Bitter messages

This chapter covers

- An overview of JMS and message-driven beans
- An example messaging application using JMS and MDBs
- Message-level design antipatterns
- Application-level design antipatterns
- Asynchronous communication antipatterns
- Performance antipatterns

We have been shredding the powdery slopes relentlessly since catching the first lift of the day. As we ascend one of Colorado's epic mountains to take yet another adrenaline ride, a storm rolls in and quickly begins blowing in a fresh layer of powder. Once off the lift, we take a seat, snowboarder style, at the top of the run to plot our line of descent. The density of snowflakes swirling in the low light conditions has decreased our visibility. Donning goggles, we push off and immediately fall into a rhythm of parallel S-turns that kick up wispy snow fans. Halfway down the mountain the slope suddenly forks, but I fail to see it through the blowing snow. After a few more turns, it hits me—I don't hear the familiar sound of another board carving across the snow. I wait at the edge of the silent slope for a while, but it's soon evident that my buddy zigged when I zagged. We're out of synch on an enormous mountain enveloped by a storm.

In *Bitter Java*, our fellow author, Bruce Tate, accurately predicted that message-driven beans (MDB) would provide fertile ground for antipatterns. Unveiled in EJB 2.0, MDB are still relatively new, yet unfortunate antipatterns have already begun to rear their ugly heads. The painful lessons these antipatterns teach aren't new. Indeed, message-based systems have been around for a relatively long time. Many seasoned developers wear the battle scars of messaging gone bad, but fueled by the need to quickly integrate applications with other internal and external applications, messaging has become increasingly pervasive. With the advent of MDBs, which promote asynchronous messaging as a first-class distributed computing model in the J2EE platform, the stakes have been raised. Yet another tool has found a home in our already brimming toolbox. And, as always, the wisdom of a craftsman will lie in knowing how and when (or when not) to use it.

In this chapter, we'll review the Java Message Service (JMS) and its recent introduction into the J2EE platform in the form of MDBs. Working through a simple example, we'll encounter potential pitfalls in designing message-based applications. Some antipatterns we'll uncover are related to application performance, while others fester at the application design level. As we look at each bitter scenario, we'll explore practical alternatives to ensure that our applications don't end up stranded.

6.1 A brief overview of JMS

JMS is an API that allows applications to communicate asynchronously by exchanging messages. JMS is to messaging systems what JDBC is to database systems. JMS is best used to glue together applications through interapplication messaging.

These applications, referred to as *JMS clients*, engage in asynchronous conversations by using a common set of interfaces to create, send, receive, and read messages. That's not to say you also couldn't use JMS for intra-application messaging to send messages between multiple threads, for example.

JMS itself is an industry-standard specification, not an implementation. Vendors of messaging products—commonly referred to as message-oriented middleware (MOM)—support JMS by providing implementations of the interfaces defined in the JMS specification. By relying only on vendor-neutral interfaces, applications are decoupled from any specific vendor. That is, the underlying vendor's implementation can be changed or substituted with another without breaking the JMS clients.

A vendor's JMS implementation is known as a *JMS provider*. A JMS provider includes the software that composes the *JMS server*, or message broker, and the software running within each JMS client. A *JMS application* therefore comprises multiple JMS clients exchanging messages indirectly through a JMS server. Figure 6.1 illustrates a common JMS application.

Notice that the JMS server acts as a middleman between JMS clients. This enables loosely coupled communication; neither client knows about the other. This loosely coupled communication improves reliability, since one client will not be dependent on the location, availability, or identity of another. Indeed, clients are free to come and go without adversely affecting reliability. This situation is in stark contrast to the remote procedure call (RPC) computing model used by CORBA and Java RMI. Applications using RPC communicate directly with each other. As such, they tend to be tightly coupled.

That's enough theory. We'll learn far more about JMS by getting our hands dirty building an example application.

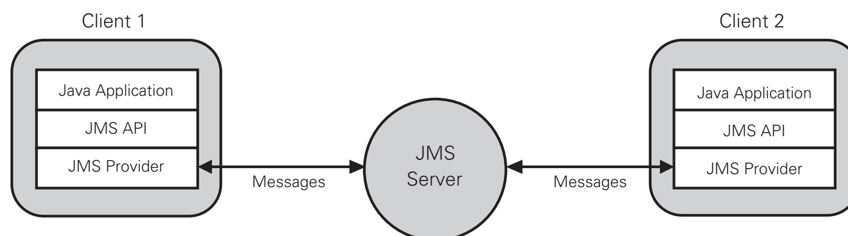

Figure 6.1 In a JMS application, applications use the interfaces of the JMS API to communicate indirectly through the JMS server. Under the hood of each JMS client and within the JMS server, a vendor's JMS implementation does all the heavy lifting.

6.2 An early antipattern: Fat Messages

Messages are the lingua franca of messaging systems. Any application that can speak in messages is welcome to join in conversations. *Message producers* are JMS clients that send messages. *Message consumers* are JMS clients that receive messages.

The message language is defined by the JMS specification, which specifies six different message types that vary with the type of payload they transport. Think of these message types as dialects of the message language. They all sound similar, but each has a slightly different accent. Their similarity lies in a common structure: headers, properties, and a payload. The headers and properties define routing and other information about the message. The payload, or message body, is the meat of the message. It contains data of specific interest to message consumers. The structure of the payload is unique to each message type. Table 6.1 breaks down each message type by its respective payload.

Table 6.1 JMS message types vary by the structure of their payload. Each can carry light or heavy loads. Fat messages clog up the messaging pipes and invariably impact the performance of a messaging application.

Message type	Payload
`Message`	No payload, just headers and properties
`TextMessage`	Java string (text or serialized XML document)
`MapMessage`	Set of name-value pairs
`ObjectMessage`	Serialized Java object
`BytesMessage`	Stream of uninterpreted bytes
`StreamMessage`	Stream of primitive Java types

Each message type is useful in different scenarios. Picking the best message for the situation is a critical design decision that affects not only the semantics of the message exchange, but also the performance of the system. Table 6.2 presents each message type that contains a payload, along with a few considerations to keep in mind when choosing a message type.

Table 6.2 Before picking a message type, carefully consider if the data being exchanged fits neatly into the payload the message type was designed to carry.

Message type	Key considerations
`TextMessage`	Because the JMS specification does not define a standard XML message type, the `TextMessage` commonly is used to transport a serialized XML document. However, any time the payload contains formatted text, such as XML, it must be parsed by consumers before it can be used intelligently.
`MapMessage`	Messages of this type are the most versatile. Predefined keys are used to read specific values of the payload. This allows the payload to grow dynamically over time without affecting consumers. Consumers that aren't aware of new keys will be ignorant of their existence. If consumers always read the entire payload in a well-defined order, carrying the keys around may become a dead weight. In these cases, `StreamMessage` may yield better performance.
`ObjectMessage`	Producers and consumers of this type of message must be Java programs. When a producer sends a message of this type, the object in the payload and the transitive closure of all objects it may reference must be serialized. That is, if the object in the payload references other objects, then consumers will receive the graph of objects reachable from the object in the payload. Deep object graphs bloat the message and restrict message throughput. Additionally, all consumers must be able to successfully deserialize the object(s) in the payload using a class loader within their respective JVMs. This means that all consumers must have access to the class definitions of the objects in the payload.
`BytesMessage`	Because this message type's payload is raw uninterpreted bytes, all consumers must understand how to interpret the payload. No automatic data conversions are applied to the payload as it's transported between consumers. This message type is rarely used, and, when it is, only to transport data of a well-known format, such as a MIME type, supported by all consumers. In most other cases, a `StreamMessage` or a `MapMessage` is more convenient.
`StreamMessage`	Unlike the `BytesMessage`, the `StreamMessage` retains the order and type of the primitives in the payload. Moreover, data conversion rules are automatically applied to the primitive types as they are read by consumers. A `StreamMessage` is a more rigid variation of a `MapMessage` in that keys do not index its data. However, because it doesn't carry around keys, this message type is generally more lightweight than a `MapMessage`. Nevertheless, unlike the `MapMessage`, a `StreamMessage` requires that consumers have explicit knowledge of the message format.

It's not always clear which message type is best to use. Some message types are used more commonly than others, based simply on the type of data being exchanged. The `TextMessage`, for example, is the natural choice for exchanging structured text. Without carefully considering the flavor of payload consumers will require, you may easily fall into the comfortable habit of using the same message type for all situations. Often, the result will be awkward, like fitting square data in a round message.

6.2.1 *One size doesn't fit all*

Designing messages in a vacuum is like designing a software component in the absence of clients. Speculation often leads to messages that are neither useful nor efficient. Take, for example, a message representing a purchase order. How much information must the message carry to be useful? The answer depends on the consumer of the message.

If the consumer is a sales automation system using the message to spot cross-selling opportunities, then including a wealth of information about the customer may be important. If the message is too brief, this type of consumer may have insufficient information to efficiently process the message. Attempting to gather more information may lead to a two-way dialogue between the producer and the consumer. Chattiness of this sort negates the benefits of loose coupling and asynchronous communication offered by JMS.

On the other hand, if the consumer is an inventory system using the message to fulfill an order, then the customer information may be unnecessary. Making the message unnecessarily verbose will fatten it up, thus requiring additional network bandwidth and CPU resources. Moreover, if the fat message is persisted to nonvolatile storage to ensure guaranteed delivery, it will require additional storage space. That being said, if the frequency at which a fat message is produced is low, then the respective overhead may be tolerable. However, as the message frequency increases, the overhead will compound until it adversely affects message throughput.

So, fat messages end up being a common problem because assumptions about consumer needs are easily made. A message that tries to be everything to everybody inevitably carries a high delivery price; it clogs up the messaging pipes and wastes space.

6.2.2 *Solution 1: Put messages on a diet*

Ideally, a message should contain just enough information to enable its consumers to handle it on their own. Designing such a message is akin to designing a programmatic interface to a distributed service. To decrease coupling and chattiness, thin interfaces generally are used to encapsulate business logic behind coarse-grained methods. Given just the right amount of information, these methods go about their business without exposing any implementation details. In contrast, fat interfaces usually are guilty of hiding monolithic business processes that are tightly interdependent. Getting anything useful to happen often requires calling multiple methods and supplying superfluous information.

Therefore, when designing loosely coupled messaging applications, it's best to follow the lessons taught by good interface-based design:

- Start by designing the interfaces—the shape and size of the messages.
- Choose a message type capable of carrying the simplest payload that meets the needs of known consumers.
- Avoid fattening up the message by speculating about the kinds of data needed by future consumers.
- Eliminate duplication by omitting data that a consumer could derive from the information already in the message.
- Take into account how often the message will be delivered and whether the delivery of the message must be guaranteed.

Knowing when to put a message on a diet isn't an exact science. No scale exists that can accurately weigh a particular message. In addition to including the size of the static payload and any application-specific headers and properties, the JMS provider may pile on additional properties at the time of delivery that contribute to the overall size of the message. If you know the approximate size of the payload, you'll find that is usually sufficient as a rough estimate when planning for performance. Remember, too, to factor in the frequency of the message delivery. Little messages can add up quickly to big performance headaches.

6.2.3 Solution 2: Use references

Sending references to information otherwise contained in a message can help reduce the size of the message. Rather than sending a fat message stuffed with raw data, it's often possible to send a lightweight message instead, one that simply contains a reference to that data. Think of it as a particular type of weight loss program where a message is encouraged to eat references. References are especially powerful in situations where large amounts of data need to be exchanged without incurring excessive performance overhead. A reference could be a URL, a primary key, or any other token pointing to the data.

For example, consider a workflow application that uses messages to route electronic documents to multiple departments. As the document transitions through its life cycle, from draft to approval, it travels from one department's message queue to the next. At each stopping point, more information may be added to the message. This workflow could be implemented using the `BytesMessage` message type to route the document in its native format. However, a downside to that approach exists: as its size increases, the document will become unwieldy. Each

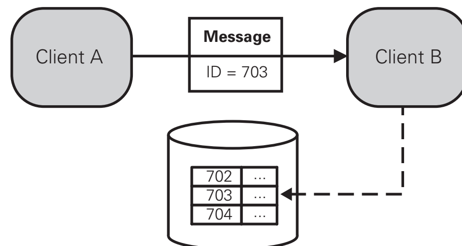

Figure 6.2
References can be used to point to the actual information otherwise contained in the message. This approach has the potential to significantly decrease the size of fat messages. In this example, the message contains a primary key for a row in a shared database table. When the consumer receives this message, it can load and interpret the data at its leisure.

department's queue will be burdened with managing the document in memory until it's been processed. The situation turns particularly sour if a copy of this fat message is broadcast to multiple consumers. Every network path from the producer to each consumer will have to swallow the fat message like an egg-eating snake. In the end, not all consumers may want the message after it's delivered. References can come in handy in these situations because they significantly decrease the size of the message.

If all message consumers have access to a shared resource, such as a database or a file system, consider trimming down messages to contain references to shared data. As an alternative to transporting an entire document, for example, the message representing the document could simply contain the name of a shared file. When a consumer receives the message, it can process the referenced document at its leisure by reading the document from the file system. Figure 6.2 illustrates the use of references to reduce the size of messages.

It's never too early to start putting messages on a weight loss program, but don't go overboard. MOM products have matured significantly over the years. Some vendors have had time to optimize their products for sending large messages over different networks. Before making any assumptions, write a few tests to measure performance. A message that may be perceived as being too large actually might transmit much faster than you think. And, if a fat message is sent infrequently, it may not be a problem. Prematurely hacking away at the message size can lead to another common problem—skinny messages.

6.3 Mini-antipattern: Skinny Messages

Despite warnings about fat messages, skinny messages are equally problematic. Striking a balance between too much and not enough information is the essence of good message design.

In general, it's always better to send a bit too much information. A couple of extra bytes on a message will generally have little overall effect on performance. On the flip side, a skinny message with too few bytes may create more work for a consumer. To successfully handle the message, the consumer may have to make extra remote calls to get more information.

Using references isn't always the right answer either. First, each consumer is burdened with resolving the references on his own. In other words, the producer can't package up all the information once and then share it with all the consumers. Worse yet, consider the case where several consumers attempt to resolve a reference to a document as soon as the message arrives. Consumers may end up competing for access to the shared file system in a flurry of network activity, causing them to block. Consequently, message throughput suffers as the consumer can't process new messages until the current message is handled. In the end, this may be far more CPU- and network-intensive than just sending the entire document in the first place.

To reiterate, not sending enough information in a message can weigh down an otherwise efficient messaging application. The virtues of asynchronous communication may be taken over by a much slower, synchronous conversation induced by contention and blocking.

6.3.1 Solution: Use state to allow lazy loading

One performance-boosting variation of references is to include some state information in the message, along with the reference. For example, in addition to the document reference, the message could also include the current state of the document. For instance, states might include: NEW, REVISED, or APPROVED. The presence of the state in the message allows consumers to make a decision about whether loading the document is necessary. Consequently, only consumers that actually need the document will access the shared file system, reducing the potential for delayed blocking. State can be added to a message in a variety of ways. Putting state in the payload is one way, although the consumer will bear the burden of filtering. In section 6.12, we'll discuss how to use message selectors. Message selectors tell the JMS server how to filter messages before delivering them to consumers. The filtering is based on the contents of each message's headers and properties.

6.4 Seeds of an order processing system

We ran into two pitfalls before starting our journey: fat messages and skinny messages. We would do well to keep these potential troublespots in mind before messages start swirling around. Now, we're ready to dive into a working example. We want an example we can sink our teeth into, so we'll develop the underpinnings of an asynchronous order processing system using JMS. Although we'll write gratuitous amounts of code, as an example of JMS, ours will fall well short of providing a comprehensive tour of JMS. Albeit easy to learn and use, JMS can be applied in a range of enterprise application integration (EAI) and business-to-business (B2B) scenarios. Our example will illustrate merely one isolated application of JMS—with a few pitfalls sprinkled in along the way to keep us on our toes. Throughout the rest of the chapter, we'll continue to refactor the application example, each time eliminating a weakness in its design.

6.4.1 Defining the system

Let's assume that we have a legacy order fulfillment application that we'd like to tie in with a J2EE online order processing system. Rather than modifying the legacy system to interface directly with the new system, we'd prefer to integrate the two worlds using a loosely coupled design. When an online order is initiated through the order processing system, it should trigger the following business logic sequentially:

1. Store the order information in an order database.
2. Deliver the order to the legacy order fulfillment application.
3. Broadcast a notification indicating the order's status.

These tasks must be completed in lock-step as an atomic business process. If any step fails, the entire process will also fail and would have to be repeated anew. However, we don't want our online customers to be blocked, waiting for the completion of this relatively lengthy business process. Customers don't need to wait; they are happy to place an order request and receive later notification—an email, for example—to confirm that the order has been fulfilled.

Reliability is paramount because we can't afford to lose any customer orders. When an order request is issued, we should be able to guarantee its disposition. In light of these requirements, we decide to use JMS as the integration glue. Using it correctly is the challenge.

6.4.2 Designing messages

As we learned in the previous section, designing the messages that form the interface between our applications will help us determine how those applications interact. Based on our admittedly simple use case, we need two messages: an `OrderRequest` message and an `OrderStatus` message.

The OrderRequest message

An `OrderRequest` message is used to initiate the order fulfillment process. Messages of this type are sent to exactly one consumer—the legacy order fulfillment application. Table 6.3 dissects the payload of an `OrderRequest` message.

Table 6.3 An `OrderRequest` message requests fulfillment of an online order.

Name	Description	Type	Example value
Order ID	The order's unique identifier	String	104-549-736
Product ID	The product's unique identifier	String	Ride Timeless 158
Quantity	The number of units to buy	int	1
Price	The product's unit price in dollars	double	479.00

At this point, we can't be certain that we've considered all possible attributes of an `OrderRequest` message. We'll keep the message simple for now.

The OrderStatus message

The second message we need, an `OrderStatus` message, is just an indication of an order's disposition. This type of message is broadcast to any application that has registered interest in the life cycle of orders. For example, the sales automation system might monitor the status of an order as it progresses through the system. This message is broadcast only after the legacy order fulfillment application has had an opportunity to process the order represented by an `OrderRequest` message.

Imagine that a message of this type contains a unique identifier for an order, an order status code, and an optional text describing the order's status. Notice that the unique order identifier is actually a reference to the original order. We don't need to include all the details of the original order in an `OrderStatus` message because subscribers of this message type are generally only interested in the order's disposition. However, if a particular subscriber wants the details of the original order, the identifier can be used to query the shared order database. In other words, the `OrderStatus` message is designed for a specific type of

consumer. A reference is used to accommodate the few subscribers that may have special interests.

Having considered the message design, we're ready to decide now how these messages should be delivered.

