The following paper was originally published in the
Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)
Monterey, California, June 1995.

The Spring Object Model

Sanjay R. Radia, Graham Hamilton, Peter B. Kessler, and Michael L. Powell
SunSoft, Inc.
Mountain View, California

The Spring Object Model

Sanjay R. Radia, Graham Hamilton, Peter B. Kessler, and Michael L. Powell

SunSoft, Inc.
2550 Garcia Avenue, MTV19-216
Mountain View, CA 94043
Sanjay.Radia@Eng.Sun.Com

Abstract

The Spring Object Model provides a basis for building operating systems, applications, and other software components for a modern distributed computing environment. All services and abstractions— whether local or remote; system, extension, or user; library or server—are structured as objects. Objects have strongly typed interfaces specified in an interface definition language. Spring forces a clear separation of interface and implementation; no implementation properties are allowed in the interface. It supports multiple interface inheritance, which is used to structure the system abstractions and provides the basis of extending and evolving them.

Interfaces are used to define boundaries of software components which can be mapped to different address space and machine boundaries. The model provides parameter passing modes that are especially useful for distributed computing but which can be optimized for the local case. This allows one to construct microkernels with the option of configuring components in the same address space for improved performance.

Objects are instances of interfaces, on which clients can perform operations. A client of an object is generally unaware of the location of the object's implementation, and can perform operations on the object and pass the object around freely. The representation of a Spring object is not a fixed piece of information such as a unique identifier; instead, it can be tailored to meet different needs using the subcontract abstraction.

1 Introduction

The Spring distributed operating system is the result of a research project to develop technology that addresses the problems of building, supporting, and using both system and application software in a distributed computing environment. This paper describes the interface and object technology we have developed and refined over the last five years and our experience with it. The design has been shown to work for a prototype distributed and secure operating system that has a flexible and powerful virtual memory system[13], name service[18][19], and file system[14]. Our technology allows the system to be easily extended and evolved, and writing distributed services and applications in Spring is much easier than in earlier systems. Some of this technology have been transferred into products and industry standards, and that process is ongoing.

Spring has been strongly influenced by ideas in programming languages such as data abstraction, strong typing, explicit interface definitions, complete separation of interface from implementation, inheritance, and, of course, the use of objects. Unlike other approaches, however, we do not assume a single programming language, nor do we assume that the type system is enforced by the operating system. Spring also builds on current good ideas in operating systems such as modular structure, multiple address spaces, transparent distribution via remote procedure calls, multiple programming languages, multi-threading, secure micro-kernels, etc. Unlike most systems, however, Spring uses objects and interfaces technology as the only software-visible structuring mechanism, for all kinds and levels of software.

1.1 Motivation

The Spring project was motivated by the problems the industry faces in building, supporting, and using software in a modern computing environment. These same problems arise when building subsystems and large applications, and our solutions are applicable to those areas as well. Some of the critical objectives are:

- security, including the ability to have different security policies from strict to unchecked,
- uniformity, from single machines to heterogeneous, enterprise-wide networks,
- consistent application model, especially for end-users, but also for administrators and developers, and
- extension, configuration, and evolution, allowing incremental change and coexistence of variations.

The object model plays a key role in handling these issues because it sets the global expectations and policies for how software components interact. In fact, simply having a single, consistent object model can help achieve important properties.

Security

Secure systems have typically been more difficult to use, because security constraints can deny service at unexpected or inconvenient times. Making a system that is truly secure requires the security policy to be rigorously implemented by all components, and almost any lapse can be fatal. We desire a solution in which most software works unchanged in environments with different security requirements.

Uniformity

Most systems use different mechanisms and different programming paradigms to connect components at different granularities. For example, a developer writes different code to access a library, an operating system service, or a network service. It is also common for the equivalent service to use different interfaces, depending on the scope or complexity of the implementation. We want a solution that allows system services to be accessed the same way independent of their implementation.

Consistent Application Model

Users can be confused when presented with a variety of inconsistent and incompatible concepts. For example, in some systems, documents, users, printers, machines, and applications are named, created, and manipulated in different ways primarily because of the different underlying implementations. Even when attempts are made to add a consistent layer on top of disparate mechanisms, problems can arise when the limitations of the implementation cause unexpected constraints that are difficult to explain at the user level. We would like a solution in which user-level abstractions correspond directly to a consistent application model.

Extension, configuration, and evolution

The rapid pace of technology advancement means that new functionality is continually being introduced. As the network grows, more powerful mechanisms are needed to provide the security, performance, robustness, etc., of a smaller environment. Some capabilities that are considered optional for some people may be considered essential for others. We need to treat all of this software in a first-class way, so that it is just as easy to use the old as the new, the extension as the built-in, the special as the general.

Even though many operating systems are implemented using high-level languages, they are hard to change, extend, and evolve. Those systems are not designed as a set of replaceable components. Component interfaces and the interactions between components are not well defined. Changes to one part of the system can easily affect another part. The procedural interfaces for a component are often the outcome of a particular implementation rather than an interface designed to define a replaceable component.

1.2 Tactics

To address these issues, the Spring object model makes a set of decisions, and applies them uniformly to all software in the Spring system. Although higher-level software could in principle obscure those choices, it is our intention for the model to "show through" to programmers and users of the system. We also apply the model down to the lowest level of the system, and use it to design the fundamental mechanisms.

Object-orientation

All software is structured as objects. Instances of abstractions such as files, documents, applications, etc., as well as services such as naming, virtual memory, user interface, etc., are modeled as objects. Functionality is accessed by invoking operations on those objects.

Strong interfaces defined in an interface definition language

An object has a type that is defined by its interface, which specifies the operations that can be performed on the object. Multiple inheritance of interfaces is supported, allowing objects to be taxonomized by their interfaces. The interface definition language plays a central role in the organization of the system and the implementation of its fundamental mechanisms.

