

Article Archive

<http://www.java.net/articles>

Rapid Web Services Development with Moose XML

Learn how Moose XML can simplify the task of prototyping and rapidly developing XML web services Rapid Web Services Development with Moose XML Wed, 2010-04-28 [Michael Quigley](#) Learn how Moose XML can simplify the task of prototyping and rapidly developing XML web services.

[Moose XML](#) is a framework that provides a set of components forgetting XML data into and out of Java applications. Moose XML was created after I had experienced some of the rough edges in existing XML marshalling solutions when used in web services environments. Wherever possible, design choices have been made to simplify Moose XML's support for "contract last" web services development.

Moose XML's schema generator is what distinguishes it from the other frameworks. The schema generator can generate an XML schema directly from your annotated Java beans. These generated schemas are useful for driving "contract last" web services development approaches. When combined with the WSDL plumbing in your favorite web services toolkit, the schema generator can make the rapid development of web services applications a reality. Because the stack is automatically managing your schemas, your mapping code is easily refactored, expanded, and maintained alongside your other Java code—changes flow directly from your annotated Java beans into your generated WSDL.

This article will give you a quick tour of Moose XML. I'll start by showing you the basics of creating a mapping. I'll finish by showing you how this can be plugged into Spring Web Services, creating a rapid web services development stack. We'll create a simple "purchase order" web service, allowing for storage and retrieval of simplified purchase order data.

More information is available at the [Moose XML website](#). The full source code of this example is available in the Moose XML source distribution. Links are provided in the [Resources](#) section at the bottom of the article.

Creating the Mapping

Our service stores and retrieves "purchase orders". Imagine that these purchase orders are for software licenses. As such, each purchase order contains a "licensee" and "license type" and a currency "amount". Purchase orders are identified by a numeric "identifier". The license type is represented using a Java 5 enum named `LicenseType`.

Each operation needs to include a "request" and "response" message. We'll start with the `CreatePurchaseOrderRequest` and `CreatePurchaseOrderResponse` types:

```
package com.quigley.moose.example.service; public class CreatePurchaseOrderRequest { // getters and
setters removed for brevity private String licensee; private LicenseType licenseType; private Float amount;}
package com.quigley.moose.example.service; public class CreatePurchaseOrderResponse { // getters and
setters removed for brevity private Long identifier;}
```

Simple POJO classes like these make up our mapping layer. We'll need to add a few annotations so that Moose XML understands how to marshal and unmarshal instances of these classes:

```
package com.quigley.moose.example.service; import com.quigley.moose.mapping.provider.annotation.*;
@Xml(name="createPurchaseOrderRequest") public class CreatePurchaseOrderRequest { // getters and
setters removed for brevity @XMLField(name="licensee") private String licensee;
@XmlField(name="licenseType") private LicenseType licenseType; @XMLField(name="amount") private
Float amount;} package com.quigley.moose.example.service; import
com.quigley.moose.mapping.provider.annotation.*; @Xml(name="createPurchaseOrderResponse") public
class CreatePurchaseOrderResponse { // getters and setters removed for brevity
@XmlField(name="identifier") private Long identifier;}
```

The `@XML` annotation defines the class-level mapping details. Each of the `@XMLField` annotations define the field-level mapping details. The Moose XML developer guide contains information on many other annotations, which are useful for more complicated mapping scenarios.

Marshalling and Unmarshalling

In order to initialize Moose XML, the application will need to acquire a Mapping instance. A Mapping instance is typically obtained by instantiating and configuring a `MappingProvider` instance. Here's an example:

```
StaticClassesProvider classesProvider = new
StaticClassesProvider();classesProvider.addClass(CreatePurchaseOrderRequest.class);
AnnotationMappingProvider mappingProvider = new AnnotationMappingProvider(classesProvider); Mapping
mapping = mappingProvider.getMapping();
```

The Mapping class is central to Moose XML. Once your application has access to a Mapping, performing operations with the framework is straightforward.

