical and by no means personal.
2. The JAMAP documentation [4] uses the term "notification" to denote SNMP notifications and the term "event" to denote events generated by the pushed-data filter or the pushed-data interpreter. In general, these terms are often interchangeable in network management.

it does not arise in the first place), we may well like to experiment with a new events servlet (e.g., with a new event correlator) without having to disable the current one.

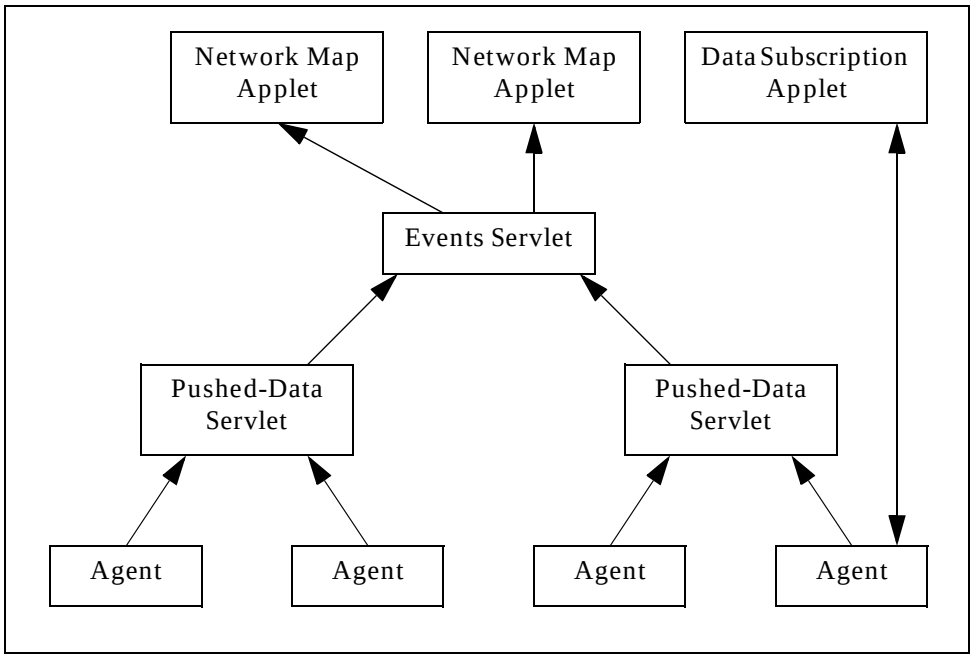

Figure 6-1: Management Nodes

JAMAP already allows for more than one pushed-data servlet or events servlet. However, pushed-data servlets connected to the same agent will receive the same data. This is because of how subscriptions are designed and implemented.

At the time of subscription, an instance of a `Subscription` subclass is created. A `Subscription` object (its dynamic type is not of relevance right now) contains three fields: `objectID`, `objectType`, and `subscriberID`. A `subscriberID` encapsulates a `collectorID` and a `consumerID` (see Figure 6-2).