Common programming model

Programmers write the same code to access objects whether they are local or remote, library or server, system, extension, or application. Objects can be accessed the same way independent of their location and implementation technique, allowing address space and machine boundaries to change without changing the software. New or optional services are accessed the same way as existing or built-in services.

Separation of interface and implementation

Multiple implementations of the same interface can coexist and alternatives can be substituted without requiring changes to the users of those objects or other components. Generally, components depend only on the interface to other components, not on the particular implementations.

Capability-style security

Possession of the object confers the right to use the object according to its interface. Objects cannot be arbitrarily constructed, but are acquired via operations on other objects, allowing appropriate access checking to take place. Authority can be delegated by passing objects to other components.

Abstract objects

Unlikely many object systems, Spring objects are abstractions in that their behavior and properties are only perceived through their interfaces. Notions such as the identity, location, or state of an object are meaningful only to the extent that they are exposed through the object's interface.

Type system by convention

Although the object model and type system provide substantial descriptive and organization power for Spring, and they are used pervasively, type information is always considered in light of trust and location. Thus the system cannot be subverted by breaking the type system, and type information can be extended and federated.

1.3 Other approaches

Microkernels have been used to unify mechanisms and address modularity and security issues in systems. Thoth[6], the V system[7], Amoeba[16] and Mach[1] use the microkernel approach. In these systems, software boundaries are tied to hardware boundaries: the system is broken into components, each of which resides in a separate address space, communicating via message passing or remote procedure calls. This approach has a number of limitations. The interface is defined by the request-response message protocol. Replacing a component by another that obeys the same message protocol is fairly straightforward, but extending the interface is more difficult. A well-defined, strongly-typed interface is preferable. Also, the restriction that the components must reside in separate address spaces is a performance penalty. While the firewalls of separate address spaces are often useful during development, it is often desirable in production use to put some components in the same address space and rely on local procedure calls to obtain better performance. Others have reported on other performance penalties of microkernel that use separate address spaces[5].

Language-based operating systems such as Smalltalk and Cedar reduce the gap between the system and the programming environment, and are more modular and easier to change. However, they have restrictions, such as the use of a single language and a single address space, and their type systems must be enforced for correctness. These limitations are unacceptable for an open, secure, heterogeneous, enterprise distributed computing environment.

Other object based systems have focused on different issues and used different tactics. Choices[4][15], an object oriented operating system, has focused on using the object oriented features of the C++ language for building a highly modular kernel for tightly coupled multiprocessors. All system services run in the kernel address space, but are given well-defined interfaces in C++. Choice has not addressed the problems of distributed computing or security. The Clouds system[2] has focused on building a distributed system with large grained persistent objects to address problem of reliability using transactions. Objects provide a uniform paradigm. Extensibility of software components was not a goal. Clouds is programmed in several languages including C, C++, Modula2 and Eiffel.

2 The Spring Object Model

The Spring Object Model includes three aspects of objects. We first describe the "client" view, that is, how objects appear and behave for those that are using them. Next, we describe the "implementation" view, that is, how an implementation is constructed to abide by the model. Finally, we describe subcontract, a technology for specifying representations and protocols that support the model. Note that it is common (but not required) for a component to be both the implementation of some object(s) as well as a client of some object(s).

2.1 The Client View of the Spring Object Model

By *client*, we mean any component that can possess an object. A client of some object might be the implementation of some other object, but is not required to be. An *object* is an instance of an interface. An *interface* specifies a set of operations that may be performed on the object. An *operation* is a transfer of control that can include the sending or receiving of a set of parameters. *Parameters* may be objects or data values (integers, strings, and simple data structures). An *attribute* is a data value that can be accessed via a pair of implicitly defined operations to store and retrieve the value.

In Spring, interfaces are expressed in an *interface definition language*.[1] The interface definition language is not a programming language, in that it is only declarations. The definitions are mapped into particular programming languages, and the mapping specifies how objects are stored, invoked, and specified as parameters by programs in that language.

The following example shows an IDL definition of an interface,

```
//    IDL
interface NamingContext {
    attribute User owner;
    void lookup(copy Name n,
                produce Object o);
    void bind(copy Name n,
              consume Object o);
};
```

and a C++ mapping of an invocation:

```
// C++ mapping of an invocation
NamingContext context;
File f;
Name filename;

context->bind(filename, f);
```

1. The original name for our interface definition language was *Contract*. The version of this language adopted by the Object Management Group is called *IDL*.

The type system and semantics of invocation are intentionally simple in order to permit a variety of languages to be used to access or implement Spring objects. Basically, the language needs some way to represent an object, and some subroutine mechanism that allows parameters. Languages with the concept of an object can probably map Spring objects to a constrained usage of the language-defined objects. Languages without such a concept can probably mimic Spring objects with a simple convention.

Objects are passive. Threads are the active entities, which can invoke operations on objects. A (thread in a) client of an object can invoke operations specified by the interface the object supports, or can pass the object as a parameter to an operation on some other object, but cannot otherwise manipulate the object. In order to invoke or pass an object, a client must first *possess* the object. A clients may be given objects when it is created, may be passed objects by being invoked, or may be returned objects as a result of invocations it performs. A client cannot declare an object into existence, nor create an object arbitrarily.[2] In general, clients obtain objects by way of operations on objects, and possession of an object gives the client the right and ability to invoke any operation defined by that object.

The Spring object model is strongly typed. The type of an object is defined by the complete set of interfaces it supports (due to inheritance, a object may support more than one interface as explained below). Each operation of an interface specifies the types of its formal parameters. An object passed as an argument to an operation must support the interface of the formal parameter.

