Introduction to Service-Oriented Programming (Rev 2.1)

by

Guy Bieber, Lead Architect, Motorola ISD

Jeff Carpenter, Software Engineer, Motorola ISD

ABSTRACT - *A new programming paradigm is forming throughout the software industry. This paradigm is driven by the exploitation of networking technology and the need to be able to create more powerful capabilities more quickly. The diversity in languages, middleware, and platforms has prevented larger constructs from being formed and the shortage of qualified software engineers only aggravates the problem. The inception of the Service-Oriented Programming (SOP) paradigm is being defined throughout the industry including: Sun's Jini™, Openwings™, Microsoft's. NET™, and HP's CoolTown™. Much like the early days of Object-Oriented Programming (OOP), certain characteristics of SOP are covered by some implementations, but no one approach covers all of them. Until the key features of OOP (encapsulation, inheritance, and polymorphism) and a design methodology (OOA/OOD) had been defined, consistency in OOP programming models was not achieved. This paper analyzes Service-Oriented technologies to identify the key characteristics and patterns of SOP, and demonstrates the value of SOP to developers and end users.*

KEY WORDS – *Service-Oriented, Contract, Component, Connector, Container, Context, Availability, Discovery, Interoperability, Security, Network-centric, Patterns.*

INTRODUCTION

With the advent of Internet technology, people are becoming more closely connected. So far, this connectivity is only skin-deep. People can send e-mail or instant messages to family members, or share files over the web. Some capabilities have been wrapped up neatly into web pages: free Internet telephony, text translation, price comparisons, and Internet garage sales (auctions). These services are useful in themselves, but the Internet is still very stove-piped; connecting these services together to do more powerful things is very difficult. For instance, it might be nice to have a service call one's cell phone when an item meeting a price constraint is found on an auction or in retail sales. It is very difficult for someone to use Internet services in ways not intended by the original author. This is typically due to poorly defined interfaces or lack of documentation on interfaces. When it becomes possible to utilize services to create new, more powerful constructs, the power of networking will be fully exploited.

To understand Service-Oriented Programming, one needs to understand some of the paradigms that preceded it, including Object Oriented Programming (OOP), Client-Server, and Component Models. OOP is built on the premise that programming problems can be modeled in terms of the objects in the problem domain. Object Oriented Programming has specific characteristics: inheritance, encapsulation, and polymorphism. Service-Oriented Programming builds on OOP, adding the premise that problems can be modeled in terms of the services that an object provides or uses.

What is a Service?

A service is a contractually defined behavior that can be implemented and provided by any component for use by any component, based solely on the contract.

Component models prescribe that programming problems can be seen as independently deployable black boxes that communicate through contracts. The traditional client-server model often lacks well-defined public contracts that are independent of the client or server implementation. This has made the client-server model brittle. In Service-Oriented Programming, components publish and use services in a peer-to-peer manner. In SOP a client is not tied to a particular server. Instead, service providers are interchangeable.

SERVICE-ORIENTED TECHNOLOGY

The software industry has been putting out strong messages that the future of distributed computing is service-oriented. These messages are coming from many of the industry big hitters: Microsoft, Hewlett Packard, Sun Microsystems, and Motorola. The message may be difficult to see, because each company has conceptualized the service-oriented model in their own technology initiatives: Microsoft .NET™, Hewlett Packard Cooltown™, Sun Microsystems Java™ / Jini™, and Openwings™.

Service-Oriented Programming is a paradigm for distributed computing that supplements Object Oriented Programming. Whereas OOP focuses on what things are and how they are constructed, SOP focuses on what things can do.

This paper examines these technologies in detail to derive what qualities make software service-oriented. We identify the architectural elements and aspects of Service-Oriented systems. We also propose a modeling language for SOP and identify useful design patterns.

MICROSOFT .NET

The Microsoft .NET™ [1, 2] effort provides an Internet Operating System, bridging applications from the traditional desktop to the Internet. Microsoft has recognized that ubiquitous network connectivity has not been fully exploited. The vision is that future applications will be built not only by integration of local services, but integration of services across the Internet. Microsoft sees this effort as a way to decrease time-to-market, to achieve higher developer productivity, and to improve quality.

Microsoft is focusing on language independence, as opposed to platform independence. This is a similar approach to that taken by the Common Object Request Broker Architecture (CORBA). .NET ignores object model issues, instead focussing on messaging. This could be interpreted as a direct attack on the Java model. The following table gives a brief summary of the core components of the Microsoft .NET™ strategy.

Core Component	Description
.NET™ Internet Operating System Services	Provide necessary services such as security (PASSPORT.NET™), file storage, user preference management, calendar management, etc.
.NET™ Development Infrastructure	This includes Visual Studio .NET™, .NET™ Enterprise Servers, the .NET™ Framework, and Windows .NET™.
.NET™ Device Software	Framework for devices to participate.
.NET™ User Experience	Delivery of a personal portal accessible from any network device.

Figure 1. The Microsoft .NET Strategy

Some parts of the .NET™ framework are well defined, but others are still immature. Recently, Microsoft demonstrated some of the technology behind the marketing at the Fall 2000 COMDEX trade show. The demonstration focused on the .NET™ Development Infrastructure. Microsoft is pushing several new technologies to enable .NET™: Service Contract Language (SCL), Simple Object Access Protocol (SOAP), Disco, C#, and the Common Language Runtime (CLR).