The following code snippet illustrates how to use a Mapping to marshall a CreatePurchaseOrderRequest object out to XML:

```
CreatePurchaseOrderRequest request = new
CreatePurchaseOrderRequest();request.setAmount(64.95F);request.setLicensee("Benjamin
Franklin");request.setLicenseType(LicenseType.Renewal); MarshallingContext ctx = new
MarshallingContext();mapping.marshall(request, ctx); String xml = ctx.getOutput().toString();
```

Here's how to unmarshall a CreatePurchaseOrderRequest from XML:

```
CreatePurchaseOrderRequest request = (CreatePurchaseOrderRequest) mapping.unmarshall(xml);
```

Generating a XML schema from your Mapping instance is no more difficult:

```
String xsd = SchemaGenerator.generate(mapping);
```

Spring Web Services Integration
For the remainder of this example, I'm going to assume that you're familiar with the Spring Web Services framework and SOAP webservices. Instead of describing every part of the configuration, we'll just look at the parts where Moose XML facilitates rapid webservices development.

Our example contains a single "endpoint" class, which is used to expose the web service operations. This example uses the Spring @Endpoint and @PayloadRoot annotations to establish a service endpoint. Here is a snippet from the example service endpoint to illustrate the pattern:

```
@Endpoint public class ServiceEndpoint { @PayloadRoot(localPart="createPurchaseOrderRequest",
namespace="http://quigley.com/moose/example/service/") public CreatePurchaseOrderResponse
createPurchaseOrder(CreatePurchaseOrderRequest req) { // }}
```

When configured, Spring will route incoming messages to the method which matches the @PayloadRoot annotations defined in your @Endpoint classes. Spring Web Services also needs to know how to convert the incoming and outgoing XML to and from the parameter types and return values of your endpoint methods. Spring Web Services provides a nice AbstractMarshaller class, which facilitates the integration of various marshalling frameworks into Spring. MooseXML extends AbstractMarshaller to provide a MooseMarshaller type.

The bulk of the magic happens in our Spring context configuration:

```
ExampleService http://localhost:8080/example/ http://quigley.com/moose/example/service/ es
```

```
com.quigley.moose.example.service.CreatePurchaseOrderRequest
com.quigley.moose.example.service.CreatePurchaseOrderResponse
com.quigley.moose.example.service.RetrievePurchaseOrderRequest
com.quigley.moose.example.service.RetrievePurchaseOrderResponse
```

Notice the GenericMarshallingMethodEndpointAdapter. It's intended to take an instance of AbstractMarshaller as a constructor argument. We're configuring it to use our MooseMarshaller. This bean configuration is what allows our service endpoint to magically marshall and unmarshall XML to and from the methods of our ServiceEndpoint class.

The most interesting bits are the mooseSchema bean and the service bean. This combination of beans connects Moose XML's schema generator to the WSDL generation capabilities in Spring Web Services. With that linkage in place, any changes you make to your mapping beans will be automatically reflected in your WSDL.

Building the Example from the Source Distribution

Download the Moose XML source distribution from the link below. Extract the distribution into a directory.

From the directory where you extracted the source distribution, execute:

```
$ ant -f build-example.xml example-service
```

When that completes successfully, you can start up a fully-configured container, pre-loaded with this example, by executing:

```
$ build/example-service/deploy/server/bin/run.sh
```

Conclusion
There are many tools and frameworks for getting XML data into and out of Java applications. Moose XML tries to smooth some of the rough edges that are typically encountered when prototyping and rapidly developing XML web services. In these scenarios, generating the schema and contract information from code ("contract last") seems to be the least cumbersome methodology.

Development is currently underway extending Moose XML to support "contract first" development approaches. The centerpiece is a new tool which can ingest an existing XML schema, and generate an annotated mapping layer. This functionality will be available before Moose XML hits version 1.0.