The interface definition language supports inheritance,[3] as shown in the following example:

```
interface IOStream {
    void get(produce Buffer b);
    void put(consume Buffer b);
};
interface File: IOStream {
    void seek(copy Position p);
};
```

2. Only the implementation of an object can create an object. See next section.

3. Note that this is inheritance of interfaces, and does not imply whether or not inheritance is used in implementations.

```
interface PropertyList {
    void setProp( copy PropName pn,
                  consume Value v);
    void getProp( copy PropName pn,
                  produce Value v);
};
interface FileWithProperties:
    File, PropertyList { }

interface Printer {
    void print(consume IOStream s);
};
```

Instances of the File interface support the get and put operations defined in the IOStream interface, as well as the seek operation. Multiple inheritance is also possible, as shown by FileWithProperties, which supports all of the operations of File as well as PropertyList. The type of the object is the one interface that includes all the operations that are supported by the object. An object may be passed as a parameter if it supports the interface specified by the formal parameter type. In this example, instances of any of IOStream, File, or FileWithProperties may be passed to the print operation.

Parameter Passing Modes

Figure 1 shows a client that contains several objects. By invoking an operation on object O and passing object p as a parameter, O comes to possess object p. The exact details of how the object is transferred is determined by the parameter *mode*.

In ordinary programming languages, it is common for parameter passing mechanisms to have ambiguous usage. For example, passing a variable by reference is used for purely *in* parameters that are too expensive to copy, for *out* parameters, and also for *in-out* parameters when the caller is expecting to use the argument after the callee has changed it in some way; that is, pass-by-reference does not indicate whether a value will be extracted from or inserted into the variable or both. More complicated issues such as whether the callee can retain a reference to the parameter or whether particular operations are considered to modify the object (for example, it may be permissible to increment the reference count, but not to change the value) are usually not specified by the parameter mode.

In programming languages, the pass-by-reference mode is based on shared memory. How could this mode be interpreted for distributed objects? While one may interpret pass-by-reference as passing references to remote object servers, the approach does not work well when you consider passing references to lightweight data-like objects (unless you have shared memory in the distributed environment). One would prefer to pass such data-like objects by value. Choosing between the pass-by-value and pass-by-reference modes for a formal parameter based on such implementation properties (data-like vs. remote) is not possible because such properties are not specified in the parameter type's interface. Indeed, a particular type may have object instances of both kinds.

In a distributed system, it is important to specify which "direction" a parameter is going. There is no reason to transmit a variable to a remote site in order to have a value placed in it. Similarly, if the target is in the local address space, it would be desirable to avoid making copies and get the same benefit as pass-by-reference.

Because we are dealing with objects, in Spring, the five parameter modes that define what happens between the *invoker* and *invokee* have an almost physical intuition to them.

- *Consume*: The argument object moves from the invoker to the invokee at the start of the operation. The invoker no longer has the object.

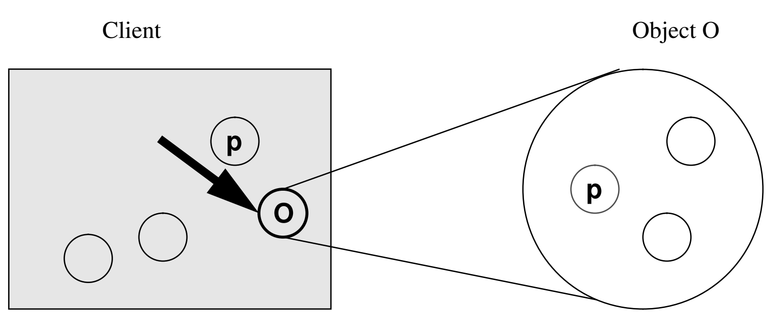

Figure 1. Passing object p as a parameter to object O during invocation

- *Produce*: The argument object moves from the invokee to the invoker at the end of the operation. The invoker has a new object.
- *Borrow*: The argument object moves from the invoker to the invokee at the start of the operation and an object (supporting the same interface, but not necessarily the same as the actual argument object) moves from the invokee to the invoker at the end of the operation. Borrow is equivalent to a consume and a produce of the argument. If the original argument is not returned, the invoker receives a new object in its place.
- *Copy*: A new object is created by calling the standard copy operation on the argument object, and this new object is passed consume. The invoker keeps the original, but no longer has the new object.
- *Share*: A new object is created and passed similar to *copy*, but the assumed semantics of the new object is that it shares behavior with the original.

The consume mode is specified for a parameter for the case when the invoker has generally no interest in using the object after the invocation, and the produce mode is specified when the invokee has generally no interest in using the object after the operation is completed. Consume and produce are two common parameter passing modes and we have found them to be natural and efficient for many interfaces. For example, in the IOStream interface above, the get operation returns the data as a produce parameter and the put operations take the data as a consume parameter.

The consume mode is the most appropriate for the put operation since the client is generally not interested in keeping a copy of the Buffer. In the few occasions when a copy is needed, the client simply makes an explicit copy of the Buffer. Similarly during a get operation, the server (say a file server implementing the IOStream) does not generally need to keep the Buffer. Indeed, we have found that for out parameters, the invokee does not generally want to keep a copy of the parameter and that the produce mode is very appropriate

Borrow is used when the invoker does not need to use the object for the duration of the operation, but wants the object back at the end of the operation. The name object (a data-like object implemented on the client side via a library) provides an equal operation that takes a borrow parameter name to be checked for equality:

```
interface Name {
    boolean equal(
        borrow Name to_be_checked
    );
    ...
};
```

The equal operation checks for equality and returns the parameter unchanged. Consume would not be appropriate since the client would generally want to keep the parameter object and the copy mode would cause an unnecessary copy.

In some uses of the borrow mode, the borrow parameter returned is different from the one passed in, so that the invoker is exchanging one object for another. For example, the interposition service takes an object as a borrow parameter, which is kept by the service, and a interposing object is returned in its place. The interposing object will intercept calls (for tracing, debugging etc.) and then forward the calls to the original object. Although borrow can be emulated using two separate parameters, one consume and the other produce, the explicit declaration of borrow is useful in specifying interfaces.