Service Contract Language (SCL) is a language for defining language-independent message interfaces. This is very similar to CORBA Interface Definition Language (IDL) and DCOM Microsoft Interface Definition Language (MIDL). The important concept to recognize here is a standard for defining service interfaces.

Simple Object Access Protocol (SOAP) is an Extensible Markup Language (XML) based means of passing messages that is intended to be language-independent. It is the equivalent of method invocation-based on-the-wire protocols such as Remote Method Invocation (RMI) Wire Protocol and CORBA Internet Inter-Orb Protocol (IIOP). SOAP is certainly less efficient than RMI or IIOP since XML messages are text-based and contain tagging information. Instead of a protocol-independent approach, Microsoft has chosen to define a new protocol.

Disco is Microsoft's upcoming strategy for service discovery. This is yet another spin on Microsoft's plug-and-play technologies. The core concept to identify here is service discovery.

C# is Microsoft's attempt to provide a programming language that is network-friendly and that has something that Active Server Pages (ASP) really lacks: security. Microsoft realized that even if they are sending mobile code between Windows machines, they must ensure the code is not malicious. C# syntax is similar to Java, except the APIs look like Win32 APIs. The key concept to recognize here is a secure platform for mobile code.

The *Common Language Runtime* (CLR) is an attempt to bring the service paradigm to Dynamic Link Libraries (DLLs). The concept is to define language-independent interfaces to DLLs that include the object code, interface definitions, and a description of the interface. The key element to notice here again is the concept of a contract.

In demonstrating their new Visual Studio .NET™ at COMDEX, Microsoft showed Web Services that had not been combined before being put together to make applications. The ability to put services together in ways not envisioned by their authors is called conjunction.

Microsoft is also working on servers to host these web services as shown in the following figure:

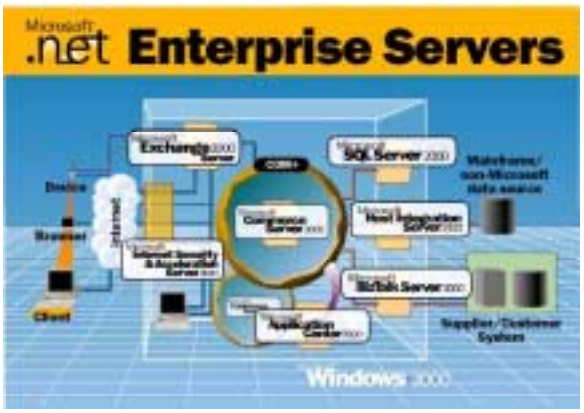


Figure 2. .NET™ Enterprise Servers

The following tables summarize the SOP elements and characteristics demonstrated by Microsoft .NET™.

Element	.NET
Contract	Service Contract Language (SCL)
Component	Web Service Providers
Container	.NET™ servers

Figure 3. Microsoft .NET SOP Elements

Characteristic	.NET
Conjunctive	Services can be combined in new ways.
Interoperable	SCL and SOAP provide some interoperability for .NET™.
Secure	C# is an attempt to provide a secure language.
Available	DISCO discovery is essential to achieving availability (i.e. fail-over)

Figure 4. Microsoft .NET™ SOP Characteristics

HEWLETT PACKARD COOLTOWN™

Hewlett Packard has elevated the user experience in service-oriented computing to the forefront through a technology called Cooltown [3]. Cooltown is built on web technologies such as Hypertext Transfer Protocol (HTTP). Cooltown promotes the idea of bridging the physical world and digital world (the web). The goal is to give people, places and things (objects) a digital presence on the web that people can then interact with. The Internet contains content relating to the objects around us, but the content is not directly linked to the objects themselves. Cooltown attempts to enrich interaction with the physical world by providing a digital presence that allows information and control to flow naturally around us.

Much of this technology focuses on the concept of discovery by location: being able to discover and interact with the objects around you. In a museum this could mean getting information delivered to you about the painting you are viewing. Cooltown envisions every object being represented by a web page. Pads are handheld devices that support web browsers and object detection. Pads can detect objects by reading bar codes, RF / IR tags, or a beacon (i.e. a Universal Resource Locator (URL) broadcaster). All of these technologies are simply used to deliver a URL. Other supporting technologies are Global Positioning Systems (GPS) and Bluetooth [4]. Objects can be very passive, providing a reference that can be associated with a URL. Other devices simply provide a URL that is hosted elsewhere. Some devices will actually contain a web server to deliver content.

The figure below shows some of the elements of HP Cooltown: beacons, tags, portals, pads, and places.

