DOVE: A Distributed Object Visualization Environment

Applying CORBA, Java Beans, and C++ to Monitor and

Visualize Distributed Systems and Applications

Michael Kircher*and Douglas C. Schmidt
mlkirche@rupert.informatik.uni-stuttgart.de, schmidt@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130, (314) 935-4215

This paper appeared in the March 1999 C++ Report.

1 Introduction

Large-scale distributed systems typically contain many heterogeneous components [16]. To manage these types of systems, applications and administrators must be able to monitor the status and proper functioning of system resources. This paper describes the design and use of a *distributed object visualization environment* (DOVE) that supports monitoring and visualization of applications and services in heterogeneous distributed systems.

In this paper, we use DOVE as an exemplar to illustrate how frameworks and components, such as CORBA services and Java Beans, can be combined with patterns, such as Visitor and Observer, to build reusable, scalable, and maintainable software monitoring tools and distributed applications. DOVE itself is a framework, which provides an integrated set of components that defines a reusable architecture for a family of related monitoring and visualization applications.

Conventional monitoring and visualization tools, such as BMC Patrol [1] or IBM NetView, have generally evolved without explicit concern for software qualities like modularity, reuse, or flexibility. Therefore, it's hard for these legacy tools to adapt rapidly to changing application requirements and endsystem/network environments. Support for adaptability is important since requirements of customers and environmental changes force developers to constantly maintain and enhance their software. In this paper, we illustrate how DOVE achieves a high degree of modularity, reuse, and flexibility by using OO techniques, patterns, CORBA, Java, and C++.

This paper is organized as follows: Section 2 gives an overview of the DOVE software architecture; Section 3 describes the design of DOVE's main components, *i.e.*, the DOVE Browser, DOVE Agent, and DOVE MIB; and Section 4 presents concluding remarks.

2 Overview of DOVE

2.1 Example Application of DOVE

It is hard to monitor distributed or embedded systems since application components in these systems are often not connected directly to user interfaces [10]. Moreover, many real-time applications lack spare computing cycles in which to process status or performance information and provide feedback to users or administrators. DOVE supports monitoring of these types of systems by minimizing the amount of computing overhead in the monitored application through the use of *agents*, as shown in Figure 1.

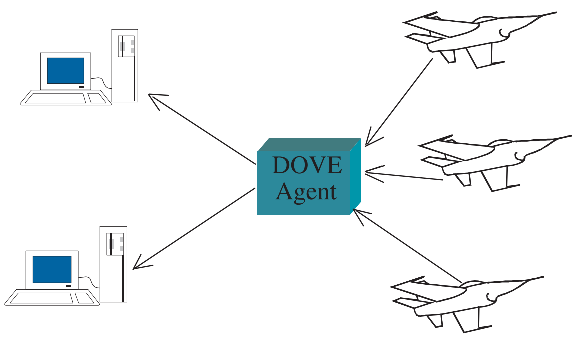

Figure 1: Applying DOVE to Avionics Simulation Systems

*Contact author.

This figure illustrates a DOVE-based monitoring system for a real-time avionics pilot training simulation system. In this application, it is important to monitor system metrics, such as scheduled jobs-per-second or communication delay and jitter, in order to provide feedback to operators who calibrate the simulation's behavior interactively. In this example, DOVE monitors the system components via a central agent that consolidates the information of many aircraft and displays the results on one or more operator consoles. A DOVE Agent, described below, is used to offload most monitoring and visualization processing from aircraft simulation components.

2.2 The DOVE Software Architecture

The key components in the Distributed Object Visualization Environment (DOVE) are shown in Figure 2. DOVE-enabled

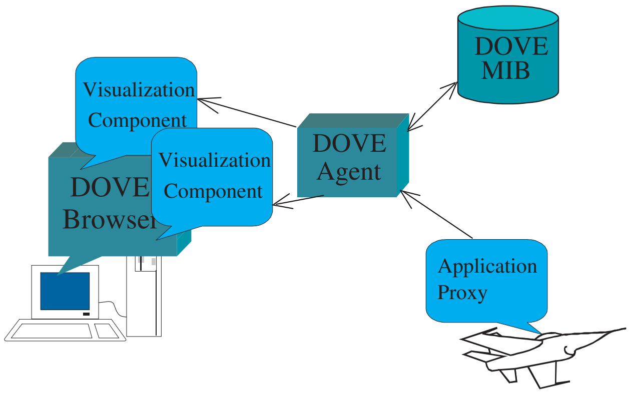

Figure 2: Components in the DOVE Software Architecture

Applications publish information regarding their status and performance metrics to DOVE *Agents* using DOVE *Application Proxies*. In turn, DOVE Agents monitor and publish the advertised information to DOVE-enabled *Browsers*. Agents store management information in DOVE Management Information Bases (*MIBs*) and/or push it to DOVE-enabled Browsers on-demand.

Browsers are connected to one or more DOVE Agents, which they query for services advertised in the distributed system. The Browsers decide which metrics to display. A DOVE-enabled Browser's display manages information published by applications via *Visualization Components*. These components control how monitoring information is presented to end-users, such as human operators. The *Visualization Components* are Java Beans that facilitate dynamic enhancement of strategies used to display information to end-users. Java Beans can be updated on-the-fly, without the need to statically reconfigure the entire system.

In complex system management environments, such as a central office network operation center or a large-scale distributed interactive simulation system, there can be many DOVE-enabled Browsers that interact with many DOVE Agents. The DOVE communication framework uses CORBA to provide a transparent and scalable infrastructure for large-scale distributed system management.