The copy and share modes are used when the invoker wants to continue possessing the object. In the case of copy, a copy of the object is made on the invoker side and the new object is passed in the same way as consume. The two objects may or may not share state and behavior depending on the semantics of the object. For example, an object of interface File has the meaning "an open file". Passing it copy gives the invokee access to the same file. The objects are different (closing one would not close the other), although their semantics are related (they share the same file data). Although the model does not specify the distinction, for some objects, both copy and share make sense. For example, a buffer might be passed copy to give access to the data and share to allow insertion of data. In case of a file, passing a file copy could mean "shared data but disjoint seek pointers" whereas passing a file share could mean "shared data and shared seek pointer". The additional share mode adds nothing to the model, but supports a common intuition.

There are similar semantics for using the above parameter passing modes for primitive types like integers, reals, strings, etc. and also for constructed types such as structs and arrays.We compare the relative costs of the parameter passing modes in Section 3.4.3.

Object versus the Object Reference

Spring has no separate notion of "object reference". This is important for security and implementation flexibility reasons, as well as to simplify the model. A client either has a particular object (which was given to it by someone else who had it, etc.), in which case it can do whatever that object can do, or it does not have it, in which case it can do nothing to the object. It is not necessary for the object to protect itself from arbitrary clients who might have forged the object. By eliminating the “object reference” vs. “object” level of indirection, implementations of some objects can be directly in the client. Because objects are abstractions, it is always possible to introduce additional levels of indirection when needed.

Many systems, such as OMG’s CORBA[17], make it explicit that the client holds an object reference that points to the object. Nonetheless, because it is designed to support interoperability between object systems and languages, CORBA is general enough about what an object reference is that the Spring object model can be viewed as an extension to it.

To a large degree whether the client is considered to hold the object or a reference to the object is simply a matter of terminology. The Spring object model allows the object state to reside on the server side, the client side, or split between the two. In the case where the entire object state is in a server and the client side simply contains some form of a pointer to the server, the notion of an object reference may make sense. However, in other situations, where the entire or part of the object state resides on the client side, the notion of a reference would have been somewhat misleading and it is more convenient to view the client as possessing the object itself rather than a reference to the object.

We chose our terminology because we wanted the client’s view to be that of having abstract objects that can be manipulated and passed around very much like the objects in many object oriented languages.When one writes Spring applications in a programming language like C++ on top of Spring, Spring objects are represented by local C++ objects which fits naturally with the view that the client holds local objects.

Identity

One feature of some object systems that is explicitly avoided is the notion of “identity” for objects. Given that objects are abstract, it is not possible for the object model to determine when two objects ought to be considered the same. An arbitrary rule could be implemented, but it would produce surprising and inconsistent answers. For example, two file objects might be replicas of the same file data. For users of files, we would like to mask the detail of which replica is being used, but for the component that is responsible for updating the replicas coherently, we clearly must distinguish them. Rather than specifying a notion of identity in the object model, we use operations on the objects themselves to answer the question. For example,

```
interface Document {
    boolean sameas(copy Document d);
    ...
};
interface ReplicatedDocument: Document {
    boolean sameReplica(copy Document d);
};
```

Constructors and Factories

The model does not provide constructors or the ability to declare an object into existence. Although the model does not permit clients to create objects arbitrarily, conventional usage makes it easy to get access to objects when needed. We call an object whose principal function is to create other objects a *factory*. Factories are widely used for commonly created objects, and language mappings can implement constructors that make calls on the default factories. Access to factories can be controlled like access to any other object, so clients cannot get a factory that creates an object they are not authorized to use.

The most common source of objects is the name service. Operations on a naming context object (similar to a directory) check to see if the client is allowed access to the named object before returning it.

2.2 The Implementation View of the Spring Object Model

The client view of the object model is independent of the choices made in the implementation. Nonetheless, there are important implications about the behavior of implementations that support the object model.

Usually, the same implementation code and associated state are used to implement several similar objects. We call such an implementation an *object manager*. Object managers generally fit into one of two categories: co-resident and server. Co-resident object managers exist in the same domain with their clients. A *domain* has an address space and is the unit of protection and resource management by the operating system. Generally, the operating system does not protect one part of a domain from the other. Server object managers can be protected from their clients by the operating system, and can even be on a different machine. Co-resident object managers can be efficient, but are insecure and typically do not outlive their clients; server object managers are more expensive, but are safer and can have an independent lifetime.

Although interfaces and objects define the logical boundaries of the software, the enforcement boundaries may be different. When a client possesses an object, it is really held by its domain. Any thread in the domain can in theory invoke operation on objects held by the domain or pass such objects as parameters. An object can be held by only one domain at a time. Domains are also the unit of distribution across machines, that is, all of a domain is on the same machine.

Although the client of a co-resident implementation could in principle refer to the object's implementation directly, it could not do so for a server implementation. In general, the client of an object has a *representation* of the object in its domain, which provides the necessary information for directing invocations to the object's implementation. The representation is typically transmitted from one client to another when an object is passed as a parameter. Client code is unaware of the details of the representation and the actual location of the state and the implementation.

As shown in Figure 2, there a logical distinction between the representation of the object, which usually is close to the client and invariant, and the state of the object, which usually is hidden from the client and manipulated by operations on the object. It is important to realize that although this distinction is often true in practice, it is not a requirement. A category of objects called "server-less" objects have co-resident object managers and the representation is the state of the object. The simplest form of a server-less object is the data object. A *data object* has an interface that consists only of attributes. Specialized implementations merely store the attribute values, and use a co-resident implementation to set or get them.