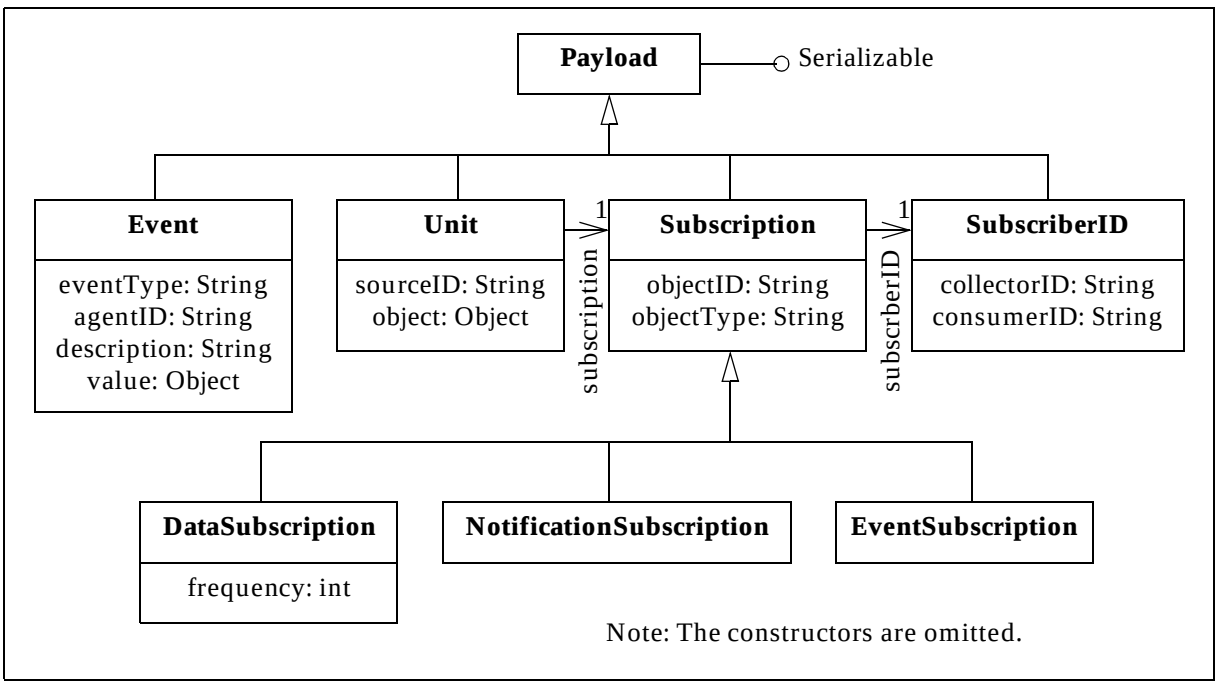

Figure 6-2: Payload and its Subclasses

The `objectID` denotes the identifier (ID) of the variable or notification to subscribe to. The `objectType` denotes the type (e.g., "SNMPNotification"). Both values are completely ignored, though! The reason they can be ignored—and the reason that only a `SubscriberID` object is necessary for unsubscription—is that `consumerID` is in essence a concatenation of the object type and the object ID. The `collectorID`, finally, is simply the ID of the agent.

We doubt the usefulness of the consumer ID[3], but the idea of collectors is a very powerful one. A pushed-data servlet will receive variables (and notifications) depending on which collector ID it used when establishing the connection to the agent. In other words, this is an example of the Observer pattern [16] with multicast (and not broadcast) semantics. But to unleash its power, the collector ID must not be set to the agent ID (as is currently the case), otherwise the pushed-data servlet either receives all the data or none at all. We therefore suggest that the `collectorID` field replaces the `subscriberID` field in `Subscription` and that `SubscriberID` is discarded.

Currently, the events servlet has to run on the same server as the (single) pushed-data servlet to which it connects. This is because there is no means yet to tell it to which pushed-data servlet to connect. We suggest to develop an applet through which an administrator can subscribe the events servlet to pushed-data servlets, by giving it a list of URIs, for instance. If the events servlet cannot store this list locally, a similar solution as for the agents should be envisioned (see Section 6.2.). Of course, the same mechanism can be used to tell pushed-data servlets to which agents to connect.

In addition to fetching a variable value once, data-subscription applets can also subscribe to variables of their agent. The exact same mechanism as for pushed-data servlets can be used, namely that the applet connects to the agent with a certain collector ID. An alternative would be that applets use the pull model when they are used for ad-hoc management. The waste of bandwidth (compared to using the push model) is only temporary.

6.2. Agent

The central piece of the agent is the `PushDispatcher` servlet which can service HTTP GET and POST requests. Upon instantiation, the `PushDispatcher` object tries to load a `SubscriptionTable` object from the local file system. If this fails, an empty one is created.

POST requests can either contain a subscription or an unsubscription (coming from the data subscription applet). The `SubscriptionTable`[4] is modified accordingly and stored on the local file system (if possible).

Not all agents will have the possibility to locally store their `SubscriptionTable`. We therefore suggest that they only provide a small non-volatile memory to store a URI. This administrator-assigned URI denotes the location and the name of the serialized `SubscriptionTable`. At the time of initialization, the agent requests (HTTP GET) the `SubscriptionTable`. By communicating the same URI to several agents, they will all use the same `SubscriptionTable`. Through the data

3. Currently, it is used to assign an interpreter rule to a variable. But each monitored variable having an interpreter rule of its own will not scale well.
4. For the sake of simplicity, we refrain from specifying whether we mean the class or an instance when it is clear from the context.

subscription applet, an administrator can instruct an agent to store (another HTTP request to the same URI) the `SubscriptionTable` or to fetch it anew. And if the administrator wants several agents to fetch it anew, she can write a simple script that issues the necessary HTTP request to the these agents.

Each HTTP GET request results in the creation of a `PushScheduler` and in the invocation of its endlessly looping `push` method. This method pushes variables to the issuer of the GET request with the help of a `MultiObjectOutputStream`.

Being designed for an SNMP agent, the `PushScheduler` is SNMP specific. Furthermore, it only works for subscriptions whose frequency—strictly speaking: period—is a multiple of 5 seconds (hard-coded).[5] The developer must have been aware of this problem, for in one of the comments, he hints at using the greatest common divisor of all subscription periods to determine when to check the subscriptions next; this is a very good idea that should be pursued.

