

Service-Oriented Programming

by

Guy Bieber, Lead Architect, ISD C4I, Motorola

ABSTRACT - *The Service-Oriented Programming (SOP) model is the most exciting revolution in programming since Object Oriented Programming. Sun's Jini™ and Java technologies are key enablers for this new paradigm. Motorola has been refining the Service-Oriented Programming (SOP) model with its Openwings™ architecture. Together these technologies enable a new generation of service-oriented computing applications. Some of the patterns required for service-oriented computing currently are only supported in the Java programming language: these include code mobility and code security. However, this may not be the case in the future. Just as Smalltalk was the only OOP language at first, other languages may someday support the primitives needed for Service-Oriented Programming. In this paper we provide a brief description of the patterns and primitives required for supporting Service-Oriented Programming.*

SERVICE-ORIENTED PROGRAMMING

To understand Service-Oriented Programming, one needs to understand some of the paradigms that preceded it. These include: OOP, Client / Server, and Component Models. Object Oriented Programming (OOP) is built on the premise that programming problems can be modeled in terms of the objects they represent. Object Oriented Programming has specific characteristics: inheritance, encapsulation, and polymorphism. Service-Oriented Programming builds on OOP, adding the premise that programming problems can be modeled in terms of the services that a component provides and uses.

Component models prescribe that programming problems can be seen as independently deployable black box client / servers which communicate through contracts. The client / server model has become brittle. Service-oriented computing contains Components that publish and use services in a peer to peer manner. In SOP a client is not tied to a particular server. Service providers are all treated equivalently.

These are the architectural elements that make up Service-Oriented Programming:

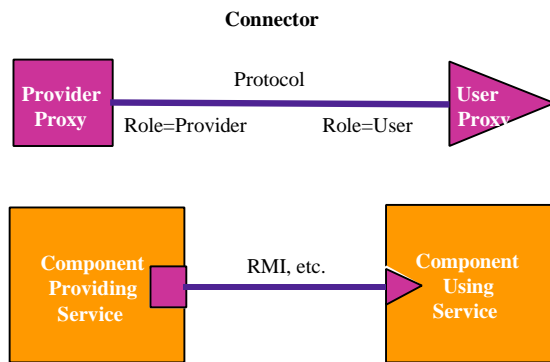
- Contracts – An interface that contractually defines the syntax and semantics of a single behavior.
- Components – Third party, reusable, deployable computing elements that are independent of platforms, protocols, and deployment environments.

Service-Oriented Programming	
Elements	Aspects
• Contracts	• Conjunctive
• Components	• Deployable
• Connectors	• Mobile
• Containers	• Secure
• Contexts	• Available

- Connectors – A connector encapsulates the details of transport for a specified contract. It is an individually deployed element that contains a user proxy and a provider proxy.
- Container – A service that can run components while managing their availability and code security.
- Contexts – A context for deploying plug and play components, that prescribes the details of installation, security, discovery, and lookup.

There are several architectural aspects important to service-oriented computing:

- Conjunctive - This refers to the ability to use or combine services in ways not conceived by their originators. This implies that services have published interfaces that can be discovered.
- Deployable – This refers to the ability to deploy or reuse the component in any environment. This requires environment independence which includes transport independence, platform independence, and context independence.
- Mobile – This refers to the ability to move code around the network. This is used to move proxies, user interfaces, and even mobile agents.
- Available – One of the premises of Service-Oriented Programming is that redundant networked resources can provide high availability. It is the goal of SOP to handle partial failures that plague distributed computing.
- Secure – The concepts of mobile code and network discoverable services provide new challenges for security. While SOP allows services to have a much broader range of use, it can not succeed without protecting these services from misuse.



Just as Object Oriented Programming has a modeling language, namely UML, Service-Oriented Programming also has a modeling language based on Architecture Description Language [5]. ADL contains the concepts of Components, Connectors, Roles, and Ports. An example diagram is shown below:

JAVA™ PATTERNS [1]

Jini™ derives much of its power from Java™, hence the relevant patterns from Java™ are presented here. In an attempt to save space mini-patterns will be throughout this document as follows: name, problem, context, and solution. The key patterns deemed from Java™ are contracts, mobility, and code security.

Pattern Name: Contracts

Problem: How can behaviors be defined independent from implementations?

Context: Anywhere it is desirable to hide complexity.

Solution: The concept of an interface construct was added to Java to describe a behavior both in syntax and semantics. The methods, method types, method parameter types, and field types prescribe the interface syntax. The comments, method names, and field names describe the semantics of the interface. In this model any object can implement the interface and interfaces can use inheritance, including multiple inheritance.

Pattern Name: Mobility

Problem: How can code be moved around the network for execution?