Because objects are abstractions, there is no complete specification of the state of a particular object. The state of a file, for example, might include disk space management information that is intermixed with that of other files.

A common reason for using a server object manager and distinguishing a representation is to accomplish sharing. For example, two clients may each have a file object that directs invocations to a shared state and implementation in some file server. Other reasons include the need for geographic separation, security, independent lifetimes, etc.

An important implementation technique is to use an object in the domain of the client to implement the mechanism for communication to the object manager. This scheme of creating a "surrogate object" is not novel, but the object model helps ensure that the same mechanism that is used to access the surrogate object can be used for co-resident objects. The simplest form of a co-resident object is the data object.

Whether co-resident or server, the object manager can assume that incoming requests come from clients that are intended to have the object. Directly or indirectly, the object manager creates all the objects it implements. Many object managers accept requests from the name server to (re)create objects in response to lookup requests. These are objects that correspond to persistent state that has existed as objects earlier, and were bound to names. Other object managers create their own factory objects and publish them when they initialize.

Interface versus Implementation Inheritance

A number of object oriented systems and programming languages provide implementation inheritance. *Implementation inheritance* is the composition of an object implementation using parts of another implementation. In some systems or languages, inheritance of an interface implies the inheritance of its implementation. This is not true in Spring: if several interfaces are related by inheritance then Spring does not require that there be any such relationships between the corresponding implementations. We use interface inheritance to structure the system interfaces and to permit substitutability of a derived interface where a base interface is expected. Implementation inheritance is concerned with sharing of code. A Spring implementation must provide all the operations supported by the object, whether in inherited interfaces or not. It may compose that implementation using shared libraries, using implementation inheritance

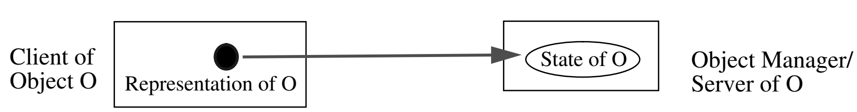

Figure 2. Client and Server of an Object

with languages such as C++, or use a disjoint implementation as it sees fit. Two closely related implementations may actually share code, whereas two radically different implementations of an interface may have disjoint implementation and would have found any forced sharing burdensome. Thus, while we do not disallow implementation inheritance (and use it in parts of the system), we do not tie it to interface inheritance so as to allow implementations of different parts of the system to change and evolve as needed.

2.3 Subcontract: Programmable Representations and Mechanisms

It may be surprising that we have described the model of objects from both the client and implementation perspective without revealing the representation of objects. This is because in Spring we support a variety of representations, and have defined a means for programming additional ones.

The basic representation in Spring is a door. A *door* is a primitive object used for making protected, remote procedure invocations to another domain, possibly on another machine. Doors are capability-like objects that cannot be forged. A door is implemented, protected, and controlled by the Spring microkernel[8]. A door may be passed as a parameter on an invocation of another door. A domain can create a door that refers to an object implementation within the domain itself. However, a domain cannot create a door that refers to an object implementation in some other domain; such doors must be obtained directly or indirectly from the target domain via other doors. Doors are not persistent—a client cannot store a door in persistent memory for later use. When a client dies, all its doors become invalid. When a server dies, all doors targeted to objects implemented by the server also become invalid.

A door is a plausible representation for an object, and is in fact used that way for many objects in Spring. However, for some objects a door may not be a suitable representation. Although invoking a door is quite fast, for some objects, it is not fast enough (it may want to use a different representation to facilitate some client side caching). Although doors are as robust as most operating system mechanisms, for some services, it is not robust enough. Some objects require a more persistent representation. The tension between performance and simplicity versus security and functionality caused us not to choose a single representation for objects, but rather to allow a variety of them to fit within the same model.

A *subcontract* is a definition of a representation and corresponding invocation protocol [9]. An object implementation selects a particular subcontract to use for its representation. Clients are always unaware of the subcontract used for the objects they possess. Object managers may or may not care which subcontract is used.

In addition to defining the representation and invocation protocol for the object, the subcontract also defines how the object is marshalled and unmarshalled as a parameter on a call on another object. Unmarshalling is interesting (although not to the topic at hand) because it must locate the appropriate subcontract which might not be at the destination [9].

A simple, common subcontract is the single door subcontract whose representation is a single door for each object. A table-entry subcontract could have a representation of a door plus an index into the table. Different table entries could all use the same door, reducing the microkernel's overhead of having many doors. Since the client could lie about the table index, this representation is appropriate only when it is not necessary to separately protect different table entries. The invocation protocol would pass the index as an additional parameter, and the object manager would know about the particular subcontract, and use the index to identify the particular object being invoked.

A persistent-door subcontract would have a representation of a door plus a name that could be used to reacquire the door. If the server crashes, or the door is revoked, the client would go to the name service to get an equivalent door. It is likely that some authentication would be necessary if the object needed to be protected. This procedure could be completely hidden in the subcontract, making it so that neither the client nor the implementation was aware that this particular representation had been chosen.

In many ways, subcontract challenges the Spring object model as much as it helps support it. Exotic subcontracts that support replication, more powerful security, caching, and other properties help to demonstrate we have indeed separated the interfaces from the implementations, at the same time that they help make it practical to use a single model for all of the software. Without subcontract, we would be mired in the traditional conflict between the desire to have pervasive functionality, which argues for putting it in at a low level, and the desire to have a simple efficient system, which argues for leaving it out. With subcontract, we can provide such properties—often transparently—for those objects or in those environments that warrant them, and omit them where they are not needed.

3 Using the Spring Object Model to build the Spring System

The Spring object model has been used to build the Spring operating system, and has been the basis for our work in Object Products at SunSoft. The system continues to evolve as we reconcile it with emerging standards and transfer the ideas and technology into products.