We suggest that the `PushScheduler` should be designed to be MIB independent, allowing it to be used in any kind of agent (e.g., CIM MIBs). This can be achieved by separating the currently implicit MIB data formatter part according to the Builder pattern [16] (see Section 5.3.4. for more).

When a variable is transmitted, it is wrapped into a `Unit` (see Figure 6-2)—along with its subscription—, and the `Unit` is then pushed by invoking the `writeObject` method of the `MultiObjectOutputStream`. This method, however, always appends a MIME separator which supposedly denotes the end of a push cycle.

The format we suggest to use in the future is depicted in Figure 6-3. The `PushScheduler` tells the `MultipartObjectOutputStream` to write the part header. It then tells it—for all variables in the push cycle—to write the object ID as well as the array that the MIB data formatter returned. And then it tells it to write the MIME separator.[6]

The object ID is a 32-bit, signed, big-endian integer (i.e., a Java `int`). The byte array can represent a serialized Java object, an XML file in UTF encoding, etc. A "smart" pushed-data collector will be able to determine where one variable ends and the next (or the MIME separator) starts by looking up the MIME type and interpreting the byte array accordingly.

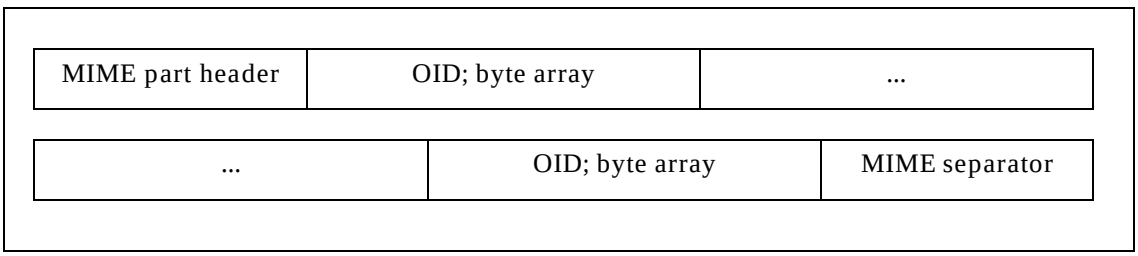

Figure 6-3: Format of MIME Parts

5. For practical management purposes, a resolution in minutes should suffice.
6. For all these kinds of output, the `MultiObjectOutputStream` can use an output stream that implements the `java.io.DataOutput` interface.

6.3. Manager

The central pieces of the manager are the `PushedDataCollector` servlet and the `EventManager` servlet.

As both Bovet [4] and Martin-Flatin *et al.* [30] indicate, a full-blown events servlet is a project of its own. We thus confine our study to the pushed-data servlet, but only after suggesting to add a filter between the events and pushed-data servlets. The task of this filter will be to translate events (e.g., a SNMP notification the notification filter forwards) in case their format does not conform to the one expected by the event correlator.

Upon instantiation, the `PushedDataCollector` creates a `MibDataLogger`. The latter is a helper object to log *SNMP* variables on the *same* server (hard-coded). These are two liabilities that are undesirable for a general-purpose pushed-data servlet. However, the only reason the `MibDataLogger` is SNMP-specific is that it contains an unnecessary cast to a third-party SNMP-variable type (see also Section 6.5.), so this can easily be fixed.

HTTP POST requests can contain one of three commands (coming from the rule editor applet). The "reinit" command tells the `PushedDataCollector` to connect to an agent to which it may not be connected yet. (When it starts to run, it is connected to no agent at all. Once connected to an agent, it cannot be disconnected on command.) The "reconnect" command tells it to connect to an agent to which it is already connected after closing the current connection. The "postrule" command, finally, tells it to compile and instantiate a rule for a given agent (collector) and a given variable (consumer). If necessary, the `PushedDataCollector` first connects to the agent.

Each HTTP GET request results in the creation of a `PushForwardConsumer` and in the invocation of its endlessly looping `waitAndPush` method. This method pushes events to the issuer of the GET request with the help of a `MultiPartObjectOutputStream`. The `PushForwardConsumer` is registered with the `ForwardConsumer` (multiplexer) of the `PushedDataCollector`. The `ForwardConsumer` receives the events of all the `PushedDataFilter`s and `PushedDataInterpreter`s.