6.4.3 Choosing messaging models

We have a couple of choices when deciding how our messages should be delivered to consumers. In general, the JMS server receives messages from producers and delivers the messages to consumers. Specifically, JMS provides two different messaging models: *publish/subscribe* and *point-to-point*.

The two messaging models use a slightly different vernacular. The publish/subscribe messaging model allows a message publisher (producer) to broadcast a message to one or more message subscribers (consumers) through a virtual channel called a *topic*. The point-to-point messaging model allows a message sender (producer) to send a message to exactly one message receiver (consumer) through a virtual channel called a *queue*. Figure 6.3 illustrates the two messaging models.

By communicating indirectly through virtual channels managed by the JMS server, producers and consumers are decoupled from one another. That is to say that a consumer's location, availability, and identity are unknown to the producer.

In our example application, an `OrderRequest` message should be processed by only one consumer—the order fulfillment application. Therefore, we'll use the point-to-point messaging model to deliver these types of messages. In contrast, an `OrderStatus` message must be delivered to all clients that have registered interest in the disposition of orders. Therefore, we'll use the publish/subscribe messaging model to broadcast these types of messages. Figure 6.4 shows an architectural diagram of the JMS components collaborating to fulfill an order.

Notice in the architectural diagram that the client that receives the `OrderRequest` message is also a publisher of `OrderStatus` messages. A JMS client can serve both roles—producer and consumer—to bridge between messaging models. Also, keep in mind that each client could be running in its own virtual machine and perhaps even on separate machines in the network.

6.4.4 Responding to change

Fortunately, the JMS API for the publish/subscribe and point-to-point messaging models are remarkably symmetrical. In general, only the names change when switching from one messaging model to the other. Every method and class name containing the substring `Topic` can be changed to `Queue`, and vice versa. A few other minor details and model-specific features exist, but by and large, the APIs

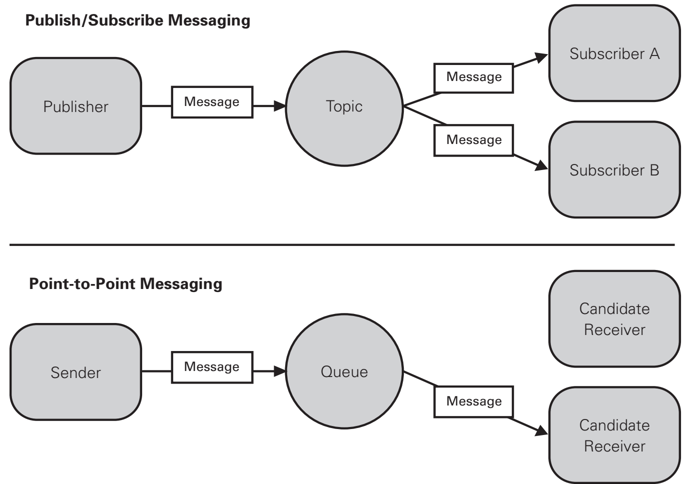

Figure 6.3 The publish/subscribe message model publishes a copy of a message to each subscriber through a topic. The point-to-point messaging model sends any given message to exactly one of possibly many receivers through a queue. The topic or queue decouples all participants to allow their location, availability, and identity to vary independently.

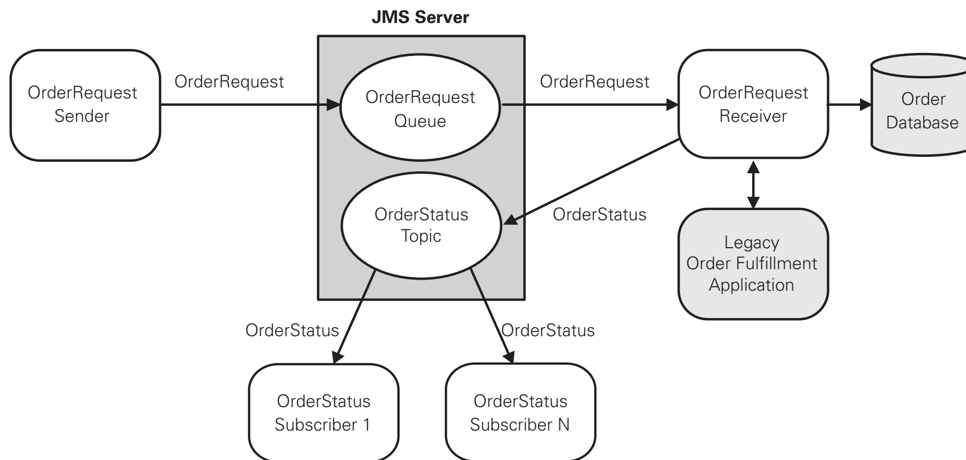

Figure 6.4 Messaging applications can be a hybrid of publish/subscribe and point-to-point messaging, depending on the number of message consumers interested in each message.

mirror each other. The upshot is that the skills you learn using one messaging model are portable to the other. It's worth mentioning that the open source Messenger (http://jakarta.apache.org/commons/sandbox/messenger/) library makes using JMS a bit easier. It effectively hides the differences between messaging models and their delivery options.

In the future, more than one consumer may want to know when an order is placed. For example, a sales automation system might also track `OrderRequest` messages to identify potential cross-selling opportunities. For now, we'll use the point-to-point messaging model, keeping in mind that the JMS APIs are on our side if needs change down the road.

We've yet to delve into how the consumer of `OrderRequest` messages is developed and packaged. We'll get there in good time, but first, let's look at the system from the perspective of the order producer. It's here that we'll gain valuable insight into the design of our application.

6.4.5 Building the OrderRequest producer

In the architectural diagram, the client that produces the `OrderRequest` messages appears to be stand-alone. However, we can safely assume that this client has more responsibilities. Indeed, if we were to zoom out a few thousand feet, we'd see that the `OrderRequestSender` is actually just a single component in a larger J2EE application. Orders are placed over the Internet through a web application that, among other things, uses this component to integrate with the order fulfillment application through messaging.

Using a flexible message format will allow us to add new attributes easily to `OrderRequest` messages later, if necessary. Several JMS message types will work, but we might be tempted to use serialized XML in the payload of a `TextMessage`. After all, the message could be easily represented as structured text, and XML offers the ultimate in flexibility and portability. Indeed, XML is a wonderful technology and a million ways exist for using it well, but this isn't one of them. To see why, let's look at listing 6.1 to see how we might send an `OrderRequest` message containing XML.

Listing 6.1 A JMS client that sends OrderRequest messages to a message queue

```
public class OrderRequestSender {

  private QueueConnection connection;
  private QueueSession session;
  private QueueSender sender;

  public void connect() throws NamingException, JMSException {   <-- Creates a QueueSender connected to the OrderRequestQueue

    Context ctx = new InitialContext();

    QueueConnectionFactory connectionFactory =
      (QueueConnectionFactory)
      ctx.lookup("OrderRequestConnectionFactory");

    connection = connectionFactory.createQueueConnection();

    session = connection.
      createQueueSession(false, Session.AUTO_ACKNOWLEDGE);

    Queue queue = (Queue)ctx.lookup("OrderRequestQueue");

    sender = session.createSender(queue);
  }

  public void sendOrder(OrderRequest order) throws JMSException {

    TextMessage message = session.createTextMessage();

    message.setText(order.toXML());   <-- Fills the OrderRequest message payload with XML

    sender.send(message);   <-- Sends the OrderRequest message to the OrderRequestQueue
  }

  public void disconnect() throws JMSException {   <-- Disconnects from the OrderRequestQueue
    connection.close();
  }
}
```

Notice that we create a `TextMessage` containing a serialized string of XML by invoking the `toXML()` method of an `OrderRequest` business object. In other words, the `OrderRequest` message is simply an XML representation of the `OrderRequest` business object. Unfortunately, the contents of the message aren't explicit. That is, without parsing the XML, we can't tell what types of data it contains.

Now that we see the world through the eyes of the `OrderRequestSender` and in the context of our architecture, we can tell all is not exactly as we imagined. Indeed, there's a bitter taste in our mouth. Using XML as the payload of the `OrderRequest` message seemed like a good idea since XML is both flexible and portable. However, data portability isn't really an issue because we'll be building the consumer of `OrderRequest` messages. Furthermore, we can achieve flexibility

with other message types. So, given that both the producer and the consumer are within our control, no clear advantages exist to using XML in this scenario. Before going much further then, let's reconsider the decision to use XML.

6.5 Antipattern: XML as the Silver Bullet

At first blush JMS and XML appear to be a match made in heaven. To some extent they are kindred spirits that can team up to solve historically vexing problems. One beauty of JMS is that it allows messages to be exchanged throughout a heterogeneous environment in a platform-neutral fashion—a noble challenge of EAI. In practice, although Java applications themselves are platform-neutral for the most part, not all systems glued together with JMS are Java applications. To further extend the reach of messaging, some JMS vendors provide support for messaging between Java and non-Java clients.

JMS is aimed at enabling the ubiquitous transfer of messages, and XML stands tall when it comes to expressing data in a portable and flexible format. Indeed, because XML distills down to a stream of text, it can be interpreted by any platform. It's also flexible in the sense that, despite conforming to a well-defined structure, an XML document can easily be extended to include new data elements without affecting current applications using it.

Sounds great, right? Not so fast. It comes as no surprise that XML has the potential to be overused. It's a fate shared by many new technologies that come on the scene with great fanfare. While some may contend that putting XML in the drinking water will make everyone's teeth whiter, XML is best used in moderation. We'll go out on a limb here and predict that a book on bitter XML wouldn't be known for its brevity. Swinging the XML hammer for the sake of XML is not without its price. When a JMS consumer receives an XML message, that message must be parsed before it can be used for anything meaningful. The overhead of parsing XML will elongate the time required for the consumer to process the message. This extra processing may in turn limit the overall message throughput of the application. As such, XML loses many of its advantages when you control all the producers and consumers. Regardless of the performance implications—which certainly must be measured before forming any conclusions—the burden of parsing should be hoisted on consumers only when a definitive advantage exists in using XML.

6.5.1 Solution: Use XML messages judiciously

XML is no panacea. In many cases, the `MapMessage` has all the same virtues as a message containing XML, without the performance hit of parsing. With respect to

portability, most JMS vendors will automatically convert a `MapMessage` produced by a Java application to an equivalent message in a non-Java environment. Native conversions of this sort are generally less expensive performance-wise than parsing XML. The format of a `MapMessage` is also flexible in that new name-value pairs can be added easily without breaking existing consumers. Moreover, messages containing XML have the disadvantage of not supporting runtime validation afforded by the explicit, strongly typed methods of a `MapMessage`.

That's not to say a powerful synergy doesn't exist sometimes between XML and JMS. For example, messages that must be represented in a hierarchical structure can certainly benefit from the flexibility of an XML message. As well, messages that travel beyond the edges of your intranet to communicate with other systems can reap the rewards of portable XML. In the future we're likely to see an even tighter coupling between these two technologies in a wide range of applications from EAI to B2B.

In any event, the best approach is to start with the simplest message type and benchmark its performance. Then, if an XML message becomes necessary, you'll have something to compare that message against. Is parsing XML messages a performance bottleneck? Wait! Don't answer that just yet. First, gather hard evidence with a performance test for the actual situation in question. Then, use that information to make an informed decision. You might be pleasantly surprised.

To reiterate, serializing XML into an `OrderRequest` message doesn't buy us much over a `MapMessage` in our application. We're designing all the producers and consumers and, at this point, the message has a flat structure. With a `MapMessage`, we're also free to add new data without affecting current clients, should that become necessary. Listing 6.2 shows the refactored method that uses a `MapMessage` type when sending an `OrderRequest` message.

Listing 6.2 Refactoring from an XML message to a MapMessage

```
public void sendOrder(OrderRequest order) throws JMSException {

  MapMessage message = session.createMapMessage();

  message.setString("Order ID", order.getOrderId());
  message.setString("Product ID", order.getProductId());
  message.setInt("Quantity", order.getQuantity());
  message.setDouble("Price", order.getPrice());

  sender.send(message);
}
```

Notice that the message is now more explicit. And we get the advantage of strong type checking. When the consumer reads the message, each attribute's type is unambiguous. For example, a consumer can now read an `OrderRequest` message as shown in listing 6.3.

Listing 6.3 Reading an OrderRequest message

```
String orderId = mapMessage.getString("Order ID");
String productId = mapMessage.getString("Product ID");
int quantity = mapMessage.getInt("Quantity");
double price = mapMessage.getDouble("Price");
```

We finally have our messages nailed down! Next, we must decide if guaranteeing their delivery brings anything to the party.

6.6 Antipattern: Packrat

Guaranteed message delivery, one of the cornerstones of messaging systems, always comes at a price in terms of resources and performance. Anything advertised as guaranteed seems to bear that caveat. Alas, no free lunch exists here.

The JMS specification contains provisions for configuring a messaging system to achieve different Quality of Service (QoS) levels. Building on that foundation, JMS vendors compete by including value-added reliability features to their product offerings. The QoS level we choose is dependent largely on our specific application's requirements. After all, we want to get something for the price we pay.

Reliability is measured on a sliding scale. It's not just a single toggle switch we flip on or off, but rather a panel of control knobs. If we turn them all to their highest setting, we'll get maximum reliability and, possibly, horrible performance. Turn them down to their lowest setting, and we'll get minimum reliability with improved performance. It's a trade-off; the right setting usually lies somewhere in the middle. Two mechanisms for guaranteeing message delivery with the highest potential for misuse are persistent messages and durable subscriptions. Failure to understand their potential cost can set us up for a big fall.

6.6.1 Putting a price on persistence

JMS defines two message delivery modes: *persistent* and *nonpersistent*. When a message marked as persistent is sent to the JMS server, it's immediately squirreled away in nonvolatile storage. Only after the message is stored safely does the message producer receive an acknowledgment that the JMS server has agreed to deliver the

message. By taking this responsibility, the JMS server guarantees that the message will never be lost. As the server metes out each persistent message to consumers, it keeps track of consumers that have actually received the message. The consumers help by acknowledging the receipt of each message. If the JMS server fails (or is restarted) while delivering messages, then upon recovery, the server will attempt to deliver all persistent messages that have yet to be acknowledged.

Non-persistent messages, on the other hand, aren't stored on disk. Therefore, they aren't guaranteed to survive a JMS server failure or restart. As such, non-persistent messages generally require fewer resources and can be delivered in less time than persistent messages. Higher levels of message throughput usually can be realized by using non-persistent messages at the expense of reliability.

By default, a message producer marks all messages as being persistent. Each message sent will be stored on disk before it's delivered. With our `OrderRequestSender`, that step works to our advantage because we can't afford to lose `OrderRequest` messages if the JMS server fails or is restarted. Figure 6.5 illustrates the sequence of events in delivering a persistent `OrderRequest` message.

Every `OrderRequest` message is guaranteed to be delivered once—and only once—to the `OrderRequestReceiver`. In contrast, ensuring that every `OrderStatus` message is received by its consumers may not be a requirement of our business. Instead, we may be able to deliver these messages once at most and avoid the overhead of guaranteed delivery. That is, it won't be the end of the world if one of these

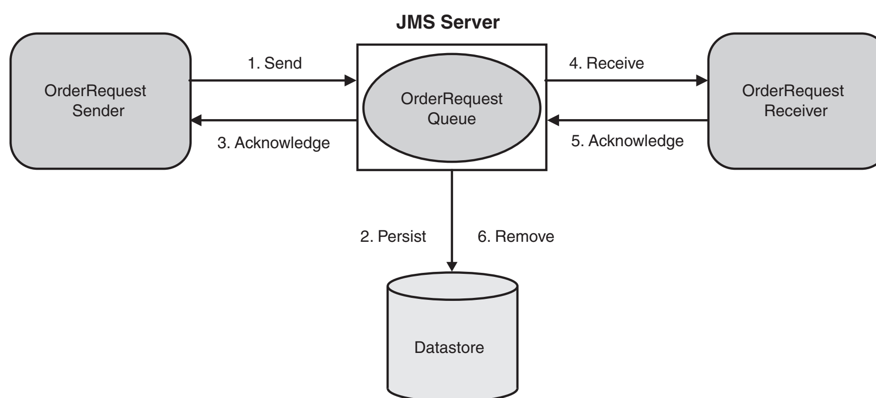

Figure 6.5 Persistent messages must be stored in nonvolatile storage by the JMS server before acknowledging the message producer. These messages are then removed from storage upon successful delivery. Not all messages require this degree of reliability. For messages that need to be delivered once at most, better throughput can be realized.

messages falls on the floor. Unless we explicitly mark `OrderStatus` messages as non-persistent, our application will suffer the burden of guaranteeing their delivery.

It's important to note that messages sent using the point-to-point messaging model must be placed on a queue in the JMS server prior to delivery, regardless of whether or not they are marked persistent. A point-to-point message not marked as persistent lives on the queue until it's consumed or the JMS server fails or restarts. Therefore, messages not consumed at a rate equal to or greater than their rate of arrival may cause the queue to grow unchecked, putting additional strain on the JMS server. Publish-subscribe messages, in contrast, don't necessarily have to be stored internally before delivery.

6.6.2 Paying for durable subscriptions

Durable subscriptions, another mechanism for guaranteeing message delivery, are a feature specific to the publish/subscribe messaging model. A durable subscription outlives a subscriber's connection to the JMS server. That is, when a message arrives at a topic for which a durable subscriber has registered interest, and the subscriber is disconnected, the JMS server will save the message in nonvolatile storage. In essence, the undelivered message is treated as a persistent message. The JMS server will continue to store any outstanding messages until the durable subscriber has reconnected. Once the subscriber has reconnected, all outstanding messages are forwarded to it. If a message expires before the subscriber reconnects, the message will be removed.

Here's the rub: If a durable subscriber is disconnected for relatively long periods of time, and messages have a long life span, the JMS server is burdened with having to manage all outstanding messages. The resulting strain on resources is similar to that of persistent messages. For each durable subscription, the message server must internally keep track of the messages each durable subscriber has missed for a given topic.

6.6.3 Solution: Save only what's important

Certain types of messages are so critical to your business that you can't afford to lose one. By all means, use the power of JMS to guarantee their delivery to the extent necessary. If, however, certain types of messages can be missed when things go bad, then you should avoid incurring the unnecessary overhead to guarantee their delivery.

In our order processing system, for example, losing an order request if the JMS server fails will adversely affect our bottom line. We must guarantee that every `OrderRequest` message ultimately arrives at our legacy order fulfillment

application. Therefore, the `OrderRequest` message is persistent. Additionally, if the legacy order fulfillment application itself fails, or is taken offline for maintenance, we must guarantee that any messages it misses will be delivered once it has recovered. Therefore, its subscription must be durable. We don't have to do anything special for durability in this case. Messages sent to a queue are implicitly durable; they'll be waiting when the consumer comes back online. At the end of the day, we're willing to incur the overhead of persistence and durable subscriptions in exchange for peace of mind.