3.1 The Interface Definition Language, IDL

In the early part of the Spring project, interfaces were specified in Contract. In submitting interface definition technology to OMG[4], we altered the syntax to be closer to C++ and omitted features such as the parameter passing modes[5] that we could not convince OMG to accept. We have also extended IDL to include formal specifications in interfaces that are used for testing (see [20] for more details).

3.2 Using Spring Interfaces

After an interface is specified in IDL, it is translated to mappings for different programming languages: client-side stubs for applications that will use objects of the interface type, and server-side skeletons for implementations of the interface. The actual mapping depends on the programming language. For C++, each IDL interface and its operations are mapped to a C++ class; the methods are client-side stubs, and the C++ object contains the representation of the Spring object which is used to direct calls to the implementation. Primitive and constructed types are also mapped to suitable C++ values.

When a domain starts, it is given some standard objects along with the program argument objects passed by its parent. From these objects it can obtain other objects. For example, one of the objects is a name context through which other objects can be looked up. Some of the objects are factories, for example, a domain uses its domain object to create threads.

4. OMG IDL was also influenced by DCE IDL.

5. OMG IDL uses “in”, which is copy for data values and share for objects. It calls borrow “inout” and produce “out”, and does not yet provide the equivalent of copy for objects or share for data values.

A domain can also implement and become a server for an object. Servers make their objects available through the name service or factory objects. For example, a printer server will bind its printer object in a well-known part of the name space for its clients. The kernel provides a factory object called a domain manager that used for creating domains.

Spring also allows lightweight data-like objects where the state and implementation of the object are local so that local procedure calls are used when operations are invoked. A domain can create such an object locally (its implementation is usually available in libraries). When such an object is passed across address space boundaries, the state of the object is passed and the receiving domain must have access to the implementation. For example, names used by the name service are implemented as data-like objects. Another example is the data parameter, called raw_data, of the read and write operations of the IO interface.

3.3 Strong Typing

The Spring object model and the system we have built with it is strongly typed. All arguments to interfaces are statically typed checked. We rely on the static type checking of programming languages: an interface is mapped to a procedure in a high level language such as C++, which performs compile type checking of arguments. Dynamic checking at runtime is performed occasionally. A client always has some perception of the type of every object it holds. The object may of course be some subtype of the perceived type, in which case the client can narrow to this subtype. At this point, the type is checked and the narrow may fail or succeed. For example when a client resolves a name, it expects the object returned by the name service to be of a particular type; the object is returned in some generic form and runtime type checking is performed to convert it to the desired type.

3.4 Experience

Interfaces, inheritance, and the separation of interfaces and implementation have proved very useful in building and extending the Spring system over the last four years. Over that time, we have refined and evolved a style and conventions for using inheritance to make the system open and extensible. A microkernel-based distributed operating system has evolved naturally out of using interfaces to define software boundaries. Unlike many other systems, we take advantage of optimizations that are possible when components are put in the same address space. Building distributed system software has proved to be easier than in conventional systems; however we need further experience with distributed applications. Our experiences with our parameter passing modes have been positive. This section briefly describes our experience in these areas.

3.4.1 Interfaces and Extending the System

The Spring system is made up of a number of components, each with a well-defined interface. Spring is focused on providing good interfaces rather than simply on providing implementations. The interfaces define and capture the various system abstractions. We have spent a fair amount of effort in defining good interfaces to the virtual memory system, the name service, the file system, etc. We allow the coexistence of radically different implementations of a given interface within a single system. For, example, all the following implement the naming context interface: persistent name server, transient name server, the file system, gateways to foreign name servers such as NIS, and the system configuration data base.

Many interfaces are related by inheritance. This has helped us structure the system abstractions so that common abstractions are captured and reused. The classic example is the IOStream interface, which is used by various objects that perform IO (The UNIX operating system exploits a similar abstraction to provide uniformity[21]). Another example is authentication; many of our objects are authenticated, and what it means to be authenticated is captured by the authenticated interface which is inherited by many interfaces. We call such an interface a mixin interface. Other examples of such mixin interfaces are caching and ACLs.

Interfaces have played a key role in extending and evolving the system. Over the last four years several implementations have changed and many interfaces have been extended. The fact that components have well-defined interfaces and are separate from implementations has made it easier to change implementations while insulating clients and localizing the impact of such changes.

Inheritance has proved useful in extending interfaces. For example, we initially defined the file interface and implemented a file system. Later, we decided to add caching. We defined a new interface called cacheable file, which inherited from file, and extended our implementations to support caching. The client's view of file did not change. This is a natural consequence of using inheritance and applications using object oriented programming languages have taken advantage of this in the past; operating system interfaces can also benefit from this and we take advantage of it widely in our system.

We have also developed a scheme for versioning interfaces: different versions of an interface are represented as different types with an inheritance relationship that minimizes the impact on existing clients and allows easy management of versions[10].

Our style of inheritance has evolved to make the system open. Rather than use the classical root inheritance where the system ends up with a few key interfaces inherited by all interfaces, we use a combination of root and leaf inheritance: fundamental properties are inherited early, near the root of the type lattice, whereas auxiliary properties are inherited late, near the leaves of the type lattice, by only those implementations that support the auxiliary properties. This provides flexibility in supporting auxiliary properties, and allows us to add new auxiliary properties as the system evolves without forcing the system to be recompiled [10]. Recompiling the entire system is a real problem in commercial systems where different components are provided by different vendors. Furthermore, shutting down and bringing up a new version of a system is nearly impossible in a distributed environment.

3.4.2 Interfaces versus Address Space and Machine Boundaries

Interfaces are used to define boundaries of software components; these can be mapped to different address space and machine boundaries. The interface definition language has carefully avoided features such as pointers which rely on shared memory which would have prevented components from being distributed across address and machine boundaries. A client of an object is generally unaware of the location of the object's implementation and can perform operations on the object and pass the object around freely.