Context: It can be advantageous to move code instead of data for security, performance, or interoperability.

Solution: By creating code that is portable, it can be easily moved from host to host. Java supports both code mobility and state mobility. State mobility is provided through serialization. Code mobility is provided through platform independent code that can be delivered in bundles called Jar files. This can be done using any form of file transfer.

Service-Oriented Programming Patterns

Java™	Jini™	Openwings™
Contracts	Lease	Component
Mobility	Discovery	Connector
Code Security	Lookup	Container
	Service Security	Context
	Service User Interface	Policy
	Transaction	Proxy
	Coordinators	Installer
		Management

Pattern Name: Code Security

Problem: When downloading and running code from numerous sources, how can it be assured the software will not harm the target system?

Context: Anywhere where mobile code is delivered from sources of various levels of trust.

Solution: Java provides a sandbox, certificate based authentication, and granular privilege assignment.

JINI™ PATTERNS [2]

The intent of this section is to give a brief description of the patterns that have arisen from Sun Microsystems Jini™ technology. Jini™ tackles the issues of distributed computing head on, unlike many distributed computing paradigms before it. For example, distributed computing is fraught with problems, such as partial failures, locality of execution, and interface mismatch. Partial failures can occur when elements, such as a network, fails between nodes. Locality of execution becomes a problem when mobile code is brought into the mix. In local invocation this is the question of call by reference versus call by value. In distributed computing this is the question of remote invocation versus object serialization. Interoperability problems occur when elements deployed at different times have incompatible interfaces. Through, the use of mobile code Jini eliminates many of these interoperability problems.

This section describes the following Jini™ patterns: lease, discovery, lookup, service security, service user interface, transaction, and coordinators.

Pattern Name: Lease

Problem: How can resources failures be detected?

Context: In a distributed environment where partial failures can occur, such as network outages.

Solution: Both sides agree to lease a resource for a given period of time. Since, lease expiration can be detected by both sides, regardless of host or network failures, it guarantees that a partial failure will be detected correctly by both parties.

Pattern Name: Discovery
Problem: How can plug and play hardware and software be achieved?
Context: In any system where reduced administration or ease of use is important.
Solution: A bootstrapping protocol is used to automatically find a lookup service. From there everything else can be found. As long as the bootstrapping technique remains the same software can participate in a plug and operate (PLOP) environment.

Pattern Name: Lookup
Problem: How can services be published and discovered based on their contracts and attributes?
Context: Used where it is desirable to publish services for general use.
Solution: Allows publication and lookup of services based on their contracts and attributes. Unlike stovepipe client / servers, service interfaces are published and are usable by any other component.

Pattern Name: Service Security
Problem: How can services be secured from unauthorized access?
Context: Anywhere services are published.
Solution: Though it is a work in progress, securing access to the Jini lookup service and to the services themselves is essential in a large service network.

Pattern Name: Service User Interface [4]
Problem: How can user interfaces be attached to services?
Context: Anywhere where a graphical, audio, or other kind of user interface needs to be provided with a service.
Solution: Jini™ uses the concept of a ServiceUI, which can be attached to any service. Factories are defined for each kind of user interface category (voice, graphics, etc.). These factories are used to get mobile code that provides a user interface. This has the advantage that workstation and server software always stays in sync.

Pattern Name: Distributed Transaction
Problem: How can distributed services have atomic behavior?
Context: Any distributed environment where synchronization is required.
Solution: Transactions have been used for a long time in databases. However, they are also useful in distributed computing. Jini provides a set of contracts for describing distributed transactions.

Pattern Name: Coordinators
Problem: How can services coordinate their actions?
Context: Any distributed environment where services need to work together.

Solution: Jini™ provides a concept called spaces, which is really a combination of a synchronization construct and an object database. The space pattern is based on a prior work called Linda, which was used for parallel computing. The concept is to allow objects to be transactionally written, taken, and read from a shared space. This concept is based on mobile code.
--

OPENWINGS™ PATTERNS [3]

Jini™ and Java™ go a long way to support Service-Oriented Programming (SOP), however, several elements are missing that would allow development of full-scale service-oriented systems. Openwings™ is focused on filling these holes, to provide the full set of elements and aspects intrinsic to SOP.

The following patterns are described in this section: component, connector, container, context, installer, policy, and proxy.

Pattern Name: Component
Problem: What is the unit of service deployment?
Context: Any system where hardware or software needs to be abstracted as services.
Solution: A component encapsulates a unit of deployment of hardware (through software) or software. Components are the basic unit of deployment for services. Services are provided and used by components are contractually specified. Components utilize a peer to peer model (instead of client server). Components are subject to third party composition and are independent of deployment contexts. Components must be independent of platforms, transport protocols, and deployment environment details (such as network topology). Components can be used as the basic unit for mobile agents.