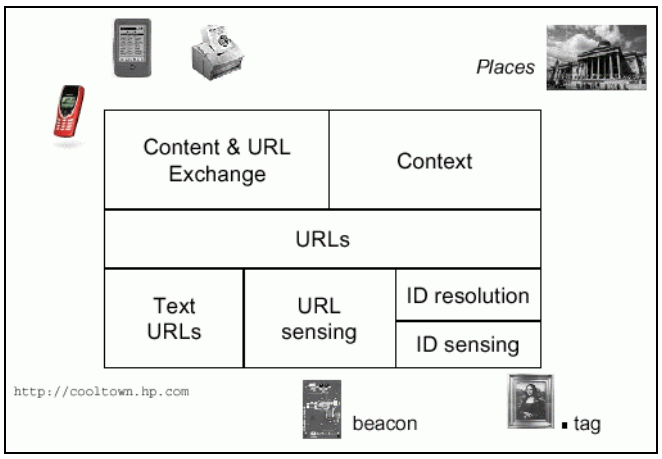


Figure 5. Cooltown™

A *place* corresponds to a location in the real world. A *tag* is provided by an object as a reference to a URL. A *beacon* delivers an object's URL. A *pad* is any device that can display a web browser and can sense beacons or tags. A *portal* provides a connection from a pad to one or more web servers. Examples of portals could be wireless Internet access such as 802.11. Hewlett Packard's Chai Server allows web servers to be embedded in devices.

The one core underlying assumption of HP Cooltown, is a web-based interface. Cooltown does not address programmatic interfaces directly, focusing instead on the user interfaces provided by web pages. This approach inhibits the conjunctive and interoperable aspects of SOP. However, it does demonstrate some of the characteristics of SOP.

The following tables summarize the SOP elements and characteristics demonstrated by HP Cooltown™.

Element	Cooltown™
Component	People, Place, Thing
Container	Web Server

Figure 6. HP Cooltown™ SOP Elements

Characteristic	Cooltown™
Mobility	Delivery of web pages and Java™ code over the web provides mobility.
Discoverable	The sensing of beacons and tags is a discovery mechanism.
Interoperable	HTTP is the standard of interoperability in HP Cooltown™.
Available	Cooltown™ location-based discovery allows users to switch between objects easily.

Figure 7. HP Cooltown™ SOP Characteristics

SUN JINI TECHNOLOGY

Sun's Jini™ Network Technology [5] is a framework for building systems spontaneously. Jini™ technology makes it possible to build a system out of a network of services. Services can be added or removed from the network, and new clients can find existing services. This all occurs dynamically, with no administration.

Services are based on well-known interfaces written in the Java™ programming language. Whether a service is implemented in hardware or software is not a concern. The service object downloaded to a user is supplied by the component providing the service. The client only knows that it is dealing with an implementation of an interface written in the Java™ programming language. A design based on service interfaces makes it possible to build systems with higher availability. A component can use any service that complies with the interface, instead of being statically configured to communicate with a certain component.

Jini™ technology is built on top of Java™ (see Figure 8 below). Java's Remote Method Invocation (RMI) provides remote garbage collection of remote objects and the ability to passing object state as well as object code around the network.

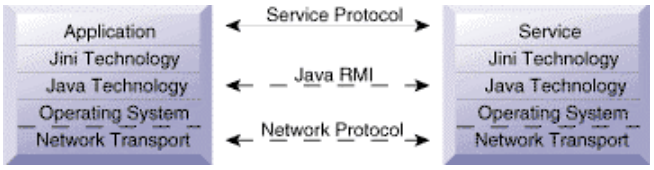


Figure 8. Jini Architecture

The Jini™ architecture has several aspects that enable Service-Oriented Programming.

Discovery is the process by which components locate a Jini™ system. The discovery protocol allows components to locate a repository of services called a Lookup Service. Lookup Services are discovered through a multicast request or a unicast request to a well-known location.

Join is the process of adding a service to a Jini™ system. After a component has discovered a Lookup Service, it registers an object for the service into the Lookup Service. This service object contains the Java™ programming language interface for the service. This includes the methods that users and applications will invoke to execute the service, along with any other descriptive attributes.

Lookup is the process by which a client or user locates and invokes a service, described by its interface type. More fine-grained selection of is made possible through the use of other attributes.

Federation is the act of associating Lookup Services together to provide a scope for Join and Lookup. Lookup Services can also be bridges to other naming and directory services.

Key aspects of the Jini™ Architecture

- Discovery
- Join
- Lookup
- Federation
- Security
- Leases
- Transactions
- User Interfaces

Security for Jini technology is currently under development, but will be built on the Java security model, including the RMI security effort.

Leases are grants of guaranteed access to a resource for a given time period, negotiated between service users and service providers. If a lease is not renewed before it's expiration then the user and the provider of the service consider the resource to be freed. A lease expires if the resource is no longer needed, the service user or network fails, or the renewal request is denied. Jini™ uses this model to reclaim resources based on the failures inherent in distributed systems.

Transactions are used to wrap a series of operations together so that the operations either succeed or fail as a unit. The Jini™ Transaction Protocol takes an object-oriented view of transactions. Instead of defining specific transaction semantics, Jini™ supplies interfaces that describe a generic two-phase commit process.

User interface adapters can be provided along with services. These adapters allow the service to be accessed directly by a user on a particular platform. These adapters can be graphical, voice-based, or other interface technologies.