Writing distributed applications in Spring is easier than in conventional RPC based systems. While the remote procedure call in conventional systems provides transparency in invoking a procedure, distributed programming in such systems is fairly difficult. One of the problems is that the target of a conventional RPC is not a first class entity: a server that is an RPC target has to register itself in some directory service, and clients have to select a target as part of a binding step before performing RCP. In particular, conventional RPC targets cannot generally be passed around as arguments.

In Spring we program using objects which can be passed around as first class entities. The RPC binding step is not necessary[6]. As a result, ordinary client-server interaction is simplified, and more sophisticated distributed computing is significantly easier than in conventional RPC systems. For example, in traditional RPC systems, passing a pointer to a piece of shared data is difficult. In Spring the client can simply implement and create an object that encapsulates the data and pass it as an argument. Similarly in traditional RPC systems, pointers to procedures cannot be passed, unless they made to be full fledged RPC targets that are registered in some directory service. In Spring objects can be passed instead.

Consider the implementation of caching for files in the Spring system. The cacher creates call-back objects on the fly and passes them to the file server for call-backs. Such objects do not have to be registered in any directory service and are simply discarded when no longer needed[7]. Another example is a service that returns a list of entities on some operation. If the list might be arbitrarily large, the service usually returns an iterator object, which can then be used to obtain the list in smaller pieces. Unfortunately, this is inefficient for the common case of small lists. In Spring one can simply return a data-like object that has a small number of entries and, if there are any more entries, the data-like object can contain within itself an iterator that can get additional entries from the server when requested. Such programs are fairly straightforward to write in Spring. The work on network objects at DEC SRC has also noted that objects simplify distributed applications when compared to traditional RPC systems[3].

Our microkernel has been a natural outcome of defining interfaces for the core operating system abstractions, such as address space, processes, virtual memory, file systems, naming, etc. In Spring optimizations are possible when components are placed together in the same address spaces. Components that are configured to reside in the same address space can use local procedure calls and pass arguments directly, avoiding the RPC and marshaling. Furthermore, as explained in more detail below, our parameter passing modes avoid further copying of arguments that would have been necessary for more traditional parameter passing modes. Thus, we can configure our microkernel operating system to reside in a few address spaces and have the same advantage of local procedure calls in monolithic systems.

Subcontracts have played an important role in giving control over the basic object mechanisms and have provided the flexibility needed in supporting features such as caching and reconnection, which are useful in a distributed systems. More detail of subcontract are given in [9].

3.4.3 Parameter Passing Modes

Consume, produce, and borrow modes are efficient since they allow the object to be passed directly without creating new ones; this is especially important when passing object in the same address space where a pointer is passed with no marshalling. The cost of copy depends on the amount of state to be copied; typically for an object with a remote server it means duplicating a door.

In examining the relative costs of different modes we need to consider two things. First, we consider the target object on which the operation is performed; parameters are passed to and from this object. Whether the target is in the local address space or remote matters because in the remote case the parameters needs to be copied to a marshall buffer anyway for all modes. Second, we consider the actual parameter objects. For example a data-like object with a large state has a large representation that needs to be passed—it can be copied or simply passed "directly".

Let us compare the consume and copy modes (the cost of the borrow and produce modes are similar to that of the consume mode). We will see that the consume mode is more efficient than the copy mode, especially when the target object is local.

For a remote target, the parameters are marshalled into a marshal buffer, and the two parameter passing modes are comparable in performance for data-like objects and for primitive data types since in both cases the data has to be copied. If the parameter is a remotely implemented object, the object representation on the client side contains one or more doors (network handles) pointing to the remote site. Thus for the copy mode, doors in the client side object representation need to be copied. This overhead is small since marshalling a door copy can be made to be almost as efficient as marshaling a door consume. Spring also, allows a flexible copy semantics for objects whereby an object's server is notified every time a copy is made. Thus, if a parameter object requires such copy semantics, then the copy mode requires an

6. Although many objects are made available through the name service, others are accessed directly. Indeed, objects are passed to and from the name service for binding and resolving names using the standard parameter passing modes.

7. M. Nelson who implemented the caching file system in both Sprite and Spring found implementing the Spring version significantly easier than the Sprite version which used traditional RPC for the reasons given above.

additional RPC; generally, most objects do not require this

However, when the target object is in the local address space (a data-like object or an object whose server is in the same address space), the consume mode is more efficient than the copy mode, since the parameter can be passed directly and does not need to be copied. The copy cost can be substantial if the parameter object is a data-like object with a large state. For parameter objects that are remote, the copy mode has the overhead of a kernel call to copy the door. Produce and borrow also avoid any copying for operations on locally implemented objects. For example, names are data-like local objects, and the equal operation on names does not have to copy the borrow parameter. Thus, although we can not assume shared memory in the general case, our parameter passing modes give us the same benefits in the local case.

When the target is remote, the copy mode is more efficient than performing a copy operation followed by passing the copied object consume because copying of data takes place anyway during marshalling. We therefore provide two subcontract methods for marshalling: marshal_consume and marshal_copy.

The parameter passing modes have turned out to be natural and easy to use. They were motivated by our need to support distributed programming without relying on shared memory. They give the same advantage as pass-by-reference in the local case.

The basic semantics of parameter passing modes in Emerald[12] is pass-by-reference. An Emerald programmer can specify special modes called *call-by-move* and *call-by-visit* based on his knowledge of the application to optimize invocation; *the semantics remains pass-by-reference* as these modes are simply hints to the system so that it can move the parameter objects if possible.

This approach is different from Spring's where each mode has its own semantics. In Emerald's call-by-move, the caller keeps a reference to the object, where as in Spring the caller does not have the object after it is passed consume[8]. Since for some objects, such as serverless objects, copying the object may have a nontrivial cost, Spring's consume and produce modes offers performance advantages over Emerald's modes.