Pattern Name: Connector
Problem: How can services using different transport protocols be interoperable.
Context: Anywhere where interoperability is important.
Solution: Connectors provide an abstraction for transport independence. Connectors are grossly divided into two categories: synchronous and asynchronous connectors. Connectors are composed of two proxies: a user proxy and a provider proxy. One proxy provides an object that implements a contract and the other takes an object that implements a contract. Connectors can naturally be chained. They also provide an insertion point for handling transport security and quality of service. Connectors can be acquired in one of three ways: they can be bundled with a service provider, looked up in a repository, or generated on the fly.

Pattern Name: Container
Problem: How can component execution be managed for security, availability, and mobility?

Context: This pattern is useful when services are deployed as components.
Solution: The most basic container is a service itself: a processing service. The container enforces code security, by setting the Java™ Security Manager. It also provides a concept missing from Java™, the ability to map multiple processes to a single JVM. The container pattern can manage pools of processing resources or JVMs and make load-balancing decisions. Containers can work together to form clusters, which guarantee clustered services are kept running. This feature can be used for cold and warm fail-over of services. Finally, the container model provides a simple environment to support mobile agents.

Pattern Name: Context
Problem: How can a context for discovery and system formation be created and managed?
Context: This pattern is useful in a distributed computing environment where systems need to be self-healing and self-forming.
Solution: A context provides an environment for self-forming and self-healing systems. Core to the context pattern is removing environment specific details from components. By doing this components become truly reusable and deployable in different contexts. Policies are another pattern that can be used to do this (see policy pattern). A context enforces a system boundary, provides for automated installation of components (see installer pattern), provides all of the core services for system formation, and prescribes how services are published and discovered beyond the workgroup.

Pattern Name: Installer
Problem: How can software be installed securely and automatically?
Context: This pattern is useful in any system using the context pattern.
Solution: The installer pattern inverts the traditional installation approach. Typically software is delivered with its own installer. Because this bundled installer has no knowledge of the deployment context, the user must answer many questions about where and how the software should be installed. In a service-oriented environment where code mobility is prevalent, this is unacceptable. Instead, components are delivered as bundles to a context running an installer. The pre-running installer service can then verify the components should be installed (by signature). Since the installer is context aware it knows how and where to automatically install the software. This can even include replicating the software across the context if it needs to be clustered.

Pattern Name: Policy
Problem: How can environment details be abstracted from code?

Context: A policy is useful anywhere a configuration file was previously used.
Solution: Policies are discoverable configuration files. For instance, the context pattern uses policies to tell deployed components information about their deployment environment. Policies have both a human readable form and an object form. The policy object knows how to store and restore itself from human readable form (in this case XML).

Pattern Name: Proxy
Problem: How can a programmatic interface be delivered in a mobile fashion.
Context: Proxies can be used behind any interface to add functionality to an object.
Solution: Proxies can provide an object that implements a contract or take an object that implements a contract. Proxies are the primitives used to create connectors and smart proxies. Smart proxies allow users to add additional layers of functionality behind an interface.

Pattern Name: Management
Problem: How can zero-administration systems be built?
Context: Any environment where administration needs to be minimized.
Solution: Every component has a management framework that allows different management aspects to be added at runtime, i.e. Management Beans (MBeans). Management Beans have a published interface much like a service, but they are intended to provide the behind the scenes service management. The management function is intended to be lightweight and typically is automated through the use of policies. At times, for instance when debugging a system, it is helpful to provide user interfaces with a MBean. This can be done with the Jini ServiceUI pattern. However, during normal operation management is intended to be as automated as possible.

CONCLUSION
Service-Oriented Programming (SOP) is a new paradigm for computer science that requires a different way of thinking of distributed problems. Though the model was originally designed for inter-process communication, it holds true for intra-process communication, i.e. between threads. The reason that threaded and distributed computing is currently fraught with errors, is largely due to the fact that contracts are not clearly defined. It is particularly bad in threading models, where calls are being made directly into the implementations of objects running in different threads.

Java™, Jini™, and Openwings™ are providing the first fully functional framework for SOP. In describing the patterns for SOP it should have become clear that some of the patterns can only be supported in a Java™ programming environment at this time (namely code mobility and code

security). Until these capabilities are added to other languages / paradigms it will be very difficult to implement Service-Oriented Programming in other languages.

REFERENCES

1. Java, <http://java.sun.com>
2. Jini, <http://www.jini.org>
3. Openwings, <http://www.openwings.org>
4. ServiceUI project, <http://www.jini.org>
5. Architecture Description Language (ADL), http://www.cs.cmu.edu/~acme/acme_documentation.html