Resources [Moose XML Framework Home](#) [MooseXML Developer Guide](#) [Moose XML Source Distribution \(Version 0.4.0 #1128\)](#) [Spring Web Services Reference Documentation](#) [Michael Quigley](#) is a software architect, a musician, a sailor, and a lover of animals. He enjoys spending his time somewhere in the confluence of technology and the arts and wants your development projects to be awesome.

<http://www.java.net/article/2010/04/28/rapid-web-services-development-moose-xml>

The Match Maker Design Pattern - a New Place for the Actions

How to add actions to a system without modifying business objects, add objects without changing actions, and still keep things reusable. The Match Maker Design Pattern Tue, 2010-04-27 [Michael Bar-Sinai](#) How to add actions to a system without modifying business objects, add objects without changing actions, and still keep things reusable.

Software systems often deal with similar concepts, whose behavior differs only slightly. Classic Object-Oriented design deals with such cases using inheritance; overriding the calculateSalary() method in different Employee subclasses allows the rest of the application to remain oblivious to the subtle differences between the salary algorithms of Manager, Engineer, and AnnoyingCeoNephew.

Sadly, inheritance doesn't scale very well. You might get away with adding a toXml() method to the business model objects, but when the customer requires data export support for Excel, OpenOffice, CSV, JSON and Lotus 1-2-3 - you know inheritance just ain't gonna cut it. Another issue with inheritance is that when one holds an Employee reference to an object and needs to know the its actual type, one has to downcast, which probably means one is going to get some runtime exceptions.

Modern Object-Oriented design sometimes separates the actions from the objects, using mainly the [Visitor](#) pattern. Sadly, this solution is not applicable in many common situations - the classes must implement the accept() method, so if you're using a third party API - you're out of luck. If you wrote that API, you would be reluctant to use the visitor pattern, as adding new visitable objects in later releases would require adding methods to the visitor interface, thus breaking existing client code that implements it.

Another approach is using the instanceof operator all over the code, which normally creates huge if-else chains, is hard to test and debug, and is prone to logic errors as the order of the if checks has to be done up the type hierarchy - if you check for Employee at line 100, the check for Manager at line 300 might never be executed.

The "where does the action go" dilemma is actually an age-old problem, normally referred to as "The Expression Problem" (See the [Resources](#) section for some links). It boils down to "it is either easy to add types, or it is easy to add operations". This article is (yet another) poke on the problem - we will see how one can add actions to the system without modifying the business objects, add objects without changing the actions, and still keep things reusable, testable, and clear. As an aside, this pattern allows us to change the application behavior on the fly, lends itself to using closures, and allows Java to implement Scala-style case class pattern matching.

Unfortunately, this is not a silver bullet - static type safety is somewhat compromised. Later in the article we will offer an effectively type-safe solution.

Making Matches

As the astute reader might have expected, the match maker pattern involves an object that makes matches between business objects to objects that act on them. The usage, shown below, is simple. At the setup, the client code registers class-handler pairs. In our example, the handler classes implement EmployeeHandler, an interface with a method handle(Employee e).

```
// SetupMatchMaker
```

<http://www.java.net/article/2010/04/26/match-maker-design-pattern-new-place-actions>

HTML5 Server-Push Technologies, Part 2

In the colclusion to his two-part series, Gregor Roth covers the new HTML5 WebSockets protocol. HTML5 Server-Push Technologies, Part 2 Mon, 2010-04-26 [Gregor Roth](#) In the colclusion to his two-part series, Gregor Roth covers the new HTML5 WebSockets protocol.

<http://www.java.net/article/2010/04/26/html5-server-push-technologies-part-2>

Using Styles, Themes, and Painters with LWUIT

Discover how to use some of the new and enhanced features in LWUIT version 1.3 Using Styles, Themes, and Painters with LWUIT Mon, 2010-04-19 [Biswajit Sarkar](#) Discover how to use some of the new and enhanced features in LWUIT version 1.3

<http://www.java.net/article/2010/04/19/using-styles-themes-and-painters-lwuit>

Rethinking Multi-Threaded Design Principles, Part 2

Introduction

In [Part 1 of this series](#), I introduced some basic principles for working in a multi-threaded application. Kevin Farnham, in his [blog](#), pointed out that our next challenge would be to design an application that fully harnesses the processing speed of next generation multi-core processors. That challenge has made me rethink some of the various means by which that goal can be achieved.

