Applying Aspect-Oriented Programming to Intelligent Synthesis

2001128382
540329

Robert E. Filman
Research Institute for Advanced Computer Science
NASA Ames Research Center MS/269-2
Moffett Field, California 94035
U.S.A.
rfilman@mail.arc.nasa.gov
http://ic.arc.nasa.gov/ic/darwin/oif/leo/filman/filman.html

I discuss a component-centered, aspect-oriented system, the Object Infrastructure Framework *(OIF), NASA's initiative on* Intelligent Synthesis Environments *(ISE), and the application of OIF to the architecture of ISE.*

Object Infrastructure Framework

A critical issue in developing component-based and distributed systems is getting the assembled set of components to follow the policies of the overall system. To achieve *ilities* such as reliability, availability, responsiveness, performance, security, and manageability, all system components must consistently perform certain actions. Unfortunately, developers of off-the-shelf or pre-existing components, blithely unaware of or indifferent to these requirements, do not code the appropriate policy support. For custom components, the developers are likely to be domain experts, not experts in systems and distributed computing, and are similarly unlikely to consistently and correctly code the calls to ility support routines. Furthermore, policies change over a system's lifetime. Tracking these changes in the components will be difficult or (lacking the component source code) impossible. Separating out the specification and implementation of ilities and providing mechanisms for weaving together the main functionality with the ility code is a prime example of the potential leverage of aspect-oriented programming (AOP).

In earlier papers, we described the Object Infrastructure Framework (OIF), a CORBA centered, aspect-oriented system for achieving ilities in distributed systems [4, 5]. OIF realized the following key ideas:

1. **Intercepting communications.** OIF intercepted and manipulated communications among functional components, invoking appropriate "services" on these calls. Semantically, this is equivalent to wrapping or filtering [1] on both the client and server side of a distributed system. The next five points can be understood as describing the architecture of a flexible wrapping system.
2. **Discrete injectors.** Our communication interceptors are first class objects: discrete components that have (object) identity and are invoked in a specific sequence. We call them *injectors*. In a distributed system, an ility may require injecting behavior on both the client and the server. (Figure 1 illustrates injectors on communication paths between components.) Injectors are uniform so we can build utilities to manipulate them.
3. **Injection by object/method.** Each instance and each method on that object can have a distinct sequence of injectors.

4. **Dynamic injection.** The injectors on an object/method are maintained dynamically and can, with the appropriate privileges, be added and removed. Examples of the application of dynamic configuration include placing debugging and monitoring probes on running applications and creating software that detects its own obsolescence and updates itself.
5. **Annotations**. Injectors can communicate among themselves by adding annotations to the underlying requests of the procedure call mechanism.
6. **Thread contexts.** Our goal is to keep the injection mechanism invisible to the functional components (or at least to those functional components that want to remain ignorant of it.) To allow clients and servers to communicate with the injector mechanism, the system maintains a "thread context" of annotations for threads, and copies between this context and the annotation context of requests. Thread contexts and annotations together provide the data space for communication between the application and injectors and among injectors. (Injectors generated by the same factory or industrial complex can also share a data space defined by their factory structure.)
7. **High-level specification compiler.** To bridge the conceptual distance between abstract ilities and discrete sequences of injectors, we created a compiler from high-level specification of desired properties and ways to achieve these properties to default injector initializations.