8. Spring clients possess objects which can be passed around. There is not a separate notion of a reference to an object.

In Emerald, an object implementor typically specifies call-by-visit if the implementation will be making frequent calls to the arguments during the call. An implementor rarely specifies call-by-move. The caller can also specify the modes based on its knowledge of its own and the server's usage; this is fairly common[11].

In contrast, in Spring, an interface writer specifies the mode based on the direction of the parameters and also on whether the callee of the interface would want to keep a copy of the object.

Emerald's approach was motivated by object mobility. In Spring, mobility is orthogonal to the parameter passing modes. However, mobility is provided automatically for data-like objects that are passed consume, produce or borrow since such an object's implementation decides what is actually transmitted when the object is marshalled.

3.5 Status of the Spring System

Spring currently exists as a fairly complete prototype that is in daily use as a desktop operating system for its own development. The operating system is based around a minimal kernel, which provides basic object-oriented inter-process communication[8] and memory management [13]. Functionality such as naming, paging, file systems, etc. are all provided as user-mode services outside of the basic kernel. The system also provides enough UNIX operating system emulation to support standard utilities such as make, vi, csh, the X window system, etc. All system interfaces are defined in IDL.

The Spring system is distributed to the research community (for details see http://www.sun.com:80/technology-research/spring)

4 Concluding Remarks

The Spring project started with the goal of supporting secure, scalable, easy-to-use, extensible software. Over the last five years we have evolved and used the Spring object model to build a fairly complete prototype operating system. We believe the Spring object model has defined a simple model of security for clients and implementations, a framework for services that span a wide range of performance and capability, a single programming paradigm for data, libraries, system services, and network services, and the means to add, replace, and extend functionality.

Interfaces, inheritance, and the separation of interfaces and implementation have proved very useful in building an extensible operating system. The location transparency provided by objects has made writing distributed software easier. Our choice to keep objects abstract with an open representation which can be controlled using the subcontract abstraction has allowed us to use objects at all levels in the system and has provided the flexibility of easily incorporating features such as caching and reconnection that are crucial in distributed environments.

5 References

[1] M. J. Accetta, R.V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian Jr., M. W. Young, "Mach: A new Kernel Foundation for UNIX Development", *Proceedings of the Summer 1986 USENIX Conference*, pp. 93-113, July, 1986.

[2] J. M. Berabeu–Auban, et al, "The Architecture of Ra: A Kernel for Clouds", *22th Hawaii International Conference on System Sciences*, January 1989, pp. 936–945.

[3] A. Birrell, G. Nelson, S. Owicki, W. Wobber," Network Objects", *Symposium on Principals of Operating System*, December, 1993.

[4] R. H. Campbell, N. Islam, P. Madany, "Choices, Frameworks, and Refinement", *USENIX Computing Systems,* 5, 3 (Summer 1992), pp. 217–257.

[5] J. B. Chen, B. N. Bershad, "The Impact of Operating System Structure on Memory System Performance", *Symposium on Principals of Operating System*, December, 1993.

[6] D. R. Cheriton, M. Malcolm, L. Melen, G. Sager, "Thoth, A Portable Real-time Operating System", *Communications of ACM*, 22(2), pp. 105-115, February 1979.

[7] D. R. Cheriton. "The V Kernel: A Software Base for Distributed Systems", *IEEE Software*, pp. 19-42, April 1984.

[8] G. Hamilton, P. Kougiouris, "The Spring Nucleus: A Microkernel for Objects", *USENIX Summer Conference*, pp. 147-160, July 1993.

[9] G. Hamilton, M. P. Powell, J. G. Mitchell, "Subcontract: A Flexible Base for Distributed Programming", *Proc. 14th Symposium on Principals of Operating System*, pp. 69-79, December, 1993. Also available SMLI technical report 93-13.

[10] G. Hamilton, S. R. Radia, "Using Interface Inheritance to Address Problems in System Software Evolution", *Workshop on Interface Definition Languages*, January 1994.

[11] N. Hutchinson, Personal Communication, 1995.

[12] E. Jul, H. Levy, N. Hutchinson, A. Black, "Fine Grained Mobility in the Emerald System", *ACM Transactions on Computer Systems*, 5(1), pp. 109-133 (February 1988).

[13] Y. A. Khalidi, M. N. Nelson, "The Spring Virtual Memory System", Sun Microsystems Laboratories Technical Report SMLI-93-9, March 1993.

[14] Y. A. Khalidi, M. N. Nelson, "Extensible File Systems in Spring", *Proc. 14th Symposium on Principals of Operating System*, pp. 1-14, December, 1993. Also available as SMLI technical report 93-18.

[15] P. Madany, Personal Communication, 1992.

[16] S. J. Mullender, A. S. Tanenbaum. "The Design of a Capability-Based Distributed Operating System", *The Computer Journal*, 29(4), pp. 289-299, 1986.

[17] Object Management Group, "Common Object Request Broker Architecture and Specification", OMG Document 91.12.1, December 1991.

[18] S. R. Radia, M. N. Nelson, M. L. Powell, "The Spring Name Service", Sun Microsystems Laboratories Technical Report SMLI-93-16, October 1993.

[19] S. R. Radia, P. Madany, M. L. Powell, "Persistence in the Spring System," *Proc. 3rd International Workshop on Object Orientation in Operating Systems (I-WOOOS III)*, pp. 12-23, December 1993.

[20] S. Sankar, R. Hayes, "ADL: An Interface Definition Language for Specifying and Testing Software", *Workshop on Interface Definition Languages*, pp. 13-21, January 1994.

[21] K. Thompson, D. M. Ritchie, "The UNIX Timesharing System", *Communications of ACM*, 17(7), pp. 365-375, July 1974.