The following tables summarize the SOP elements and characteristics demonstrated by Sun Jini™:

Element	Jini™
Contracts	Service Interfaces

Figure 9. Jini™ SOP Elements

Characteristic	Jini™
Mobility	Delivery of service proxy objects and associated code over the network.
Conjunctive	Services provided over the network can be aggregated to perform new tasks.
Interoperable	The Java language™ is the standard of interoperability.
Available	Equivalent services can be utilized in the case of service failure.
Secure	Planned for a future release, based on Java RMI security.

Figure 10. Jini™ SOP Characteristics

OPENWINGS™

Openwings™ [6] is a service-oriented architectural framework for building systems and systems of systems. Although not tied specifically to Jini™, it builds upon Java™ and Jini™ concepts to provide a more complete solution. The figure below shows a high level diagram of the Openwings™ architecture.

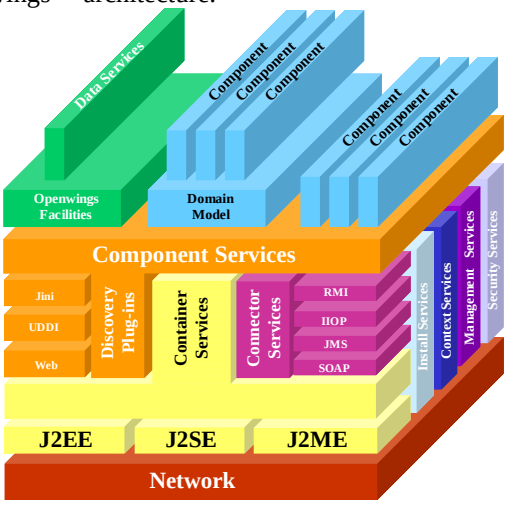


Figure 11. Openwings Architecture

Several of the core services provide aspects of service-oriented computing, as described below:

Component Services provides convenient abstractions of the service discovery / lookup semantics – components simply provide and use services. Component services allows multiple service discovery mechanism to be used transpartently. Additionally, there are simple semantics for adding service attributes and user interfaces.

Connector Services provides an abstraction of protocols (both synchronous and asynchronous). This enhances the model of service-oriented computing by making it possible for service providers and users to be completely transport-independent. Connectors can be distributed with components or built dynamically at runtime. The Openwings reference implementation contains a component called a Connector Repository, which provides a service that generates and stores connectors based on the applicable service interface.

Container Services provide processing as a network service. The container is responsible for lifecycle, availability, and code security for components. Container services provide a process abstraction for the Java virtual machine as well as the ability to control non-java executables. Container services also provide a clustering capability for high availability systems.

Context Services creates a context for service discovery and component deployment. An Openwings context is a grouping of hardware and software components that use each other's services (i.e. a system). When a component is placed in a context, its services can be utilized by other components in that context. Contexts can enter into relationships with other contexts to share services, thus forming systems of systems.

Security Services meets the challenges posed by evolving, unpredictable threats in a network-based service framework. Openwings focuses on three major aspects of securing service-oriented systems: service security (authorization & authentication for service usage), mobile code security, and transport security (integrity and confidentiality).
Install Services

Install Services provide the ability to easily deploy components. Install Services allow zero-interaction installation of components onto platforms. Install Services provide component authentication and resolution of dependencies.

Management Services abstracts the management of software components, hardware components and networks. Management services provide a simple management framework that allows plug-ins for various legacy management frameworks .

Openwings Facilities are non-core Openwings services that are deemed of general use. *Data Services* provides a simple abstraction of persistence mechanisms by providing an object view of database, regardless of the underlying storage mechanism.

The following table summarizes the SOP elements demonstrated by Openwings™

Element	Openwings™
Contracts	Service Interfaces
Component	Component Services
Connector	Connector Services
Container	Container Service
Context	Context Services

Figure 12. Openwings™ SOP Elements

Openwings™ shares the SOP characteristics provided by Jini™, with the following additions:

Characteristic	Openwings™
Secure	Role-Based Access Control to services
Deployable	Openwings Components can be installed over the network into Spaces or specific platforms. Connector Services provides protocol independence and Policy Services provides environment independence.

Figure 13. Unique Openwings SOP Characteristics

SUMMARY OF SERVICE-ORIENTED TECHNOLOGIES

All of the initiatives discussed so far have key focus areas and key technologies they depend on, as seen in the following table.

Initiative	Focus	Dependencies
.NET™	Language-Independent Services	SOAP, IP, SCL, XML, DISCO, WINTEL, HTTP
Cooltown™	User / Service Interaction	HTTP, IP
Jini™	Platform-Independent Service Discovery	Java, HTTP, IP
Openwings™	Service-Oriented Programming	Java, HTTP, IP

Figure 14. Summary of SOP Technologies

ELEMENTS OF SOP

The analysis of several Service-Oriented technologies has yielded a set of common architectural elements that make up Service-Oriented Programming:

- Contract – An interface that contractually defines the syntax and semantics of a single behavior.
- Component – A third-party deployable computing element that is reusable due to independence from platforms, protocols, and deployment environments.
- Connector – An encapsulation of transport-specific details for a specified contract. It is an individually deployable element.
- Container – An environment for executing components that manages availability and code security.
- Context – An environment for deploying plug and play components, that prescribes the details of installation, security, discovery, and lookup.

