TEST DRIVEN

Practical TDD and Acceptance TDD for Java Developers

BONUS CHAPTER

manning

LASSE KOSKELA

contents

13 Test-driving EJB components

Enterprise JavaBeans are all over the place!

—Yours truly, back in 2003

Enterprise applications have always been considered a demanding field, technology-wise. Everyone knows we need to have support for transactional processing, have high-availability properties, and be able to scale our systems by adding more hardware into our clusters. Ever since the new Millennium, the Java community's standard solution to these enterprise-grade requirements has been J2EE, or Java EE—Java Platform, Enterprise Edition.

Although it could be argued that a mass of other fundamental building blocks in the Java EE platform support fulfilling our enterprise-grade requirements, there's no denying that the most visible piece of the Java EE platform is its component model, Enterprise JavaBeans (EJB). For some reason, most enterprise applications developed in the corporate world in the beginning of the twenty-first century have made use of this technology. Whether those reasons had any basis or not, the fact is many corporate developers today are facing a legacy of Enterprise JavaBeans. The arrival of the latest revision of the EJB specification, version 3.0, has breathed new life into the technology and is contributing to the rebirth of EJB-based architectures.

In this chapter, we'll talk about both the 2.x series of the technology and the new EJB 3.0. Although few developers would prefer the old specification to the new revision, the application-server vendors have recently started supporting EJB 3 with their latest product releases. Knowing how some corporate IT departments are sluggish in upgrading their rubber-stamped technology stack, it'll be a while before EJB 3 becomes mainstream.

We're going to look into the future and focus on disarming EJB 3 with our test-driven mastery. Whenever we talk specifically about the 2.x version of the specification, I'll be sure to make a note.

We'll start our journey by exploring test-driven EJB development in the context of *session beans,* the most used class of Enterprise JavaBeans. Once we've got the basics in check, we'll discuss other types of Enterprise JavaBeans, such as *message-driven beans* and *entity beans*—or *entities,* as they're called in EJB 3 parlance.

Enterprise JavaBeans are here to stay, and that's why we're going to pull up our sleeves and get a grip on how to work with this technology in a test-driven way. Talking about enterprise applications doesn't mean we shouldn't be interested in producing flexible, testable designs and high-quality software. Let's start by learning what has traditionally made working with Enterprise JavaBeans problematic for test-infected developers.

13.1 The problem with testing managed objects

EJB is a managed object–based technology. Managed objects can be a boon for developer productivity in domains where applications can be written reliably without worrying about how the container—the system that manages the managed objects—instantiates, uses, and reuses our managed objects. There's nothing fundamentally wrong with managed objects. It's the implementation of the idea that sometimes causes developers to grow grey hair.

From a test-driven developer's point of view, the biggest issue with EJB as a technology has been that the blueprints for writing EJB components have not—at least, not until EJB 3—taken testability into consideration. Developers are facing a situation where the components are tightly coupled into the presence of the EJB container as they rely on the services of the container. Moreover, a key prerequisite for TDD is that we're able to run our unit tests quickly; having to spin up a heavy-weight container is hardly good news in that respect.

To contrast the domain of unit-testing managed objects with unit-testing regular Java objects, consider the usual structure of our tests. We typically start by instantiating the object under test. Then, we populate the object with any dependencies it requires, often using mock implementations rather than the real thing. Finally, we invoke the functionality we want to test, and perform assertions on the object's and the collaborators' states to verify correct behavior.

Now, let's think about the managed-objects scene. Managed objects are, by definition, managed by someone else—the container. We typically don't instantiate the object ourselves directly; rather, we ask the container to give us a reference to our managed object. Figure 13.1 illustrates this difference between managed and regular Java objects.

For the same reason, we also typically don't populate the dependencies on managed objects but instead rely on the container giving us a chance to do that from within the component. This is the case with EJB 2.x, where dependencies have traditionally been actively acquired by the object through lookup requests to

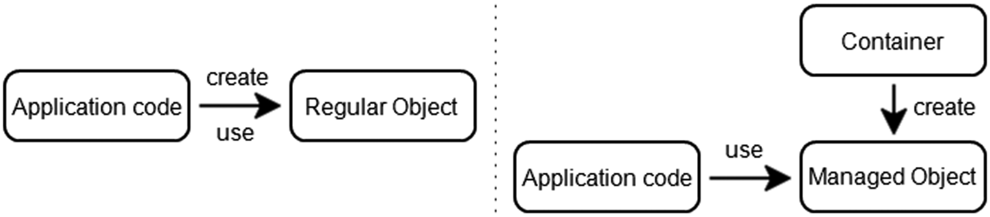

Figure 13.1 Distinction between container-managed objects and plain vanilla objects

the application server's Java Naming and Directory Interface (JNDI) tree. It's all happening in reverse compared to regular objects.

All these problems have solutions. EJB 3 incorporates many of these solutions, but those who are stuck with the legacy of EJB 2.x won't have to suck it up. Once you're finished with this chapter, you'll have a good understanding of the necessary tricks and workarounds in bending EJB technology to the test-driven method of developing software.

The most common type of EJB is the session bean, so let's start from there.

13.2 Conquering session beans

We want to test-drive session beans, which are objects managed by an EJB container. The container is of no interest to us while writing unit tests for the code, so let's get rid of the container as far as the unit tests are concerned. At some point, we're likely to want to do some integration testing to verify that the transaction settings and other related configurations are consistent with our intent, but right now we're talking about test-driving application functionality into our component. No container in the mix, please.

In this section, we'll see how to take the session beans out of the EJB container and survive in the standalone environment of a unit-testing framework. Let's begin by talking about how to simulate the lifecycle of session beans and how to write contract tests as a safety net to give an early warning when we're about to do something that's against the specification. I'll talk in terms of the EJB 3 specification here, but most of the techniques are directly applicable to the older EJB 2.x API too. I'll explicitly mention it when this isn't the case.

After breaking out of the chains of containment, we'll talk about *faking* dependencies—how to sneak test doubles into our session beans in our unit tests. We'll first discuss a couple of ways to intercept JNDI lookups performed by the session bean under test. This is most relevant for EJB 2.x session beans, which are expected to actively acquire their dependencies.

After working around JNDI lookups, we'll turn our attention to dependency injection as the alternative. Dependency injection is the way to go with the newer EJB 3 spec and is most relevant for EJB 3 session beans. The same technique can be used with the 2.x generation of the EJB specification, too—with a little extra effort, as we'll learn.

But now, let's get out of that enterprise jail cell—the EJB container.

13.2.1 Breaking out of the container

The good news with managed-object frameworks such as EJB is that the component lifecycle is documented—otherwise, developers wouldn't know how to implement the components properly. The documented lifecycle means we can often step into the shoes of the real container and *simulate* its function for the purpose of a unit test, invoking the lifecycle methods in the proper order and passing in test doubles as necessary for each test.

This is the case with EJB as well. Each type of EJB component (session beans, entity beans, and message-driven beans) has a set of lifecycle methods that the container promises to invoke under certain conditions. What's left to do is create the necessary test-double implementations of the associated `javax.ejb.*` interfaces—or let tools like EasyMock do that. Let's see what's needed to simulate the lifecycle.