Each component in DOVE's software architecture is described in more detail below:

DOVE application: DOVE applications use pre-defined and/or automatically-generated C++ or Java APIs to communicate with DOVE Agents in order to store and retrieve information residing in a DOVE MIB. DOVE's configuration process supports highly dynamic applications via the Service Configurator pattern [6]. It is not necessary, therefore, to recompile, relink, or restart applications in order to change their monitoring configurations.

DOVE MIB: A DOVE MIB is a repository of information in DOVE. It stores configuration information, such as the identify of advertised services, about monitored applications. The DOVE MIB also can be used to store monitoring data, which can be retrieved later by applications and browsers. In addition, DOVE Agents can use this information to advertise their services.

DOVE Agent: One or more DOVE Agents run in separate processes on nodes in the distributed system and perform the following tasks:

- *Service advertisement* – The DOVE Agent advertises the monitored applications, locations, metrics and control services available.
- *Change notifications* – A DOVE-enabled Browser can automatically be updated when monitored metrics change values.
- *Data reduction and correlation* – Filtering and correlation are supported in DOVE Agents via the CORBA Events Service [5].
- *Visualization configuration* – *Visualization Components* can be integrated dynamically by specifying the name of a Java Beans repository. The Java Beans are then loaded and presented to the operator in order to display the monitoring information in novel ways.
- *MIB management* – MIB management is performed through DOVE-enabled Browsers, which can specify the monitored information to be stored for later retrieval. If an Application Proxy (dis)connects, the appropriate DOVE MIB records this event.

DOVE-enabled Browsers: A DOVE-enabled Browser can either run as a stand-alone application or a Java applet (*e.g.*, in a standard Java-enabled Web browser like Netscape or Internet Explorer). It serves as the display front-end to human operators. During system initialization, the Browser connects with one or more DOVE Agents and provides operators a list of services available from the configured DOVE Agents. The operator then selects which metrics to display. These metrics are derived from information advertised by the Agent.

The following tasks are perform by DOVE-enabled Browsers:

- *Service discovery* – The Browser uses Agents to discover which server applications in a distributed system are offering DOVE services. This discovery process is similar to JavaSoft's Jini framework [15].
- *Visualization builder* – This builder allows end-users to bind graphical or data gathering components to data published by DOVE-enabled Applications. The Browser then packages the resulting tool in its own applet or application. Likewise, the Browser can save this tool to a file for later use. DOVE's "build once, use frequently" mechanism is implemented using Java Beans, as described in Section 3.4.

Visualization Component: The DOVE Visualization Component is the conduit through which information from a DOVE-enabled Application reaches the end-user. It's also the conduit though which configuration information from the user reaches the application. Each Visualization Component is a Java Bean, which registers for updates from Agents or Application Proxies. The Visualization Component makes new information public as it arrives and triggers events related to those changes. Through the DOVE-enabled Browser, the user can connect Visualization Components to these properties and events to monitor and change the state of the application.

Application proxy: The Application Proxy extends a DOVE Agent with application-specific functionality. An Application Proxy is necessary when developers want the Agent and Visualization Component to interact in ways that the generic DOVE framework cannot provide. For example, developers may want to send special messages to the Visualization Component when the application sets a flag in the DOVE MIB.

Figure 3 illustrates the relationships between the various DOVE Components in more detail, using a system developed to monitor distributed electronic medical imaging systems [7]. In this figure, an Image Server publishes the outgoing number of megabytes and updates the value every time a client finishes downloading an image. A network management application may want to use the Visualization Component to chart

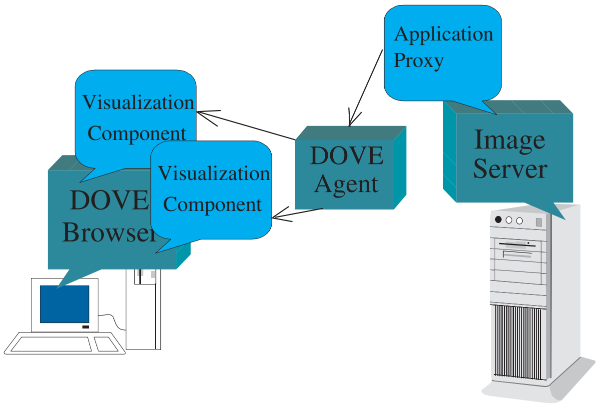

Figure 3: Components in the DOVE Software Architecture, monitoring an Image Server

the average outgoing throughput. To achieve this functionality, the Visualization Component would have a property called "outgoing Megabytes," the point data, timestamps indicating the start and finish time of the connection, and an event called "outgoing Megabytes updated," as shown in Figure 4.