ASPECTS OF SOP

There are also several architectural aspects that characterize service-oriented computing:

- Conjunctive - This refers to the ability to use or combine services in ways not conceived by their originators. This implies that components have published interfaces for the services they provide.
- Deployable – This refers to the ability to deploy or reuse a component in any environment. This requires transport independence, platform independence, and environment 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. This is the key enabler for interoperability.

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

- 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 the 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 cannot succeed without protecting these services from misuse.
- Interoperable – Interoperability is the ability of components from different sources to use each other's services. SOP supports this through two key features: code mobility and contracts. Interoperability is achieved by sending code across the network that complies with the contract. This code could be a proxy or even an entire application.

ARCHITECTURE DESCRIPTION LANGUAGE

Just as Object Oriented Programming has a modeling language, namely Unified Modeling Language (UML), Service-Oriented Programming needs a modeling language as well. Architecture Description Language (ADL) [7], developed at Carnegie Mellon University, is a modeling language that provides notation for most of these architectural elements of SOP. ADL contains notation for Components, Connectors, Roles, and Ports. A proposed modification to ADL for SOP adds notation for containers and contexts, which could be viewed as specialized components. An example diagram is shown below:

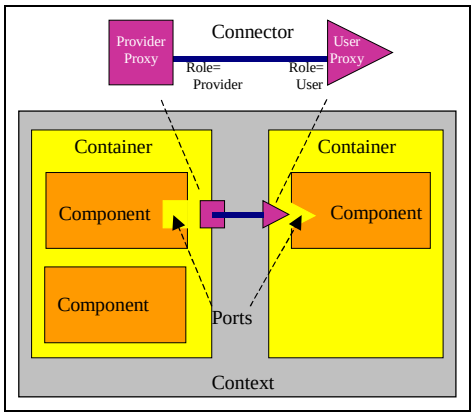


Figure 15. Architecture Description Language

PATTERNS FOR SERVICE-ORIENTED PROGRAMMING

In the following pages we present design patterns that apply to Service-Oriented Programming derived from Java™, Jini™, and Openwings™. The patterns from Microsoft .NET™ and HP Cooltown™ are covered by these patterns. In the interest of brevity, mini-patterns are used throughout this section as follows: name, problem, context, and solution.

JAVA™ PATTERNS

Jini™ derives much of its power from Java™, hence relevant patterns from Java™ are presented here: contracts, mobility, and code security.

Pattern Name: Contracts
Problem: How can behaviors be defined independent of 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. An object can implement multiple interfaces. This pattern occurs in .NET™ SCL.

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 making code 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, delivered in bundles called Java Archive (JAR) files using any form of file transfer.

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 security sandbox, certificate based authentication, and granular privilege assignment.

JINI™ PATTERNS

Unlike distributed computing paradigms before it, Jini™ tackles the hard issues of distributed computing. Distributed computing is fraught with problems such as partial failures, locality of execution, and interface mismatch. Partial failures can occur when elements, such as networks, fail between nodes. Locality of execution refers to the question of call by reference versus call by value. In distributed computing, this becomes 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 these interoperability problems. This section describes the following Jini™ patterns: lease, discovery, lookup, service security, and service user interface.

Pattern Name: Lease
Problem: How can resource 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. Lease expiration is detected by both sides, regardless of host or network failures, so partial failures are guaranteed to 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. This pattern occurs in .NET™ Disco and in the Cooltown™ tag / beacon discovery.

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: Securing access to the Jini™ lookup service and to the services themselves is a work in progress.

Pattern Name: Service User Interface
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 [8], 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 client and server software always stay in sync.

OPENWINGS™ PATTERNS

Jini™ and Java™ provide features that enable 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 (a component can provide or use many services). Services provided and used by components are contractually specified. 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.

Pattern Name: Connector
Problem: How can components using different transport protocols be interoperable?
Context: Anywhere where interoperability and efficient bandwidth use is important.
Solution: Connectors provide an abstraction for transport independence. Connectors are categorized as synchronous or asynchronous. Connectors are composed of a user proxy and a provider proxy. A user proxy provides an object that implements a contract and a provider proxy takes an object that implements a contract. Connectors can naturally be chained. They also provide an insertion point for transport security and quality of service. Connectors can be bundled with components, located 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: An application server is an example of a container. A Java container would enforce code security by configuring the Java™ Security Manager. The container also provides a concept missing from Java™, the ability to map multiple processes to a single Java Virtual Machine (JVM). This reduces the startup time for Java programs. The container pattern can manage pools of JVMs and make load-balancing decisions. Containers work together to form clusters, which guarantee clustered services are kept running. This feature enables cold and warm fail-over of services. Finally, the container provides an environment to support mobile agents.

Pattern Name: Context
Problem: How can a context for system formation and service discovery be created and managed?
Context: This pattern is useful in a distributed computing environment where systems need to be self-forming and self-healing.
Solution: A context provides an environment for self-forming and self-healing systems. A context enforces a system boundary, provides for automated installation of components (see the Installer pattern), provides the core services for system formation, and prescribes how services are published and discovered beyond the workgroup. The context pattern relies on environment-independent components (see the Policy pattern).