Simulating the EJB 3 lifecycle

The EJB 3 specification defines several optional callback methods that a session bean class can implement. The callback methods are tagged with an annotation so the container can recognize them from regular business methods. The annotations for the main callback methods are called `@Init`, `@PostConstruct`, `@PrePassivate`, `@PostActivate`, and `@PreDestroy`, as shown in figure 13.2.

Methods annotated with `@Init` or `@PostConstruct` are called by the container immediately after the object is instantiated. A method annotated with `@PrePassivate` or `@PostActivate` is called just before or after a stateful session bean is passivated or activated, respectively. A method annotated with `@PreDestroy` is called

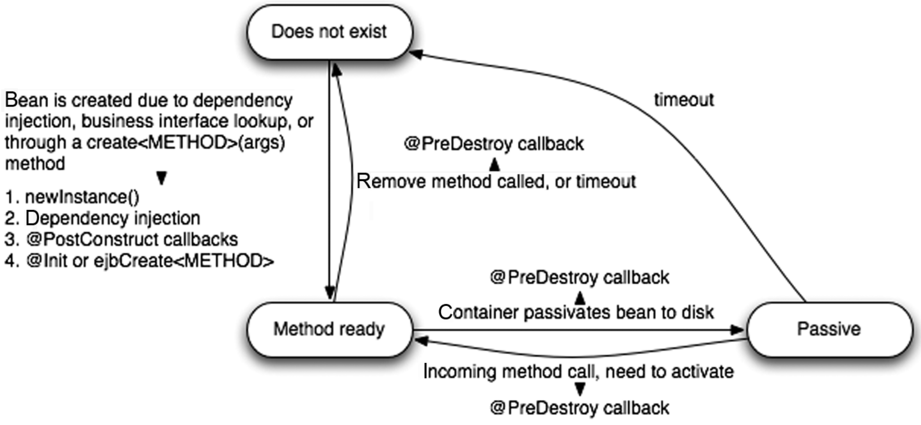
Does not exist
Bean is created due to dependency injection, business interface lookup, or through a create<METHOD>(args) method
1. newInstance()
2. Dependency injection
3. @PostConstruct callbacks
4. @Init or ejbCreate<METHOD>
@PreDestroy callback
Remove method called, or timeout
timeout
@PreDestroy callback
Container passivates bean to disk
Method ready
Passive
Incoming method call, need to activate
@PreDestroy callback

Figure 13.2 Simplified lifecycle of an EJB 3 session bean focusing on the lifecycle callback methods

right before the container removes the bean instance from its object pool. This is a simple contract between the container and the session bean and is straightforward to carry out in a unit test. We instantiate our bean class and invoke the appropriate lifecycle methods before and after invoking any business methods we want to test. With this in mind, we could write a test similar to listing 13.1 with the intention of test-driving a business method into an EJB 3 session bean.

Listing 13.1 Simulating container by invoking lifecycle methods on a session bean

```
@Test
public void discountIsCalculatedBasedOnVolumeOfPastPurchases()
        throws Exception {
    PricingServiceBean bean = new PricingServiceBean();
    bean.myInitMethod();
    bean.myPostConstructMethod();
    Product product = new MockProduct();
    Account account = new MockAccount();
    Price p = bean.discountedPrice(product, account);
    // ...
}
```

Invoke lifecycle methods

It's as simple as that. We invoke some methods, and no container is in sight! We have a clear contract that our bean class and the container vendor must fulfill, so we can use that contract as a natural boundary for isolating the unit under test from its production runtime environment—the EJB container.

Lifecycle methods in EJB 2.x

We can simulate the lifecycle similarly with the old EJB 2.x specification. The main difference is that although EJB 3 gives the developer the freedom to declare only those lifecycle methods they need (and with arbitrary names), the EJB 2.x specifications dictate that all the methods be declared with specific names such as `ejbCreate`, `ejbPostCreate`, and so forth.

If you're familiar with the EJB 3 specification, you're probably aware that beans are no longer required to implement any lifecycle methods. Plus, the lifecycle methods can be named virtually anything we want. Assuming that the test in listing 13.1 would be our first test for the `PricingServiceBean` we're developing and that we know the implementation won't need to perform any initialization in the `@Init`- or `@PostConstruct`-annotated methods in order to calculate a discount, why should we invoke those methods? Our test would be more compact if we omitted those method calls.

We could omit them. There's a point to be made about consistency, however. Considering that a later refactoring might move some code from our business methods into an initialization method, for example, it would be useful to know that all of our tests comply with the lifecycle defined in the specification and that the production code passes those tests.

Speaking of being consistent and complying with the specification, it might make sense to build just enough testing infrastructure to verify that the bean developer is abiding by the contract. Let's see how much of an effort that would be.

Automating the EJB 3 session bean contract

Those with experience in EJB development are bound to be familiar with the EJB container's aggravating complaint about the developer having missed an implicit contract, such as a constraint on a lifecycle method's signature that isn't enforced by the compiler. This is something we'd rather find out immediately during development than when we try to deploy the bean to a container. With that in mind, it makes sense to establish some automation in this field.

There are two main aspects where extra automation might come in handy. First, it would be useful if we had unit tests to verify those aspects of the contract between our beans and the container that the IDE or compiler doesn't catch. Second, it might be useful to encapsulate the invocation of lifecycle methods into common utility methods to avoid the arbitrary `NullPointerException` in situations where our tests mistakenly invoke the bean's methods differently than a standards-compliant container would, leaving the bean in an invalid state.

Listing 13.2 illustrates the first aspect in the form of a sketch of an abstract base class providing tests for verifying the correct usage of annotations in the bean class under test. These types of tests are generally referred to as *contract tests.*

Listing 13.2 Abstract base class contributing test methods for the session bean contract

```
public abstract class EJB3SessionBeanTestCase {

    protected abstract Class<Object> getBeanClass();
    @Test
    public void ejb3CompliantClassAnnotations() throws Exception {
        Class bean = getBeanClass();
        Annotation stateless = bean.getAnnotation(Stateless.class);
        Annotation stateful = bean.getAnnotation(Stateful.class);
        assertFalse("Only one of @Stateless or @Stateful allowed",
                (stateless != null && stateful != null));
    }

    @Test
    public void backwardsCompatibilityWithEjb2Interface()
```

Subclasses need to implement this

```
            throws Exception {
        if (SessionBean.class.isAssignableFrom(getBeanClass())) {
            assertAnnotation(PreDestroy.class, "ejbRemove");
            assertAnnotation(PostActivate.class, "ejbActivate");
            assertAnnotation(PrePassivate.class, "ejbPassivate");
            assertAnnotation(PostConstruct.class, "ejbCreate");
        }
    }

    @Test
    public void validNumberOfLifecycleAnnotations()
            throws Exception {
        assertMaxNumberOfAnnotations(1, PreDestroy.class);
        assertMaxNumberOfAnnotations(1, PostActivate.class);
        assertMaxNumberOfAnnotations(1, PrePassivate.class);
        assertMaxNumberOfAnnotations(1, PostConstruct.class);
    }

    // assertion methods' implementations omitted for brevity
}
```

With a base class like that in listing 13.2, we could declare our test class for a session bean to extend the base class, implement a one-liner method for returning the bean class we're testing, and inherit automatic checks for things like proper use of annotations.