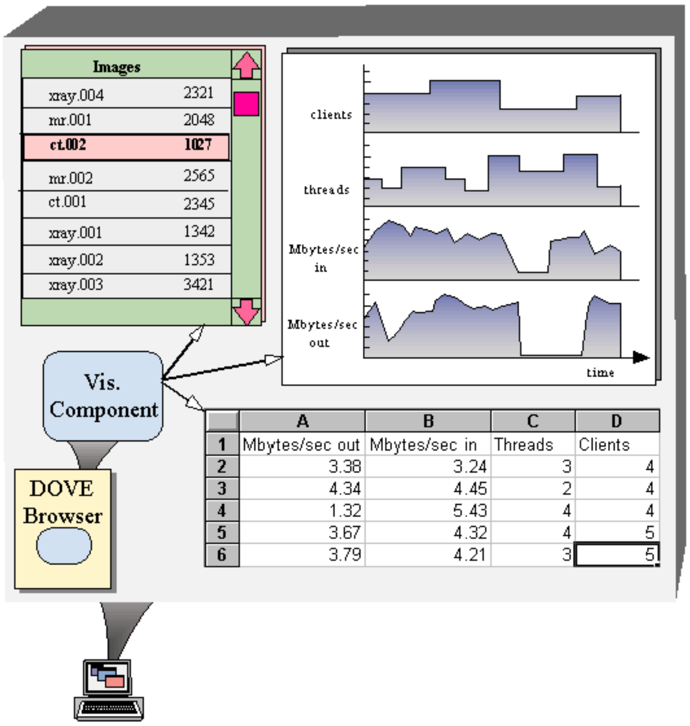

Figure 4: Example DOVE Visualization

To interact with DOVE, users can simply connect an averaging graph to the "outgoing throughput" property. This causes the graph to invisibly attach itself to the timestamp properties and update events. When an update event occurs, the graph updates the current average throughput in megabits per/sec using the new point datum and time elapsed. Finally, it plots the next point on the graph.

2.3 The DOVE Middleware Infrastructure

To increase our productivity and to leverage existing development effort, DOVE is based on the ACE framework [11], the TAO real-time CORBA ORB [13], and Java Beans [14]. Each middleware infrastructure component is summarized below.

Overview of ACE and TAO: The ADAPTIVE Communication Environment (ACE) is an OO framework developed by the Distributed Object Computing (DOC) group at Washington University [12]. The ACE framework implements fundamental design patterns for communication software. ACE is targeted for developers of high-performance communication services and applications on UNIX and Win32 platforms. ACE simplifies the development of OO network applications and services that utilize interprocess communication, event demultiplexing, explicit dynamic linking, and concurrency. ACE automates system configuration and reconfiguration by dynamically linking services into applications at run-time and executing these services in one or more processes or threads.

The ACE ORB (TAO) is an ORB endsystem architecture for high-performance, real-time CORBA [13]. TAO implements the CORBA 2.x standard and addresses performance limitations with conventional ORBs. Like ACE, TAO was developed by the DOC group at Washington University.

Both ACE and TAO are freely available using an open source development model. They are used for many commercial projects at companies like Bellcore, Siemens, Motorola, Kodak, and Boeing, as well as in many academic and industrial research projects. ACE and TAO have been ported to a variety of OS platforms including Win32, most UNIX/POSIX platforms, and real-time operating systems. C++ and Java versions of ACE are available for downloading at `www.cs.wustl.edu/~schmidt/ACE.html`.

Overview of Java Beans: A Java Bean is a reusable software component. Common type of Java Beans are usually small control programs, though Beans can also be complete stand-alone applications. The Java Beans framework is designed to allow objects to be written in such a way that their properties and behavior can be modified without have to recode or recompile existing components.

In general, Java Beans are Java classes that conform to certain standards. Inheritance from a special base class is not needed to create a Java Bean, though developers must conform to certain design guidelines [14]. For instance, method names should conform to a naming convention so that reuse and configuration are facilitated.

The naming convention for Java Beans is simple: a method that sets any parameter of an Java Bean should be named `set<parametername> (<parametername>)` and a method that gets some parameter of a Java Bean should be named `<parametername> get<parametername>`.

3 The OO Design of DOVE

This section presents the OO design of DOVE, focusing on key design challenges we faced and the solutions we employed to resolve these challenges.

3.1 Motivating the Need for CORBA

3.1.1 Context

The first prototype of DOVE was developed before the version that is the focus of this paper. This initial prototype used Java as the programming language and was programmed using sockets for communication. Moreover, it was very restrictive, *e.g.*, only one Application Proxy and one DOVE-enabled Browser were allowed, and the types of data transported included only weapons status, navigation information, and statistical data.

The prototype consists of a Java front-end that displayed a fix set of aircraft metrics, such available weapons, an artificial horizon, and several statistical data items. Statistical data were displayed in their own Java-Canvas, the weapons were listed in a Java-Panel (class `Display_Weapons`) and the artificial horizon (class `Display_Art_Horizon`), and all metrics were displayed within one Java-Panel. Class `Display_Weapons` and class `Display_Art_Horizon` inherited from an interface called `Display_Object` to allow other classes to access them polymorphically. The Abstract Factory pattern was used to define a `Display_Factory_Object` that manage weapon and artificial horizon displays and creates `Display_Objects`.

Figure 5 illustrates the dependencies between the classes. The communication between Application Proxy, the visualization, and the DOVE-enabled Browser is solely performed through sockets and TCP/IP. The Application proxy generates random input, which is then fed into a socket connected to the DOVE-enabled Browser.

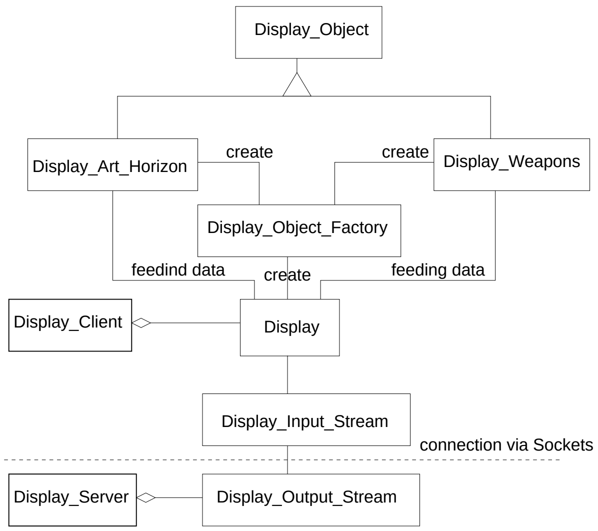

Figure 5: Class Diagram for the Original DOVE-enabled Browser

3.1.2 Problem

Although the first prototype served as a reasonable proof-of-concept, is was hard to scale to large distribution topologies and hard to extend with new functionality. The main problem centered on the hard-coded socket-based connection between the Application Proxy, which collected and sent metrics, and the Visualization Components, which displayed the metrics. In particular, the prototype didn't support multiple instances of Application Proxies, nor did it support multiple instances of DOVE-enabled Browsers. Moreover, the first prototype was not extensible since it used exposed the details of the data types used to communicate between the various DOVE components.