Laws governing a processor's speed and performance:

In terms of concurrency, when multiple threads run simultaneously, only one thread gets executed at a time. But to achieve parallelism, the processor should support multiple threads that can be executed at any given point in time. [Flynn's taxonomy](#) classifies processing platforms based on Instruction Set and Data stream, as follows: Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD), and Multiple Instruction Multiple Data (MIMD). As modern processors fall under either SIMD or MIMD, it makes it possible for us to write a data or task centric application that could run in parallel.

Since historically processor clock speed was following [Moore's law](#), there was no real need for most programs to execute multiple instructions at a time. Applications simply ran faster because new processors were faster. However, today processor speeds are no longer increasing rapidly.

To increase overall processing power, chip manufacturers began developing processors that provide instruction level parallelism: the processors evolved to have multiple processing units in them. While some processor architectures include a logical processing unit within the same space to allow Simultaneous Multi-Threading, others go a step further by implementing two or more execution units (Cores) in a processor.

To see how much performance improvement the parallelism contributes, we can easily derive an equation:

Speed Enhancement = (Time needed for Sequential execution) / (Time needed for Parallel execution)

Considering number of processor cores, the equation has been further simplified by [Amdahl's Law](#) as:

Speed Enhancement = $1 / (F_s + ((1 - F_s) / N))$,

where F_s is the fraction of entire program spent running sequentially and N being number of cores in the processors. Here, if N is 1, we get no speedup, but as N increases (tends to infinity, $(1 - F_s)/N$ becomes 0), the speed is totally dependent on the $1/F_s$ part - which leads to the conclusion that, to improve execution speed of an application, we need to figure out ways or patterns to make more code execute in parallel.

Finding different ways to lock

As we go on finding ways to refactor the code into parallelized parts, it becomes quite clear that some way or other, different parts of the code must contend for some resource that requires a lock. Whenever any thread tries to access a lock that's already being held by someone, it gets suspended first and awakened later when lock is available, thus spending some time doing nothing. In this case the JVM manages everything for you, but after holding the lock, if the thread is blocked forever (Database connection down or some I/O operation), it would suspend all waiting threads, causing resource starvation. To ameliorate the situation, we can implement our own lock and let client code handle acquiring or releasing the lock by itself. Though this sounds appealing, the task to implement that would undoubtedly be daunting, as you have to manage the lock by yourself. A simple mistake can make you fail miserably.

Lets revisit our previous CarBookings example:

```
class CarBookings{ final ReentrantLock lock = new ReentrantLock(); private final Set bookings = new HashSet(); public boolean putCar(CarBooking booking){ boolean status = false; if(!lock.isLocked() ){ lock.lock(); try { bookings.add(booking); } finally { lock.unlock(); } status = true; } return status; }}
```

So, when BookingRequester calls the putCar method, if it's already being used (locked) by someone else it returns false, providing better control to the requester on whether to retry or abort, without going through a Run - Suspend - Run phase. Though it looks simple, it gets more complicated with multiple locks; it becomes very easy for someone to forget to put in the unlock code, resulting in all threads waiting forever. So, make

sure you keep track of all locks, and make sure they will be unlocked properly.

Non-Blocking operation

In addition to strictly following the rules of using explicit locks, another way to achieve concurrency is to perform non-blocking operation on the contention points. The [Compare-And-Swap \(CAS\)](#) is one of such operation, where a field can be updated with new value only when certain precondition is met. So, when multiple threads try to make a non-blocking call, only one succeeds at a time, but others are given a choice either to continue re-trying or abort the operation without falling prey to being suspended and awakened later on. This allows them to be in a Running state even if the lock is not acquired, unlike the case of a synchronized lock where they have to be in a Suspended state for some time and come back to Running state when the lock is available. Clearly, as the Non-blocking-Lock-acquiring process reduces the extra transition (Run-Suspend-Run) time, it becomes the better choice for developing highly scalable application.