Contract tests in EJB 2.x

Just as we can codify the contract for an EJB 3 session bean into a reusable base class, we can do the same for the EJB 2.x generation of beans. The main difference is, once again, that the EJB 2.x specifications nail down the contract in more detail (such as standard names for lifecycle methods and mandatory interfaces to implement).

Building these kinds of base classes may not be worth the effort if our application has only one or two beans. However, for a project making heavy use of Enterprise JavaBeans, it will pay off in improved speed of development, because certain types of errors are caught in our TDD cycle rather than during integration testing.

Off-the-shelf contract tests with Ejb3Unit

If you're not excited about writing your own contract tests, the open source project ejb3unit (http://ejb3unit.sf.net) offers a set of abstract base classes for putting your EJB 3 session beans and entities under contract tests.

Container as a single point of access

Removing the container from the picture by simulating the component lifecycle doesn't seem too bad. But what happens if our session bean needs to use container services, such as programmatic transactions (through the `javax.ejb.UserTransaction` interface), or look up something from the JNDI tree—such as a `javax.sql.DataSource`, another EJB, or a JMS queue or topic? The container is also the single point of access to such dependencies.

For example, we might want our `PricingServiceBean` in listing 13.1 to delegate the lookup for the purchase history of the given `Account` object—perhaps to an `AccountHistoryBean` class. Although the delegation of responsibility is a good design practice, these kinds of dependencies between beans have been annoying in the past when it comes to testing. Let's talk about dependencies and simple ways to alleviate the pain associated with them.

13.2.2 Faking dependencies

We can't discuss dependencies in the context of EJB without bumping into JNDI. JNDI, the mother of all trees in enterprise Java, is the one place to which all the various resources are bound and from which they can be obtained. For instance, every EJB component we've deployed as part of our application can be found from the application server's JNDI tree with a unique identifier—the JNDI name—either given explicitly in our configuration files or derived from the bean's class name.

Figure 13.3 illustrates a simple case of one EJB having a dependency to another within the same JNDI context. EJB #3 needs EJB #1 to fulfill its responsibility, whatever that is. Somehow, we need to be able to test beans that have such dependencies without the implementation of that dependency.

Managing and working around dependencies in our unit tests is such a big topic that we should draw a couple of lines in the sand before rushing ahead. The solution to many testability issues, as you've probably learned by now, is to add indirection or isolation and then exploit the new lever we just added to simplify our test—which, by association, has the same effect on our production code.

Speaking of isolation and indirection, we have a few main approaches from which to choose when it comes to refactoring our beans' dependencies to something more testable:

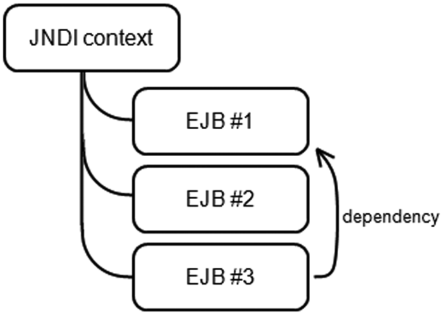

Figure 13.3 Enterprise JavaBeans and their dependencies in the JNDI tree

- *Fake the JNDI tree*—We can strive to keep the component we're testing as much of a black box as possible and focus our efforts on substituting the JNDI tree with one that's under our tests' control.
- *Override JNDI-related code*—We can cut the JNDI-related code out of the picture, leaving that portion of the code base to be tested as part of our integration test suite.
- *Reverse dependencies*—We can turn the dependencies around with something called *dependency injection*.

These approaches are as much about design decisions as they are about testing trickery. We'll see how much they affect our design as we look at each of these approaches in more detail. Let's start with faking the JNDI tree.

Faking the JNDI tree

If we want our tests knowing as little as possible about how—and when—the bean under test obtains its dependencies, one of our best options has been an open source project named MockEJB.[1] Among other things, the MockEJB project supplies us with a fake implementation of a JNDI tree. More accurately, it provides mock implementations of the `javax.naming.InitialContextFactory` and `javax.naming.Context` interfaces, which our beans indirectly use for connecting to the JNDI tree.

This means we can tell MockEJB to configure the current JVM runtime to use MockEJB's own mock implementation of the JNDI interfaces and start running our tests—during which the EJB being invoked will do JNDI lookups—knowing that what the bean will get from JNDI is exactly what we put there. What we put into the mocked-up JNDI tree is either plain old Java objects implementing the expected interfaces or mock objects we've created with tools like EasyMock.

Listing 13.3 shows a snippet from the MockEJB source code for the `MockContextFactory` class, the mock implementation of one of the key JNDI interfaces.

Listing 13.3 Snippet from MockEJB, substituting a mock JNDI implementation

```
public class MockContextFactory implements InitialContextFactory {

    private static Map savedSystemProps = new HashMap();
    ...

    public static void setAsInitial() throws NamingException {
        // persist original system properties
```

[1] http://www.mockejb.org.

```
        savedSystemProps.put(Context.INITIAL_CONTEXT_FACTORY,
                             System.getProperty(key));
        savedSystemProps.put(Context.URL_PKG_PREFIXES,
                             System.getProperty(key));
        // set system properties for mock JNDI implementation
        System.setProperty(Context.INITIAL_CONTEXT_FACTORY,
                           MockContextFactory.class.getName());
        System.setProperty(Context.URL_PKG_PREFIXES,
                           "org.mockejb.jndi");
    }

    public static void revertSetAsInitial() {
        Iterator i = savedSystemProps.entrySet().iterator();
        while(i.hasNext()) {
            Map.Entry entry = (Map.Entry) i.next();
            restoreSystemProperty((String) entry.getKey(),
                                  (String) entry.getValue());
        }
        rootContext = null;
    }
}
```

What we can see from listing 13.3 is that the `MockContextFactory` class gives us a static method called `setAsInitial` for replacing the current JNDI implementation with MockEJB's own. Similarly, a method called `revertSetAsInitial` reverts the system properties back to the way they were before calling `setAsInitial`. The key is that between calls to these two static methods, the following code *du jour* for connecting to the JNDI tree gets back an object talking to the MockEJB implementation:

```
MockContextFactory.setAsInitial();
...
// connect to JNDI, getting a mock rather than the real thing
Context mockedUpJndiContext = new InitialContext();
...
MockContextFactory.revertSetAsInitial();
```

Obviously, we'll want to ensure that the original system properties are reset after each test, which means we'll likely call these static methods from our tests' setup and teardown methods, respectively. Let's write a test case, shown in listing 13.4, which makes use of the `MockContextFactory`.