Conversely, we may be willing to tolerate the loss of an `OrderStatus` message, a temporal message reflecting an order's state at a given instant. If a subscriber misses an `OrderStatus` message, the worst-case scenario is that the subscriber must check the order status in the order database. The inconvenience of missing a message just doesn't warrant the cost of burdening the JMS server with the tasks of storing each message, then deleting the message later once all interested subscribers have successfully acknowledged it. And remember, because we can easily bolt on more reliability later, if necessary, we'll do best by starting simple.

6.7 Mini-antipattern: Immediate Reply Requested

When I reach the base of the slope, there's no sign of my snowboarding buddy. He may have waited for me patiently somewhere, or already started back up. I could wait at the base to see if he shows up, but if he's already on the lift, I'll miss the opportunity of another ride. All slopes on this side of the mountain converge in this spot, and at the head of the lift line, there's a small whiteboard. I decide to scribble a message for him. The next time he gets on the lift, he will be sure to see it, and we'll hook back up. In the meantime, the powder is getting deeper, and I'm ready for the next ride.

If you're blocked waiting for a reply, you're stuck. You can't move on or coordinate new activities. As a result, you may miss out on opportunities. In other words, waiting creates an opportunity cost. To work (and play) efficiently, you'd like to rendezvous when it's most convenient. Asynchronous messaging frees you from waiting and lets you get in a few more runs.

Excessive coupling is the enemy of asynchronous messaging. Indeed, it flies in the face of a powerful aspect of asynchronous messaging—loose coupling. If message producers have intimate knowledge of the consumers with which they communicate, then assumptions are inevitably made. In particular, a producer may rely on a particular consumer's identity, location on the network, and possible

connection times. Consequently, the system can't grow and shrink dynamically. In other words, producers are susceptible to the changes of the consumers on which they rely. If, for example, a consumer on which a producer relies disconnects or moves to a new host on the network, then the producer may end up waiting indefinitely for the consumer to reconnect.

That said, JMS does support a synchronous request/reply style of communication. Message producers can send a request in the form of a message to an outbound destination (topic or queue). When a message consumer receives the message—either synchronously or asynchronously—it can then reply by sending a message to a predetermined inbound destination. The two participants may agree on well-known destinations ahead of time. Alternatively, the producer can dynamically create a temporary inbound topic and assign it to the request message's `JMSReplyTo` property. Figure 6.6 illustrates a synchronous request/reply conversation.

It's true that the producer and consumer are decoupled in the sense that they are unaware of each other's identity or location, but an implied association exists. Indeed, their life cycles are coupled. After publishing the request message, the `TopicSubscriber.receive()` method invoked by the producer blocks until a consumer sends a reply message. The producer must wait on the line until a consumer is connected. Even then, the producer is at the mercy of the consumer's duty cycle. If the consumer is never able to connect and send a reply, the producer will continue to block, forever waiting for a reply. To avoid freezing the producer indefinitely, use the `receive(long timeout)` or `receiveNoWait()` method. These methods will break the producer free of the synchronous bonds before it's too late.

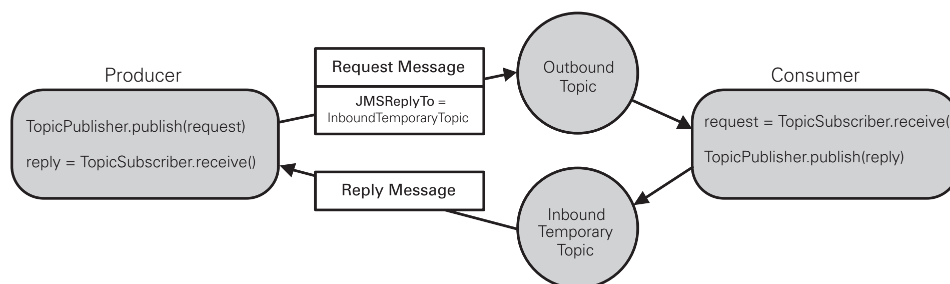

Figure 6.6 Although JMS does support synchronous request/reply messaging, if used extensively this messaging tends to create undesirable coupling between the message producer and consumer. From the producer's perspective, the round trip is synchronous; it blocks waiting for a consumer to reply. If a consumer isn't able to reply, the producer may block indefinitely.

The sequence of steps required by a message producer to engage in a request/reply conversation can be executed in one fell swoop using the `javax.jms.TopicRequestor` or `javax.jms.QueueRequestor` utility classes. These classes define a `request()` method that encapsulates the lock-step process of sending a request message and blocking until a reply message is received. If not executed in a separate thread, invoking the blocking `request()` method from a message producer will block the calling thread until a reply has been received. This risk alone may warrant a move from convenience to safety by using a variant of the `receive()` method directly.

In general, asynchronous messaging is utilized best for a fire-and-forget style of communication. When a request/reply conversation is needed, the power of asynchronous communication is diminished, and the scales start to tip back in favor of RPC communication. Therefore, before using JMS, carefully consider if your system has the potential to benefit from asynchronous messaging. Indeed, asynchronous communication should sometimes be eschewed in favor of synchronous communication. If specific use cases require an immediate reply in response to a request, consider using synchronous protocols such as Java RMI or SOAP. Although JMS may afford better reliability through guaranteed message delivery, it may also be overkill for the task at hand. Remember that it's just another tool whose value is derived from the circumstances in which you use it.

Speaking of new tools, we've now learned enough about JMS to start cracking message-driven beans. It's been a long journey to this point, but you won't want to miss what's around the next corner.

6.8 Using message-driven beans (MDBs)

Let's pick up where we left off on our order processing system. We built the `OrderRequestSender`, picked the best message type, and then dialed in the right amount of reliability. It's high time we designed the consumer of `OrderRequest` messages. We could choose to create the consumer as a stand-alone JMS client. However, we want to scale our application to handle many `OrderRequest` messages concurrently. So, in the spirit of this book, and because we already have an investment in an EJB server, we'll design the message consumer as a message-driven bean (MDB).

6.8.1 Pooling with MDBs

MDBs were introduced in EJB 2.0 as server-side components capable of concurrently processing asynchronous messages. In contrast, while session and entity beans can

produce asynchronous messages, they can only consume messages synchronously. An MDB's life cycle is similar to that of a stateless session bean. Instances of a particular MDB are identical. They hold no state that makes them distinguishable. Therefore, MDB instances can be pooled. Message producers unknowingly interact with an MDB instance by sending a message to a topic or queue subscribed to by the MDB. Figure 6.7 illustrates the advantage of pooling MDB instances.

An MDB is equipped to handle JMS messages by implementing the `javax.jms.MessageListener` interface. This interface defines a single `onMessage()` method. When a message is delivered to the topic or queue, an MDB instance is plucked from the pool and its `onMessage()` callback method is invoked with the message. If more messages are delivered to the topic or queue before the instance's `onMessage()` method returns, then other instances are called into action to handle the messages. When an instance's `onMessage()` method returns, the instance is returned to the pool to await the next message.

To summarize, the use of MDBs offers a distinct advantage over managing multiple JMS clients. Instead of trying to load balance messages between stand-alone JMS clients for optimal throughput, the container effectively distributes the load using a pool of available MDB instances. So, let's take advantage of an MDB to handle requests for orders.

6.8.2 Building the OrderRequest consumer

Unlike a session or entity bean, an MDB does not have a home or remote interface. In other words, an MDB does not define business methods accessible directly from remote clients. Instead, it simply defines the `onMessage()` method that

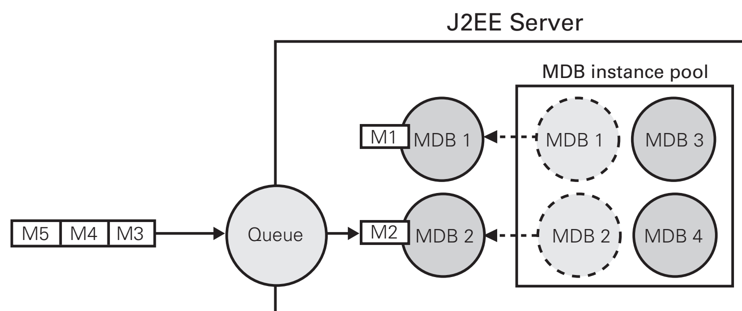

Figure 6.7 MDB instances are pooled in preparation for handling incoming messages. In this example, two MDB instances have been enlisted from the pool and are now busily handling messages. When the next message arrives (M3), if MDB1 and MDB2 are still busy, then an idle MDB instance (MDB3 or MDB4) will be plucked from the pool to handle the message. In this way, multiple messages can be consumed concurrently for better performance.

contains the business logic for handling a message. The business logic encapsulated in the `onMessage()` method is executed in response to asynchronously receiving a message. Listing 6.4 shows how an `OrderRequest` is consumed by our MDB.

Listing 6.4 A message-driven bean that handles OrderRequest messages

```
public class OrderRequestReceiverMDB
  implements javax.ejb.MessageDrivenBean,
             javax.jms.MessageListener {

  private MessageDrivenContext ctx;

  public void setMessageDrivenContext(MessageDrivenContext ctx) {
    this.ctx = ctx;
  }

  public void ejbCreate() {}

  public void onMessage(Message message) {

    if (message instanceof MapMessage) {

      MapMessage mapMessage = (MapMessage)message;

      try {

        String orderId = mapMessage.getString("Order ID");        <-- Crack the
        String productId = mapMessage.getString("Product ID");        message
        int quantity = mapMessage.getInt("Quantity");
        double price = mapMessage.getDouble("Price");

        OrderRequest orderRequest =      Create an order value object
          new OrderRequest(orderId, productId, quantity, price);

        recordOrder(orderRequest);     Store order in order database

        OrderStatus status = fulfillOrder(orderRequest);   <-- Send order to
                                                               fulfillment system
        notifyOrderStatusSubscribers(status);   <-- Broadcast
                                                    notification to
      } catch (JMSException jmse) {                 OrderStatus
        jmse.printStackTrace();                     subscribers
      }

    } else {
      System.err.println("OrderRequest must be a MapMessage type!");
    }
  }

  public void ejbRemove() {}
}
```

In addition to the performance benefits gained by MDB pooling, this type of bean is much easier to develop than a stand-alone JMS consumer. Notice that we didn't

have to write all the boilerplate setup code needed to connect to the JMS server through JNDI and subscribe to a queue as we did in building the `OrderRequest-Sender`. The EJB container takes care of all that plumbing, based on the contents of deployment descriptors. Listing 6.5 shows the standard XML deployment descriptor (`ejb-jar.xml`) relevant to our MDB example.

Listing 6.5 XML deployment descriptor for the OrderRequestReceiverMDB

```
<message-driven>
  <ejb-name>orderRequestReceiverMDB</ejb-name>
  <ejb-class>com.bitterejb.order.ejb.OrderRequestReceiverMDB</ejb-class>
  <transaction-type>Container</transaction-type>
  <message-driven-destination>
    <destination-type>javax.jms.Queue</destination-type>
  </message-driven-destination>
</message-driven>
```

By declaring the message destination as a queue using the `<destination-type>` tag in the deployment descriptor, the code in the MDB itself becomes oblivious to a message's point of origin (topic or queue). That is, this same MDB could be configured to subscribe to a topic without changing any code. This means that the business logic this MDB encapsulates easily can be reused across messaging models. That's a markedly easier solution than developing a new JMS consumer client.

The actual JNDI name of the queue to which our `OrderRequestSender` is sending `OrderRequest` messages is declared in a vendor-specific XML deployment descriptor. The EJB container automatically subscribes MDB instances to this message queue when the instances are created. This same vendor-specific deployment descriptor may also declare the initial and maximum size of the MDB instance pool. Sizing the pool allows us to easily throttle the message throughput, based on expected message volumes.

We haven't yet discussed the actual business logic involved in handling a message. Once a message arrives, the business logic can do whatever is necessary to fulfill the order. That process isn't all that relevant to the antipatterns in this chapter. We could imagine the business logic using the J2EE Connector Architecture (JCA) to communicate with the legacy order fulfillment application, for example. The logic might even collaborate with other EJB components—session and entity beans—in a more complex workflow. For example, to create an easily supported order status notification, a subscriber of `OrderStatus` messages could use JavaMail to send an email to the person who placed the order. The email could contain an

indication of the order's status. In any event, this arbitrary business logic should be decoupled from JMS as we'll see in our next antipattern.

6.9 Antipattern: Monolithic Consumer

At this point, we might be tempted to walk away from the MDB that consumes `OrderRequest` messages, satisfied that it dutifully handles messages. If we did, we'd miss a golden opportunity to improve the design. Before wandering off, let's take a minute to reflect on ways to keep the code clean and the design pristine. A small investment to pay off design debt now will help prevent interest payments from accumulating down the road.

As it stands, our MDB's `onMessage()` method creates the unfortunate side effect of undesirable coupling. It's not a particularly long method, but after cracking the message, it inlines a sequence of method calls. As a result, the business logic `onMessage()` encapsulates—the real meat of the order fulfillment process—is intimately tied to an asynchronous messaging infrastructure. No clean separation of concerns exists between the communication mechanism used to interact with the business logic and the logic itself. Left untouched, this tightly wrapped ball of code is, and forever will be, a JMS consumer. This coupling has a severe consequence. The only way to execute the business logic is by publishing a JMS message for consumption by the MDB. However, we'd like to reuse this business logic in the absence of JMS. Without overengineering the design, what's the simplest thing we can do now to head off potentially painting ourselves in a corner later? It might surprise you to hear that a test is in order.

6.9.1 Listening to the test

If we had attempted to write a test for the business logic before digging into the implementation of the MDB, the pain that would be caused by undesirable coupling would have been evident. By paying attention to the test, we would have uncovered a better design opportunity much sooner. Indeed, when writing a test is painful, we can usually assume that something's wrong.

Consider how difficult it is to write a test for the business logic through the MDB's `onMessage()` method. To do so, we would have to follow this procedure:

1. Write a full-blown JMS message producer similar to the `OrderRequestSender`.
2. Register the message producer as a subscriber of `OrderStatus` messages.
3. Create and publish an `OrderRequest` message.

4. Wait for the asynchronous `OrderStatus` message.
5. Validate that the resulting `OrderStatus` message contains the expected status.
6. Query the order database to ensure the order was properly recorded.

That's a lot of work! And most of our effort is geared toward appeasing the JMS infrastructure. While this approach might create a good integration test, we're once again forced to use JMS. We really just want to know if the business logic works. However, given the current design, testing the business logic independent of JMS proves difficult because the test doesn't distinguish between the two. The test forces us to separate JMS from the business logic by refactoring the MDB to delegate its work to a testable component.

6.9.2 Solution: Delegate to modular components

Modular designs that use cohesive and loosely coupled components are generally easier to test. Imagine how the design improves if we look at it first in light of a test. Without worrying about how a JMS message arrives, the test is simply concerned with validating the business logic. After all, the test really only cares about the guts of the `onMessage()` method. This tells us that inside the `onMessage()` method is a unique component just waiting to be let free. So, let's refactor the logic contained in the `onMessage()` method into a separate component, called the `OrderRequestHandler` class. Listing 6.6 shows the updated `onMessage()` method.

Listing 6.6 Refactoring onMessage() to delegate to an order request handler

```
public void onMessage(Message message) {

  if (message instanceof MapMessage) {

    MapMessage mapMessage = (MapMessage)message;

    try {

      String orderId = mapMessage.getString("Order ID");     <-- Crack the message
      String productId = mapMessage.getString("Product ID");
      int quantity = mapMessage.getInt("Quantity");
      double price = mapMessage.getDouble("Price");

      OrderRequest orderRequest =     Create an order value object
        new OrderRequest(orderId, productId, quantity, price);

      OrderRequestHandler handler = new OrderRequestHandler();
      handler.handle(orderRequest);  <-- Delegate to encapsulated business logic

    } catch (JMSException jmse) {
      jmse.printStackTrace();
    }
```

```
    } else {
      System.err.println("OrderRequest must be a MapMessage type!");
    }
  }
```

If we extract the inlined code into the `OrderRequestHandler` class, then the code's business logic is decoupled from asynchronous messages. The `OrderRequestHandler` class is solely responsible for the order fulfillment process: recording an order, submitting the order to the legacy order fulfillment application, and notifying order status subscribers. In addition, the business logic easily can be tested outside the MDB container, completely separate from JMS technology. Once we've gained confidence that the handler works as expected, it can be used by many clients. Local clients within the same JVM, for example, can submit an order request simply by invoking a method directly on an instance of the class. We've successfully put JMS in its rightful place—as a glue technology.

Now, imagine we want to expose the logic of the `OrderRequestHandler` to remote clients. Using the Session Façade design pattern, a session bean can service remote synchronous clients by delegating directly to an `OrderRequestHandler` instance. Moving a step further, we can expose the same business logic to remote asynchronous clients by creating an MDB that either delegates directly to an `OrderRequestHandler` instance or indirectly through the Session Façade. Figure 6.8 illustrates the multiple communication paths used to access the business logic that processes an order request.

Notice that by decorating a modular component in a layered fashion, we've effectively created two communication paths: one synchronous and the other asynchronous. Moreover, no code duplication exists. We need only to change the business logic in one place to affect the synchronous and asynchronous clients uniformly. That is, the business logic can be varied, independent of the client types that may chose to use it.

The moral of the story is to remember that an MDB is simply a conduit between JMS clients and business logic. As such, it should be kept as thin as possible. After receiving a message, and possibly converting it into a lightweight business object, the MDB should delegate to other components that act on the contents of the message. And we discovered all that by starting from a testing perspective. Go figure!

Now for a little fun with a familiar, albeit tiresome, game: hot potato.

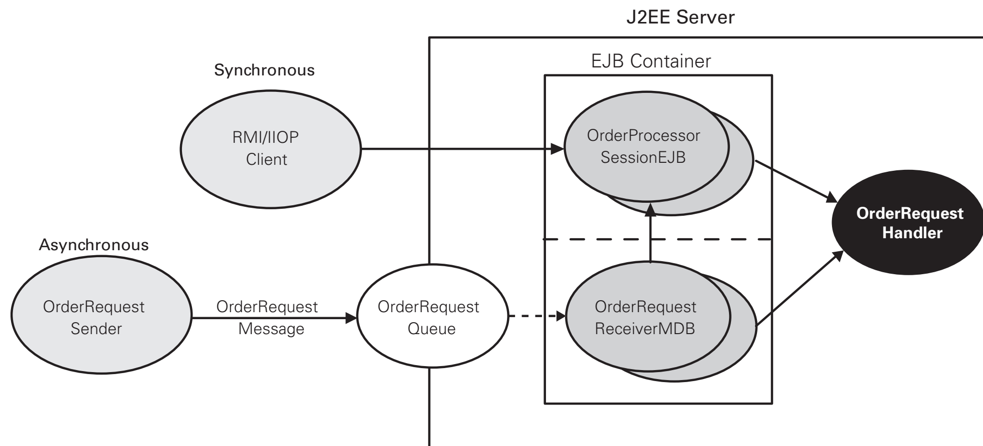

Figure 6.8 Layering is a design technique used to build loosely coupled systems capable of servicing disparate clients. By decorating a simple component that encapsulates business logic, enabling technologies can serve as thin communication adapters. This maintains a clean separation of concerns, improves testability, and allows the business logic to be changed in one location to affect all clients.