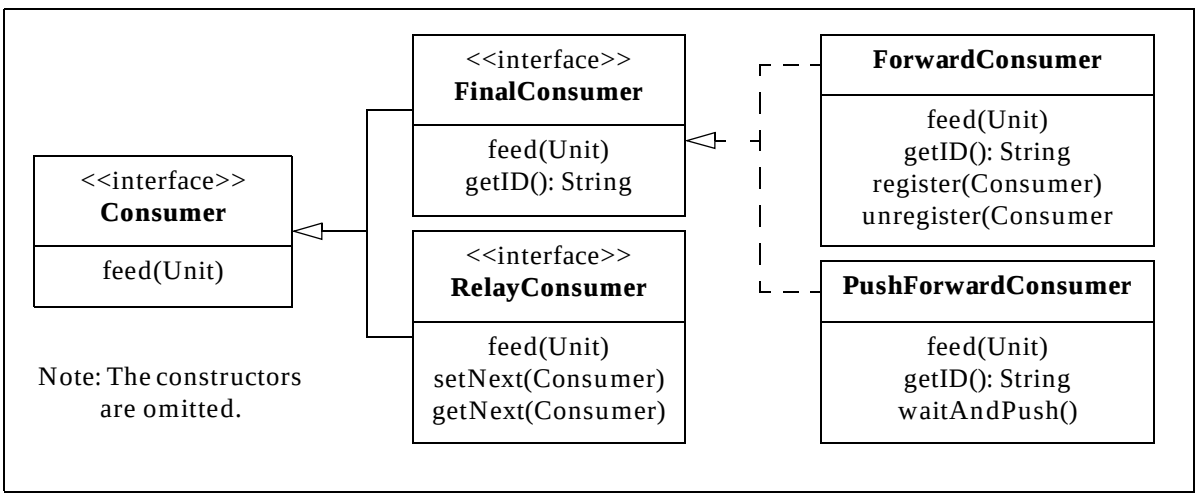

Figure 6-4: Consumer & Co.

The problem is that JAMAP `Consumer`s handle `Unit`s (see Figure 6-4). So every `Event` first needs to be wrapped into a `Unit` before being serialized and sent through the network. This wastes both time and bandwidth. And we can spare neither when things go wrong in the network, resulting in many events being generated and transmitted.

6.4. Analysis of the Design

Visibility. The fields of the JAMAP classes are either declared `public`[7] or nothing at all (i.e., neither `private` nor `protected`). In Java, the default is package visibility which most certainly is not what was intended. All classes in the same package can access and modify those non-`private` fields which in turn may hamper the evolution of classes that expose their fields in such a way. Data hiding is a proven principle of programming in general and of object-oriented programming in particular. The Bridge pattern [16] goes even further by not only hiding the implementation, but by decoupling it from the abstraction.

False Family. `PersistentObject` is a JAMAP class that directly inherits from `java.lang.Object` and that implements `java.io.Serializable` without modifying any behavior or adding any fields. The intention seems to be that it acts as the supertype of all serializable JAMAP classes. But there is no need for such a class as this is the role of `java.io.Serializable`. Furthermore, classes that do not inherit from `java.lang.Object` directly or from a `PersistentObject` subclass cannot inherit from the latter anymore, and some classes that could inherit do not. Most important, however, is that the five classes that do do not really belong to the same family (e.g., `PushNetworker` and `DefaultMonitorController`). The Family pattern [36] should not be applied when the purpose of the child classes is quite different from the purposes of their parent and sibling classes.

Wrong Reference. `Payload`[8] (see Figure 6-2) is the superclass of all classes exchanged via the network. Important for these classes is that they are serializable. However, two of these classes—`Event` and `Unit`—each contain a reference to a `java.lang.Object`. At run time, this may cause the raise of an exception if the dynamic type of the object is not serializable. Therefore, these references in `Event` and `Unit` should be replaced by references to `java.io.Serializable`. No generality is lost and errors are already discovered at compile time. (The argument type of the `writeObject` method in `MultiPartObjectOutputStream` can be changed accordingly.)

Reinvent the Wheel. `MIMEPart` and `MultiPartStreamHandler` are a prime example for the Reinvent the Wheel antipattern [5]. They implement functionality that `java.util.StringTokenizer` already provides, namely the division of `java.lang.String`s into substrings wherever certain delimiter characters are found.

Naming. `MultiPartObjectOutputStream` is not a subclass of `java.io.OutputStream` despite its name and despite featuring a `writeObject`

7. We find this to be ok, but object-orientation purists would probably prefer to provide access to those fields through (`final`) accessor methods only.
8. `Payload` could have been declared `abstract`. This raises the interesting question of why `java.lang.Object` is not abstract either.

method with the exact same signature. A naming convention in the JAMAP documentation [4] may also confuse the uninitiated reader: the term "MIB" denotes a SNMP MIB and is not used in a generic sense.

6.5. Analysis of the Implementation

Comments. The biggest problem we faced while studying the source code of JAMAP is that comments are virtually non-existent. It was difficult at times to figure out what the responsibility of a class is, or to understand the inner workings of a method. Note that because we are dealing with a first prototype, this is not necessarily an instance of the Architecture by Implication antipattern [5]. Nevertheless, this may eventually cause the Lava Flow antipattern [5].