3.1.3 Solution

We introduced CORBA IDL definitions for DOVE component interfaces and replaced the socket connection between the Application Proxy and the DOVE-enabled Browser by CORBA [18, 17] calls using the newly defined CORBA interfaces. The DOVE-enabled Browser is implemented using Java [2], which supports the use of Java Beans. Our scalability problems were alleviated by implementing the DOVE Agent with the CORBA Events Service, as discussed in Section 3.3. Finally, the hard-coded connection between metrics and visualization is alleviated by Java Beans, as described in Section 3.4.

3.2 Bootstrapping the Naming Service

3.2.1 Context

Once DOVE was rewritten using CORBA, components like Application Proxies and DOVE-enabled Browsers communicate only through their IDL interfaces. Therefore, an object reference to each remote component must be obtained in order for clients to make calls. The CORBA Naming Service allows easy storage and retrieval of these object references. Object references are correlated with names, which can be used to obtain bindings to the appropriate object references. If a Naming Service was not used, these object references must be provided through more tedious means, such as supplying them via command-line parameters.

3.2.2 Problem

How does an ORB obtain the initial object reference to its Naming Service?

3.2.3 Solution

We obtained the initial Naming Service object reference via a "bootstrap protocol." TAO implements the bootstrap protocol for the Naming Service in the following way:[1]

- TAO's Naming Service runs in a process that listens to a well-defined multicast address and a well-defined port for client multicast requests. TAO uses multicast instead of a fixed host address to locate the Naming Service so that it can reside on any machine in a domain. Thus, clients in this domain need not know which machine the Naming Service resides on since it's transparent to them.
- The Naming Service expects to receive two bytes that define the port number in network byte order on which the requesting machine expects the response from the Naming Service. The Naming Service then responds to this address with a datagram containing the IOR of the Naming Service.
- Any client requesting the Naming Service IOR must multicast a UDP datagram containing the port number to the multicast address. The port number in the packet refers to the port on which the requesting client listens for the response from the Naming Service. The client then waits until the response arrives from the Naming Service or a timeout occurs. If the Naming Service IOR is received, the client ORB converts it into an object reference using the standard CORBA `string_to_object` operation.

[1] The OMG has recently standardized a "Interoperable Naming Service" specification [3]. However, at the time this paper was written no ORBs implement this specification. We plan to make TAO conform to this specification shortly.

Once a client has resolved its Naming Service object reference, other object services and application components can be resolved via the `resolve` method on the Naming Service's `Naming_Context`.

3.3 Enhancing the DOVE-enabled Browser with the Events Service

3.3.1 Context

The communication mechanism between DOVE components must be scalable so that multiple DOVE-enabled Browsers can communicate efficiently with multiple DOVE-enabled Applications. Likewise, monitoring information must be transported generically so that its schemas can change without having to change the communication protocols or DOVE component interfaces.

3.3.2 Problem

The first DOVE prototype uses sockets to communicate by sending bytestreams between the Application Proxy and the DOVE-enabled Browser. Therefore, as discussed in Section **??**, the first protoype of DOVE scales poorly since it allows only uni-direction communication between various DOVE components. Moreover, this prototype hard-codes the schema information directly into the applications, which makes it hard to change the communication protocols without changing the DOVE components.

3.3.3 Solution

Use the CORBA Events Service [5] as the communication mechanism to multicast events from event suppliers to registered event consumers through an Event Channel. The DOVE Agent is then implemented as an Event Channel. Figure 6 illustrates how event suppliers push events to the Event Channel, which then filters, correlates and finally forwards these events to event suppliers. In this figure, DOVE-enabled Browsers are event consumers and DOVE-enabled Applications are event suppliers.

The events in the OMG COS Specification of the Events Service are defined as type `CORBA::Any`. The `CORBA::Any` type consists of the following two fields:

- *The typecode* – The typecode field describes which built-in or derived data type the value field points to. The field actually contains a pointer to a structure, in which information about the data type is stored.
- *The value* – The value is a pointer to `void`. The CORBA specification allows an application to assign any type to a `CORBA::Any`.

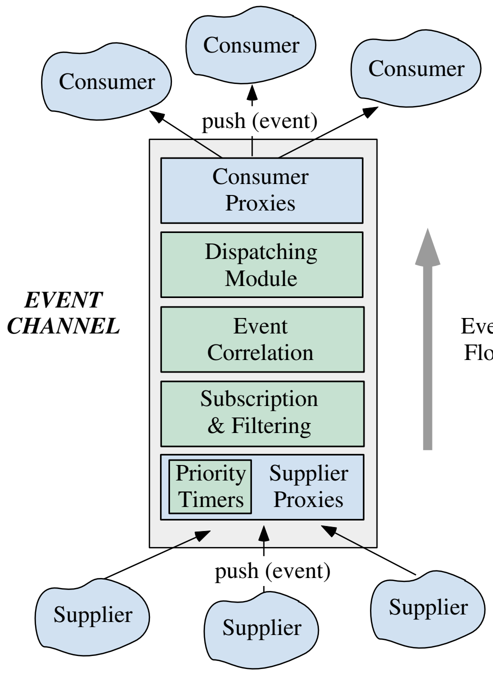

Figure 6: Components in TAO's CORBA Events Service

The value and type of a `CORBA::Any` can be set using the copy-constructor, the assignment operator, the `replace` method, or `operator <<=`. The stub or skeleton generated by the IDL compiler must ensure that the value(s) and the type(s) are properly marshaled for transmission as the stub and skeleton are responsible for proper (de)marshaling.