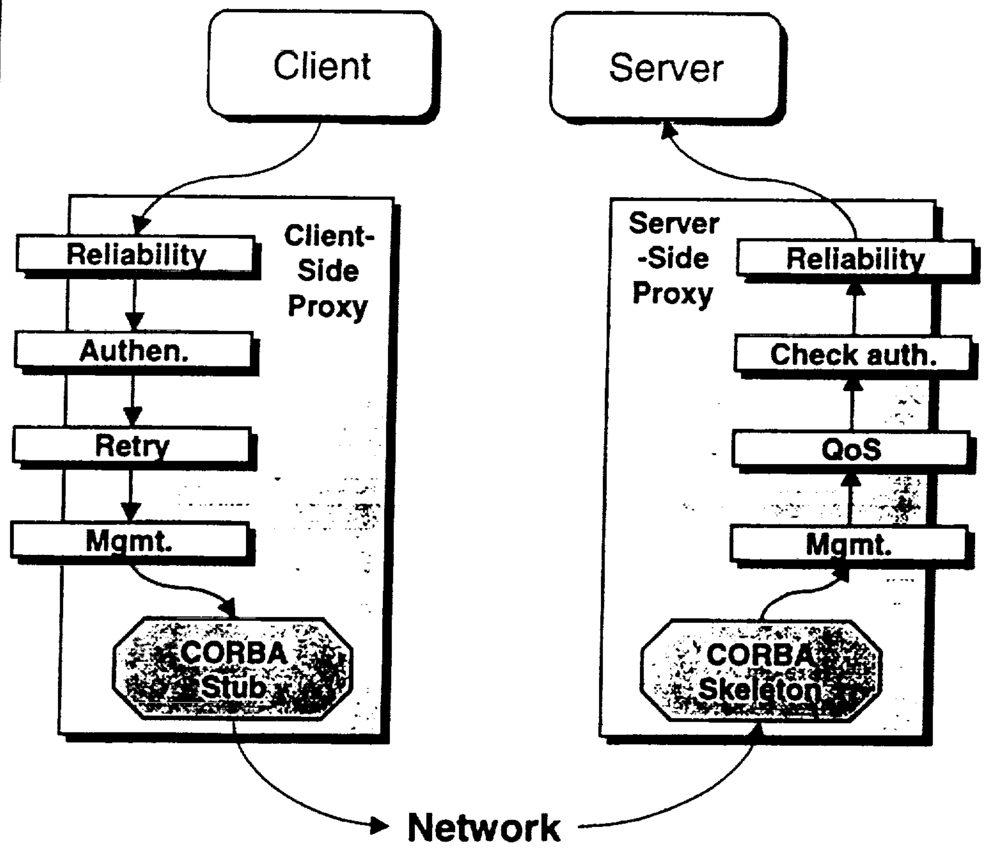

Figure 1: Injectors on the Communication Path Between Components

OIF was developed to simplify distributed computing. We developed our prototype system for CORBA [10] components written in Java. CORBA is a distributed object technology (DOT) that presents remote objects as *proxy* objects in the local computing environment. Client-side proxies are responsible for encoding and forwarding a local call to the remote service; server-side proxies for decoding and returning the result. CORBA requires the description of an object's interface in its interlingual Interface Description Language (IDL). The CORBA IDL compiler then "compiles" that IDL into the code for the proxies in the desired execution language or languages. We implemented OIF by creating an alternative IDL compiler whose proxies both included calls to the proxy-specific sequence of injectors and maintained the request object/annotation/thread-context relationships.

A premise of the OIF work was that components are black boxes whose internal structure cannot be examined or manipulated. This contrasts with source-language–level approaches to AOP like AspectJ [9] and subject-oriented programming [8] which express aspects as code fragments to be woven together into a working program. Thus, OIF injectors are reusable—they have their own "target-independent" semantics and the same injector can be used in multiple places.

Examples of injectors we have developed or are developing include are listed in Table 1. Several of these injectors are discussed in greater detail in [5].

Intelligent Synthesis Environment

The American National Aeronautics and Space Administration (NASA) Intelligent Synthesis Environment (ISE) program is a grand attempt to develop a system to transform the way complex artifacts are engineered. The program "aims to link scientists, design teams, manufacturers,

Ility	Injector	Action
Security	Authentication	Determines the identity of a user.
	Access control	Decides if a user has the privileges for a specific operation.
	Encryption	Encodes messages between correspondents.
	Intrusion detection	Recognizes attacks on the system.
Reliability	Replication	Replicates a database.
	Error retry	Catches network timeouts and repeats call.
	Rebind	Notices broken connections and opens connections to alternative servers.
	Voting	Transmits the same request to multiple servers (in sequence or parallel) combining the results by temporal or majority criteria.
	Transactions	Coordinates the behavior of multiple servers to all commit or fail together. Requires additional interface on application objects.
Quality of service	Queue-manager	Provides priority-based service.
	Side-door	Provides socket-based communication transparently to application.
	Futures	Provides futures transparently to the application.
	Caching	Caches results of invariant services.
Manageability	Logging	Reports dynamically on system behavior.
	Accounting	Reports to accounting system on incurred costs.
	Status	Accrues status information and reports when requested.
	Configuration management	Dynamically test for incompatible versions and automatically updates software.

Table 1. Injectors

suppliers and consultants in the creation and operation of an aerospace system" [6]. That is, geographically distributed engineers and scientists creating the design for, say, a spacecraft, should be able to conveniently collaborate. This collaboration includes not only sharing information, but also being able to use advanced analysis systems and virtual reality environments to explore the properties of the design. The underlying system should track the changes and versions of design artifacts and protect the intellectual property of the participants. The ISE vision thus shares intent of applying analysis tools to designs with other initiatives (e.g., DARPA's Simulation-Based Design [3] and the National Industrial Information Infrastructure Protocols Consortium's STEP standard [7]) but differs in its extension to the human-computer interface.