Let's implement the queue (BookingQ or AdviceQ) from the previous article.

```
public class BookingQ{ private static class Node { CarBooking booking; Node nextBooking;
Node(CarBooking cb, Node next) { this.nextBooking = next; booking = cb; } boolean
compareAndSetNext(Node en, Node sn){ if(this.nextBooking==en){ this.nextBooking = sn; return true; } return
false; } } boolean compareAndSetNode(Node node, Node en, Node sn){ if(node==en){ node = sn; return true;
} return false; } private volatile Node head; private volatile Node tail; public BookingQ() { tail = head = new
Node(null,null); } public void putBooking(CarBooking booking){ Node newBooking = new Node(booking,null);
while(true){ Node currBooking = tail; Node nextTail = tail.nextBooking; if(currBooking == tail){ if(nextTail !=
null){ //Line 1 compareAndSetNode(this.tail, currBooking, nextTail); }else{ //Line 2
if(currBooking.compareAndSetNext(null, newBooking)){ // Line3 compareAndSetNode(this.tail, currBooking,
newBooking); // Line4 break; } } } } public CarBooking getBooking(){ while(true){ Node currHead = head;
Node currTail = tail; //Line 5a Node headNext = currHead.nextBooking; //Line 5b if(currHead == head){
if(currHead!=currTail){ //Line6, Some bookings have been made if(compareAndSetNode(this.head, currHead,
headNext)){ //Line7, Advancing head CarBooking firstBooking = headNext.booking; if(firstBooking!=null){
headNext.booking = null; //Line8 return firstBooking; //Line9 } } }else{ //No booking yet if(headNext != null){
//Line10 compareAndSetNode(this.tail, currTail, headNext); }else{ return null; } } } } }
```

Here, the unit element of the queue is a Node that has a booking property holding the CarBooking and nextBooking to point to the next Node in queue. The BookingQ starts with its head and tail pointer set to a Node with NULL booking and NULL nextBooking. The process of putting any node into the queue requires updating two pointers:

Step 1: set the next pointer of current node to the new node
Step 2: advance the tail pointer to the new node
also

After successfully adding a node to the queue, the head's nextBooking would point to new not-NULL node, but the booking element still remains NULL.

The CAS is applied wherever there is a contention point, like updating a node in compareAndSetNext or compareAndSetNode.

Now consider a situation when a new booking is being added in putBooking by a thread (T1). It checks to find tail of the queue in Line 2 and does compareAndSetNext. But between Step 1 and 2 if another thread (T2) tries to put another booking and finds in Line 1 that another thread's work is underway (meaning that Line 3 is done, but 4 is not, so the tail's nextBooking no more NULL), it then completes T1's unfinished job (Step 2) of updating the tail pointer to the new node. After that, T2 goes to the next iteration to put a new booking.

In case of getBooking, when the tail and head are not same (Line 6), it advances the head pointer to the next node of first booking node (Line 7) and returns its booking element (Line 9). Setting headNext.booking to NULL (Line 8) resets the head node back to a starting point. As head no longer references the previous node, the Garbage Collector would collect it eventually, making the queue length shorter. In line 10, a similar situation to putBooking may arise when a thread tries to read off of the empty queue and between Line 5a and 5b another thread puts a booking (Line 3), and in that case it also advances the tail pointer to the new booking.

Note, we could do the same CAS using the Java AtomicReference class, but for clarity of understanding we've just elaborated it a bit.