Serialization. The `java.io.Serializable` interface is primarily a tagging interface [22]. In other words, classes that implement it let the run-time environment know that it is ok to serialize their objects. And even though serialization can be customized by defining private `writeObject` and `readObject` methods, one does not have to do so as long as the default serialization mechanism is appropriate. Furthermore, as soon as a class implements a certain interface, all its subclasses are of the latter's type as well. Most of the JAMAP classes that implement the `Serializable` interface override `writeObject` and `readObject` even though they do not change the default serialization mechanism. This unnecessarily clutters the source code.

Casts. When working with object references, the only kind of explicit cast that is occasionally necessary (and checked at run time) is the downcast.[9] The JAMAP source code, however, contains both upcasts (e.g., in `EventManager`) and casts of fields to types they are declared as anyway (e.g., in `SubscriptionTable`). While syntactically and semantically correct, this may cause performance penalties if the compiler does not remove the unnecessary casts and is potentially very confusing for someone reading the source code. As for the Functional Decomposition antipattern [5], the cause may be thinking in a non-object-oriented programming language such as C.

Exception Handling. Most `try` blocks in JAMAP are only followed by a single "catch-all" `catch(Exception)` block which simply prints the stack trace. We find that this is ok for a prototype. Nevertheless, one should keep in mind that beyond doing more useful and important things within a `catch` block, we can profit from Java's strong typing to separate the `catch` blocks of different exceptions on the one hand, and not to erroneously catch exceptions that should have propagated out of the method on the other hand.

Consistency. The same MIME part boundary is defined in more than one JAMAP class (i.e., in `MultipartObjectOutputStream` and `MultipartStreamHandler`). Defining the "same" constant in more than one class is very dangerous. Instead, it should be publicly defined in one only class or in an interface without any methods. In the former case other classes can use it by referencing the constant, in the latter case by implementing the interface.

---

9. Downcasts are *not* necessary to determine the method that is performed when an operation is invoked on an object (e.g., as in `MibDataLogger`)—polymorphism takes care of that.

Efficiency. First, when the same reference or value is requested or computed more than once in the same block but does not change in between, it should be assigned to a local variable the first time it is needed. Second, in cases where information has to be read from an input stream but can be processed before being complete, it seems like a good idea to read information until the input stream blocks, to process what has been read (instead of first waiting for all the data), and to go back to reading afterwards. The JAMAP classes that make use of this idea (e.g., `Proxy`) do so by defining an array of length one into which to read. The problem is that with `java.io.InputStream`s, the number of bytes read is at most the number of bytes available without blocking or the length of the array into which to read the bytes—whichever number is smaller.[10] Third, whenever a `String` is constructed by appending several `char`s or `String`s, it is more efficient to use a `StringBuffer` than performing `String` concatenation (e.g., as is done in `MIMEPart` or `MultiPartStreamHandler`).

10. By the way, we came across a bug that only appears on occasion. If the `InputStream` is still blocked at the time of the next read attempt, the old content of the array will be processed again. One therefore not only has to check for a return value of -1 (denoting the end of stream), but also for 0 (meaning no byte read).

7. Conclusion

7.1. Summary and Contributions

In this thesis, we first introduced design patterns and two approaches to IP network management, the Simple Network Management Protocol (SNMP) and Martin-Flatin's proposal for Web-based network management (JAMAP).

We then compiled a list of ten design patterns, summarizing each one and explaining their potential roles in SNMP-compliant network-management software. We also discussed four antipatterns in the context of SNMP-based network management.

Next, we characterized the architecture of JAMAP in terms of design patterns, starting with a coarse-grained view and refining it. Additionally, we presented a design pattern that may lead to a solution for distributing management tasks.

Finally, we briefly reviewed the current JAMAP prototype and made some suggestions for improvement, both at the design and implementation level.

Overall, we provided a software-engineering view of IP network management. We showed that general-purpose design patterns—first found in designs of editors, graphical user interfaces, frameworks, and many other software systems—can support the documentation and design of network-management software.

7.2. Benefits for the Student

Building on what I was taught in the lectures on computer networks, distributed systems, and Web technologies, I learned the foundations of network management. In particular, I learned how IP networks are managed with SNMP and in one instance of Web-based network management.

As to my software-engineering skills, I gained more experience in the design of object-oriented software systems, broadened my knowledge of Java (threading, networking, servlets), and learned the core parts of the Unified Modeling Language. Furthermore, I saw clear examples of what to avoid when developing software while I studied antipatterns.

The main benefit, however, stems from my intensive study of design patterns in the course of this thesis. I now know about 40 design patterns, most of which I can put into practice without having to consult the literature.