Structurally, the ISE program divides into groups concerned with Collaboration (tools for sharing information), Human-Centered Computing (immersive environments), Life Cycle Integration and Validation (version management) and Infrastructure for Distributed Computing (software architecture). Software architecture is our primary concern. The ISE software architecture (or ISA, for short) needs to provide (1) a database of design artifacts, information about those artifacts, design tools, information about those tools, users and their rights and privileges, and so forth; (2) a straightforward architectural layer to the application programmer that transparently supports distribution and the invocation of distributed analysis tools, (3) access control, transaction, version, and translation mechanisms, (4) scripting mechanisms (the ability to chain together tool applications). I have been working with the group defining the ISE software architecture to try and realize these goals. In the next section, I present a view of an ISA shaped by an OIF-based AOP model. Providing these additional aspects to an ISE-like environment makes good challenge problem for aspect-oriented programming.

Applying AOP to ISE

In this section, I present the outline of an Intelligent Synthesis Architecture for ISE. The key themes of this architecture are (1) ISA as a repository of synthesis artifacts (including information about both the synthesis artifacts themselves and the tools used to create synthesis artifacts), (2) ISA as an enabler of distributed computing and (3) ISA as a collection of synthesis services. Critical to the ISE problem is incorporating and integrating legacy analysis tools. Such tools often exist in their own particular environments, take their own idiosyncratic inputs, and do who knows what else.

The key ideas here are

- We reify (treat as objects to be talked about) users, tools, and design artifacts (the inputs and outputs of tools.) Call such things *entities*.
- We record annotations about entities. (For example, file XYZ is the output of running tool ABC on inputs G and H. This was done on May 15th by STU. It has the following permissions, precision, version, and so forth.) We cannot anticipate all the annotations that users might want, so the set of annotations must be dynamic. We may also profitably apply the knowledge representation mechanisms of expert system building tools (e.g., default values, inheritance, access-oriented programming).
- Since we know about the available tools and the annotations they impose on their outputs, and since tools are expressed as distributed objects, we can run these tools (remotely) and can script together collections of tools.
- Tool scripts and application programs can take advantage of a number of system-provided services. (The repository can be seen as be one such service.)

We argue for a four layer architecture for ISA (Figure 2). The lowest layer, the transport layer, enables secure remote invocation. Above it, a repository provides a database of synthesis artifacts and their attributes. The Common Services layer provides services on which every user of ISA can rely, and the Extended services layer uses the mechanisms of the repository and transport layer to provide optional additional facilities.

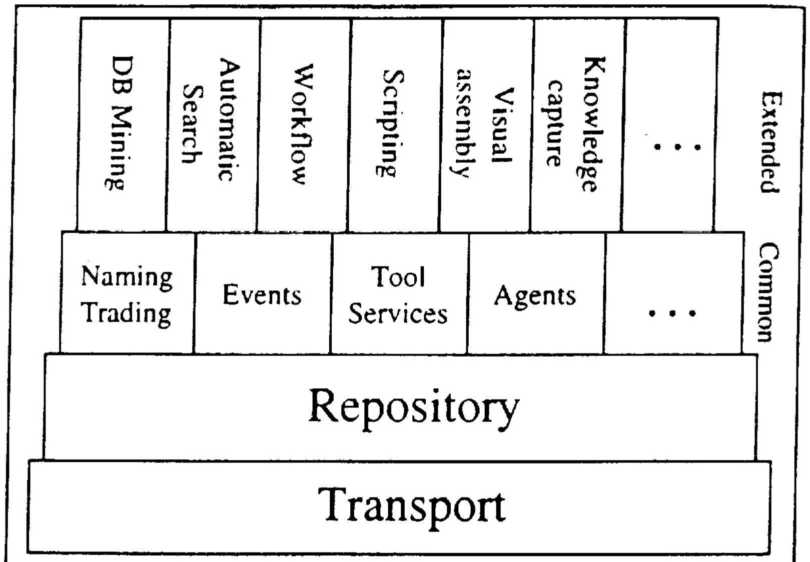

Figure 2. Intelligent Synthesis Architecture

The obvious (from our point of view) choice for the transport layer is CORBA extended with OIF. By wrapping existing tools in CORBA wrappers, tools can be accessed by a distribution-transparent, software-bus mechanism. CORBA is the most mature of such architectures. In contrast with DCOM, it runs on a large variety of operating systems and with programs compiled in many languages. In contrast with Java RMI, it is particularly tuned to legacy applications. And in contrast to the next generation of HTTP, well, for one thing, it's already here.