6.10 Antipattern: Hot Potato

When a JMS server doesn't receive an acknowledgment for a message sent to a consumer, the server's only recourse is to attempt to redeliver the message. This sets the stage for a potentially wicked game of hot potato. The game goes something like this:

- The JMS server sends a message to a message-driven bean.
- The MDB raises an exception or rolls back its transaction.
- As a result, the MDB container doesn't acknowledge the message.
- So the server attempts to redeliver the message.
- The message again causes the MDB to raise an exception or roll back its transaction.
- Once again, the server does not receive an acknowledgment.
- Rinse and repeat.

The JMS server and the MDB container continue to toss the message back and forth, neither one wanting to get caught with the message when its timeout expires (if ever). Round and round they go; where they stop, nobody knows.

This begs a question: what might cause a message to go unacknowledged by an MDB. As first-class EJB components, MDBs are transaction-aware in their own right. Often we want to execute the business logic, triggered by the arrival of a message, as an atomic business process. Let's look at our example again. The arrival of an `OrderRequest` message kicks off a sequence of actions: updating a database, accessing an external system, and sending notification. If any step fails, we want the entire business process to be rolled back.

Using MDBs greatly simplifies handling messages within a transaction. MDBs can manage their own transactions or let the container manage transactions on their behalf. If CMT are used, then message consumption is included in the same transaction as the message handling logic. Only if the transaction succeeds will the message be acknowledged. It's an all-or-nothing proposition. If either of the following occurs while executing the `onMessage()` method, the transaction will be rolled back and the JMS server will attempt to redeliver the message:

- A system exception (e.g., `EJBException`) is thrown from the `onMessage()` method.
- The `MessageDrivenContext.setRollbackOnly()` method is invoked.

Because the message acknowledgment is tied directly to the success of a transaction, MDBs that use CMTs are easy candidates for a game of hot potato. Rolling back the transaction because business logic fails causes the message to never be acknowledged. Figure 6.9 depicts the hot potato game played between the JMS server and an MDB instance (note that the steps are presented in clockwise sequence).

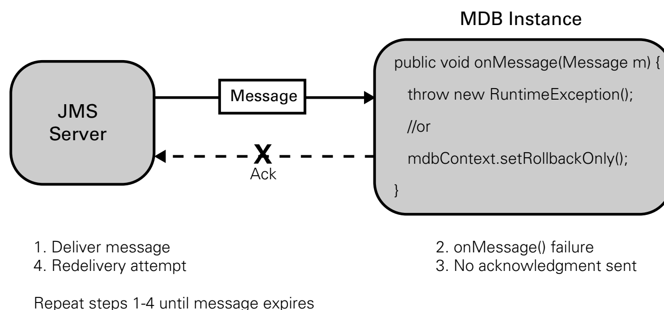

Figure 6.9 If an MDB instance continuously throws a system exception from its `onMessage()` method, or rolls back the transaction, then the MDB container doesn't acknowledge receipt of the message. Consequently, the JMS server assumes the message wasn't successfully delivered. In an effort to set things straight, the server will attempt to redeliver the message. The message becomes like a hot potato tossed back and forth between the JMS server and MDB instances.

MDBs that choose to manage their own transactions are slightly less likely to get into a game of hot potato, though they are not immune to it. When BMT are used, the consumption of a message is not included in the same transaction as the message handling logic. Messages are acknowledged, regardless of whether the transaction is committed or rolled back. To force the JMS server to redeliver a message if the transaction is rolled back, a system exception can be thrown from the `onMessage()` method.

Although hot potato appears to be good, clean fun, it's not a game with many prizes. The JMS server on one side is frazzled, juggling all outstanding messages. On the other side, the MDB instances are thrashed, trying to deal with recurring messages they can't handle. Until an acknowledgment is made, the games will continue.

6.10.1 Solution: Acknowledge the message, not its result

The easiest way to avoid a game of hot potato is to acknowledge the successful receipt of the message, not whether the resulting business logic was successful. The JMS server can't do anything about the latter, so don't put the server in a position to let you down.

Toward this end, you don't want to throw system exceptions in response to business logic errors. System exceptions should be raised only in response to genuine system (or container) failures. Because application exceptions cannot be thrown from the `onMessage()` method, it's best to log any business logic errors and return gracefully from the `onMessage()` method. This lets the JMS server know that the consumer got the message, which is all the server really cares about anyway. A variation on this theme is to send an error-related message to a special error queue. To handle unexpected error conditions intelligently, exception-handling consumers can subscribe to this queue.

Be mindful of the repercussions of rolling back an MDB transaction by invoking the `MessageDrivenContext.setRollbackOnly()` method. It, too, will force the JMS server to attempt redelivery. Ask yourself whether the next MDB instance chosen to handle the message will be able to execute the business logic successfully, or if it will suffer the same fate. If the problem that triggers the rollback is unrecoverable, then the next MDB instance to receive the redelivered message will likely encounter the same problem. Incoming hot potato! If it's possible that the next MDB instance to receive the redelivered message will be able to recover from the error, then rolling back the transaction may be appropriate.

Some JMS providers automatically support the use of a Dead Message Queue (DMQ). If, for example, an attempt to deliver a message is unsuccessful after a

preconfigured number of redelivery tries, the message is automatically redirected to the DMQ. Our application is then responsible for monitoring this queue and taking appropriate action when a doomed message arrives. JMS providers may also support a configurable redelivery delay whereby the JMS server waits a predefined amount of time before attempting redelivery. Understanding the conditions under which a message will be redelivered helps minimize the chance of creating a berserk message. Equally troublesome is the subject of our next antipattern: a message that takes a while to chew on.

6.11 Antipattern: Slow Eater

An MDB can chew on only one message at a time. Until its `onMessage()` method returns (swallows what's in its mouth), an MDB cannot be used to handle other messages. That is, an MDB instance is not re-entrant. If another message is delivered to the MDB container before a busy instance's `onMessage()` method returns, then the container will pluck another MDB instance from the instance pool to handle the new message. This is true for both publish/subscribe and point-to-point messaging. Messages published to a topic are delivered to one MDB instance in every MDB container registering interest in the topic. Messages sent to a queue are delivered to one MDB instance in exactly one MDB container registering interest in the queue. In either case, an MDB instance can only handle messages serially.

When the `onMessage()` method takes a relatively long time to handle a message, more and more instances in an MDB instance pool will be needed to handle high message volumes. Messages will start to back up any time the average arrival time of messages is greater than the average time to consume each message, thereby creating a bottleneck that restricts message throughput. In general, anytime the ratio of message production to consumption is high, message throughput will suffer.

6.11.1 Solution: Eat faster, if you can

If high message volumes are expected, it's wise to keep the `onMessage()` method as fast as possible. An MDB with a short and sweet `onMessage()` method can achieve higher levels of message throughput with a smaller number of MDB instances in the pool. Because every MDB instance in the pool is stateless and identical, any idle instance can handle an incoming message. As soon as the `onMessage()` method returns, it can immediately handle another message. That's all well and good, except for one minor detail: MDBs are usually tasked with time-consuming work on which message producers can't afford to block waiting.

Indeed, if faced with a quick and dirty job, we might just use a synchronous method call.

We should strive to keep the code paths invoked by the `onMessage()` method optimized as necessary to support a tolerable message throughput. Delegating to modular components that perform the actual message handling makes it much easier to write isolated performance tests that continually measure the response time of the logic encapsulated in `onMessage()`.

When we dig a bit deeper in our toolbox, we can find a few performance tricks for helping slow eaters. If we spend time reading our JMS vendor's documentation, we can get a feel for the possible tools we could use. For example, some vendors have support for throttling, which effectively slows down producers if consumers are lagging behind. We might as well use what's already available to our advantage—we paid for it! Once our MDBs are efficiently consuming messages, we need to make sure they aren't eating more than their fair portion. This is the subject of our next antipattern.

6.12 Antipattern: Eavesdropping

As messages fly around in a message-based system, consumers must pick and choose the messages they'll consume. The potential for information overload increases with each new message producer participating in the system. A consumer eating a relatively small portion of messages today may be faced with significantly larger portions tomorrow.

Take, for example, a publish/subscribe scenario with subscribers eavesdropping on a high-traffic topic. As more and more messages are sent to that topic, the subscribers may experience an abysmal signal-to-noise ratio. Similarly, in a point-to-point scenario, receivers consuming messages from a high-volume queue may be burdened with handling low priority work. Developing custom message filtering logic in each message consumer is both time consuming and prone to error. It also makes it difficult to uniformly improve message filtering logic and performance.

As a work-around, multiple destinations (topics and queues) can be set up to partition messages according to their intended use. In other words, we can break up coarse destinations into multiple fine-grained destinations for more selective listening. For example, we could configure two queues for our order processing system: one for standard orders and the other for premium orders. However, the process of setting up special interest destinations starts to fall apart at some point, and ultimately leads to a proliferation of topics and queues, which must be administered and managed. This work-around also places the burden on message

producers to send only relevant messages to each destination. Message consumers are in turn burdened with registering interest in only the appropriate destinations necessary to get all the information they need.

6.12.1 Solution: Use message selectors

Message selectors are one way for message consumers to easily tune out messages they don't need or want to hear. Each message consumer can be configured with a unique message selector, much as we use mail and news filters to receive only information we're interested in reading.

A message's filtering can be based only on its headers and properties, not on the payload it carries. The SQL-92 conditional expression syntax—which makes up SQL `WHERE` clauses—is used to declare the filtering criteria. The JMS provider filters messages, so the process is automatic from the consumer's perspective.

The use of message selectors is one way to easily design *queue specialization* into a message-based system. Referring back to our example order processing system, we can see it may make good business sense to handle premium orders differently than standard orders. Rather than creating two different queues—one for standard orders and another for premium orders—a single queue could be used by all message consumers interested in orders. We could then create two different types of `OrderRequest` handlers modeled as MDBs: a standard order handler and a premium order handler.

Assuming an `OrderRequest` message contained the total price of the order as a message property, the standard order handler would be created with a message selector on that property so the standard order handler would only see orders on the queue with a total price up to $1,000. The premium order handler's message selector would restrict its view of the queue to only those orders that exceeded $1,000. We could then vary the size of the respective MDB instance pools independently. For example, the premium order handler's pool might be increased to improve the throughput of fulfilling premium orders. Figure 6.10 illustrates the flow of messages when message selectors are used to handle premium orders differently than standard orders.

Using the same example, each subscriber of `OrderStatus` messages could use message selectors to select the messages they receive. Messages that didn't match the selection criteria for a given subscriber wouldn't be delivered to that subscriber. Each subscriber would pick up a good, clean signal without any of the noise.

How and where messages are filtered is an implementation detail of the JMS provider. Any specific JMS vendor's implementation may apply the message selection logic in the server-side message router or in the client-side consumer's JVM.

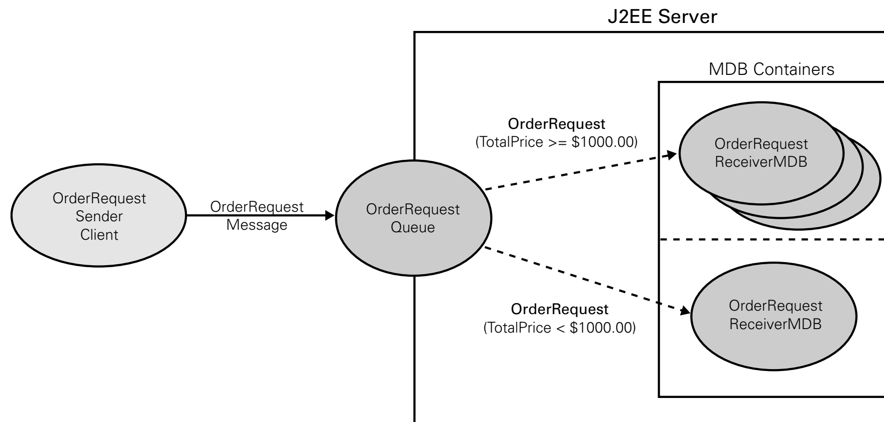

Figure 6.10 Message selectors can be used to improve signal quality by filtering messages based on header and property values. Rather than having to eavesdrop on all messages for fear of missing an important message, consumers can be created with unique message selectors. This allows the QoS to be varied according to the business value of the message being handled by a pool of MDB instances.

Depending on the implementation, message selectors may create a measurable drag on performance. In general, however, filtering on the server side is less expensive and may actually improve performance by minimizing network traffic for unwanted messages. In any event, it pays to have performance benchmarks and automated tests that can continually check whether performance is going off the rails. Running these tests can help determine objectively how the performance of message selectors stacks up against custom message filtering logic in each message consumer.

At the end of the day, anything that can be done with message selectors can also be achieved using multiple destinations. The use of message selectors over multiple destinations ultimately boils down to striking a balance between resource management and performance.

Now, let's apply what we've learned by setting up a message selector.

6.12.2 Declaring message selectors

Message selectors are declared for each message consumer when the consumer is created. With MDBs, no extra coding is necessary. The message selection criteria simply are declared in the XML deployment descriptor. The MDB container creates all MDB instances in the pool with the same message selection criteria.

Assuming an `OrderRequest` message contains a property defining the total cost of the order, adding the following XML snippet to the standard XML deployment descriptor (`ejb-jar.xml`) causes only those orders exceeding $1,000 to be delivered to the MDB instances managed by this container:

```
<message-selector>
  <![CDATA[TotalPrice > 1000.00]]>
</message-selector>
```

That's all there is to it! Of course message selectors can be arbitrarily complex, depending on the number of message properties and the conditional logic involved. Using a CDATA section around the message selector text means the text won't be subjected to XML parsing. Therefore, we won't need to escape all the logical operators to appease the XML parser.

6.12.3 Going beyond message selectors

Many JMS vendors have value-add features for going beyond message selectors. If you choose to take these paths, just remember that you're straying away from portability. Fortunately, with MDBs we can take advantage of these extensions at deployment time. That is, we usually won't have to change any code to put these extensions in or take them out. The deployment descriptor conveniently includes all configuration details.

As we learned, the JMS specification restricts message selectors to filtering, based on a message's headers and properties. In other words, we can't filter a message by inspecting its payload. In response, many vendors have added proprietary extensions to the message selector syntax to support content-based routing. For example, many vendors can use XPath to filter either proprietary XML message types or a `TextMessage` containing XML.

Another proprietary extension for message filtering is the use of wildcard topic names. By using a dot notation when naming topics, we can set up a hierarchy of information. Consumers can then easily subscribe to groups of messages. Take, for example, a financial application that sends stock quote updates to either the STOCKS.NYSE.IBM or STOCKS.NASDAQ.SUNW topics. If consumers want to subscribe to all NASDAQ prices, they simply register interest in the STOCKS.NASDAQ.* topic. Alternatively, they can listen to a specific stock by registering interest in the STOCKS.NASDAQ.SUNW topic, for example.

Up to this point we've covered many antipatterns related to the design of applications using JMS and MDBs. As a parting shot, let's look at a final antipattern, one that usually reveals itself at the end of your development process.

6.13 Antipattern: Performance Afterthoughts

We've touched on performance in many ways throughout this chapter. However, just because some antipatterns had performance side effects doesn't mean we should focus on performance too early. Premature optimization is speculative at best. On the other hand, casting performance absolutely to the wind is a recipe for disaster. Every design decision we make, including the selection of a JMS vendor, ultimately has the potential to affect performance. Our path deviates away from a successful deployment each time a decision is made without objectively measuring its performance implications.

Although the JMS specification defines two messaging models and various QoS features that may influence performance, the specification does not address the performance implications of these decisions. This lapse gives JMS vendors a lot of room to compete and tailor their product offerings to shine in certain deployment scenarios. Indeed, vendors have different strengths and weaknesses. It's entirely possible that a vendor's implementation designed specifically to excel in certain scenarios may fall down in other scenarios. And then there's the code we write!

Simply measuring the time it takes a JMS server to transport a single message from a producer to a consumer doesn't give us a full picture of performance. The performance of an individual message's delivery cycle may be markedly different when the JMS server is under load—for example, delivering fat, persistent messages to multiple consumers. Without a rough measure of success based on realistic usage patterns, the measurements are useless.

6.13.1 Solution: Measure early and often

Our defense against performance-related antipatterns is a solid foundation of automated tests that validate our application's performance requirements. When faced with decisions that may affect performance, these automated tests can be rerun to objectively measure any impact. As our application's design takes shape, we'll get confidence by continually running its performance tests to measure progress. If a change improves performance, we can raise the bar by modifying the tests to use the new benchmark. If performance degrades, we can undo whatever changes were made and try again.

Automated performance tests can also be used as a yardstick when evaluating different JMS implementations in terms of their performance. Before making an investment in a specific JMS vendor's implementation, we should create a few benchmarks. We should start by using a simple driver that can be configured easily to produce/consume arbitrary numbers of messages and report performance

metrics for each action. Because the performance of messaging models will vary between vendors, we need to make sure the test is indicative of our application's needs. Then we can proceed to write automated tests that use the test rig to simulate a representative use case and automatically check that performance is within tolerable limits.

Given the variation in vendor implementations and design decisions, performance cannot be treated as an afterthought without facing potentially dire consequences. Writing performance tests early and running them often illuminates unforeseen bottlenecks and reduces the effects of downstream thrash tuning. The following list represents factors that should be considered when writing and running performance tests:

- **Message throughput** The number of messages a JMS server is able to process over a given period of time can be a telling metric. It quantifies the degree to which an application can scale to handle more concurrent users and a higher message volume.
- **Message density** The average size of a message impacts the performance and scalability of an application. Smaller messages use less network bandwidth and are generally easier for the JMS server to manage.
- **Message delivery mode** Persistent messages must first be stored in nonvolatile memory before being processed by the JMS server. Effective tests must produce messages representative of the production system to get an accurate picture of production performance.
- **Test under realistic load scenarios** Load testing with multiple users often illuminates bottlenecks that aren't evident in a single producer/consumer scenario. Write tests that measure the message throughput capable under average and peak concurrent user loads. Consider both the ratio of users to actual JMS connections, and the resources required.
- **Production rate versus consumption rate** If the rate at which messages are produced exceeds the rate at which messages are consumed, the JMS server must somehow manage the backlog of messages. Watch for any significant disparities between the send rate and the receive rate.
- **Go the distance** Endurance testing over an extended period of time can identify problematic trends such as excessive resource usage or decreased message throughput. Running performance tests overnight, for example, may highlight problems that may be encountered when the system goes live.