Listing 13.4 Using MockEJB and EasyMock to mock the JNDI tree

```
import javax.naming.*;
import org.junit.*;
import static org.junit.Assert.*;
import static org.easymock.EasyMock.*;
```

```
public class PricingServiceBeanTest {

    @Before
    public void setUp() throws Exception {
        MockContextFactory.setAsInitial();
    }

    @After
    public void tearDown() {
        MockContextFactory.revertSetAsInitial();
    }

    @Test
    public void discountBasedOnVolumeOfPastPurchases()
            throws Exception {
        Account account = new MockAccount();
        Product product = new Product("", 10000); // price: $100

        DiscountService ds = createMock(DiscountService.class);
        expect(ds.getDiscountPercentage(account)).andReturn(25);
        replay(ds);

        InitialContext context = new InitialContext();
        context.bind("example/DiscountServiceBean/local", ds);

        PricingServiceBean bean = new PricingServiceBean();
        bean.ejbCreate();
        Price price = bean.discountedPrice(product, account);

        assertEquals(new Price(7500), price);  // $100 - 25% = $75
        verify(ds);
    }
}
```

❶ **Use mock JNDI implementation**

❷ **Reset everything after each test**

❸ **Create mock of dependency**

❹ **Bind mock dependency to JNDI**

Listing 13.4 isn't rocket science. First, we ❶ surround our test methods with calls to the `MockContextFactory`'s `setAsInitial` and ❷ `revertSetAsInitial` methods in order to preserve isolation between tests. Then, inside our test method, we ❸ create a mock implementation of the dependency (`DiscountService`) using EasyMock and ❹ bind the mock object into the JNDI tree using the name we expect our bean under test to use to look it up.

From there, we invoke the code under test using a combination of real objects and mock objects, afterwards asserting that the outcome was what we expected and that the expected collaboration took place between the code under test and our mock objects.

With the test in listing 13.4 in place, the simplest implementation that will make the test pass would look something like listing 13.5.[2]

Listing 13.5 Implementation of the `PricingServiceBean`

```java
import javax.ejb.Stateless;
import javax.naming.InitialContext;

@Stateless
public class PricingServiceBean implements PricingService {

    public Price discountedPrice(Product prod, Account account) {
        try {
            InitialContext ctx = new InitialContext();
            DiscountService ds = (DiscountService) ctx
                    .lookup("example/DiscountServiceBean/local");

            int percentage = ds.getDiscountPercentage(account);
            float multiplier = (100 - percentage) / 100.0f;
            return new Price((int) (prod.getPrice() * multiplier));
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Look up dependency

Use dependency

All we need to do is acquire the `DiscountService` bean (which happens to be the mock implementation), ask it for an account-specific discount percentage, and adjust the product's list price accordingly. It's simple, and that's what we want from an example focused on dealing with dependencies. Having a snippet of code to look at is also helpful, considering what we're going to do next: ignore the JNDI lookup in our tests.

Ignoring JNDI lookups

In listing 13.5, we saw an implementation of the `discountedPrice` business method on our `PricingServiceBean`. We faked the JNDI implementation to provide the bean under test with its dependencies. Although using the `MockContextFactory` and binding things to the JNDI tree in the tests was hardly a laborious task, it's not the only trick. Focus on getting the functionality into place rather than thinking about the JNDI name with which the dependencies should be bound to the JNDI tree. In those cases, *extract and override.*

[2] Please don't round numbers like this when implementing systems for the bank where I keep my money.

Let's take another glance at listing 13.5. Two steps are involved: one to obtain the dependency and one to use the dependency for implementing the business functionality. Why don't we refactor that method by pulling the first part into a separate method? The result is presented in listing 13.6.

Listing 13.6 Extracting the acquisition of a dependency into a separate method

```
public Price discountedPrice(Product product, Account account) {
    try {
        DiscountService discounts = getDiscountService();
        int discount = discounts.getDiscountPercentage(account);
        float multiplier = ((100 - discount) / 100.0f);
        return new Price((int) (product.getPrice() * multiplier));
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

protected DiscountService getDiscountService()
        throws NamingException {
    String name = "example/DiscountServiceBean/local";
    return (DiscountService) new InitialContext().lookup(name);
}
```

Use getter to acquire dependency

This refactoring not only improves the code by increasing the cohesion within `discountedPrice` but also gives us the option of writing the tests by overriding the `getDiscountService` method rather than faking the JNDI. To illustrate this approach, look at listing 13.7, which shows how we might have written our test for the `discountedPrice` functionality had we wanted to ignore the JNDI lookup rather than fake the JNDI tree.

Listing 13.7 Test that would've pushed our implementation toward listing 13.6

```
public void discountBasedOnVolumeOfPastPurchases()
        throws Exception {
    Account account = new MockAccount();
    Product product = new Product("", 10000); // list price: $100

    final DiscountService ds = createMock(DiscountService.class);
    expect(ds.getDiscountPercentage(account)).andReturn(25);
    replay(ds);

    PricingServiceBean bean = new PricingServiceBean() {
        @Override
        protected DiscountService getDiscountService() {
            return ds;
```

Create mock DiscountService as before

Override getter instead of binding mock to JNDI

```
        }
    };
    Price price = bean.discountedPrice(product, account);
    assertEquals(new Price(7500), price);  // $100 - 25% = $75
    verify(ds);
}
```

Override getter instead of binding mock to JNDI

The advantage with this approach is that the test is slightly simpler than the one in listing 13.4, where we initialized a mock JNDI tree and bound the mock dependency into the tree. In this example, we use fewer lines of code—mostly because of not having to initialize and reset the mock JNDI implementation in the setup and teardown methods. The downside is we're not testing that the proper JNDI lookup is performed. Also, the test knows too much about the bean implementation's details—the existence of a getter method—for my taste.

Having said that, the JNDI lookup is a trivial piece of code that could be considered something that can't break; remembering that we'll have automated integration and system tests, this might be an acceptable trade-off. We could still write a test for the getter only, verifying that the JNDI lookup happens. It's largely a matter of personal preference, and balancing these pros and cons is ultimately left to us.

There's a third approach to dealing with dependencies, which helps us avoid such a trade-off. The magic words are familiar from chapter 4: *dependency injection*.

Injecting dependencies with EJB 3

The previous two techniques, faking the JNDI tree and overriding JNDI lookups, are valid whether our beans are written against the 2.x or 3 version of the EJB specification. The third technique, dependency injection, is the one to go with if we're using EJB 3. That's how the specification suggests we handle dependencies! But dependency injection is worth serious consideration even if we're using EJB 2.x.

The fundamental reason for having to fake the JNDI tree or override getter methods to substitute dependencies is that the bean under test is actively acquiring its dependencies. Now, consider a situation where the bean would instead expect the dependencies to be handed to it by someone else. If this was the case, we wouldn't have the problem in the first place. That's what dependency injection is all about—moving the responsibility for obtaining dependencies outside of the component.

As I hinted already, EJB 2.x versus 3 is important when we're talking about Enterprise JavaBeans and dependency injection. In EJB 2.x, the specification leaves little room for integrating a dependency-injection framework such as the Spring Framework or PicoContainer. EJB 3, on the other hand, has taken advantage of the state of the art in the open source world and incorporated dependency injection into the specification as the preferred way for Enterprise

JavaBeans to acquire their dependencies. An EJB 3 container *is* a dependency injection framework.

Let's refresh our memory of how dependency injection works. Three things happen:

1. The component advertises its dependencies, typically in terms of interfaces or named services.
2. The component is registered to a dependency injection container along with other components.
3. Knowing the capabilities of all registered components, the container makes sure each advertised dependency gets injected with a matching implementation.

In the case of Enterprise JavaBeans, the EJB 3 container acts as a dependency injection container. Beans advertise their dependencies either through metadata annotations on setter methods or member variables or, alternatively, in the external deployment descriptor. In EJB 2.x, it's more complex. Listing 13.8 shows how we could (and should!) rewrite the earlier discounted price test, knowing we have the dependency-injection features of EJB 3 at our disposal.

Listing 13.8 Using dependency injection to populate dependencies