OIF injectors on CORBA-wrapped tools and services can be used to

- Maintain the annotations of artifacts created by running tools. Injectors can note, for example, the "owner" of a process and include in the repository the information about that owner, or store pointers to the "inputs" of a tool on the annotations of its outputs.
- Enforce complex, not-yet-anticipated access control rules on data, particularly as contractors form federations to deal with design subproblems.
- Enforce automatic data set transformations to translate between representations.
- Supply alternative servers of the same service.
- Report on the status of jobs to distributed managers and debuggers.
- Support "session" environments reflecting user privileges downstream and carrying the user environment.
- Support "long-lived" transactions needed by the design process, (in contrast "database" transactions, such as bank account updates and airline reservations.)
- Obtain and assure the appropriate versions of datasets.
- Provide software redundancy and mobility, enabling moving computations and data for increased efficiency.

Each of these concepts can be realized by inserting the appropriate injectors on some or all of the methods of a tool or service. It is also the case that for each of these, we don't really know at this point what exactly we want done—what we mean by versions, access control restrictions, system management, transactions, and so forth. However, this is a strength of the injector approach, in that we can experiment and resolve these issues by the results of these experiments, rather than demanding omniscience at the original design time.

The Common Services layer would support services such as name serving, matchmaking, trading, events, system management, agent infrastructure (whatever that might mean) and tool wrapping. The Extended Services layer would support services such as repository mining, automatic design search, workflow automation (notifications and actions on events), visual assembly, and knowledge capture. I mention these more for completeness than for their relevance to AOP, though some might encode well as injectors.

Concluding Remarks

The Object Infrastructure Framework provides a mechanism for performing Aspect-Oriented Programming at the component level. We have discussed how the OIF mechanism could be applied to the development of an architecture for an Intelligent Synthesis Environment and how ISE could serve as a challenge problem for AOP

Acknowledgments

The ideas expressed in this paper have emerged from the work of the NASA Ames Variational Design group, the NASA ISE project, and the MCC Object Infrastructure Project. I thank Stu Barrett, Chris Knight, David Korsmeyer, Diana Lee, Barry Leiner, Ted Linden, Johnny Medina, Tarang Patel, and Don Van Drie for their contributions.

References

1. Aksit M. and Tekinerdogan B. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. AOP '98 workshop position paper, 1998. http://wwwtrese.cs.utwente.nl/Docs/Tresepapers/FilterAspects.html
2. Beugnard, A. How to make aspects re-usable, a proposition. In Lopes, C., Bergmans, L., Black, A. and Kendall, L. *Proc. Aspect-Oriented Programming Workshop at ECOOP '99*
3. Dabke, P. Enterprise Integration via CORBA-Based Information Agents. *IEEE Internet Computing 3* (5) September 1999, pp. 49–57.
4. Filman R., Barrett S,. Lee D., and Linden T. Inserting Ilities by Controlling Communications. To appear in *Comm. ACM*, 2000. http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/oif/oif-cacm-final.pdf
5. Filman, R. E., Korsmeyer, D. J., and Lee, D. D. A CORBA Extension for Intelligent Software Environments. To appear in *Advances in Engineering Software*, 2000. http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/oif/williamsburg-submit.pdf
6. Goldin, D. S., Venneri, S. L., and Noor, A. K. Beyond Incremental Change. *IEEE Computer 31*(10), October 1998, pp. 31–39.
7. Hardwick, M., Spooner, D. L., Rando, T. and Morris, K. C. Data Protocols for the Industrial Virtual Enterprise. *IEEE Internet Computing 1* (1), January 1997, pp. 20–29.
8. Harrison, W. and Ossher, H. Subject-Oriented Programming (A Critique of Pure Objects). Proc. OOPSLA '93. *ACM SIGPLAN Notices* 28 (10), October 1993, pp. 411–428.
9. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-Oriented Programming. Xerox PARC Technical Report, February 97, SPL97-008 P9710042. http://www.parc.xerox.com/spl/projects/aop/tr-aop.htm
10. Siegel, J. *CORBA: Fundamentals and Programming*. New York: Wiley, 1996.