7.3. Future Work

The compilation of design patterns for IP network management can be pursued by taking into account WBEM, JMX, and other proposals on the one hand, and further design-patterns literature on the other hand.

It is probably worthwhile to investigate how recent Java technologies—such as JavaSpaces, Jini, and especially the Java Management Extensions—can be incorporated into Web-based network management. It would also be interesting to see whether

JAMAP and the design patterns we suggest contain ideas that have been missed in these Java technologies.

Finally, it would be useful to implement and test the modifications that we suggested for JAMAP. But before doing so, one may want to address the issues of security and delegation of management, which probably require further design modifications.

Appendix A:
Project Description

Diploma Project (M.S. thesis) for 1999-2000: Design Patterns for the Management of IP Networks

---

This project has been assigned to Paul E. Sevinç, M.S. student at ETH Zurich.

Professor: R. Guerraoui (LSE)

Assistant: J.P. Martin-Flatin (ICA), INR 130, Tel. 4668, <jp.martin-flatin@ieee.org>

Hosting lab: ICA

Work place: INR 111

Maximum number of students for this project: 1

Project Description

The management of IP networks (especially intranets) is typically based on a network management platform, such as HP OpenView or Cabletron Spectrum, and vendor-specific management GUIs called add-ons. In this setup, management software is expensive, but hardware is expensive too, since we need a dedicated management station. Transfers of management data between the manager and the agents are based on the SNMP protocol and the manager-agent paradigm [8]. The manager performs some polling at regular time intervals.

A new approach to IP network management, known as "Web-Based management", consists in using Web technologies instead. An applet is uploaded by a Web browser, and management data is exchanged via HTTP (instead of SNMP) between the manager and the agents. This requires that HTTP servers be embedded in all agents. We proposed recently a new model whereby data polling is replaced by push technologies, which saves a lot of network bandwidth [5].

Some work is currently under way at ICA to develop a management platform called JAMAP [6], which supports Web-based management and is based on the push model. The network management model adopted for this platform is fairly simple, though. The purpose of this project is to devise a more comprehensive management framework that would extend significantly the current model. The student will first study how network monitoring, data collection, notification delivery and event handling are currently performed. He/she will then analyze them, and build object-oriented models of these management tasks. Based on the so-called Gang of Four's book [4], he/she will propose some design patterns to model the network management application.

Benefits for the Student

In the course of this diploma project, the student will:
- become an expert in object-oriented design and design patterns
- become familiar with the Unified Modeling Language
- get an in-depth understanding of the management of IP networks
- apply theoretical software engineering concepts to a practical application domain

---

Prerequisites

Prior to starting this project, the student should:
- be enrolled in a computer science or communication systems program (EPFL or Erasmus exchange)
- have a good knowledge of object-oriented methods and Java
- be familiar with the Web
- know the basics of design patterns
- not necessarily know anything about network management

Bibliography

[1] G. Booch. Object-Oriented Analysis and Design. 2nd edition. Addison-Wesley, Menlo Park, CA, USA, 1994.

[2] G. Booch, J. Rumbaugh and I. Jacobson. The Unified Modeling Language User Guide. Addison-Wesley, Reading, MA, USA, 1999.

[3] M. Fowler and K. Scott. UML Distilled: Applying the Standard Object Modeling Language. Addison-Wesley, Reading, MA, USA, 1997.

[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] J.P. Martin-Flatin. "Push vs. Pull in Web-Based Network Management". In *Proc. 6th IFIP/IEEE International Symposium on Integrated Network Management (IM'99), Boston, MA, USA, May 1999*. IEEE Press, New York, NY, USA, 1999.

[6] J.P. Martin-Flatin, L. Bovet and J.P. Hubaux. "JAMAP: a Web-Based Management Platform for IP Networks". Submitted to the *9th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM'99), Zurich, Switzerland, October 1999*. Springer-Verlag, Berlin, Germany, 1999.

[7] P. Sridharan. Advanced Java Networking. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[8] W. Stallings. SNMP, SNMPv2, SNMPv3, and RMON 1 and 2. 3rd edition. Addison-Wesley, Reading, MA, USA, 1999.

References

1. C. Alexander. "The Origins of Pattern Theory: The Future of the Theory, and the Generation of a Living World". *IEEE Software*, 16 (5): 71-82, 1999.

2. N. Anerousis. "Scalable Management Services Using Java and the World Wide Web". In A.S. Sethi (Ed.), *Proc. 9th IFIP/IEEE Int. Workshop on Distributed Systems: Operations & Management (DSOM'98), Newark, DE, USA, October 1998*, pp. 79-90.

3. B. Appleton. "Patterns and Software: Essential Concepts and Terminology". Available at <http://www.enteract.com/~bradapp/docs/>.