```
@Test
public void discountBasedOnVolumeOfPastPurchases()
        throws Exception {
    Account account = new MockAccount();
    Product product = new Product("", 10000); // list price: $100

    DiscountService ds = createMock(DiscountService.class);
    expect(ds.getDiscountPercentage(account)).andReturn(25);
    replay(ds);

    PricingServiceBean bean = new PricingServiceBean();
    bean.setDiscountService(ds);

    Price price = bean.discountedPrice(product, account);
    assertEquals(new Price(7500), price);  // $100 - 25% = $75
    verify(ds);
}
```

Call setter like container would

The simplicity of calling a setter is hard to compete with. Constructor-based dependency injection is arguably even simpler than this. However, that comes with the trade-off of not being able to substitute the dependency after construction, which makes it more difficult to organize test code where we'd like to use a different mock implementation for different test methods.[3] It's up to us to choose what fits our current need the best.

The change toward simpler code with the adoption of dependency injection is obvious on the implementation's side as well, as we can see in listing 13.9.

Listing 13.9 Bean class getting the dependency through a setter

```
@Stateless
public class PricingServiceBean implements PricingService {

    private DiscountService ds;

    @EJB(businessInterface = DiscountService.class)
    public void setDiscountService(DiscountService impl) {
        this.ds = impl;
    }

    public Price discountedPrice(Product prod, Account account) {
        try {
            int percentage = ds.getDiscountPercentage(account);
            float multiplier = (100 - percentage) / 100.0f;
            return new Price((int) (prod.getPrice() * multiplier));
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

We could go even further in simplifying our production code by employing field injection rather than introducing a setter. All we'd need to do would be to move the `@EJB` annotation on the `private` field and vanquish the setter method. However, that would mean that in our test we'd need to resort to using the Reflection API for populating the field—just like the EJB container would when facing such an annotation on a `private` field. Another option would be to make the field more visible. My personal preference is to not make fields `public`, so I generally recommend using either setters or reflection to inject to `private` fields.

Although the plumbing necessary for injecting to a `private` field requires some work, there's another point to consider that might tilt us toward going for field injection (regardless of whether we're talking about `private`, `protected`, or

[3] Then again, that might be an indicator of the two tests belonging to two different test classes.

public fields): By using a utility method to inject dependencies to a named field, we can also check for the presence of the appropriate dependency-injection annotation. For example, the utility method in listing 13.10 alerts us to add a missing @EJB or @Resource annotation on a field we're injecting to from our test.

Listing 13.10 Injecting a dependency using the Reflection API

```
/**
 * Inject the given dependency into the named field on the
 * target object, also verifying that the field has been
 * annotated properly.
 */
public static void injectField(Object target, String fieldName,
                               Object dep) throws Exception {
    Field field = target.getClass().getDeclaredField(fieldName);
    Annotation ejb = field.getAnnotation(EJB.class);
    Annotation res = field.getAnnotation(Resource.class);
    boolean wasFound = (ejb != null || res != null);
    assertTrue("Missing @EJB or @Resource annotation", wasFound);
    field.setAccessible(true);
    field.set(target, dep);
}
```

❶ Verify field has appropriate annotation

❷ Force (possibly private) field to be accessible and inject dependency

The injectField method in listing 13.10 isn't too shabby. It ❶ ensures that the field to which we inject has either an @EJB or a @Resource annotation, and it ❷ performs the injection.

Injecting dependencies with EJB 2

What about the EJB 2.x specification? How can we use dependency injection with EJB 2.x session beans? The injection of dependencies in our tests works the same as with EJB 3 (except we wouldn't check for the presence of an annotation). The question is how the injection takes place when we deploy our code to a real EJB container.

I've done this by having the appropriate lifecycle method (typically ejbCreate) look up the dependencies and inject them through a setter or directly into a private field. Our contract with the EJB container effectively guarantees that our dependency injection will take place when running in a container, and we still have the benefit of being able to inject dependencies ourselves in a test. Nice!

There's one more thing. How do we know the lifecycle method is injecting the dependencies correctly? We write a test for the expected behavior, of course. This test invokes the lifecycle method—either faking the JNDI tree or overriding the lookups—and asserts the correct objects were populated into the right places. Listing 13.11 shows this test.

Listing 13.11 Testing that an EJB 2.x session bean injects dependencies

```
@Test
public void dependencyInjectionShouldHappenInEjbCreate() {
    final Dependency fakeDependency = new FakeDependency();
    SomeEJB2xSessionBean bean = new SomeEJB2xSessionBean() {
        @Override
        protected Object lookupDependencyFromJNDI() {
            return fakeDependency;
        }
    };
    assertNull(bean.dependency);
    bean.ejbCreate();
    assertSame(fakeDependency, bean.dependency);
}
```

❶ **Override JNDI lookup**

❷ **Dependency should be there after ejbCreate()**

In listing 13.11, an EJB 2.x–style session bean should obtain a dependency of type `Dependency` by performing a JNDI lookup when the container invokes the `ejbCreate` callback method. We codify this scenario into a test by ❶ overriding the JNDI lookup to return a fake implementation of the dependency and by ❷ checking that the same object gets assigned to the expected field during the invocation of `ejbCreate`. Another option would've been to let the bean do the lookup and fake the JNDI tree instead. Whichever way we go, dependency injection isn't difficult to adopt for EJB 2.x, although it's easier in EJB 3.

That's all we need to know about dealing with the dependencies our Enterprise JavaBeans may have. Knowing these tools and tricks, we can write tests that are independent of the real container and run blazing fast, and verify the bean under test collaborates with its dependencies as expected. And it's all simple enough to let us write the test first!

We've been talking a lot about session beans—the all-round workhorse of the EJB specification—but that's not all there is to Enterprise JavaBeans. Next, we'll jump to the asynchronous world of message-driven beans and the Timer service. No need to fret; the asynchronous nature doesn't show much in the tests. Let's move on before we've spilled the beans on it all. (Pun intended.)

13.3 Commanding asynchronous services

Although session beans represent the majority of Enterprise JavaBeans usage, they are inherently about synchronous computing. Enterprise systems, however, often have requirements that need asynchronous execution where multiple threads converse with each other, as illustrated in figure 13.4.

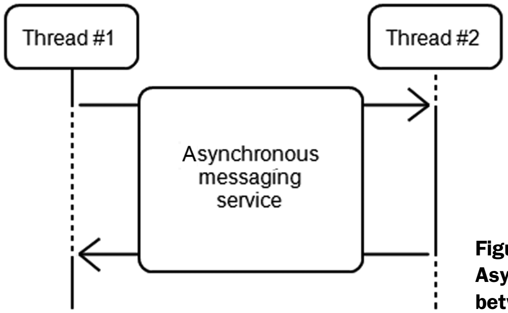

Figure 13.4
Asynchronous execution introduces time between actions as a new variable.

The existence of legacy systems over several decades might call for a messaging bridge or a message bus, the nature of certain messages might call for a resequencer queue, and so forth. There is a host of reasons for introducing asynchronous messaging concepts like queues and topics into our application, and enterprise Java developers can't avoid encountering messaging-related and event-based technologies.