- **Know your options** JMS vendors generally support proprietary runtime parameters and deployment options for tuning the performance and scalability of their product offering. Know what options are available out-of-the-box so that the JMS provider can be configured to yield optimal performance and scalability relative to your application.
- **Monitor metrics** Some JMS vendors include an administrative console for monitoring internal JMS metrics such as queue sizes and message throughput. Monitor these metrics to gain insight into the usage patterns of your application. If an API is available for obtaining these metrics programmatically, such as through JMX, write tests to check continually whether the metrics are within tolerable ranges.
- **Chart metrics** Simple charts serve as early warning systems against undesirable performance trends. For example, plotting the number of messages processed as a function of time will help pinpoint where message throughput plateaus. When using point-to-point messaging, plotting the queue size over time will clearly indicate when messages are being backlogged.

Automated performance tests are invaluable for their ability to keep all these considerations continually in check. Don't settle for having to manually recheck performance every time you make a change. Invest early in tests that check their own results and run them often to gain confidence. You'll be glad you did!

6.14 Summary: Getting the message

JMS is easy to use and extremely powerful, yet subtle implications must be carefully considered when using JMS to build message-based applications. In this chapter, we discussed several common pitfalls as we developed an example order processing system glued together with asynchronous messaging. In many cases, we were able to side-step problems by applying relatively simple refactorings. In other instances, we avoided potential problems altogether by understanding the consequences of design decisions and planning accordingly.

Although many antipatterns discussed in this chapter are applicable to JMS in general, we specifically put MDBs under the microscope. As first-class EJB components making their debut in EJB 2.0, MDBs enable asynchronous access to server-side business logic and resources. Moreover, they simplify the development of message consumers that can scale to handle high-volume message traffic. Nevertheless, designing and configuring MDBs to meet the challenges of today's business needs

requires attention to detail. As we watch MDBs mature to include support for other messaging technologies, we'll likely bear witness to new MDB antipatterns.

Many antipatterns we discussed in this chapter are related to performance. JMS is used primarily as a glue technology to integrate multiple applications through the exchange of portable messages. As such, the quality of a message-based application is measured according to the message throughput it can reliably scale to handle. The important lesson to be learned from these antipatterns is to size and test your application early and often to ensure a successful deployment.

6.15 Antipatterns in this chapter

This section covers the Fat Messages, Skinny Messages, XML as the Silver Bullet, Packrat, Immediate Reply Requested, Monolithic Consumer, Hot Potato, Slow Eater, Eavesdropping, and Performance Afterthoughts antipatterns.

FAT MESSAGES

DESCRIPTION
Using the same message type for all situations and not designing messages for their intended consumers leads to bloated messages.

MOST FREQUENT SCALE
Application

REFACTORED SOLUTION NAME
Message dieting

REFACTORED SOLUTION TYPE
Software

REFACTORED SOLUTION DESCRIPTION
Design messages to carry just enough information to allow their consumers to autonomously handle the messages. Send references to data when sending the data itself is size prohibitive.

ANECDOTAL EVIDENCE
"This message contains a plethora of information, just in case consumers need it."

SYMPTOMS, CONSEQUENCES
The messaging pipes are clogged with fat messages, and message throughput suffers.

SKINNY MESSAGES

DESCRIPTION

Messages that don’t contain enough information burden their consumers with making extra remote calls to get more information.

MOST FREQUENT SCALE

Application

REFACTORED SOLUTION NAME

Put some meat on the bones.

REFACTORED SOLUTION TYPE

Software

REFACTORED SOLUTION DESCRIPTION

Err on the side of sending a bit too much information. Add state information to references to let consumers decide when and if to load referenced data.

ANECDOTAL EVIDENCE

“Why is the application spending all of its time I/O blocked?”

SYMPTOMS, CONSEQUENCES

Asynchronous communication breaks down into synchronous communication to clarify the intent of messages. Misuse of references causes contention of a shared resource and ends up being slower than a fatter message.

XML AS THE SILVER BULLET

DESCRIPTION
Filling messages with XML by default in the name of flexibility and portability

MOST FREQUENT SCALE
Application

REFACTORED SOLUTION NAME
Use XML on the edges.

REFACTORED SOLUTION TYPE
Software

REFACTORED SOLUTION DESCRIPTION
Use XML messages to communicate with applications beyond your control. The `MapMessage` has similar flexibility and portability when communicating with application within your control.

ANECDOTAL EVIDENCE
"XML is the only way to make this message portable." "All the cool developers are using XML."

SYMPTOMS, CONSEQUENCES
Messages containing XML may incur unnecessary overhead that limits message throughput. Message handling logic isn't explicit or type-safe.

PACKRAT

DESCRIPTION
Storing all messages, regardless of whether delivery must be guaranteed

MOST FREQUENT SCALE
Application

REFACTORED SOLUTION NAME
Save only the important messages.

REFACTORED SOLUTION TYPE
Software

REFACTORED SOLUTION DESCRIPTION
Consider the ramifications of losing a message before deciding to guarantee its delivery.

ANECDOTAL EVIDENCE
"Let's be safe and store all messages by default."

SYMPTOMS, CONSEQUENCES
Storing all messages limits message throughput and unnecessarily burdens the JMS server.

IMMEDIATE REPLY REQUESTED

DESCRIPTION
Using JMS for a synchronous request/reply style of communication

MOST FREQUENT SCALE
Application

REFACTORED SOLUTION NAME
Use synchronous communication technologies where appropriate.

REFACTORED SOLUTION TYPE
Software

REFACTORED SOLUTION DESCRIPTION
If a request/reply style of communication is needed, consider using Java RMI or SOAP.

ANECDOTAL EVIDENCE
"How can I return a result once the consumer handles the message?"

SYMPTOMS, CONSEQUENCES
Undesirable coupling between producers and consumers, negating the benefits of asynchronous messaging

MONOLITHIC CONSUMER

DESCRIPTION

Inlining business logic in the class that consumes a message

MOST FREQUENT SCALE

Application

REFACTORED SOLUTION NAME

Delegate.

REFACTORED SOLUTION TYPE

Software

REFACTORED SOLUTION DESCRIPTION

Design the message consumer to simply crack the message then forward the message's data to a separate class defining the actual business logic.

ANECDOTAL EVIDENCE

"I can't test the business logic without starting the JMS server." "That's too hard to test." "Our system's only API is through asynchronous messaging."

SYMPTOMS, CONSEQUENCES

Business logic is tightly coupled to the use of JMS and can only be accessed by sending it a message.

HOT POTATO

DESCRIPTION

A message is continuously tossed back and forth between the JMS server, and a message consumer that won't acknowledge it has received the message.

MOST FREQUENT SCALE

Application

REFACTORED SOLUTION NAME

Acknowledge the message receipt, not its result.

REFACTORED SOLUTION TYPE

Software

REFACTORED SOLUTION DESCRIPTION

Consumer should always acknowledge that they've received a message. This acknowledgment should not be dependent on the success of the business logic handling the message. Log failures in business logic to a separate error queue.

ANECDOTAL EVIDENCE

"Where did all these messages come from?"

SYMPTOMS, CONSEQUENCES

The JMS server is burdened with attempting to redeliver messages that no consumer will ever acknowledge.

SLOW EATER

DESCRIPTION

Message consumers that take a relatively long time to consume a message negatively affect message throughput

MOST FREQUENT SCALE

Application

REFACTORED SOLUTION NAME

Eat as fast as you can.

REFACTORED SOLUTION TYPE

Software

REFACTORED SOLUTION DESCRIPTION

Measure the consumption rate of messages as an early warning system against bottlenecks. Optimize the code paths of message consumers as necessary.

ANECDOTAL EVIDENCE

"We have to frequently increase the size of the message-driven bean instance pool." "The message queues continue to grow unchecked."

SYMPTOMS, CONSEQUENCES

Message throughput is negatively affected when the production rate is greater than the consumption rate.

EAVESDROPPING

DESCRIPTION

Listening in on high-traffic message queues and topics for fear of missing an important message

MOST FREQUENT SCALE

Application

REFACTORED SOLUTION NAME

Use message selectors.

REFACTORED SOLUTION TYPE

Software

REFACTORED SOLUTION DESCRIPTION

The use of message selectors lets consumers tune out messages they aren't interested in hearing. Specialized message consumers can handle high-priority messages with a better QoS.

ANECDOTAL EVIDENCE

"This consumer keeps getting spammed with unwanted messages."

SYMPTOMS, CONSEQUENCES

Message consumers are burdened with handling throw-away messages and high priority work is intermixed with low priority work. Network and CPU utilization increases.

PERFORMANCE AFTERTHOUGHTS

DESCRIPTION

Focusing on performance without requirements or engaging in premature optimizations without a baseline

MOST FREQUENT SCALE

Application

REFACTORED SOLUTION NAME

Measure early and often.

REFACTORED SOLUTION TYPE

Process

REFACTORED SOLUTION DESCRIPTION

Gather performance requirements early and often. Build automated performance tests that continuously validate performance criteria. Use performance tests to benchmark JMS vendors based on your application's requirements.

TYPICAL CAUSES

Poor planning

ANECDOTAL EVIDENCE

"We will have plenty of time to performance tune at the end of the development cycle." "We'll let our QA department measure performance." "We're using a reputable JMS server, so it should scale well."

SYMPTOMS, CONSEQUENCES

Repeated delivery of poorly performing software, redesign of critical use cases late in the development cycle, and last-minute tuning activities that are ineffective.

Part 4

Broader topics

In Arkansas, we gaze at the rain-swollen Little Missouri River. We look forward to running this Ozark jewel, which is rarely this high. Our egos, too, are swelled. This river, now merely the promise of a fun diversion, would have been well beyond our skill level a mere two years ago. As we suit up, we plan our run, discussing strategy and safety issues. I notice a partner frantically scrambling through our gear and realize that we've forgotten to pack a spray skirt. The function of a skirt is to seal water out of the kayak. After eight hours of driving, we'll have no run today.

In Arkansas, we painfully learned that issues like packing and strategizing can be as important as fundamental skills like paddling. The same holds true of EJB development. Part 4 of *Bitter EJB* addresses secondary issues like tuning and packaging.

In chapter 9, we discuss the importance of good performance tuning techniques. We emphasize the need for an automated test suite and the importance of testing before making assumptions about performance. In chapter 10, we discuss the issues of building, testing, and packaging an application. We look into tools like XDoclet and Ant that make the build process easier to automate. And we underscore the importance of running automated tests. In chapter 11, we peek into the future of EJB, pointing to technologies that may play a crucial role in the future of EJB.

9 Bitter tunes

This chapter covers

- Definitions of performance
- Antipatterns related to the EJB performance tuning process
- A JUnitPerf tutorial
- Tuning an example EJB application using JUnitPerf tests
- A step-by-step performance testing methodology
- Techniques for automating performance testing

It's early in the morning, and I'm locked in tightly to my new snowboard, staring anxiously down the impossibly steep slope. I'm a skier who's grown increasingly addicted to the freedom of snowboarding, and I've learned quickly. But I'm having a tough time getting to the next level—the confident level of the elite boarder. With a twist of the hips, I accelerate downhill. I mechanically hammer through a couple of turns, reacting to each tiny groove and bump in the ungroomed morning snow. My brain gradually falls behind, and my body only barely keeps up with the descent. I'm in a purely reactionary mode now, with my eyes tracking the terrain only inches in front of me. I fear that I may be unable to stop, and I certainly can't keep up this reckless pace. I wonder if I will even see the crash come.

In this chapter, we'll tour a few common pitfalls related to the EJB performance tuning process. We'll focus on developing a disciplined performance testing methodology driven, not by irrational fears or wild speculation, but by automated tests whose objective results aren't distracted by emotion. By continually measuring the performance of our code—and the impact of our changes to it—these tests will help us stay ahead of the pain endured when undetected performance problems sneak into our code.

Ah, but tuning isn't a development activity, you say. Configuring the application for its operational environment is a job suited for those other geeks—the operations folks strolling safely around the lodge—not those of us still on the mountain. Well, we could pass the buck that way, but letting performance tuning roll too far downhill is an incredibly inefficient way to develop software. At best, it introduces a costly delay in the feedback cycle between making a change intended to improve performance and seeing whether that change actually did any good. At worst, failure to start measuring performance early invites the danger that significant problems will crop up later, when redesigns are no longer economical. Instead, to maximize our time and ensure a successful rollout of our application, we must obtain immediate results on early performance testing.

In this chapter, we will consider an EJB application that suffers from poor performance. The application will employ a familiar antipattern that will serve as a crash test dummy for our performance testing methodology, letting us focus on tuning the application and measuring the impact of that tuning. Each time we ratchet the performance gear a notch, we'll receive immediate data that indicates unambiguously whether we've truly improved performance. By taking the guesswork out of the tuning process, we'll increase our confidence, allowing us to tackle new performance requirements without fear.

9.1 Measures of success

Before we shift into high gear, let's first nail down a definition of performance as a measurement. In general, two ways exist for viewing performance: response time and throughput. We tune and test applications differently, depending on the aspect on which we're focusing our improvements.

9.1.1 Response time

The *response time* of our application refers to the speed at which the application is able to service a given request, such as a user requesting a web page through a browser. The request may be serviced by any number of resources in our application, including servlets, EJB, a database, or a legacy system. We can manage certain types of resources by placing a limit on the maximum number of concurrent requests each resource can safely handle. That is, managed resources are control valves that help us throttle the application for consistent performance and stability. Consequently, each time a request requires the use of a managed resource, it may need to wait in a queue until the resource is available.

Take, for example, a limited resource familiar to most enterprise developers—database connections. Figure 9.1 (a) shows a database connection servicing a request for data. In this case, the database connection pool is sized with ample available connections capable of servicing requests without queuing. So no cost is incurred in waiting for a database connection to become available. In contrast, figure 9.1 (b) shows a queue of active requests waiting to be serviced by a single database connection. In this case, the size of the database connection pool is not able to keep up with the number of new requests without queuing. Step right up… and take a number!

From figure 9.1, we can infer that the response time of a request will include any time spent waiting in the request queue for an available database connection. The response time may also include any network latency in obtaining a database connection via a remote call. Furthermore, as concurrent requests for a database

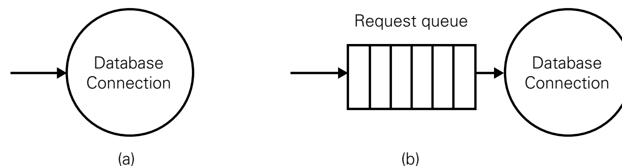

Figure 9.1 Database connections are examples of managed resources that may cause incoming requests to be queued before being serviced. Queuing incurs additional response time overhead.

connection increase, more requests will be queued, waiting for the connection. Therefore, to characterize the response time of our application accurately, we must take two essential measurements: the response time for a single request and the response time for the same request under a load of concurrent requests.

In section 9.7.2, we'll roll up our sleeves and write automated tests that measure the response time of a use case from our application. By continually running these tests, we should gain confidence, knowing that any optimizations we make have indeed improved response time. For now, let's begin by considering the possible measurements of such tests.

9.1.2 Throughput

While response time focuses on the speed of a specific request, *throughput* measures the number of requests our application can service in a given amount of time. For EJB applications, throughput is typically measured as the number of business transactions per second (tps). What constitutes an average business transaction is certainly application-specific, so throughput metrics always must be taken in appropriate context. For example, our application might be capable of processing 10 product catalog queries per second, with each query returning an average of five products.

From a slightly different angle, we use throughput as an indicator of our application's potential to scale. *Scalability* is a measure of a load's affect on our application's performance. For example, if we say that our application can scale to handle five concurrent users, then we're referring to the application's ability to maintain a linear (not exponential) average response time for each user, while under the stress of a five-user load.

Applications that scale well can deliver increasingly higher levels of throughput by adding resources, such as more hardware or more connections within a pool. When the average response time of a business transaction becomes intolerable under load, the application has reached its *maximum effective throughput*. Stressing the application beyond this point by piling on a heavier load will further degrade its responsiveness.

Revisiting our example of database connections as managed resources, figure 9.2 shows how a database connection pool can be used to work off requests efficiently. As the number of concurrent requests increases, the single database connection in figure 9.2 (a) will eventually hit a wall. Try as it might, the connection won't be able to keep up with the number of pending requests in the queue. Consequently, the request queue will continue to grow, adding to the response time of all waiting requests. By tuning the size of the database connection pool to include two

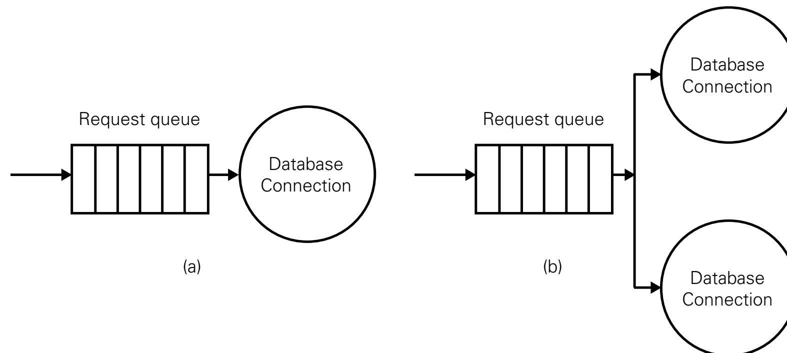

Figure 9.2 Pooling limited resources, such as database connections, is a common technique for improving an application's throughput. By increasing the size of the connection pool, a scalable application can take advantage of these additional resources to deliver better performance.

available connections, as shown in figure 9.2 (b), more requests can be serviced in a given amount of time. All other things being equal, figure 9.2 (b)'s connection pool will double the application's throughput depicted in figure 9.2 (a).

When attempts to get an application to scale prove frustrating, or even downright impossible, bottlenecks are the usual suspects. In general, a bottleneck is any chokepoint that restricts throughput. When a bottleneck is suspected, load testing tools are called to the rescue. Load testing tools are invaluable. They first put a load on an application, then shine light on any bottlenecks that rear their ugly heads. In section 9.7.5, we'll capture a nasty little bottleneck in the wild by writing an automated load test. Before we do, however, we must discuss an antipattern everybody knows but nobody likes to talk about: Premature Optimization.