The CORBA Events Service uses the `CORBA::Any` type to allow any type of application-specific data, such as avionics sensor data, to be transported within its events. These data items can then easily put into the `CORBA::Any` and transported generically throughout the network and DOVE components.

Figure 7 shows the relationships of the Events Service parts to the rest of DOVE. First the Application Proxy, serving as event supplier, pushes information into the event channel, which filters and correlates these events (containing the metrics) and pushes the events to the event consumer which then updates information inside the DOVE-enabled Browser.

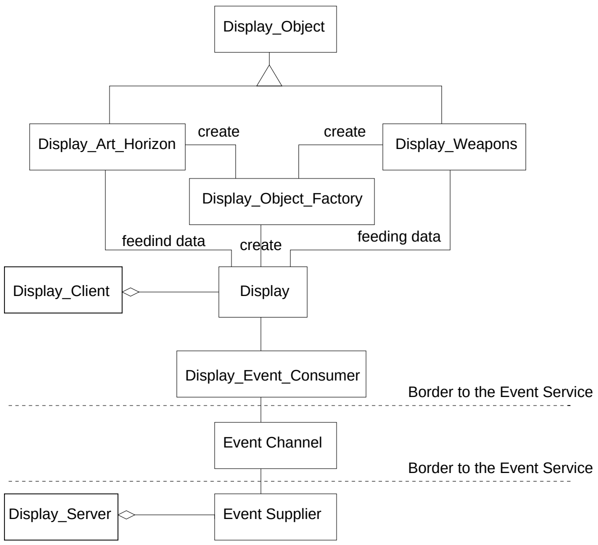

Figure 7: Class Diagram for the Integration of the Events Service into DOVE

3.4 Enhancing the DOVE-enabled Browser with Java Beans

3.4.1 Context

The original DOVE prototype consisted of the following Visualization Components: (1) a graph component, (2) a list-panel to display the weapon status, and (3) an artificial horizon. The way the viewer was built made it possible to have several of these components running simultaneously, connected to different data sources. Thus, it was possible to display metrics and graphs dynamically, though the set of Visualization Components was hard-coded.

3.4.2 Problem

In the original DOVE prototype, both the set of Visualization Components and the metric that could be viewed with which Visualization Component were hard-coded. Thus, it wasn't possible to display a metric using different Visualization Components, which is too inflexible. Having a generic way to connect Visualization Components with metrics makes it possible to view the same metric with different Visualization Components, each displaying a different aspect of the metric. Moreover the Visualization Components were not dynamically loadable, so the set of Visualization Component could not be enhanced at run-time.

3.4.3 Solution

We used the Observer pattern [4] to provide a generic mechanism for registering and accessing Visualization Components. This pattern decouples the metrics and the Visualization Components, making it possible to connect a metric with any Visualization Component. It also allows several different Visualization Components to be connected to the same metric.

The Observer pattern defines two roles for objects: *Observables* and *Observers*. Observers register with one or more Observables. An Observable informs one or more registered Observers about changes as soon as its own state changes. Applying the Observer pattern to the DOVE-enabled Browser encapsulates the metrics in Observables and the Visualization Components as Observers.

In addition, DOVE's Visualization Components are both Observers *and* Java Beans. Figure 8 shows the architecture of new DOVE-enabled Browser using Java Beans. This figure shows how all concrete Observables inherit from the Observable interface and all concrete Visualization Components, *i.e.*, the Java Beans, inherit from the Observer interface. Because communication between Observables and Observers is performed only via the base class interface, independence between concrete Observables and Observers is ensured.

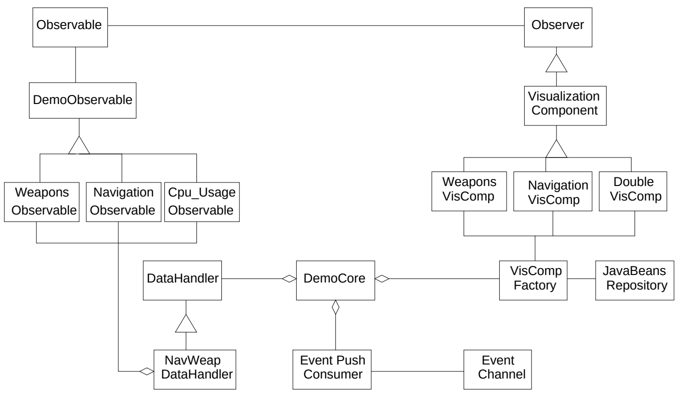

Figure 8: Class Diagram of the DOVE-enabled Browser

The functional model of the new DOVE-enabled Browser is simple, as shown Figure 9. The `DemoCore` object instructs the `VisCompFactory` to create a new Visualization Component with the properties necessary to display the chosen `Observable`. Once created, the `DemoCore` is responsible for connecting `Observables` with `Observers` (*i.e.*, Visualization Components). Connecting `Observables` is done through the `DataHandler`, which is a generic interface that allows the `DemoCore` to request a list of Observables and then to pick one specifically to connect it with an appropriate Visualization Component.