We're building on top of the Java Messaging System (JMS) API. JMS is a standard interface through which application developers can connect to the underlying messaging system implementation. Although JMS is a simple API, we must manage connections to the messaging system and other related resources. This is where message-driven beans (MDB) come into play: They let a developer focus on the application functionality, leaving the nasty details of resource management to the EJB container.

It's time to pull up our sleeves again and test-drive some asynchronous components.

13.3.1 Message-driven beans

Message-driven beans are possibly the simplest kind of Enterprise JavaBeans. This is because the interface (`javax.jms.MessageListener`) consists of one simple method:

```
public void onMessage(javax.jms.Message msg);
```

The only parameter to the `onMessage` method, `javax.jms.Message`, is an interface that's easy to fake—it's a map of key-value pairs—and a number of readily available mock implementations exist.[4]

[4] The MockEJB framework comes with one, for example.

What do message-driven beans do in the `onMessage` method? They typically process the message contents and either manipulate the system state (either by connecting to a database directly or by invoking business logic through session beans) or pass the message forward to the next queue or topic for further processing. That's not an issue, though, because we've just learned how mocking dependencies is child's play when we know the right tricks.

I'm sure you're itching to see some action, so let's look at a sample scenario for which we want to implement an MDB.

An example of asynchronous messaging

Say we're developing a search engine that works on a large, multi-gigabyte data set sitting on a remote file system where a single search might take up to 60 seconds to complete. Let's also say the system has thousands of simultaneous users. Such a scenario would be good for performing the search operation asynchronously. Figure 13.5 describes the workflow we've designed for handling the search operations using a JMS queue and message-driven beans.

Our design in figure 13.5 says we'll have two queues—one for delivering incoming search requests and one for delivering the results of completed searches. When a new search request comes through to the searches queue, the EJB container allocates an instance of our message-driven bean to consume the message. Our bean is then expected to execute the search request synchronously using a session bean; once the session bean generates the results, our bean sends a new message containing the search results to the results queue.

Now that we have an idea of what we want to build, where should we start? We'll need an instance of the message-driven bean, so let's figure out how to instantiate the bean class.

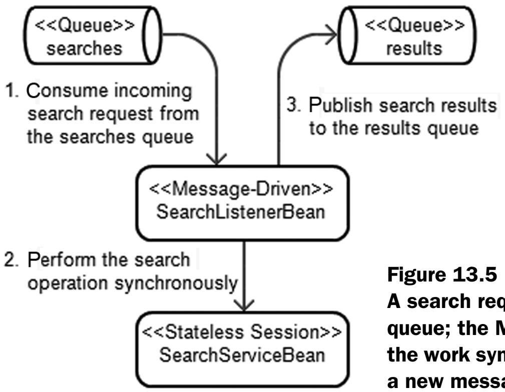
<<Queue>>
searches
<<Queue>>
results
1. Consume incoming search request from the searches queue
3. Publish search results to the results queue
<<Message-Driven>>
SearchListenerBean
2. Perform the search operation synchronously
<<Stateless Session>>
SearchServiceBean

Figure 13.5
A search request message comes in to the searches queue; the MDB consumes the message, delegates the work synchronously to a session bean, and sends a new message to the results queue.

Just create one

Sound familiar? Just like with session beans, there's a strong case for the "just create one" approach to leaving the container out of the picture when test-driving message-driven beans. The real EJB container might dynamically generate and use a subclass of the bean class at runtime, but it's sufficient for our purposes to create an instance of the bean class with the `new` operator, call the appropriate lifecycle methods if applicable, and invoke the functionality.

What next? We have an initialized instance of the message-driven bean, but we're not specifying what we expect it to do; let's set up some expectations using mock objects, invoke the business logic, and verify that the bean does what we want.

First, we need a bunch of mock objects for setting up the expected collaboration between the message-driven bean and the JMS API. Listing 13.12 shows the fixture and the associated setup code for the test class.

Listing 13.12 The fixture for our first message-driven bean test

```
import javax.jms.*;
import org.junit.*;
import static org.easymock.EasyMock.*;

public class SearchListenerBeanTest {

    private SearchService searchService;
    private QueueConnectionFactory factory;
    private QueueConnection connection;
    private QueueSession session;
    private QueueSender sender;
    private Queue queue;
    private ObjectMessage resultMessage;
    private ObjectMessage searchMessage;

    @Before
    public void setUp() {
        searchService = createMock(SearchService.class);
        factory = createMock(QueueConnectionFactory.class);
        connection = createMock(QueueConnection.class);
        session = createMock(QueueSession.class);
        sender = createMock(QueueSender.class);
        queue = createMock(Queue.class);
        resultMessage = createMock(ObjectMessage.class);
        searchMessage = createMock(ObjectMessage.class);
    }
}
```

There are a lot of mock objects—proof that the JMS API is deep in terms of its object hierarchy. The good news is the test itself is nothing special. We have mock objects that we pass to the production code being tested, and we expect certain method calls to occur. Listing 13.13 shows the test method where we set up our expectations, instantiate the bean, and invoke the bean's onMessage method.

Listing 13.13 A test for our message-driven bean's business logic

```
public class SearchListenerBeanTest {
    ...

    @Test
    public void searchRequestIsDelegatedAndResultsPassedForward()
            throws Exception {
        // non-mock input objects passed to the production code
        String[] keywords = new String[] { "keyword1", "keyword2" };
        String[] results = new String[] { "match1", "match2" };

        // expect the MDB to retrieve search request contents
        expect(searchMessage.getObject()).andReturn(keywords);

        // expect a call to the SearchService EJB
        expect(searchService.search(aryEq(keywords)))
                .andReturn(results);

        // expect MDB to submit search results to results queue
        expect(factory.createQueueConnection())
                .andReturn(connection);
        expect(connection.createQueueSession(
                        false, Session.AUTO_ACKNOWLEDGE))
                .andReturn(session);
        expect(session.createSender(queue))
                .andReturn(sender);
        expect(session.createObjectMessage(aryEq(results)))
                .andReturn(resultMessage);
        sender.send(resultMessage);
        connection.close();

        // done recording expectations...
        replay(searchService, factory, connection, session, queue,
                sender, resultMessage, searchMessage);

        // ...ready to invoke business logic
        SearchListenerBean bean = new SearchListenerBean();
        bean.searchService = searchService;
        bean.connectionFactory = factory;
        bean.resultsQueue = queue;
        bean.onMessage(searchMessage);
```

Expectations for call to SearchService ❶

Expectations for submitting results to JMS queue ❷

Create one, inject dependencies, and invoke onMessage() ❸

```
        // verify expectations
        verify(searchService, factory, connection, session, queue,
                sender, resultMessage, searchMessage);
    }
}
```

We ❶ declare that we expect the bean to acquire the message payload—a `String` array—from the incoming JMS message and pass the `String` array to the `SearchService`. Then, in ❷ we proceed to recording expectations for the sending of search results as an `ObjectMessage` to the results JMS queue. After moving all mock objects to replay mode, we ❸ instantiate the bean class, inject dependencies—the mock implementations for the `SearchService` and the JMS resources—and invoke the bean's `onMessage` method. Finally, we verify that the expected collaboration took place.

Listing 13.14 shows the implementation that satisfies our test.