9.2 Antipattern: Premature Optimization

Making a chunk of seemingly slow code faster can be quite satisfying. You can get a thrill from strutting your programming prowess and seeing immediate performance improvements. However, that thrill can cloud your judgment. Time passes in a flash as performance tweaking takes over, often resulting in an overly complex tangle of code that might get run only once in a blue moon. But when that code does run—man, is it fast!

Nobody is immune to the allure of Premature Optimization. We've been both victims of and witnesses to its use. Premature Optimization can take many forms: a speculative architecture decision, the choice of a particular design, a change in the runtime environment, or convoluted code.

Low-level code optimizations tend to attract your attention first. The trouble is, in most cases, the code paths you decide to optimize aren't called frequently enough to justify the time spent tuning the code. To make matters worse, the risk of wasting time optimizing arbitrary code paths increases with the code base size. Sure, you can always find a chunk of code that can be made faster, but it's usually the wrong one. Although you want your code to be reasonably efficient at all times, you gamble each time you blindly optimize code at the expense of code clarity and precious development time. Indeed, code clarity is often compromised as a result of optimization.

Premature Optimization has another insidious side effect. Tweaking code to make it faster tends to break something that's already working. That is, as you contort the code to squeeze out the last little bit of performance, you inevitably make compromises that can come back to bite you big later. Alas, your code won't win any awards for producing the wrong result quickly.

9.2.1 Tuning EJB applications blindfolded

EJB applications are especially unforgiving when you tune them in the dark. They inherently use at least one other resource, such as a database. As such, any arbitrary EJB method may spend more time blocked, waiting on a resource, than actually using the CPU. In that case, optimizing code won't show any significant performance improvement.

The runtime environment of an EJB application is also particularly fertile ground for the Premature Optimization antipattern. You must consider the virtual machine, database, and application server versions, as well as a dizzying array of internal tuning parameters for each version. If you try to get all these parameters adjusted for optimal performance before understanding their effect on your application, you incur at least two costs. First, you divert time away from those activities that really pay the bills. Second, you can complicate deployment unnecessarily by assuming that certain configuration parameters must be used.

Another reason EJB applications are prone to Premature Optimization is the myriad design decisions that must be weighed against performance. Indeed, it's easy to speculate on possible optimizations from the design perspective. Once you swerve onto that path, you may waste a significant amount of development time before you see any possible gain. Table 9.1 describes a few premature high-level design optimizations prevalent in EJB applications, along with their potential consequences.

At the end of the day, time spent tuning one area of our application is time not spent tuning another. It's a game of opportunity cost. Without a deep understanding of your application and the behaviors of its users, arbitrary

Table 9.1 Premature design optimizations in EJB applications may degrade performance and increase complexity. Deferring these types of optimizations until deemed necessary and beneficial is a better use of our time and resources.

Premature Optimization	Potential consequences
Entity beans	If your business object doesn't require concurrent read and write access while retaining stringent transactional integrity, then the use of an entity bean may incur unnecessary complexity and performance overhead. A servlet or session EJB using JDBC is often sufficient.
Stored procedures	Although stored procedures allow your database to do the heavy lifting, the business logic they encapsulate is tightly coupled to the database schema and may be written in a proprietary language that's difficult to maintain. Designing a business logic layer in your middle tier generally is easier to develop and maintain.
Bean-managed persistence	BMP entity beans may suffer from hard-coded SQL, difficult-to-maintain database logic, and $n + 1$ database calls to load n bean instances. Entity beans using container-managed persistence generally are more efficient and easier to develop, if used properly.
Custom primary key generator	If not designed carefully, custom primary key generators may require synchronization that becomes a scalability bottleneck. Better scalability, with less work, may be realized by using automatic primary key generators already provided by your database. To help, JDBC 3.0 includes new methods to facilitate the retrieval of automatically generated fields.
Caches	If the data in your cache is changing more often than it's being used, then the number of cache hits may not justify the complexity of caching while preserving data integrity.
Custom resource pools	The use of custom resource pools in the name of better performance may prevent your application server from managing resources effectively. Stability may deteriorate unless the pools already provided by your server are used.

optimizations are pure speculation. However, by first identifying the most valuable optimizations, whether at a high or low level, you can concentrate your efforts where they're most needed.

9.2.2 Solution 1: Plan, but don't act (yet)

Performance requirements are the solution for Premature Optimization. Without well-defined goals, you'll try forever to optimize every line of code you write to mitigate a performance backlash. However, by defining measurable goals for performance-critical use cases, you can optimize pragmatically, based on patterns in user behavior and data usage. Your energy is focused on solving the most critical performance issues first.

In our experience, the best performance gains are realized when following the advice given in the simple motto, "make it run, make it right, make it fast." Notice

that this advice speaks to the order in which you take action, not necessarily the order in which you consider the necessity of those actions. In other words, you should take action to improve performance only when not doing so would preclude you from delivering a successful application. The earlier you know what determines success, the better.

Knowing when to take action isn't always clear-cut. We're constantly trying to strike a delicate balance between optimizing the code we write today and building an application that can achieve expected levels of performance. On one hand, you want to keep the code efficient without racking up too much time tuning. On the other hand, you need to consider the performance requirements of your application early and often. If you don't keep in mind how your decisions might impact performance, chances are when you finally do look up, you'll be aimed straight for a tree. Nevertheless, you can avoid possible disasters by expending effort to improve performance only when you've gathered sufficient evidence to let you prioritize and focus on optimizations that will truly make a difference.

If you defer performance tuning until it's proven to be a high-yield investment, you'll have a chance to validate your design with working code and tests. At this point in the development cycle, you will be able to understand the design well enough to consider the potential benefits of global and local optimizations toward meeting performance goals. Better yet, with a solid foundation of tests, you will be able to tune safely, knowing that the tests will fail if tweaking code causes existing functionality to break. In the meantime, writing well-factored, modular code puts you in a position to tune economically down the road, if necessary.

9.2.3 Solution 2: Write well-factored, modular code

Until performance improvements are necessary, write code that is as simple and clean as possible. The time you'll save if you write the simplest and cleanest code as a matter of course can be used later to optimize those few places where code accidentally gets complex and laden.

If you find opportunities to improve performance, remember that simple designs that use well-factored, modular code are more amenable to performance tuning than more complicated designs. In general, well-factored code is easier to change. And code that's easy to change is easier to tune. By encapsulating implementation details, modular components can respond to change, allowing us to change their underlying code without breaking their clients. Moreover, well-factored, modular code exposes succinct methods that serve as excellent starting points for optimizing a particular code path. Take, for example, the method in

listing 9.1 responsible for withdrawing an amount from a bank account, designed as an entity EJB.

Listing 9.1 Code that is not well factored is also difficult to tune

```
public double withdraw(int accountId, double amount)
  throws Exception {

   InitialContext context = new InitialContext();

   Object homeRef = context.lookup("AccountHome");

   AccountHome accountHome =  (AccountHome)PortableRemoteObject.
    narrow(homeRef, AccountHome.class);

   Account account = accountHome.findByPrimaryKey(accountId);

   account.setBalance(account.getBalance() - amount);

   return account.getBalance();
 }
```

Find the specified account by its ID

Withdraw the specified amount of money

Notice how all the logic is inlined in the method. If we were to run a code profiler on this code, we would find it difficult to ascertain which piece of logic—finding the appropriate account or withdrawing the specified amount—consumes the most time. The profiler would merely decompose the overall method time into the individual execution times of each method invoked. However, we'd like to know which coarse-grained code path could benefit most from tuning. To make this monolithic method easier to read, let's refactor it a bit, as shown in listing 9.2.

Listing 9.2 Well-factored code enables a code profiler to help you tune effectively

```
double withdraw(int accountId, double amount) {

  Account account = findAccount(accountId);

  return account.withdraw(amount);
}
```

Find the specified account by its ID

Withdraw the specified amount of money

Our refactoring organized the inlined code into two distinct methods: `findAccount()` and `withdraw()`. In applying this refactoring, not only have we made the code simpler and more modular, we've also enabled the code profiler to help us. The code profiler can now quantify the individual cost of each code path and point us directly to the starting point of the most expensive path. And, as an added bonus, once we optimize a particular method, that method can be used by other components in our application, providing better performance many times over.

Now, let's consider what would happen if, instead of building performance into our design, we attempt to bolt on performance after the design is finished.

9.3 Antipattern: Performance Afterthoughts

Developing applications that perform well requires prior intent. If performance is important, it must be baked in to the application, not bolted on afterwards. We learned this lesson the hard way a few years back. (And one of us has the hairline to prove it!) The database we were using had a serious bottleneck in its locking strategy. When used with applications that read more data than they wrote, the database was lightning fast. Yet whenever multiple concurrent users attempted frequent database updates, this particular database was clearly the wrong tool for the job. Unfortunately, we didn't know the bottleneck existed until it was too late. Although we knew from the beginning that the application needed to scale in order to be successful, we didn't plan for scalability early enough. We assumed that the application would scale, and if it didn't, we figured we would have time later to refactor the application's design to be more scalable. Building a prototype that demonstrated a few performance-critical use cases under load would have alerted us earlier to the impending doom.

Just because you're using EJB technology doesn't mean you can disregard performance concerns. It's true that any worthy EJB container will help you manage resources for the best possible performance. However, perfuming a poorly performing application with the scent of EJB won't keep the flies away.

We can increase our application's probability of success by using simple designs tactically with well-factored, modular code, but that, too, is no substitute for strategically planning for performance. This presents a conundrum. If we delay considering performance until right before the application goes live, it's usually too little too late. Then again, we don't want to speculate on performance at the expense of rapidly delivering valuable software. The answer to this problem lies in continuous planning and measurement.

9.3.1 Solution: Plan early and often

To counterbalance premature optimization, we need to plan proactively for performance. That's not to say we should attempt to predict future performance demands and carve a plan in stone. We'll be sorely disappointed when, as often happens, things don't go according to that plan. Indeed, our perspective inevitably will alter as we learn more about our application and its users. Consequently, the performance plan is subject to frequent change. Planning for performance

requires that we constantly consider the current state and goals of our application, by taking measurements and making course corrections throughout the project.

As the delivery date approaches, no doubt we'll know which aspects of our application suffer from poor performance. Furthermore, we'll have more accurate estimates of the production load on our application and the respective hardware necessary to handle that load. Performance plans may change as a result of this information, and we should consider this a good thing.

In the meantime, the performance planning process will help us head off potential problems at the earliest opportunity. If we continually plan for performance, we'll be able to see obstacles in the terrain ahead and react in time to avoid a crash. Let's consider the following guidelines for performance planning:

1. ***Understand the application's usage patterns*** Users generally expect different levels of service, depending on the feature of the application they are using at the time. Users expect some use cases to respond rapidly and understand when others are slower. A web user, for example, expects to navigate a product catalog quickly. Yet, when an online order is placed, the same user will accept a delayed order confirmation via email. Understanding patterns in user behavior, and the data and resources required to support that behavior, provides invaluable input into the performance planning process.
2. ***Prioritize performance requirements*** To maximize your time and dollars, you should satisfy the performance requirement with the highest business value first. For example, optimizing a product catalog for maximum responsiveness when browsed under load is arguably a better investment than optimizing your email server for faster order confirmation. Once the top performance requirement has been demonstrated successfully, you can work on the next highest priority requirement. Rinse and repeat.
3. ***Write automated performance tests*** Performance tests that unambiguously define and validate the performance requirements of your application are essential in helping you meet desired performance goals. Without a target, you'll never know when you've hit the mark. Good performance tests express objective exit criteria in an executable format. In other words, running these tests will help you decide if tuning is necessary, and, if so, when tuning should stop. Tests also prevent tendencies to overoptimize based on speculation or commit too early to designs and infrastructure that seem to promise improved performance.
4. ***Build modular components*** Components that hide their implementation insulate the rest of the application from changes made to improve

performance. Using these components, you can start with a simple algorithm that works, even if it may incur a few extra seconds of overhead. As you learn more about your application and its uses, you can easily swap in a new, blazingly fast algorithm or data structure, for example.

5. ***Revise plans based on feedback*** Once performance goals have been identified and prioritized, you must demonstrate performance as early as possible to get feedback. You'll want to know sooner rather than later if you're making design decisions that may prohibit your application from meeting user demands. If you feed this information back into the planning process, you can steer the design to meet your performance goals continually. You can respond more readily to change, rather than dutifully marching to an inflexible master plan.
6. ***Understand your EJB server's configuration options*** In your haste to tear the wrapper off your new EJB server, a few finer features may go unnoticed. Server vendors differentiate themselves from their competitors, providing different knobs and levers you can twist and pull to improve performance. If you understand the available options, you'll know when to leverage rather than build for successful performance. Study and investigate the contents of the box. Then experiment and see what happens. Your tests will announce impending danger if your application starts to sputter.
7. ***Schedule the availability of production hardware*** Your plan should include testing on production-quality hardware as soon as possible. Indeed, how your application performs for real users is what matters most ultimately. Everything else is just preparation.

Remember, it's not the plan that's important, but the planning. With that in mind, let's get down in the trenches with a real, live application.

9.4 Grist for the tuning mill

Let's say we've accepted a mission to develop yet another online product catalog. Our customer group—those folks defining the requirements of our application—has decided that users want to browse a list of all products for a particular category within a product catalog. The initial user interface will be an HTML browser, but it's imperative that the catalog browsing service be available to other types of distributed clients. This requirement is not unlike the others we've been delivering, from which a service-oriented architecture has emerged.

We conclude that the simplest approach would be to publish a remote façade that encapsulates the business logic of querying a product catalog. We dub it the catalog service. A stateless session bean seems like the logical choice given our architecture and experience, so let's make it the centerpiece of our design.

9.4.1 Putting an EJB to the test

Before diving into the implementation of our catalog service, let's start by writing a test. Why? Well, how else will we know what code to write? If we write a test first, we'll have an example of the catalog service's intended use. In addition to demonstrating the intent of the catalog service, the test can validate automatically that the catalog service returns the correct results. Once the test passes, we're done!

Listing 9.3 shows the JUnit test, which queries for all products in the snowboard category of the product catalog. (Note: a full tutorial on JUnit goes beyond the scope of this chapter. The *JUnit Primer*[1] will help get you up and running quickly.)

Listing 9.3 Unit testing the product catalog service

```
public class CatalogTest extends TestCase {

  public CatalogTest(String name) {
    super(name);
  }

  public void testGetProducts() throws Exception {

    String snowboardCategory = "Snowboard";

    Catalog catalog = (Catalog)getCatalogHome().create();

    Collection products =
       catalog.getProductsByCategory(snowboardCategory);

    assertEquals(25, products.size());

    Iterator productIter = products.iterator();
    while (productIter.hasNext()) {
      ProductDetails product = (ProductDetails) productIter.next();
      assertEquals(snowboardCategory, product.getCategory());
    }

    catalog.remove();
  }
}
```

1 http://www.clarkware.com/articles/JUnitPrimer.html

The test case has a single test method, `testGetProducts()`, that starts by using the `Catalog` home interface to create a remote reference to a `Catalog` session bean instance. The `Catalog` remote interface represents a façade—a black box from the client's perspective—that finds products in a catalog. Using the remote `Catalog` reference, the test then queries for all products in the snowboard category. We expect exactly 25 products in this category because before running the test we created 25 example products in the snowboard category of the product catalog database. Iterating over the resulting collection of products, the test validates that the catalog service only returns the products in the snowboard category.

Excellent! Now there's just one problem: We have to get the test to pass.

9.4.2 *Passing the test*

To get the test to pass, we have to write the code for the `Catalog` EJB. All EJB components require a remote interface, home interface, and bean class. We won't actually show the code here because, frankly, the implementation doesn't matter. Instead, we'll just show the design of one possible solution. Our priority should be to begin by writing clean and simple code. We won't worry about performance here. We just want to validate that our design is usable and our code produces the correct results, thus avoiding the risk of overspending on performance too early. Running the test from the remote client's perspective gives us confidence that the catalog service is working as we expect. We can change and tune the underlying code without the fear that existing functionality might silently break. If it does, the test will surely let out a scream.

Under the hood in the `Catalog` bean class, we code the bean to use JDBC to query our product table directly through a database connection. The values in our database table are then packaged and returned to the client in a collection of lightweight `ProductDetails` DTOs. In future use cases, administrative users may update the products in a catalog. This updating might require a more complex persistence mechanism, for example, the type of persistence afforded by an entity bean. We'll cross that bridge when we get there. Right now we're concerned only with retrieving read-only product information, so a simple stateless session bean wired up to the database will do just fine. Figure 9.3 shows a UML sequence diagram illustrating the interaction of our recently built components.

Notice that all business logic occurs on the server side, behind the façade of the `Catalog` interface. As such, our design is modular. If performance becomes an issue, we can tune the code behind the `Catalog` interface without adversely affecting its remote clients. That's reassuring because we have a sneaky suspicion that tuning may be in our immediate future.

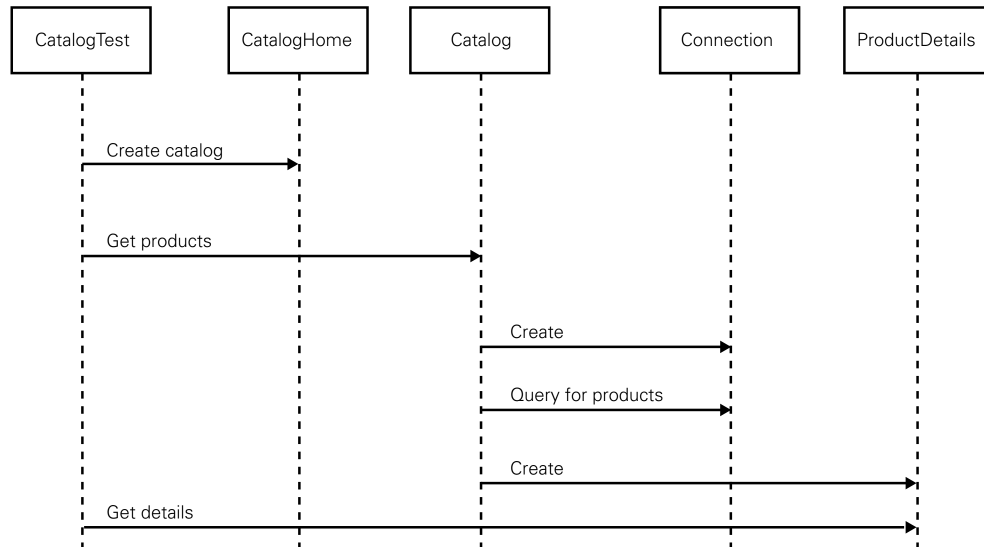

Figure 9.3 The Catalog stateless session bean, wired directly to the database, is used to query the catalog for all products in a product category. The test stands in for a client, demonstrating a usage scenario and validating that expected results are returned.