Conclusion

Referring back to important laws, it becomes clear that though modern processors provide better means to improve application performance, the program is where the work of implementing parallelism must occur. The more the program can be decomposed into parallelized parts, the better it performs on multicore computers. In addition, it becomes clear that while working with explicit locks gives you better control than the JVM intrinsic lock, non-blocking operations also enable you achieve higher performance than any synchronized operation. In any case, parallelized programs must be designed carefully and tested properly. Even small oversights may result in undesirable consequences in multithreaded applications.

[Dibyendu Roy](#) has more than ten years of design and development experience in various domains including Banking and Financial Systems, Business Intelligence tools and ERP products.

<http://www.java.net/article/2010/04/14/rethinking-multi-threaded-design-principles-part-2>

HTML5 Server-Push Technologies, Part 1

Server-Sent Events and WebSockets explained HTML5 Server-Push Technologies, Part 1 Wed, 2010-03-31
[Gregor Roth](#) Server-Sent Events and WebSockets explained

<http://www.java.net/article/2010/03/31/html5-server-push-technologies-part-1>

Flexible Swing Reporting Using JIDE Aggregate and Pivot Tables

Learn about a Swing report alternative that provides 90% of the solution with 10% of the effort. Flexible Swing Reporting Using JIDE Aggregate and Pivot Tables Mon, 2010-03-22 [Malcolm Davis](#) Learn about a Swing report alternative that provides 90% of the solution with 10% of the effort.

<http://www.java.net/article/2010/03/22/flexible-swing-reporting-using-jide-aggregate-and-pivot-tables>

Getting Started with Java and SQLite on Blackberry OS 5.0

Get started in creating applications that utilize the SQLite database engine on Blackberry OS 5.0. Getting Started with Java and SQLite on Blackberry OS 5.0 Wed, 2010-03-17 [Bruce Hopkins](#) Get started in creating applications that utilize the SQLite database engine on Blackberry OS 5.0.

<http://www.java.net/article/2010/03/17/getting-started-java-and-sqlite-blackberry-os-50>

Rethinking Multi-Threaded Design Principles

Designing and improvising a thread based application is a challenge. But by following certain design principles and guidance, this can be easily overcome. Rethinking Multi-Threaded Design Principles Wed, 2010-03-03 [Dibyendu Roy](#) Designing and improvising a thread based application is a challenge. But by following certain design principles and guidance, this can be easily overcome.

Introduction

For many years, we have been using Java Threads for developing numerous Client-Server based applications. While it's always desirable to have a highly responsive Client application, handling a high volume of client requests has always been a prime goal of any Server application. However, designing a highly-scalable server requires lots of analysis. Without careful design and adherence to a well thought out underlying policy, a thread based system can fail to produce the desired outcome.

In this article, we'll discuss certain design principles that should be followed when the objective is to build an efficient, thread-safe application.

Thread Safety lies in Domain

As a design principle of thread safety, the vital point to look into is "Domain First." After all, the whole structure of an application lies on the Domain itself. If you get your domain model right, most of the problems are automatically taken care of. A model represents certain aspects of business logic consisting of various constraints and invariants. Understanding the pre and post conditions of a model are much needed in order to make the realized application thread-safe.

Let's imagine an travel agency that arranges budgeted cars for their traveler guests. They need an interactive car rental application that allow them to book multiple cars at different tourist locations on different dates within a specified price range. As they keep making multiple rental requests, at the same time if the system fails to find a car at particular location within price range, they want to be notified with suggested locations on the same screen.

We can think of this as a booking Requester taking the rental requests and putting them in a thread safe collection CarBookings that would be read concurrently by an actual CarFinder process to provide suggested locations. If CarFinder succeeds, it forwards this via the booking engine to the booking rental location. If CarFinder does not succeed, it tries to find the cars based on the price range and nearest location. Once found, it adds the info to another thread safe collection CarBookingAdvices.