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 how the software should be installed. This pattern is simply unworkable for mobile code. Instead, components are delivered as bundles to a context with an installer. The installer service verifies and installs the software automatically throughout the context.

Pattern Name: Policy
Problem: How can environment-specific 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 (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 and insight into system operation is important.
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 published interfaces much like services, but they provide behind-the-scenes service management. The management function is lightweight and can be automated through the use of policies.

SERVICE FAILURE HANDLING

One aspect of Service-Oriented Programming that deserves closer examination is how availability is achieved at the component level. Components that use services must deal with the realities of distributed computing; namely, that services in distributed systems can fail in unique ways [9]. In distributed technologies such as Java™ RMI, Jini™, and Openwings™, this is brought home to developers by the convention that every method of a service interface must throw a `java.rmi.RemoteException`. How a component deals with a service failure is purely the decision of the developer. Realistically, there are an unlimited number of ways to handle service failures. However, it is helpful to consider a number of possible strategies for exception handling.

DO-NOTHING SOLUTION

The simplest approach is the do-nothing solution: the broken service object is discarded, and no immediate attempt is made to recover or locate another equivalent service. The component proceeds to do other things and avoids activity that involves using the service.

```
try
{
  service.doSomething();
}
catch (RemoteException e)
{
  service = null;
  // log the error
  System.out.println(e.toString());
}

// continue…
```

Figure 16. Do-nothing code example

PROPAGATE THE EXCEPTION

The do-nothing solution is obviously not adequate for most situations. Alternatively, the `RemoteException` may be rethrown and handled at a higher level:

- Components that have user interfaces could enter a state where they await some user action that initiates the lookup and use of another service.
- Components using session-oriented (stateful) services where information pertinent to the component's use of the service is remembered over a series of calls instead of passed in every call. Recovery in this case requires code that attempts to roll back to a well-defined state at which it is appropriate to find and use a different service.

LOCATE ANOTHER SERVICE

Another approach is to attempt to recover by locating another equivalent service. The code that uses the service can be placed in a loop to attempt to locate another equivalent service. If an appropriate service is found, the component can continue normal operation.

```
boolean success = false;
int tries = 0;

while (!success && tries<3)
{
  tries++;
  try
  {
    service.doSomething();
    success = true;
  }
  catch (RemoteException e)
  {
    // log the error
    System.out.println(e.toString());

    // attempt to locate a new service
    service = ...
  }
}
```

Figure 17. Simple recovery code example

This code attempts to limit the number of attempts made to locate a new service so that the code does not go into an infinite loop. There are several variations on this method:

- The number recovery attempts can vary. The code above makes three attempts.
- The recovery attempt could be governed by a timeout instead of a specific number of tries. This requires a multi-threaded approach.
- This approach could also benefit from Aspect-Oriented Programming, which is provided in the Java language by AspectJ technology [10]. This would allow common exception handling blocks to be applied across entire applications.

This is still a fairly primitive approach. It may be cumbersome to surround every method call on a service

with a block of code like the one above.

USE A SERVICE PROXY

A technique to avoid the introduction of extra code blocks is to use a layer of abstraction to hide service failures as much as possible. This approach uses the Proxy pattern. The client code receives a service object that is a proxy for actual services. This proxy is a local object, and when one of its methods is called, it in turn invokes the same method on the service object, catching `RemoteException`. If the remote service should fail, the proxy uses the strategy outlined above to locate and use an equivalent service. If the proxy cannot succeed in completing the call using some service object, it raises a `RemoteException`.

This approach is actually fairly simple to implement. As of version 1.3, the Java programming language contains a feature called dynamic proxies [11]. Using the reflection capabilities of the language, the JVM can dynamically create objects that implement one or more interfaces. These dynamic proxies pass method invocations to another object that handles method calls. This method-handling object must implement the `java.lang.reflect.InvocationHandler`. A properly written `InvocationHandler` would be used to map method calls to one or more service objects and trap remote exceptions.

A variation on this method made possible by the use of a proxy is the ability to continually maintain a pool of equivalent services, so that no time is spent on lookup if a new service instance is needed.

SERVICE-ORIENTED EXAMPLE

The real power of Service-Oriented Programming may not be apparent until it is put into action. This section develops an example problem and proposes a solution using Service-Oriented Programming. The goal is to demonstrate the value provided by SOP throughout the life cycle of a product. As is often true in real life, the evolution of the problem can be even more interesting and challenging than the original problem.

THE PROBLEM

A MP3 player vendor has decided to one-up the competition by differentiating their product. They have decided to produce an extensible product that allows any source of audio to be directed to any audio capable device. Users are no longer content just to search for files, download or rip them, organize them, and play them on their PC or MP3 device. Users want to be able to speak the name of the song, artist, or type of music they want and have music found, downloaded, organized, and played for them wherever they are. They want to take advantage of the many sources of audio content available on the web. The user should be able to play their audio on a variety of devices.

THE SERVICE-ORIENTED SOLUTION

To decompose this problem, one must define the components and service interfaces. The first step in this process is to look for standard service interfaces for this problem domain. If these interfaces already exist, it is likely that one or more components that implement this interface are already available as products that can be integrated into the solution. For this example, the assumption is that the service interfaces need to be created. In the interest of space, the interfaces for the example are summarized below.