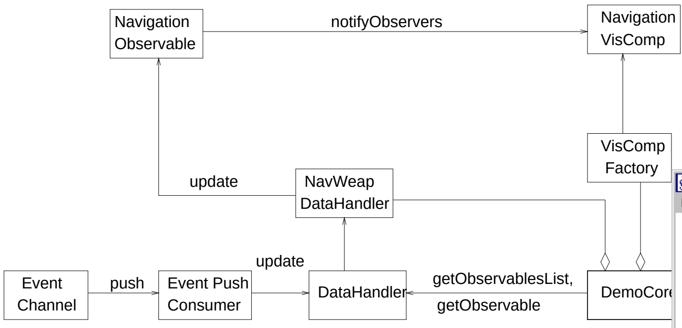

Figure 9: Functional Model of the DOVE-enabled Browser

Updates of the Visualization Components occur as follows: TAO's CORBA Event Channel pushes events to the `PushConsumer`, which forwards the event data field to the Data Handler, which is actually an concrete Data Handler, *e.g.*, `NavWeapDataHandler`. This Data Handler then demultiplexes the event data structure into several metrics. The metrics recognize the value they contain has changed and trigger a notification event to their Observers. These events contain an object with the changed data. The Observers read the data field of the notification event and update their graphical front-end in the DOVE-enabled Browser. Figure 10 shows the interface between Observables and Observers.

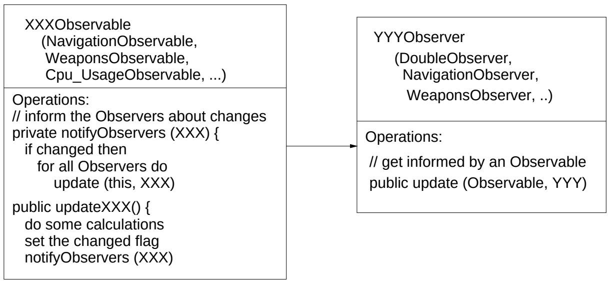

Figure 10: Interface Description between Observables and Observers

Observer/Observable pairs can be categorized into two classes depending on the data for which they are responsible. The first category uses only built-in Java types, such as `Double` or `Int`. The second category uses derived types (user defined classes) containing more sophisticated information, such as Weapon status or Navigation information.

3.4.4 Implementation

As mentioned earlier, Java Beans is used as the framework for the generic DOVE Visualization Components. The Java JDK implements the Observer pattern in the `java.util.Observer` and `java.util.Observable`. Figure 11 illustrates the GUI of a DOVE-enabled Browser that is configured for the avionics simulation system described in Section 1. Six

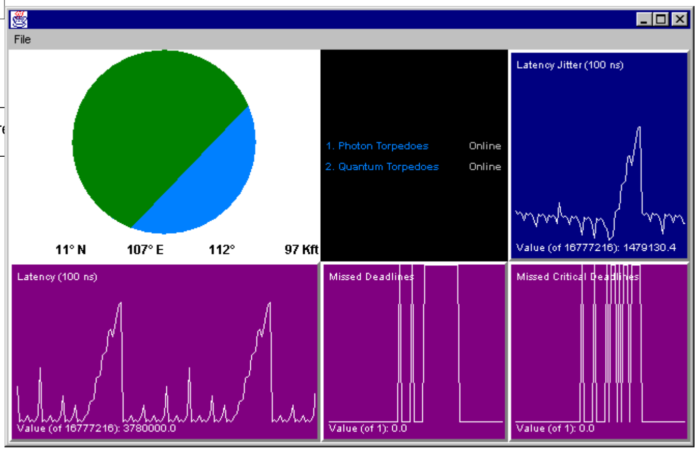

Figure 11: The GUI of the DOVE-enabled Browser

Visualization Components are connected in this figure. One of them is the Navigation Visualization Component, one is the Weapon Status Visualization Component, the others are Double Visualization Components connected to corresponding metrics.

3.5 Building the DOVE Management Information Base

3.5.1 Context

A Management Information Base (MIB) is a database containing management information about so-called managed objects, such as routers, PCs, workstations, or software running on machines. In this paper, we use MIBs to store not only standard management information but also historical values of monitoring statistics and associated metrics. Thus, a DOVE MIB can store a complete history of information from monitoring applications.

3.5.2 Problem

One requirement for DOVE is to save all monitoring information on persistent storage in the MIB. Once stored persistently, this information can be used for analysis and debugging of the monitored system. The stored information must be saved in a human readable way.

The following are several alternative ways to build a DOVE MIB:

Have a MIB connected directly to the Events Service: The Events Service could implement the persistence storage of events in itself. The CORBA Object Services specification [9] states that an Event Channel can store events for "a specified time, passing it along to any consumer who registers with the channel during that period of time (*e.g.*, it may keep event notifications about changes to engineering specifications for a week)." The advantages of this approach is the simple interface and the fact that only a small number of objects would be involved in a request for old messages, namely the Events Service and the requesting object.

Have the MIB external to the Events Service: In this case, the MIB would be a consumer to the Event Channel. The basic idea is that this external object listens to all the events, stores them on persistent storage with time, type and source ID. It retrieves them on-demand, filtering them from the persistent storage and supplying them to the Event Channel again. The challenge with this approach is that care must be taken to ensure that no other components are affected by the repetition of the events.

If the MIB is treated as an external object, there are the following two ways to contact the MIB:

- One way is to ensure that the external object, *i.e.*, the MIB, pushes the events directly to the consumer who requested the stored events.
- Another way is to have the MIB contacted without involving the Events Service. This requires that the requesting objects know whom to ask. The object reference of the MIB could be supplied in response to a request to the Events Service.