There are many thread safe data structures available, but if you're sure about the Thread Safety policy that you want to adhere to, it's best to create a domain model on your own. For example:

```
//Defining first CarBookings class CarBookings{ private final Set bookings = new HashSet (); public synchronized void putCar(CarBooking booking){ bookings.add(booking); } public synchronized Set getBookings(){ return Collections.unmodifiableSet(new HashSet(bookings)); }}
```

Since the methods putCar() and getBookings() provide a secure way of accessing the non-thread-safe instance variable bookings, the CarBookings can be used safely by both the BookingRequester and CarFinder process running in different threads. This meets our simple thread safety requirement.

```
//Defining BookingAdvice class BookingAdvice{ private final String bookingId=""; private final BigDecimal rate=new BigDecimal(""); private final String location=""; //Getter and Setter omitted } class BookingAdvices{ private final Set advices = new HashSet(); public synchronized void putAdvice(BookingAdvice advice){ advices.add(advice); } public synchronized Set getAdvices(){ return Collections.unmodifiableSet(advices); }}
```

```
//Defining Requester class BookingRequester{ private CarBookings bookings; private BookingAdvices advices; public void requestBooking(CarBooking booking){ bookings.putCar(booking); //Line 1 displayAdvice(advices.getAdvices()); //Line 2 } private void displayAdvices(Set advices){ //Displays the advices on the same screen }} //Defining Finder class CarFinder{ private BookingAdvices advices; private CarBookings bookings; public void processAdvice(){ List adviceList = getGeneratedAdvices(); for(BookingAdvice advice: adviceList) advices.putAdvice(advice); //Line 3 findAndProcessBookings(bookings.getBookings()); //Line 4 } private List getGeneratedAdvices(){ //Generate advice based on requests } private void findAndProcessBookings(Set bookings){ //Finds advice for the
```

booking if required }} Hiding Model State

But, there is a caveat: if the CarBooking class is mutable, then there is a possibility of modifying each CarBooking instance in the set returned by getBookings(). Does it make our model brittle? Well, as long as we can make sure that it won't be modified later on, we'll be fine with that. But, sometimes it's easier to get thread safety by hiding the model state and blocking the access to it thereafter. Here, we could do that by adding a copy constructor to the mutable CarBooking or making it immutable:

```
//Defining mutable CarBooking class CarBooking{ private String bookingId; private String location; private
BigDecimal rate; //Copy constructor public CarBooking(CarBooking booking){ this.bookingId=
booking.getBookingId(); this.location= booking.getLocation(); this.rate= booking.getRate(); } //public Getter
methods below} class CarBookings{ //Modified version public synchronized Set getBookings(){ Set
newBookings = new HashSet (); for(CarBooking b : bookings){ CarBooking newBooking = new
CarBooking(b); newBookings.add(newBooking); } return Collections.unmodifiableSet(newBookings); }}
```

So, when getBookings() returns, it returns a copy of the CarBooking, thus not allowing modification of the original state--which makes it purely thread safe.

```
//Defining immutable CarBooking class CarBooking{ private final String bookingId; private final String
location; private final BigDecimal rate; public CarBooking(String bookingId, String location, BigDecimal rate){
this.bookingId= bookingId; this.location= location; this.rate= rate; } //Getter omitted}
```

In case we make CarBooking immutable, we can just return the bookings set from getBookings() as shown in our first CarBookings example.

Do Thread Delegation if you can

Sometimes you might find yourself being overburdened with handling the thread safety in your domain model; in that case you can simply delegate it to a range of Java built-in data structures that are already proven and tested to be thread safe. Like in our example, if we had to use bookings as HashMap, where String could represent bookingId, we could have easily replaced that with ConcurrentHashMap--which is a thread safe HashMap. Likewise there are many other data structures available (under the java.util.concurrent package) like CopyOnWriteArrayList and ConcurrentLinkedQueue, to name a few.

Figuring Single Thread Confinement

To coordinate multiple threads in a lock-based application, the OS and JVM have to burn a fair amount of CPU cycles for context switching and thread scheduling--making the system less responsive sometimes.