4. L. Bovet. *The Push Model in a Java-Based Network Management Application*. Master's Thesis, Swiss Federal Institute of Technology Lausanne, Switzerland, March 1999. Available at <http://icawww.epfl.ch/~jpmf/students.html>.

5. W.H. Brown, R.C. Malveau, H.W. McCormick III, and T.J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, New York, NY, USA, 1998.

6. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Chicester, England, 1996.

7. G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. 2nd edition. Addison-Wesley, Wokingham, England, 1994.

8. W. Cunningham and K. Beck. "Using Pattern Languages for Object-Oriented Programs". In L. Power and Z. Weiss (Eds.), *Addendum to the Proc. 2nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '87), Orlando, FL, USA, October 1987*, p. 16 & pp. 94-95. ACM Press, New York, NY, USA, 1995. Available at <http://c2.com/doc/oopsla87.html>.

9. D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, Reading, MA, USA, 1999.

10. L. Deri. *A Component-based Architecture for Open, Independently Extensible Distributed Systems*. Doctoral Thesis, University of Bern, Switzerland, June 1997. Available at <http://www.iam.unibe.ch/cgi-bin/oobib?scg-phd>.

11. Distributed Management Task Force. *WBEM Initiative*. Home Page. Located at <http://www.dmtf.org/wbem/>.

12. P.H. Feiler and W.F. Tichy. "Propagator—A Family of Patterns". *Proc. 23rd International Conference on Technology of Object-Oriented Languages and Systems (TOOLS '97), Santa Barbara, CA, USA, July 1997*.

13. P. Felber, R. Guerraoui, and M.E. Fayad. "Putting OO Distributed Programming to Work". *Communications of the ACM*, 42 (11): 97-101, 1999.

14. M. Fowler with K. Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 2nd edition. Addison-Wesley, Reading, MA, USA, 2000.

15. A. Fuggetta, G.P. Picco, and G. Vigna. "Understanding Code Mobility". *IEEE Transactions on Software Engineering*, 24 (5): 342-361, 1998.

16. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesely, Reading, MA, USA, 1995.

17. B. Garbinato and R. Guerraoui. "Using the Strategy Design Pattern to Compose Reliable Distributed Protocols". In *Proc. 3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS '97), Portland, OR, USA, June 1997*, pp. 221-232.

18. B. Garbinato. *Protocol Objects and Patterns for Structuring Reliable Distributed Systems*. Doctoral Thesis no. 1801, Swiss Federal Institute of Technology Lausanne, Switzerland, May 1998.

19. G.S. Goldszmidt. *Distributed Management by Delegation*. Doctoral Thesis, Columbia University, New York, NY, USA, 1996.

20. M. Grand. *Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML*. John Wiley & Sons, New York, NY, USA, 1998.

21. M. Grand. *Patterns in Java, Volume 2*. John Wiley & Sons, New York, NY, USA, 1999.

22. C.S. Horstmann and G. Cornell. *Core Java: Fundamentals*. 4th edition. Sun Microsystems Press, Palo Alto, CA, USA, 1999.

23. C.S. Horstmann and G. Cornell. *Core Java: Advanced Topics*. 4th edition. Sun Microsystems Press, Palo Alto, CA, USA, 2000.

24. H. Hüni, R. Johnson, and R. Engel. "A Framework for Network Protocol Software". In *Proc. 10th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '95), Austin, TX, USA, October 1995*, pp. 358-369. ACM Press, New York, NY, USA, 1995.

25. D. Levi. "Introduction to the Script MIB". *The Simple Times*, 7 (2): 5-6, 1999.

26. J.P. Martin-Flatin, S. Znaty, and J.P. Hubaux. A *Survey of Distributed Enterprise Network and Systems Management*. Technical Report SSC/1998/024, version 2, Swiss Federal Institute of Technology Lausanne, Switzerland, August 1998.

27. J.P. Martin-Flatin. *IP Network Management Platforms Before the Web*. Technical Report SSC/1998/021, version 2, Swiss Federal Institute of Technology Lausanne, Switzerland, December 1998.

28. J.P. Martin-Flatin. *The Push Model in Web-Based Network Management*. Technical Report SSC/1998/022, version 3, Swiss Federal Institute of Technology Lausanne, Switzerland, November 1998.

29. J.P. Martin-Flatin. "Push vs. Pull in Web-Based Network Management". In M. Sloman, S. Mazumdar, and E. Lupu (Eds.), *Proc. 6th IFIP/IEEE International Symposium on Integrated Network Management (IM'99), Boston, MA, USA, May 1999*, pp. 3–18. IEEE Press, New York, NY, USA, 1999.