We've written just enough code to get the test to pass. We've made it work and made it right. Now, let's make it fast.

9.4.3 *Specifying response time as a measure of success*

Up until now, we've written simple code so that the test would pass. In doing so, we certainly didn't forget the hard-earned wisdom we gained designing distributed systems over the years. Rather, we made a concerted effort to determine the workability of our design and code first, before speculating on performance.

With much pride, we demonstrate the new catalog service to our customer, who is delighted. After kicking the tires a bit, our customer hunkers down at our test machine and proceeds to press the button that lists the snowboard category products over and over again. Our customer likes the fact that the new catalog service is working this soon in the schedule, so now we're ready to start polishing. The customer consensus is that the service just isn't fast enough; we need to improve performance.

The average response time of the web page to query and list 25 products is approximately 1.4 seconds. We think we can do better than that with a little tuning, but we need a way to measure success. So we ask our customer group to write

a new performance requirement. We watch as they scratch their heads and mumble a bit, but finally they draft a performance:

The average response time of the catalog web page listing up to 25 products should not exceed 1 second.

Great, now we have a goal!

9.4.4 Seeing light at the end of the tuning tunnel

To satisfy our new performance requirement, we need to shave off about a half-second of the response time, then stop tuning before we hit the point of diminishing returns. However, we're not sure whether to start chiseling in the presentation layer or the business logic layer. We want to focus our attention on tuning the sections of code that contribute the most overhead to the overall response time. Guessing would be a fool's game, so we put our trusty code profiler on the case to hunt down the busiest code. Table 9.2 shows the results:

Table 9.2 Using a code profiler takes the guesswork out of tuning by identifying hot spots worth optimizing. Always seek the advice of a code profiler before attempting to tune code.

Method	Total time (ms)
`CatalogEJB.getProductsByCategory()`	1327.0
`CatalogServlet.service()`	10.0
`CatalogEJB.getConnection()`	10.0
`CatalogEJB.runQuery()`	15.0
	1362.0

The code profiler identifies the `CatalogEJB.getProductsByCategory()` method as the major contributor to the total response time. Well, that rules out tuning the presentation layer for any observable gain in performance. The top contributor is the EJB method, not the servlet method that presents its results. The profiler results also rule out tuning the database interaction. The time it takes to obtain a database connection and execute the SQL query is insignificant compared to the business logic in the EJB.

We have no question that the culprit is the code in the EJB that transforms database rows into products. But when we're done tuning, how will we know we've made good progress? What if our tuning activities cause performance to take a step backward? Fear begins to envelop us as we're reminded of endless hours spent with the subject of our next antipattern—Thrash-tuning.

9.5 Antipattern: Thrash-tuning

> *I am barely a third of the way down the hill, contemplating disaster, when something finally clicks. Instead of focusing on every slight ripple in the snow, I concentrate only on obstacles that might impact my balance or my course down the mountain. I'm able to keep my eyes further ahead, improving my ability to plan and react. With this improvement I husband my waning strength better, saving it for the biggest challenges—like that tree just ahead! I've entered the zone that my mentors so often describe.*

If you're always looking at your feet, you can't anticipate what's up ahead. And if you can't anticipate, you can't determine whether your microadjustments are making any progress toward the ultimate goal. Before long, you're bound to diverge hopelessly off course.

Thrash-tuning is a nasty habit born out of undisciplined performance tuning. You know the drill: change a tuning parameter here, tweak some code there, then run the application. *Is it faster? No, it actually seems to be slower! Interesting. Now which change caused that to happen?* Lacking clear knowledge, you repeat the cycle, over and over again.

Without a baseline to measure against, Thrash-tuning is entirely unpredictable. You may spend days tuning in circles with only a minor improvement to the application's overall performance. Sometimes you get lucky and increase performance by a single order of magnitude in a quick round of thrashing, but you'll soon find that's the exception. As a general rule, Thrash-tuning can consume hours or days of your life, without amounting to much good.

The following are common ways to invoke the curse of Thrash-tuning:

- **Changing more than one thing at a time** In the rush to get the most bang for your tuning buck, you change multiple things at once, but doing so makes the individual contributions of those changes indistinguishable from each other. Multiple changes also makes backing out a change that degrades performance difficult.
- **Forgetting to measure between changes** Without quantifiable evidence that a particular change improves performance, you can't clearly determine its impact. Performance goals always seem to be just beyond your grasp.
- **Not knowing when to stop** Remember, Thrash-tuning is a habit and a particularly hard one to kick. You may find yourself tuning endlessly. Performance tests are the cure, telling you when enough is enough.

If these scenarios sound all too familiar, you're not alone. Both novice and veterans alike have suffered similar fates at the hands of EJB applications. The sheer number of opportunities for improving EJB performance makes falling into a vicious tuning cycle deceivingly easy. At one end of the spectrum, you can change deployment descriptors quickly to dynamically influence performance. At the opposite end, you can apply high-level design patterns, using general knowledge gained from distributed computing.

The difficulty lies in choosing an approach and measuring its impact in isolation. A solution applied in hopes of improving performance for one aspect of our application may cause unwanted secondary effects in other areas. Consequently, performance tuning quickly turns into a delicate balancing act. We become painfully aware that multiple controls often exist for performance with complex interactions. Each new interaction increases the odds that making a change—one that theoretically should improve performance—may not make a difference.

Besides being incredibly frustrating, when thrash-tuning runs rampant, it has the potential to rob us of enormous amounts of development time. The more you scratch it, the more it itches. A sure-fire way to stop this irritation is to use a sound methodology with information derived from automated performance testing.

9.5.1 Solution: Use a performance testing methodology

The only defense we've found against falling prey to Thrash-tuning is the use of tests to gather evidence first, before making an attempt to improve performance. We've tried predicting whether an optimization would improve performance, and yet, after hours of navel gazing, we remained undecided. Although not as therapeutic as navel gazing, writing tests that measure the impact of changes has given us much better success.

To ensure that you're always ratcheting forward toward optimal performance, current performance must be automatically compared to a baseline. Doing so keeps the performance of your application from going off the rails. If you use a methodology like the following, your performance stays on track, never more than one change away from the last baseline:

1. Begin with clean and modular code that's easy to understand and modify, and driven by tests that express its intentions and expectations.
2. Choose quantifiable performance goals for the code.
3. Profile the code to identify hot spots with the highest return on investment.
4. Write and run an automated test that baselines current performance.

5. Make a change intended to improve performance.
6. Run the automated test again to measure the gain (or loss) in performance.
7. Repeat as often as necessary until the application meets its performance goals.

Notice that we use a test to measure both before and after a tuning change is made. The test tells us if the tuning did any good. If not, we can back out the change to arrive safely at the last good baseline. Bear in mind that we might need to give caches, pools, and other performance-enhancing mechanisms a chance to warm up before taking a measurement. Otherwise, the observed performance may be thrown off by a cold start. In other words, the test environment has to be predictable, and the tests must be repeatable.

This easy-to-use formula really shines when applied incrementally to solve the most pressing performance problem at any given time. Once performance has improved in one area, and a test is in place to keep that problem continually in check, you can repeat the formula with the next most important performance problem.

We'll use this methodology to confidently tune our catalog service throughout the remainder of this chapter. In fact, we've already taken the first steps toward our goal. We wrote clean and modular code to make our test pass. Demonstrating our results to our customer prompted them to draft a realistic performance requirement we can measure. Before beginning to hack and slash, we used a profiler to find high-yield tuning opportunities. Now we're ready to begin tuning so that we can deliver on the goal our customer has given us. First, let's make sure we get started on the right foot by avoiding the tedium of manual testing.

9.6 Mini-antipattern: Manual Performance Testing

When we first turned over our catalog service to our customer, we watched as they poked and prodded, and we noticed that they grew weary of manually hitting the web page to assess performance. A couple times, they had to redo tests because the manual process wasn't followed consistently. Clearly, we need a more efficient and less error-prone testing strategy. As the suite of performance tests grows, running them all manually just doesn't scale. So many tests to run, so little time. Our team has earned a reputation for cranking out high-quality features like clockwork. To live up to that great reputation, we can't drag ourselves to the test lab to play the role of simulated web users every time we change something that impacts performance. That's what computers are for!

When push comes to shove and deadlines loom, manual tests are always the second thing (right after documentation, of course) that gets selectively ignored. Peril usually isn't too far behind. The next time we tune something without the safety of automated tests, our confidence will wane, and we'll fall back into a Thrash-tuning cycle again. Our stress goes up, the number of defects increases, and pretty soon we're burnt out.

9.6.1 *Solution: Automate performance testing*

Automated performance tests are like canaries in a coal mine. If we keep them running, they'll continue to measure whether performance goals are being met in the face of change. If a change causes performance to backslide, well, we'll know it at the poor birdie's expense.

Good performance tests offer many advantages, including

- automatically checking their own results
- providing immediate feedback in the form of a simple pass or fail status
- retaining their value over time through repeated testing of expectations
- running continuously without manual intervention
- instilling confidence to change code with impunity

At the least, we should run our performance tests once a day as a sanity check. If we're actively tuning code or changing the runtime environment, we should run them more often. The repeatability of the tests will prove our application's readiness for production.

A plethora of tools already exists for performance testing automation. Apache JMeter,[2] for example, is an open source desktop application that measures the performance of an application's behavior under load. Traditionally used to test web applications, over time JMeter has been made more extensible. You can now write custom extensions to put almost any server, network, or object under load. JMeter is also highly configurable; it includes a collection of test listeners for graphically visualizing performance. It can also be configured to include test assertions. For example, you can assert that a request for a web page returns within a specified amount of time. Yet, although JMeter is a valuable tool for performance testing, it falls short of our goals for automation. Specifically, the test results must be manually inspected each time the tests are run. While we could

2 http://jakarta.apache.org/jmeter/

probably extend JMeter to satisfy our desire for automation, instead we opt to use a complementary tool.

In this chapter we'll use JUnitPerf, an open source performance-testing tool that wraps existing functional tests written in JUnit. We choose JUnitPerf because it allows us to write tests that automatically check their own results and provide immediate feedback with an unambiguous pass or fail status. JUnitPerf is also tightly integrated with JUnit, so our performance tests can be run automatically alongside our functional JUnit tests.

9.7 Automated performance testing with JUnitPerf

We've already validated the feasibility of our EJB design with working code and a functional test. Making it fast enough to please our customer is our next order of business. However, we don't want to risk breaking functionality by complicating our code with performance optimizations. To be useful, the catalog service has to work right and perform well at the same time. Let's explore how JUnitPerf can keep both these interests in check.

9.7.1 JUnitPerf overview

JUnitPerf[3] is an open source set of JUnit extensions for automated performance testing. JUnitPerf tests transparently wrap standard JUnit tests and measure their performance. In other words, we can build upon our existing functional test to make sure the code continues to work right. The JUnitPerf tests tell us if the code is fast enough. If a performance test doesn't meet expectations, the whole test fails. If the functional test fails, the performance test fails. Conversely, if the performance test passes, then we have confidence that tuning didn't cause existing functionality to break. Table 9.3 describes the major JUnitPerf classes and interfaces.

Because JUnitPerf tests can run any class that implements JUnit's `Test` interface, we could use JUnitPerf to measure the performance of any test conforming to this interface. In this section, we use it to wrap the JUnit test we wrote earlier for our catalog service. We could also use JUnitPerf to run HttpUnit tests and measure the performance of our entire web application, for example. Another option might be to use JUnitPerf to run Cactus tests to validate our catalog service's business logic from within the EJB container.

---

3 http://www.clarkware.com/software/JUnitPerf.html

Table 9.3 JUnitPerf is a collection of classes and interfaces for performance testing JUnit tests.

Class/Interface	Description
`TimedTest`	Runs a JUnit test and measures its elapsed time. A `TimedTest` is constructed with a specified maximum elapsed time. By default, a `TimedTest` will wait for the completion of its JUnit test and then fail if the maximum elapsed time was exceeded. Alternatively, a `TimedTest` can be constructed to immediately fail when the maximum elapsed time of its JUnit test is exceeded.
`LoadTest`	Runs a JUnit test with a simulated number of concurrent users and iterations. The load can be incrementally ramped by registering a `Timer` instance to control the delay between the additions of each concurrent user.
`Timer`	An interface implemented by classes that define timing strategies to optionally control the delay between additions of users in a `LoadTest`.
`ConstantTimer`	A `Timer` with a constant delay.
`RandomTimer`	A `Timer` with a random delay and a uniformly distributed variation.

But that's another day. Right now, our customer is getting nervous. Let's put our money where our mouth is with an automated response time test for the catalog service.

9.7.2 Testing response time

Recall that we're staring down the barrel of a response time of approximately 1.4 seconds to display 25 products on a web page. We won't sleep well until the response time is under 1 second. Luckily, we know what to do. The code profiler indicated earlier that optimizing the `CatalogEJB.getProductsByCategory()` method would be the smart move. How will we know when we're done? Well, when a performance test passes, of course.

We want to write a test that will fail if the response time of our use case exceeds 1 second. To do that, we create a JUnitPerf `TimedTest` instance that wraps our existing `CatalogTest.testGetProducts()` test case method. Listing 9.4 shows the JUnitPerf test used to validate our performance expectations.

Listing 9.4 Testing the response time of the catalog service

```
public class CatalogResponseTimeTest {

  public static Test suite() {

    long maxTimeInMillis = 1000;

    Test test = new CatalogTest("testGetProducts");
    Test timedTest = new TimedTest(test, maxTimeInMillis);
    return timedTest;
```

```
    }

    public static void main(String args[]) {
      junit.textui.TestRunner.run(suite());
    }
  }
```

As a convenient way to run our test, our test defines a `suite()` method called by the JUnit test runner in the `main()` method. We run the `CatalogResponseTime-Test`, and it fails with the following output:

```
.TimedTest (WAITING):
  testGetProducts(com.bitterejb.catalog.ejb.CatalogTest): 1352 ms
F
Time: 1.352
There was 1 failure:
1)testGetProducts(com.bitterejb.catalog.ejb.CatalogTest)
Maximum elapsed time exceeded! Expected 1000ms, but was 1352ms.

FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```

All right, we knew that would happen. We just wanted to see if the test was really measuring anything. The test expected the response time to be less than 1 second, but sure enough, it measured the same response time as observed by our customer—1.4 seconds. Now we have a solid baseline from which to work. We should be able to optimize code, improving the response time until the test passes. If the test doesn't eventually pass, we'll need to start turning the performance knob in the other direction or look for another knob to turn.

Be aware of a subtle "gotcha!" when writing JUnitPerf tests. The response time measured by a `TimedTest` includes the elapsed time of the `testXXX()` method and its test fixture—the `setUp()` and `tearDown()` methods. Therefore, the maximum elapsed time specified in the `TimedTest` should be adjusted accordingly to take into account any cost of the existing test's fixture.

9.7.3 Tweaking code

To get the test to pass, we follow the code profiler's advice and optimize the logic that created `ProductDetails` objects from the rows in our database. The SQL query is sufficiently fast, according to the profiler, so nothing is gained barking up that tree. After optimizing, we run the `CatalogResponseTimeTest` again, and it gives us the following output:

```
.
```

```
TimedTest (WAITING):
testGetProducts(com.bitterejb.catalog.ejb.CatalogTest): 751 ms

Time: 0.751

OK (1 test)
```

Hey, that did the trick! The test tells us that we made good progress. Had the test failed, we could have continued to optimize until it passed.

After showing off the improved catalog service to our customer, we add this test to our suite of performance tests. As we go forward, the automated test will continue to keep the response time of this use case in check.

9.7.4 Specifying scalability as a measure of success

At this point we know how long it takes for one user to get a list of 25 products using the catalog service. How long will the same process take if our application is under the stress of multiple concurrent users? Until now we haven't thought much about scalability, but we're confident that our simple design won't let us down. Here's where the rubber meets the road.

Our customer is impressed with our track record of meeting goals, and now is ready to hand us a new challenge. Performance planning early and often has enabled the customer to accurately estimate the expected load on the production system. The customer now wants to improve upon our performance success by writing a new performance requirement, which states:

The response time of the catalog web page listing up to 25 products should not exceed 1 second under a load of five concurrent users.

In other words, the catalog service should scale to handle five concurrent users while consistently maintaining the single-user response time we demonstrated earlier. That's a tall order. Let's use JUnitPerf to see how far off the mark our application currently is.

9.7.5 Testing response time under load

We have an automated JUnitPerf test that measures single-user response time. We'd like to use a similar testing technique to put this test under a load of five concurrent users while measuring each user's response time. We want the test to fail if any user's response time exceeds one second. Then we can follow our performance testing methodology to tune until the test passes.

To do so, we write a JUnitPerf test that creates a `LoadTest` instance passing in a `TimedTest` instance and a number of concurrent users. The `TimedTest` in turn

wraps our existing `CatalogTest.testGetProducts()` test case method. Listing 9.5 shows the JUnitPerf test used to validate our scalability expectations.

Listing 9.5 Testing the scalability of the catalog service

```
public class CatalogLoadTest {

  public static Test suite() {

    long maxTimeInMillis = 1000;
    int concurrentUsers = 5;

    Test test = new CatalogTest("testGetProducts");
    Test timedTest = new TimedTest(test, maxTimeInMillis);
    Test loadTest = new LoadTest(timedTest, concurrentUsers);
    return loadTest;
  }

  public static void main(String args[]) {
    junit.textui.TestRunner.run(suite());
  }
}
```

We run the `CatalogLoadTest` , which fails with the following output:

```
..........
TimedTest (WAITING):
testGetProducts(com.bitterejb.catalog.ejb.CatalogTest): 771 ms
TimedTest (WAITING): testGetProducts(com.bitterejb.catalog.ejb.CatalogTest):
   1372 ms
F
TimedTest (WAITING): testGetProducts(com.bitterejb.catalog.ejb.CatalogTest):
   1963 ms
F
TimedTest (WAITING): testGetProducts(com.bitterejb.catalog.ejb.CatalogTest):
   2584 ms
F
TimedTest (WAITING): testGetProducts(com.bitterejb.catalog.ejb.CatalogTest):
   3255 ms
F
Time: 3.40
There were 4 failures:

1)testGetProducts(com.bitterejb.catalog.ejb.CatalogTest)
  Maximum elapsed time exceeded! Expected 1000ms, but was 1372ms.
2)testGetProducts(com.bitterejb.catalog.ejb.CatalogTest)
  Maximum elapsed time exceeded! Expected 1000ms, but was 1963ms.
3)testGetProducts(com.bitterejb.catalog.ejb.CatalogTest)
  Maximum elapsed time exceeded! Expected 1000ms, but was 2584ms.
4)testGetProducts(com.bitterejb.catalog.ejb.CatalogTest)
  Maximum elapsed time exceeded! Expected 1000ms, but was 3255ms.
```