1. *Audio Player* - This service interface controls the playback of an audio file, stream, or play list (represented by a URL). This includes standard features such as play, pause, stop, fast forward, rewind, mute, skip forward, and skip backward.
2. *Audio Recorder* – This service interface is used to control recording of audio and can produce an audio stream or audio file.
3. *Audio Finder* - This service interface takes in an audio description (title, artist, album, etc.) and returns one or more hits as URLs.
4. *Audio Store* – This service interface takes in a audio file or audio stream reference and organizes available information about it, such as artist, title, genre, etc.
5. *Audio Codec* – This service interface is used to publish audio codecs.
6. *Speech to Text Service* – This service interface takes in an audio stream or file. It performs speech recognition and outputs text. One of the attributes of the service is the language being translated.
7. *Text to Speech Service* – This service interface takes in a text stream and outputs the audio representation as a stream or file.
8. *Audio Provider* – This service interface is used for net broadcasters or audio providers (such as Napster) to publish their services.
9. *Audio Stream* – This service interface provides steaming audio using some codec.

The figure below shows the components and services in the initial design in ADL notation. The connectors have a user role (triangle) and a provider role (square). The numbers on the ends of the connectors correspond to the previously defined service contracts.

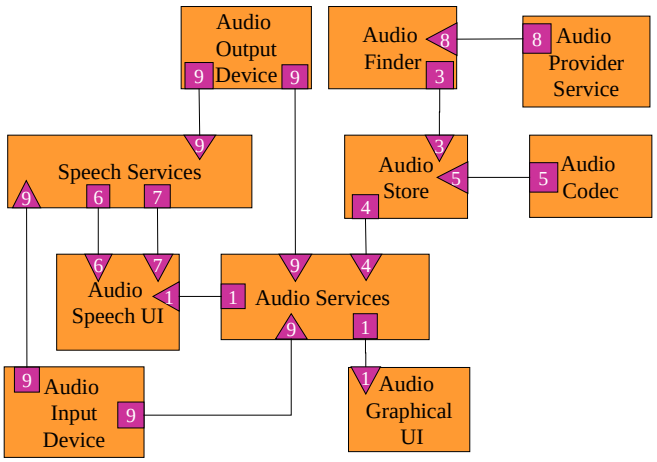


Figure 18. ADL Problem Model

Here is a brief summary of the components in this diagram. Components in bold are included as part of the core product offering:

- ***Audio Services Component*** – This is the vendor's existing product. It is an audio player, audio recorder, and CD ripper.
- ***Audio Graphical User Interface*** – This is a service user interface delivered directly from the audio services component.
- ***Audio Speech UI*** - This is another service user interface delivered from the audio services component.
- *Speech Services Component* – This is a third party tool that provides text-to-speech and speech-to-text conversion. The intent is to allow audio services to work with speech control.
- ***Audio Store Component*** – This component manages audio files, audio stream references, and codecs. The Audio Service goes to the audio store to access any available audio providers. This is packaged with the product.
- *Audio Provider Component* –Any provider of audio content locally or on the web. This is not packaged with the product.
- ***Audio Finder Component*** - This service locates audio providers to find specific audio content. This is packaged with the product.
- *Audio Codec* – These codecs are published on the web.
- *Audio Output De*vice - Any device that accepts an audio stream.
- *Audio Input Device* - Any device that provides an audio stream.

Certain components work better if they are shared. For instance, speech services usually require training to achieve high quality recognition. If every product contained its own speech service, users would waste time training different services. By making speech services shared, they can integrate easily with whatever products and services the user buys.

The figure below shows an example deployment. The hardware includes two PCs (with broadband connections), one stereo, and multiple radios throughout the house. From a software perspective, there are four components that implement the Audio Player interface (Interface 1). One player is attached to a stereo directly, another is on a PC, and a third is tied to a short-range FM transmitter.

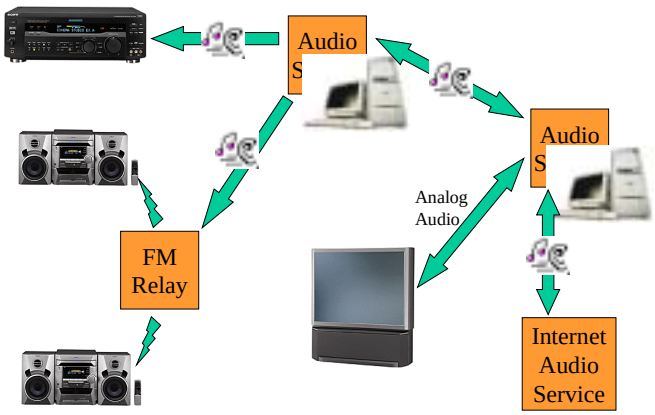


Figure 19. Example MP3 system

Some of the components in this diagram lookup the services they use, but others delegate lookup to an external controller called a coordinator. A coordinator component can be used to configure connection of passive services. The coordinator may assign services itself, or it may have a user interface to allow the user to do the configuration. The Audio Services component acts as a coordinator, allowing the user to direct a particular audio stream to particular rooms in the house. This is easy to visualize for the user: relevant components are shown as pictures of what they really are and service contracts show up as labels or icons. Then it is simply a matter of connecting the dots to do powerful things.