The main problem of the DOVE MIB is how to retrieve the information published by the DOVE Agents. The information is contained in a `CORBA::Any`, as mentioned in Section 3.3.3. The DOVE MIB must read the typecode, analyze it, extract the information, and timestamp the new entry in the MIB. It is also necessary to ensure that the order of the events is maintained. In particular, it's important that new events aren't processed before earlier events.

3.5.3 Solution

DOVE's MIB is designed as an external MIB, which receives all events from the DOVE Agent's Event Channel and stores them as timestamped "type, member name and value" tuple in persistent storage. This design requires that an external object is connected as event consumer to the Event Channel. A DOVE MIB listens to all events sent on the Event Channel and stores the event data in persistent storage, using in a format similar to the format used for declarations in C++.

Event data is contained in a `CORBA::Any` and no assumptions are made about the type of data in an `Any`. For instance, the `CORBA::Any` could contain a `struct`, a `double`, a `long`, etc. Moreover, several layers of types could be contained, *e.g.*, a `struct` may contain a nested `struct`, which contain a `string`, etc. Figure 12 illustrates a class diagram for this type of recursive type configuration.

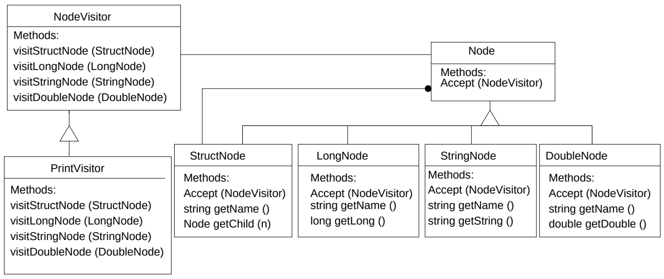

Figure 12: Visitor and Tree Representing the Content of a `CORBA::Any`

Figure 12 illustrates how all types of nodes, `StructNode`, `DoubleNode`, `LongNode`, `ULongNode`, `StringNode`, etc., inherit from `Node`. Therefore, we used the Visitor pattern [4] to traverse these trees. All Visitors inherit from `NodeVisitor`. which provides the basic functionality to traverse a tree. The visitor method `PrintVisitor` accepts a file name and a tree as input. It prints the value of the tree in a format similar to the declaration format in C++.

Figure 13 shows the relation between the objects and methods in the Visitor pattern used in the DOVE MIB. The

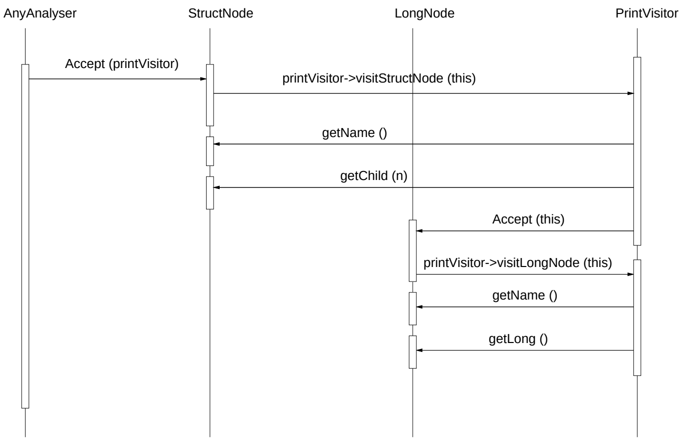

Figure 13: Functional Model of the Visitor Pattern in the DOVE MIB

`AnyAnalyzer` creates a tree out of the type code and value information in the `CORBA::Any`. It instantiates the `PrintVisitor` and calls the root of the tree with a reference to the `PrintVisitor`. The node, in this case the `StructNode`, then calls the `PrintVisitor`, which in turn calls the StructNode again to obtain the properties of it. In this case the properties are the name and references to children.

The primary advantage of the Visitor pattern is that the nodes do not depend on what is done with the properties, *e.g.*, if they are used for further computation, just printed or further traversed. This independence assures that new Visitors can be added without changing the node objects.

For example, the `PrintVisitor` uses references to the children of the `StructNode` to traverse the tree. The first child is a `LongNode`, it gets called by the `PrintVisitor` via its `Accept` method and replies with the `visitLongNode` method call of the `PrintVisitor`. In turn, the `PrintVisitor` gets the properties of the `LongNode` and prints them. The `LongNode` need not care about what is done with its properties.

A wrapper called `AnyAnalyzer` wraps the building of the trees with the `PrintVisitor` so that the `AnyAnalyzer` can be seen as one object to which `CORBA::Anys` can be given. The content of the `CORBA::Any` will be written to a file.

The MIB application itself consists out of a `Event Push Consumer` and an `AnyAnalyzer`, as shown in Figure 14. This design decouples the components as much as possible.

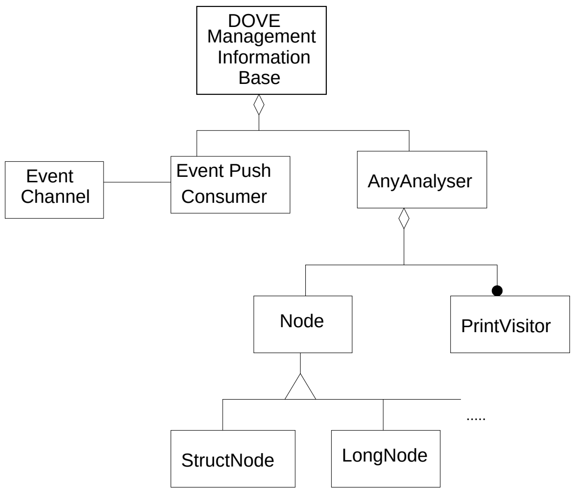

Figure 14: Class Diagram of the DOVE Management Information Base (MIB)

Therefore, a change in the `Event Push Consumer` does not influence the `AnyAnalyzer` and vice versa. Figure 14 shows the relations between the classes.