```
. . .
FAILURES!!!
Tests run: 5,  Failures: 4,  Errors: 0
```

Ouch! Our application can't scale beyond one user. Notice that the first user's response time is within the 1-second limit, but the other users' response times bust the threshold. Worse yet, the response times increased for each successive user, indicating that our application has a bottleneck restricting its ability to scale.

So we fire up the code profiler and run the `CatalogLoadTest` to obtain clues. The code profiler doesn't let us down. Table 9.4 shows what the profiler finds when the catalog service is under load.

Table 9.4 Running a profiler on the catalog service under load reveals contention for a database connection. Load testing tools help illuminate scalability bottlenecks.

Method	Average time (ms)
`CatalogEJB.getProductsByCategory()`	716.0
`CatalogServlet.service()`	10.0
`CatalogEJB.getConnection()`	1248.0
`CatalogEJB.runQuery()`	15.0
	1989.0

The `CatalogEJB.getConnection()` method that was only taking around 10 milliseconds in our initial run of the code profiler is now taking up the majority of the overall response time. Let's tune that method while continuing to test the single-user response time.

9.7.6 *Using a connection pool to increase throughput*

Based on the evidence provided by the code profiler, we conclude that a single database connection is the limiting factor to scaling our application. Consequently, requests for a connection are being queued. Each successive user's response time in turn rises above the desired threshold.

In this case, pooling database connections is a low cost, high reward change sure to improve scalability. Before you start rolling your eyes over yet another example demonstrating the virtues of connection pooling, allow us to explain. We realize connection pooling is the poster child for many discussions on performance. Indeed, it's a well-known performance problem, and that's exactly why we're using it here. We want the problem—and the solution—to be familiar so

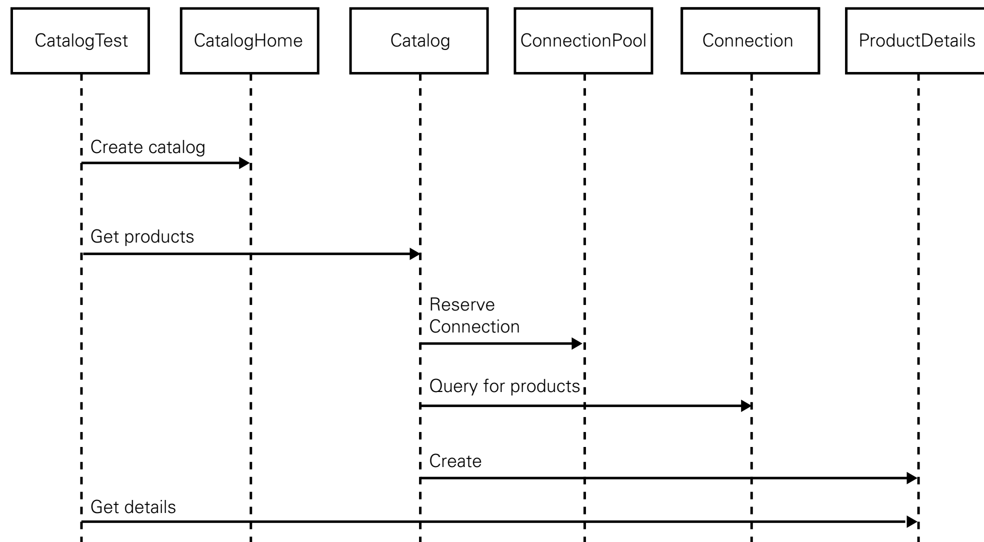

Figure 9.4 Refactoring the catalog service to use a database connection pool improves the scalability without sacrificing code complexity.

you can focus on how to test it. It's not about the connection pool; it's about the technique to discover it.

It's also worth noting that we've run across more than one improperly sized connection pool. Worse yet, we've seen custom connection pools that were implemented incorrectly. (Yet another victim of the Not Invented Here antipattern.) In other words, just because you're using a connection pool doesn't necessarily mean you get instant scalability. You have to test for that, so let's get back to the technique.

Instead of synchronizing access to a single database connection shared by multiple users, we refactor our `Catalog` EJB to use a database connection pool. We then configure the pool size to five active connections to improve scalability. Figure 9.4 shows a UML sequence diagram illustrating the use of the database connection pool.

Now we run the `CatalogLoadTest` again, and it passes with the following output:

```
.....
TimedTest (WAITING): testGetProducts(com.bitterejb.catalog.ejb.CatalogTest):
   751 ms
TimedTest (WAITING): testGetProducts(com.bitterejb.catalog.ejb.CatalogTest):
   812 ms
```

```
TimedTest (WAITING): testGetProducts(com.bitterejb.catalog.ejb.CatalogTest):
   822 ms
TimedTest (WAITING): testGetProducts(com.bitterejb.catalog.ejb.CatalogTest):
   831 ms
TimedTest (WAITING): testGetProducts(com.bitterejb.catalog.ejb.CatalogTest):
   811 ms

Time: 0.972

OK (5 tests)
```

Outstanding! Our scalability test is passing, and the underlying functional test continues to pass. This tells us that refactoring to use a database connection pool didn't break anything. As we expected, the refactoring actually improved scalability. Because requests don't need to be queued before being serviced, the response times are fairly consistent for each concurrent user. The test validates our design as able to handle five concurrent users without any specific user experiencing a delayed response time. If, in the future, the response time of any user increases beyond the limit set in our load test, the test will fail.

9.7.7 *Testing throughput*

We may end up with performance requirements expressed as throughput rather than as response time under load. For example, we might want to write an automated test to measure the total amount of time elapsed while servicing all five concurrent users. Using JUnitPerf, we simply reverse the order in which we create the tests, this time wrapping the `LoadTest` in a `TimedTest`, as indicated in listing 9.6.

Listing 9.6 Testing the throughput of the catalog service

```
public class CatalogThroughputTest {

  public static Test suite() {

    long maxTimeInMillis = 1000;
    int concurrentUsers = 5;

    Test test = new CatalogTest("testGetProducts");
    Test loadTest = new LoadTest(test, concurrentUsers);
    Test timedTest = new TimedTest(loadTest, maxTimeInMillis);
    return timedTest;
  }

  public static void main(String args[]) {
    junit.textui.TestRunner.run(suite());
  }
}
```

The `CatalogThroughputTest` will fail if the catalog service is unable to process at least five catalog queries per second. After refactoring to use a database connection pool, the `CatalogThroughputTest` passes with the following output:

```
.....
TimedTest (WAITING): LoadTest (NON-ATOMIC): ThreadedTest:
  testGetProducts(com.bitterejb.catalog.ejb.CatalogTest)(repeated): 972 ms

Time: 0.972

OK (5 tests)
```

Now that we've written JUnitPerf tests to measure both response time and throughput, let's put all the numbers together into a performance model.

9.8 Modeling performance

Using our existing JUnitPerf tests, we can ramp up the user load to sketch out a model that represents our application's overall performance. Doing so will answer questions in the performance planning process like, "Will our application scale to meet the demands of 10, 100, or 1,000 concurrent users?"

As an example, figure 9.5 shows the average response time as a function of the number of concurrent users. The figure example compares the use of a database connection pool with 10 active connections to that of a single shared database connection.

Notice that with a single database connection the application cannot maintain a linear response time as the number of concurrent requests increases. That is, as more users attempt to use the application, their observed response times are elongated. In contrast, using a database connection pool allows the application to service requests to up to 25 users at a relatively constant response rate.

Figure 9.6 shows the throughput as a function of the number of concurrent users. This figure also compares the use of a database connection pool with 10 active connections to that of a single shared database connection.

Notice that, regardless of the number of concurrent users, the bottleneck caused by a single database connection limits the throughput to one catalog query per second—the application's maximum effective throughput. In contrast, by configuring the connection pool with 10 active connections, the application is able to consistently process almost 10 catalog queries per second. The application can scale to handle at least 25 concurrent users with only 10 shared connections.

Models such as these are great information radiators. You can look at them quickly and know how your application performs. Many performance testing

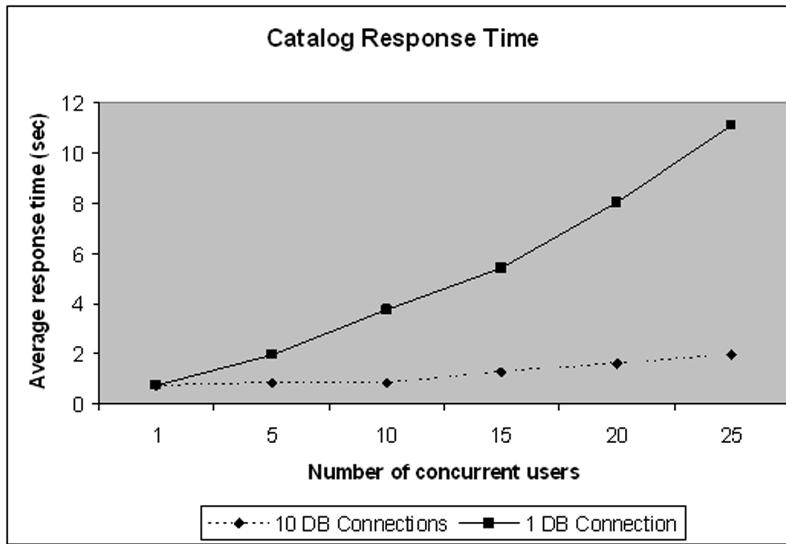

Figure 9.5 The use of a database connection pool, as indicated by the dotted line, yields a fairly constant response time for up to 25 concurrent users. With a single shared connection, as indicated by the solid line, the response time curve is exponential.

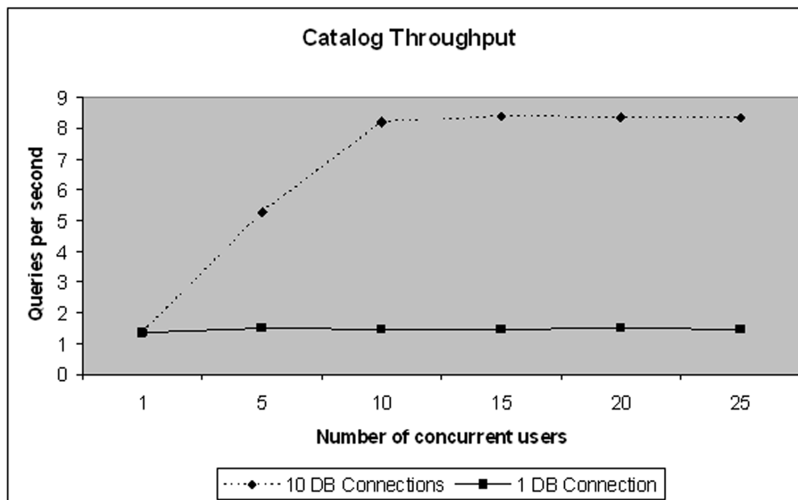

Figure 9.6 The use of a database connection pool, as indicated by the dotted line, delivers a throughput roughly equivalent to the number of active connections in the pool. With a single shared connection, as indicated by the solid line, the throughput bottoms out at one query per second.

tools, including JMeter, will automatically generate charts of this sort. Use them to your advantage.

9.9 *Mini-antipattern: Stage Fright*

If we don't test our application's performance early and often in a production-like environment, when the curtain goes up, the application may fall down in front of its live audience. Often we assume that, when an application meets its performance goals in development, it will perform equally well for its intended audience. We're usually disappointed.

To simulate realistic production traffic and usage patterns, we need to test our application's performance with representative data, tool versions, workloads, network latency, and hardware capacity. Tests that merely simulate users continually buying Chihuahuas from our online pet store won't cut it. We'll be in for an unwelcome surprise when a real user tries to buy a furry friend not already cached in the middle tier.

9.9.1 *Solution: Practice on stage*

To alleviate the fear and risk of embarrassment on stage, practice is our only recourse. Running performance tests in a production-staging environment as soon as possible, and keeping those tests running, will give us the confidence we need. The best approach to practicing for a production setting is to write tests that address our worst fears. What will happen when 10 users log in at the same time? We won't know until it happens, but we do know it's better to have it happen when we're practicing. Writing a passing login test under a 10-user load goes a long way toward boosting our confidence for the big show. In other words, tests let us safely play "what if" games with performance. By simulating a load, they can help us determine the amount of hardware we'll need to support our expected user load.

We need to give our application a dress rehearsal by testing under realistic scenarios. We'll use the same version of the virtual machine, application server, database, and other tools that will be deployed in the production environment. If caching and pooling is used to boost performance, then we can let the application warm up before running the performance tests. In other words, our tests should measure the actual performance, as observed by real users, to the maximum extent possible.

Once we've done our best to design and tune for performance under realistic loads, no substitute exists for tuning an application in production. Don't

underestimate the value of a tool that can monitor a live performance. Usage patterns in a live system may behave differently than expected.

9.10 Summary: Tuning with confidence

In this chapter, we looked at several antipatterns that commonly plague the EJB performance tuning process. In our quiet moments, when we're sure nobody is listening, we've probably all admitted to ourselves that we've been bitten by the need for speed. However, now that we've resolved to put speed in its place, we want to avoid these antipatterns by adopting an approach that's best summarized in the carpenter's motto: measure twice, cut once.

Before tuning to improve performance, we profiled our code to find hot spots. We then measured the performance again with a failing performance test written using JUnitPerf. Only after reviewing this evidence did we attempt to change code or the runtime environment to improve performance. We also put our trust in our gauges—automated tests that specify the performance requirements set by our customer. We leveraged these tests, using them as the qualifying measure of success, to ensure that our application's performance continually improved.

The methodology proposed for side-stepping the pitfalls encountered in this chapter is applicable to any type of performance tuning activity—EJB or otherwise. The bottom line: When making a case for or against applying changes in the name of performance, don't assume facts not already in evidence. Test first, then tune with confidence.

9.11 Antipatterns in this chapter

This section covers the Premature Optimization, Performance Afterthoughts, Thrash-tuning, Manual Performance Testing, and Stage Fright antipatterns.

PREMATURE OPTIMIZATION

DESCRIPTION

Good programmers frequently try to optimize every line of code and speculate in the name of performance, without considering which code/design elements are actually performance problems.

MOST FREQUENT SCALE

Application

REFACTORED SOLUTION NAME

Performance test automation

REFACTORED SOLUTION TYPE

Process, technology

REFACTORED SOLUTION DESCRIPTION

Use the simplest code/design that will work. Establish concrete criteria and run automated performance tests against the criteria to establish the need for performance tuning. Tune only problem areas. Write well-factored and modular code that's easy to tune later, if necessary.

ANECDOTAL EVIDENCE

"It works fine, but I suspected future performance problems so I spent the afternoon making it fast." "All of my code is a little tough to read, but it's very fast." "That design/technology is going to be too slow."

SYMPTOMS, CONSEQUENCES

Fewer development cycles are left for customer requirements or meaningful optimization when unforeseen problems arise. Design and code becomes unnecessarily complex and difficult to maintain. Functionality breaks when it's tweaked to be faster.

PERFORMANCE AFTERTHOUGHTS

DESCRIPTION

Attempting to bolt performance on to an application at the end of the development cycle rather than bake it in from the beginning

MOST FREQUENT SCALE

Application

REFACTORED SOLUTION NAME

Continuous performance planning

REFACTORED SOLUTION TYPE

Process

REFACTORED SOLUTION DESCRIPTION

Gather performance requirements early and often. Build automated performance tests that continuously validate performance criteria. Performance tests help to define exactly which areas do not meet criteria to focus testing efforts. Make any necessary course corrections throughout the project based on quantifiable measurements.

TYPICAL CAUSES

Poor planning

ANECDOTAL EVIDENCE

"We will have plenty of time to performance tune at the end of the development cycle." "It's a good design. We do not need to tune for performance." "We'll let our QA department measure performance." "We're using Enterprise JavaBeans, so it should scale well."

SYMPTOMS, CONSEQUENCES

Repeated delivery of poorly performing software, redesign of critical use cases late in the development cycle, and last-minute tuning activities that are ineffective

THRASH-TUNING

DESCRIPTION

Performance tuning is difficult without a solid baseline or when multiple configuration parameters are changed at once between measurements. Attempting performance tuning in these conditions makes it difficult to gauge progress and correct problems, lengthening the overall cycle time and giving the appearance of thrashing.

MOST FREQUENT SCALE

Application

REFACTORED SOLUTION NAME

Good performance methodology

REFACTORED SOLUTION TYPE

Process

REFACTORED SOLUTION DESCRIPTION

A sound performance testing methodology and a good testing environment are the primary keys. Baseline measurements are mandatory to gauge progress. All tests should start from a common configuration and changes should be made one at a time. Focus on performance problems demonstrated by failed tests.

ROOT CAUSES

Haste, inadequate performance testing tools

ANECDOTAL EVIDENCE

"It feels faster, don't you think?" "When are we done tuning?" "What did we change to make it slower?"

SYMPTOMS, CONSEQUENCES

Inefficient performance testing and tuning, longer than expected performance tuning cycles, and unclear results of performance improvements

MANUAL PERFORMANCE TESTING

DESCRIPTION

Manually running performance tests every time something is changed doesn't scale and the tests aren't easily repeatable

MOST FREQUENT SCALE

Organization

REFACTORED SOLUTION NAME

Automated performance testing

REFACTORED SOLUTION TYPE

Process, technology

REFACTORED SOLUTION DESCRIPTION

Use a performance testing tool like JUnitPerf to build automated tests. Let a computer run the tests continuously and consistently.

TYPICAL CAUSES

Inadequate performance testing tools

ANECDOTAL EVIDENCE

"I don't have time to run that test." "I can't repeat the results of the test from run to run."

SYMPTOMS, CONSEQUENCES

The time it takes to run tests manually increases the pressure to fall back into a thrash-tuning cycle. Performance problems aren't detected consistently.

STAGE FRIGHT

DESCRIPTION

Failure to test software in its production environment with representative data, tool versions, workloads, network latency, and hardware capacity

MOST FREQUENT SCALE

Application

REFACTORED SOLUTION NAME

Production environment testing

REFACTORED SOLUTION TYPE

Process

REFACTORED SOLUTION DESCRIPTION

Test application performance in settings as close as possible to the production environment.

TYPICAL CAUSES

Pride, ignorance

ANECDOTAL EVIDENCE

"We don't have time to test in production. The system is going live tomorrow!" "Don't worry. This is good code. It should work fine in the production environment." "It was fast on my development machine."

SYMPTOMS, CONSEQUENCES

Software that performs well in development environments, but fails miserably in production settings