Listing 13.14 Implementation of the message-driven bean

```
import javax.annotation.*;
import javax.ejb.*;
import javax.jms.*;

@MessageDriven(activationConfig = {
        @ActivationConfigProperty(propertyName="destinationType",
                                  propertyValue="javax.jms.Queue"),
        @ActivationConfigProperty(propertyName="destination",
                                  propertyValue="queue/testQueue")
})
public class SearchListenerBean implements MessageListener {

    @EJB(businessInterface=SearchService.class)
    public SearchService searchService;

    @Resource(name="QueueConnectionFactory")
    public QueueConnectionFactory connectionFactory;

    @Resource(name="resultsQueue", type=javax.jms.Queue.class)
    public Queue resultsQueue;

    public void onMessage(Message message) {
        try {
            ObjectMessage request = (ObjectMessage) message;
            String[] keywords = (String[]) request.getObject();
            String[] results = searchService.search(keywords);
```

Expose dependencies for injection

Call SearchService

```
            QueueConnection qc =
                    connectionFactory.createQueueConnection();
            QueueSession qs = qc.createQueueSession(
                    false, Session.AUTO_ACKNOWLEDGE);
            QueueSender sender = qs.createSender(resultsQueue);
            message = qs.createObjectMessage(results);
            sender.send(message);
            qc.close();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Publish search results

This test, although not difficult to follow, is verbose. Proceeding with further tests (to add proper handling of JMS exceptions, for example), we could refactor the setting of expectations into private methods on the test class. That would simplify our tests to a more compact form. We should also consider extracting the different sections of the `onMessage` implementation into separate methods, making our production code more pleasant to the eye. I'll leave such refactorings as an exercise for you, because I have a lot to discuss regarding the services provided by the EJB specification.

Using an embedded JMS broker

It's possible to use a lightweight, embedded JMS broker instead of mock objects when testing code that needs to use JMS interfaces. The Apache ActiveMQ project provides such an implementation.

Although I've left rewriting the test in listing 13.13 as an exercise for you, listing 13.15 presents an abstract base class that can be used as the basis of such tests.

Listing 13.15 Abstract base class for tests using the embedded ActiveMQ JMS broker

```
import java.util.logging.*;
import javax.jms.*;
import org.junit.*;
import org.apache.activemq.ActiveMQConnectionFactory;

public abstract class AbstractJmsTestCase {

    protected QueueConnectionFactory queueConnectionFactory;
    protected QueueConnection queueConnection;
    protected QueueSession queueSession;

    protected TopicConnectionFactory topicConnectionFactory;
    protected TopicConnection topicConnection;
    protected TopicSession topicSession;

    @Before
    public void setUpActiveMQ() throws Exception {
        disableActiveMqLogging();
        ActiveMQConnectionFactory factory =
                new ActiveMQConnectionFactory(
                        "vm://localhost?broker.persistent=false");
        setupQueueConnection(factory);
        setupTopicConnection(factory);
    }

    @After
    public void tearDownActiveMQ() throws Exception {
        queueConnection.stop();
        queueConnection.close();
        topicConnection.stop();
        topicConnection.close();
    }

    private void setupQueueConnection(QueueConnectionFactory f)
            throws Exception {
        queueConnectionFactory = f;
        queueConnection = f.createQueueConnection();
        queueConnection.start();
        queueSession = queueConnection.createQueueSession(false,
                Session.AUTO_ACKNOWLEDGE);
    }

    private void setupTopicConnection(TopicConnectionFactory f)
            throws Exception {
        topicConnectionFactory = f;
        topicConnection = f.createTopicConnection();
        topicConnection.start();
        topicSession = topicConnection.createTopicSession(false,
                Session.AUTO_ACKNOWLEDGE);
    }

    private void disableActiveMqLogging() {
        Logger.getLogger("org.apache.activemq").setLevel(
                Level.WARNING);
    }
}
```

This class provides concrete subclasses with instances of the key JMS interfaces for dealing with either pub-sub (topic) or producer-consumer (queue) type of connections. The embedded ActiveMQ JMS broker is reasonably fast—my laptop runs 10 simple integration tests against ActiveMQ in one second. Not blazing fast, but acceptable if we have only a handful of JMS-related tests.

Before moving on to discussing entity beans and EJB 3 persistence, let's look at one more type of asynchronous service. JMS and message-driven beans aren't the only type of asynchronous service available to the enterprise Java developer. The EJB 2.1 specification introduced the Timer service, a facility for scheduling one-off and repeating events, optionally with an initial delay. Let's look at how the Timer service works and an example of test-driving a piece of code that uses this service.

13.3.2 Working with Timer service API

With the introduction of the Timer service in EJB 2.1, enterprise Java developers got a standard facility for executing business logic based on scheduled events or on repeating intervals. The Timer service is made available to EJB components through the `EJBContext` interface (as well as through `SessionContext`, `EntityContext`, and `MessageDrivenContext`, which extend `EJBContext`). We ask the `EJBContext` object for a `TimerService` interface and then ask the `TimerService` to create a `Timer`. In EJB 3, the `TimerService` can also be injected directly into a `private` field of a managed object through the `@Resource` annotation.

The `TimerService` implementation is aware of the identity of the EJB instance creating the timer. After the timer's timeout interval, the EJB container takes care of obtaining a reference to the EJB instance and invokes its `ejbTimeout` method, passing the `Timer` object as the sole argument. Figure 13.6 illustrates these events.

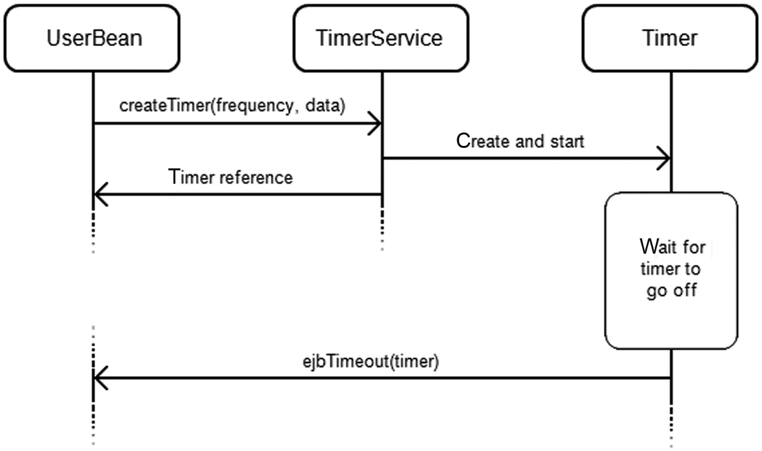

Figure 13.6 The `TimerService` creating a `Timer` that performs a callback when it times out

For example, consider a scenario where we have an entity bean representing a user account. Call the bean class UserBean. Now, implement a password-expiry feature by letting the UserBean instances set a password-expiry timer every time the bean's password field is changed. Such a timer is started when a new UserBean instance is created. When the timer times out, the bean sets a flag indicating that the password has expired and should be changed upon next login.

Say our UserBean is an EJB 2.1 container-managed persistence (CMP) type of an entity bean, meaning we don't write any JDBC code but rather let the EJB container generate that plumbing for us. For now, we're only interested in test-driving the timer logic in place to our entity bean, so we'll ignore most of the lifecycle methods required from the entity bean and focus on the handful we need to set up our timers.

Time to get busy.

Setting expectations

I've decided to start small. We're first going to write a test for the initial timer, which is started when a new bean instance (a new user account) is created. We've determined that ejbPostCreate (or @PostConstruct if we're using EJB 3) is a suitable lifecycle method for doing that. Knowing that the ejbPostCreate method will need to obtain a TimerService from an EntityContext and a Timer from the TimerService, we set out to write a skeleton for our first test using the EasyMock framework for obtaining mock implementations of these three interfaces. The code is shown in listing 13.16.

Listing 13.16 Expected collaboration toward the TimerService