This can be avoided by allowing only one thread to gain access to work on a unit model (either CarBooking or BookingAdvice) at a time, and also making sure that same unit model should not be re-worked by the same thread. To make this happen, we can employ bounded BlockingQueues (BookingQ and AdviceQ) that would act as a mediator between Requester and CarFinder. So, for a rental request, the Requester would just create a CarBooking and put it in the BookingQ queue, and on the other hand, CarFinder would take that off of the queue. The same thing will be done by the CarFinder to return a BookingAdvice in the AdviceQ queue. This way, multiple Requesters and CarFinders can work asynchronously, making the system highly responsive.

Use synchronized lock wisely

While synchronized lock provides an easy way to work with multiple threads, improper usage of the lock may cause unpredictable results. This becomes obvious when multiple locks are acquired to perform an atomic operation where the first lock is held until the last lock operation is performed. Let's consider our BookingRequester and CarFinder that are using simple Java Sets instead of CarBookings and BookingAdvices. Here two operations: putting request and getting advices for Requester (vice versa for CarFinder, lines 5,6,7 and 8) become implicitly atomic under the same lock:

```
class BookingRequester{ private final Set bookings = new HashSet(); private final CarFinder finder; public
BookingRequester(CarFinder cf){ finder = cf; } public synchronized void requestBooking(CarBooking
booking){ bookings.add(booking); //Line 5 displayAdvice(finder.getAdvices()); //Line 6 } public synchronized
Set getBookings(){ return bookings; } } class CarFinder{ private final Set advices = new HashSet(); private
BookingRequester bookings; public CarFinder(BookingRequester br){ bookings = br; } public synchronized
void processAdvice(){ List adviceList = getGeneratedAdvices(); for(Advice advice: adviceList
advices.add(advice); //Line 7 findAndProcessBookings(bookings.getBookings()); //Line 8 } public
synchronized Set getAdvices(){ return advices; } }
```

Now, a problem starts to creep in when two threads T1 and T2 try to call Requester requestBooking() and

CarFinder processAdvice() respectively at the same time. During requestBooking, before T1 calls getAdvices() on CarFinder (Line 6), it has to wait for T2 to complete processAdvice. But to complete processAdvice, T2 needs to call getBookings() on Requester (Line 8), which won't happen until T1 completes requestBooking()--leading to a deadlock situation.

In our previous implementation of BookingRequester and CarFinder, we have already delegated the two operations to be handled independently under different locks (one for CarBookings and another for BookingAdvices, lines 1, 2, 3 and 4) to avoid any deadlock.

But, here we can simply resolve it by moving synchronized from the entire method to only the time required to put the booking (Line 5) and advice (line 7).

Another important point regarding deadlocks would be the order in which the resources are accessed in an atomic operation. If two threads work on an operation with two resource locks being held in a different order, it can lead to a deadlock situation too. So, make sure all threads calling an atomic operation get all resource locks in the same order.

Conclusion

Designing and improvising a thread based application is a challenge. But by following certain design principles and guidance, this can be easily overcome. At the same time, clear understanding of thread safety policy is also essential as it helps you simplify the design. There are many other techniques available that we couldn't cover here. However, the principles presented here will always assist you in overcoming some of the thread safety related hurdles you might be facing as you develop thread-safe applications intended for operation on modern multicore and multiprocessor computers.

[Dibyendu Roy](#) has more than ten years of design and development experience in various domains including Banking and Financial Systems, Business Intelligence tools and ERP products.

<http://www.java.net/article/2010/03/03/rethinking-multi-threaded-design-principles>

Has JDBC Kept up with Enterprise Requirements?

Look beyond Type 4 architecture to address the latest requirements of the enterprise Java ecosystem. Has JDBC Kept up with Enterprise Requirements? Mon, 2010-02-22 [Jesse Davis](#) Look beyond Type 4 architecture to address the latest requirements of the enterprise Java ecosystem.

<http://www.java.net/article/2010/02/22/has-jdbc-kept-enterprise-requirements>