30. J.P. Martin-Flatin, L. Bovet, and J.P. Hubaux. "JAMAP: a Web-Based Management Platform for IP Networks". In R. Stadler and B. Stiller (Eds.), *Active Technologies for Network and Service Management-Proc. 10th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM'99), Zurich, Switzerland, October 1999*. LNCS 1700: 164-178, Springer, Berlin, Germany, 1999.

31. M. Mattsson, J. Bosch, and M.E. Fayad. "Framework Integration: Problems, Causes, Solutions". *Communications of the ACM*, 42 (10): 80-87, 1999.

32. K. McCloghrie. "The SNMP Framework". *The Simple Times*, 4 (1): 9-10, 1996.

33. P.E. Mellquist. *SNMP++: An Object-Oriented Approach to Developing Network Management Software*. Prentice-Hall, Uppder Saddle River, NJ, USA, 1998.

34. Metrowerks. *PowerPlant White Paper*. Metrowerks, Austin, TX, USA.
Available at <http://www.metrowerks.com/whitepapers/>.

35. Microsoft. *MFC Documentation*. Microsoft, Redmond, WA, USA.
Available at <http://msdn.microsoft.com/library/devprods/vs6/visualc/vcmfc/mfchm.htm>

36. H. Mössenböck. *Objektorientierte Programmierung in Oberon-2*. 3rd edition. Springer, Heidelberg, Germany, 1998.

37. Object Management Group. *OMG Unified Modeling Language Specification*. Object Management Group, Framingham, MA, USA, June 1999.
Available at <http://www.rational.com/uml/index.jtpml>.

38. A.J. Riehl. *Object-Oriented Design Heuristics*. Addison-Wesely, Reading, MA, USA, 1996.

39. D. Riehle and H. Züllighoven. "Understanding and Using Patterns in Software Development". *Theory and Practice of Object Systems*, 2 (1): 3-13, 1996.

40. D. Riehle. "The Event Notification Pattern—Integrating Implicit Invocation with Object-Orientation". *Theory and Practice of Object Systems*, 2 (1): 43-52, 1996.

41. M.T. Rose. *The Simple Book: An Introduction to Networking Management*. revised 2nd edition. Prentice-Hall, Uppder Saddle River, NJ, USA, 1996.

42. D.C. Schmidt. *Patterns for Concurrent, Parallel, and Distributed Systems.* Home Page. Located at <http://www.cs.wustl.edu/~schmidt/patterns-ace.html>.

43. D.C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Distributed Objects.* John Wiley & Sons, Chicester, England. To appear in 2000. (Drafts available at <http://www.cs.wustl.edu/~schmidt/patterns/patterns.html>.)

44. P.E. Sevinç. *Ein Framework für die Mehrzieloptimierung mit Genetischen Algorithmen*. Term Thesis, Swiss Federal Institute of Technology Zurich, Switzerland, February 1999. Available at <http://www.stud.ee.ethz.ch/~psevinc/FEMO/>.

45. A. Silberschatz and P.B. Galvin. *Operating System Concepts.* 5th edition. Addison-Wesely, Reading, MA, USA, 1998.

46. W. Stallings. *SNMP, SNMP v2, SNMP v3, and RMON 1 and 2*, 3rd edition. Addison-Wesely, Reading, MA, USA, 1999.

47. Sun Microsystems. *Java 2 Platform, Micro Edition.* Home Page. Located at <http://java.sun.com/j2me/>.

48. Sun Microsystems. *Java Technology.* Home Page. Located at <http://java.sun.com/>.

49. Sun Microsystems. *JMX white paper.* Sun microsystems, Palo Alto, CA, USA, June 1999. Available at <http://java.sun.com/products/JavaManagement/index.html>.

50. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesely, Reading, MA, USA, 1998.

51. C. Szyperski. "Components and Objects Together". *Software Development*, 7 (5), 1999. Available at <http://www.sdmagazine.com/breakrm/features/s995f2.shtml>.

52. A.S. Tanenbaum. *Computer Networks.* 3rd edition. Prentice-Hall, Upper Saddle River, NJ, USA, 1996.

53. J.P. Thompson. "Web-Based Enterprise Management Architecture". IEEE Communications Magazine, 36 (3): 80-86, 1998.

54. C. Wellens and K. Auerbach. "Towards Useful Management". *The Simple Times*, 4 (3):1-6, 1996.

55. E. Wilde. *Wilde's WWW: Technical Foundations of the World Wide Web.* Springer, Berlin, Germany, 1999.

56. Y. Yemini, G. Goldszmidt, and S. Yemini. "Network Management by Delegation". In I. Krishnan and W. Zimmer (Eds.), *Proc. IFIP 2nd Int. Symposium on Integrated Network Management (ISINM'91), Washington, DC, USA, April 1991*, pp. 95-107. North-Holland, Elsevier, Amsterdam, The Netherlands, 1991.