3.5.4 Implementation

Since the MIB does not require a GUI it was implement using C++ and TAO. The `Node`, {`Struct, Double, Long` ... }`Node, NodeVisitor, PrintVisitor` and `AnyAnalyzer` classes are mapped directly to C++ classes. The `AnyAnalyzer` class contains detailed knowledge about the internals of a `CORBA::Any`. The `CORBA::Any` in TAO holds the value in two different ways, depending on if the ORB owns the data or not.

If the ORB does own the data, a simple pointer to an allocated memory area can be retrieved. If the ORB does not own the data, it must be copied into a newly allocated memory area and decoded. Analyzing the `CORBA::Any` involves checking the type code information and creating a tree node with a proper pointer to the actual memory location. The number of bytes defined by the type code size is then skipped and this scheme is reapplied recursively.

4 Concluding Remarks

The DOVE monitoring framework has proven to be a very flexible and generic approach to meet the rapid pace of change and growth in heterogeneous distributed systems. We learned the following lessons while developing DOVE:

- Patterns are valuable for software development since they simplify reuse and extensibility. In particular, applying the patterns to DOVE made the software easier to understand and maintain compared with earlier systems we developed that weren't explicitly designed using patterns. Some of the key design patterns we used in DOVE are Observer, Factory, and Visitor [4].
- Using an interface definition language like CORBA IDL simplifies the development of heterogeneous components written in different programming languages. CORBA IDL makes it easy to use the right programming language for the right tasks, *e.g.*, using Java for the GUI and C++ for the Events Service, while ensuring smooth inter-language interoperability between languages.
- CORBA takes the concept of generic interfaces a step further and supports the transparent distribution of software components. With its powerful features and common object services, such as the Naming Service and Events Service, it is an effective way to integrate C++ and Java code. A good example of this in DOVE is its use of the TAO's Real-time Events Service, where end-to-end quality of service can be guaranteed throughout the distributed objects, up to the GUI components in the DOVE-enabled Browsers that are implemented in Java. Because human operators typically interact with their consoles slower

than even the slowest Java implementation, the perceived performance is generally meets the display requirements.

- Java Beans help increase the adaptability and maintainability of DOVE applications and tools. For instance, developers and even end-users can modify and enhance different Beans rapidly.
- Though the performance of Java ORBs like Java IDL or VisiBroker for Java is not as good as highly-optimized real-time C++ ORBs like TAO, they are sufficient to support the connection of Java user interfaces to distributed applications and components.
- When used properly, exception handling shortens component debugging and testing time considerably since it's not possible to "ignore" errors [8]. Points of failure can quickly be tracked down to the source.
- Memory management is highly problematic in C++ applications. Therefore, it's essential to use memory management tools, such as Purify and Bounds Checker.

The complete C++ and Java source code, examples, and documentation for ACE, TAO, and DOVE is freely available at URL `www.cs.wustl.edu/~schmidt/TAO.html`.

References

[1] BMC. *http://www.bmc.com*. BMC, 1998.

[2] David Flanagan. *JAVA in a Nutshell 2nd ed.* O'Reilly, 1997.

[3] Dan Frantz, Michi Henning, Michael Neville, Tod MacFadden, Jeff Mischkinski, and Martin Chapman. *ftp://ftp.omg.org/pub/docs/orbos/98-10-11.pdf*. OMG, 1998.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97*, Atlanta, GA, October 1997. ACM.

[6] Prashant Jain and Douglas C. Schmidt. Service Configurator: A Pattern for Dynamic Configuration of Services. In *Proceedings of the* 3^{rd} *Conference on Object-Oriented Technologies and Systems*. USENIX, June 1997.

[7] Prashant Jain, Seth Widoff, and Douglas C. Schmidt. The Design and Performance of MedJava – A Distributed Electronic Medical Imaging System Developed with Java Applets and Web Tools. In *Proceedings of the* 4^{rd} *Conference on Object-Oriented Technologies and Systems*. USENIX, April 1998.

[8] Harald Mueller. Patterns for Handling Exception Handling Successfully. *C++ Report*, 8(1), January 1996.

[9] OMG. *CORBA services: Common Object Services Specification*. OMG, 1997.

[10] Douglas C. Schmidt. The Reactor: An Object-Oriented Interface for Event-Driven UNIX I/O Multiplexing (Part 1 of 2). *C++ Report*, 5(2), February 1993.

[11] Douglas C. Schmidt. ACE: an Object-Oriented Framework for Developing Distributed Applications. In *Proceedings of the* 6^{th} *USENIX C++ Technical Conference*, Cambridge, Massachusetts, April 1994. USENIX Association.

[12] Douglas C. Schmidt. *The ADAPTIVE Communicaton Environment*. DOC group, Washington University, 1994.

[13] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4):294–324, April 1998.

[14] Sun. *http://java.sun.com/beans/spec.html*. Sun Microsystems, Inc, 1997.

[15] Sun. *http://www.javasoft.com:81/products/jini/index.html*. Sun Microsystems, Inc, 1998.

[16] Steve Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 14(2), February 1997.

[17] Visigenic. *VisiBroker for JAVA 3.1 documentation*. Visigenic, 1998.

[18] Andreas Vogel and Keith Duddy. *Java Programming with CORBA*. Wiley & Sons, 1997.