One important thing to note here is that since discovery is being used, the system is self-forming and self-healing. The system is self-forming because new components can be added dynamically and used in the system without changing the existing components. For instance, if another player is added to the system, it simply appears as another option for audio output. If a player is removed from the system, it is no longer shown as an option for audio output. In a service-based system, it doesn't matter if the device provides a single service or all-in-one capability, since each service is a separate entity.

EVOLVING THE PROBLEM

Competitors will inevitably develop competing standards for audio services on the web. The strategy to overcome this is to publish an adapter service that complies with the competitor's service interface definition and translates it to the vendor's definition.

One way the system might evolve is to extend audio services to the car. Motorola started with car radios and has come back to them, with a twist, in the iRadio [12]. What if playlists of music and information from the home could follow users wherever they go, including the car? If a car can be detected over Bluetooth or HomeRF [13], the audio can be downloaded to the car stereo. The car in this instance would contain an audio store and audio player.

As wireless connectivity increases, even children's toys can participate in service discovery. For instance, Motorola's new cable modems have HomeRF wireless access built in. In fact, Sally's doll can play her favorite songs or allow Mom to call her down to dinner. Again, this is simply another audio recorder and player.

This capability could be extended to the daily workout, using a wireless audio player that can store audio. When the device is in the house it is discovered and audio can be sent to it. This could even be the audio recorded from a favorite nighttime comedy. The cable box could become an audio source. More instances of these same components are used to achieve this.

This same player could be used at house parties or nightclubs. Everyone brings their wireless audio devices containing their favorite songs. The house playlist is generated from this collection of music, creating a more interactive experience.

BENEFITS OF THE SOP APPROACH

The technologies to achieve this vendor's evolutionary goals actually exist today. The advantages to the developer are that their existing work was never broken, and extending the system is easy. For users, SOP allows them to connect the services in new ways that add capabilities and value. The ability to redirect audio anywhere in the house, to the car, or to personal devices utilizing wireless technology provides tremendous new value. Instead of running wires between stereo components, the user draws lines between components to achieve the desired configuration. A traditional stovepipe client-server system would have never allowed the expansion or flexibility the SOP solution provides.

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. communication between objects contained in different threads within the same program. The reason that both threaded and distributed computing are currently fraught with errors is that contracts are not clearly defined. The problem is particularly bad in threading models, where calls are often made directly into the implementations of objects running in different threads.

Key benefits of Service-Oriented Computing

For developers	For users
• Truly reusable components • Simplified system integration • Extensible Systems	• Available systems • Secure data • Plug and Play • Zero-administration • Accessibility of services

The service-oriented approach and service-oriented frameworks such as Openwings™ provide many benefits for developers and system integrators. Building software components is simplified by the enforcement of good object-oriented design principles. Component design is driven by the interfaces of the services provided. This in turn simplifies integration of systems. The prototyping of components is also simplified. A fully service-oriented component framework such as Openwings™ Component Services makes it possible to build truly reusable, non-trivial software components.

The benefits of simplified development and integration are passed on to users. Systems that are designed with redundant services will be highly available, satisfying expectations for systems that always work. Systems that are designed with security at all levels, especially at the service level, will meet user's expectations of systems that protect their secure data. Systems that are designed based on well-known interfaces will be true plug-and-play, satisfying user demand for zero-administration systems.

Because service interfaces are clearly separated from user interfaces, the same service can be accessible to a wide variety of users who access the service in a variety of ways. Users with increasing levels of expertise can take advantage of more features of service interfaces. Users around the world can use an interface that works in their language. Users will be able to access services with all kinds of different devices. In the near future, users will be able to combine off-the-shelf components to build their own custom systems.

Java™, Jini™, and Openwings™ are providing the first fully functional framework for SOP. In describing the

patterns of SOP, it should have become clear that some of the patterns can only be supported in a Java™ programming environment at this time, especially code mobility and code security. Until these capabilities are added to other languages and paradigms it will be very difficult to implement Service-Oriented Programming in other languages.

REFERENCES

[1] Microsoft .NET, http://www.microsoft.com/net

[2] The Programmable Web: Web Services Provides Building Blocks for the Microsoft .NET Framework, http://msdn.microsoft.com/msdnmag/issues/0900/WebPlatform/print.asp

[3] HP Cooltown, http://cooltown.hp.com/

[4] Bluetooth, http://www.bluetooth.com

[5] Jini, http://www.jini.org

[6] Openwings, http://www.openwings.org

[7] Architecture Description Language (ADL), http://www.cs.cmu.edu/~acme/acme_documentation.html

[8] ServiceUI project, http://www.jini.org

[9] A Note on Distributed Computing, http://www.sun.com/research/techrep/1994/abstract-29.html

[10] Aspect J, http://aspectj.org

[11] Dynamic Proxy Classes, http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html

[12] iRadio, http://www.motorola.com/ies/telematics/iradio/infrastructure.html

[13] Surfboard SB4100W, http://broadbandstore.motorola.com/