```
import static org.easymock.classextension.EasyMock.*;
import org.junit.*;
import javax.ejb.*;

public class PasswordExpiryTimerTest {

    @Test
    public void timerIsCreatedUponCreatingNewUserAccount()
            throws Exception {
        EntityContext context = createMock(EntityContext.class);
        TimerService timers = createMock(TimerService.class);
        Timer timer = createMock(Timer.class);

        expect(context.getTimerService()).andReturn(timers);
        expect(timers.createTimer(
                BunchOfConstants.TWO_MONTHS, "password expired"))
                .andReturn(timer);
        replay(context, timers, timer);

        // TODO: invoke ejbPostCreate()

        verify(context, timers, timer);
    }
}
```

We expect the `EntityContext` mock to receive a call to `getTimerService` and the returned `TimerService` mock to receive a call to `createTimer`, with the parameters indicating that we expect the bean to set the timer to use a two-month delay before triggering. After recording the expected collaboration toward the `EntityContext` and `TimerService` interfaces, we move our mock objects to replay mode, invoke `ejbPostCreate`, and then ask our mock objects to verify that the expected collaboration took place.

The stuff that should happen between the `replay` and `verify` calls doesn't exist yet. Let's see how we can realize the comment into Java code.

Instantiating the bean under test

The next thing to do is figure out how to instantiate the entity bean. We want the bean to use container-managed persistence, which means—according to the EJB specification—our bean class must be abstract and enumerate all persistent fields as pairs of abstract getter and setter methods. After briefly sketching on a piece of paper, we reach a decision: Our user bean needs to have a primary key, a username field, a password field, and a `boolean` field for storing the knowledge of whether the password has expired. For the purposes of our test, we're only interested in the "password expired" field. So, for now, treat `UserBean` as a regular Java class.

Listing 13.17 shows the remainder of our test, invoking the `ejbPostCreate` method, which should trigger the creation of a new timer.

Listing 13.17 Expected collaboration toward the `TimerService`

```
import static org.easymock.classextension.EasyMock.*;
import org.junit.*;
import javax.ejb.*;

public class PasswordExpiryTimerTest {

    @Test
    public void timerIsCreatedUponCreatingNewUserAccount()
            throws Exception {
        ...
        replay(context, timers, timer);
        UserBean entity = new UserBean();
        entity.setEntityContext(context);
        entity.ejbPostCreate("someusername", "somepassword");

        verify(context, timers, timer);
    }
}
```

There! A failing test. EasyMock is complaining that nobody's asking our `EntityContext` for a `TimerService` and, thus, nobody's asking the `TimerService` to create any timers. It's time to write some production code to make our test pass. Listing 13.18 shows the simplest and yet most sufficient implementation that comes to mind.

Listing 13.18 Letting `ejbPostCreate()` create a `Timer`

```
public class UserBean {

    private EntityContext ctx;

    public void setEntityContext(EntityContext ctx) {
        this.ctx = ctx;
    }

    public void ejbPostCreate(String username, String password) {
        ctx.getTimerService().createTimer(
                BunchOfConstants.TWO_MONTHS, "password expired");
    }
}
```

We know the `UserBean` class should be abstract, implement the `javax.ejb.EntityBean` interface, and so forth, in order to be a valid entity bean implementation, according to the EJB 2.1 specification. Consider writing some contract tests for checking such things for our entity beans. Leaving that thought behind for a while, we set out to tackle the other half of creating a timer—the stuff that happens when the timer goes off.

Triggering the timer

In practice, we're looking for a call to set a `passwordExpired` field to `true` when the timer goes off (when the EJB container invokes the bean's `ejbTimeout` method). The simplest way to express our intent is to declare an anonymous subclass of the `UserBean` in our test method and override the setter from the parent class to set our `boolean` field. Listing 13.19 shows this approach in action.

Listing 13.19 Test for setting the flag when the timer goes off

```
public class PasswordExpiryTimerTest {

    protected boolean passwordExpired;

    @Before
    public void setUp() {
        passwordExpired = false;
    }

    // ...

    @Test
    public void flagIsSetWhenTimerGoesOff() throws Exception {
        UserBean entity = new UserBean() {
            @Override
            public void setPasswordExpired(boolean expired) {
                passwordExpired = expired;
            }
        };
        Assert.assertFalse(passwordExpired);
        entity.ejbTimeout(null);
        Assert.assertTrue(passwordExpired);
    }
}
```

Noticing the @Override annotation on the setPasswordExpire method, the compiler prompts us to add the missing setPasswordExpired to our UserBean class. The ejbTimeout method doesn't exist yet, so we'll need to add that too. In a short time, we have a failing test and permission to write production code. Listing 13.20 shows the full implementation.

Listing 13.20 Implementation for setting the "password expired" field upon timeout

```
public class UserBean {

    private EntityContext ctx;

    public void setEntityContext(EntityContext ctx) {
        this.ctx = ctx;
    }

    public void setPasswordExpired(boolean expired) {
    }

    public void ejbPostCreate(String username, String password) {
        ctx.getTimerService().createTimer(
                BunchOfConstants.TWO_MONTHS, "password expired");
    }

    public void ejbTimeout(Timer timer) {
        setPasswordExpired(true);
    }
}
```

We've just stepped through what constitutes a representative example of test-driving EJB components that make use of the Timer service. Our UserBean class didn't reach a state where it could be deployed to an EJB container (being non-abstract and missing a bunch of other lifecycle methods, as well as not implementing the EntityBean interface), but it's getting there.

In practice, if we would continue developing our UserBean further from here, we'd probably add the setters and getters, make the UserBean class abstract, let it implement the proper interfaces, and then—in order to not bloat our test code with empty implementations of a gazillion getter and setter methods—add a stub class (such as UserBeanStub), which would be a concrete class, implementing all the abstract methods. In our tests, we could then extend UserBeanStub instead of UserBean and override just those methods in which we're interested for that specific test.

In the tradition of lazy book authors, I'll leave that as an exercise for you, and move on to the next topic. The last subject on the agenda for this chapter is entity beans and the new EJB 3 Persistence API. Even though entity beans can be a pain in the behind, you'll be surprised by the glaring simplicity and the ease of test-driving persistent entities using EJB 3.

13.4 Entities and entity beans

EJB 3 entities and EJB 2.x entity beans are different beasts from the other kinds of Enterprise JavaBeans, session beans, and message-driven beans. *Entities* represent data and operations related to finding entities by various criteria, modifying the data represented by a given entity bean instance, and creating and deleting data—in other words, persistent objects.

The majority of entity beans written by corporate developers have traditionally had no functionality beyond what's generated by the EJB container for implementing the persistence to a backing database—with the exception of entities that use bean-managed persistence (BMP), implementing their own persistence logic (typically using the JDBC API). Figure 13.7 illustrates this common division of responsibility, which hasn't promoted good object-oriented design in many enterprise systems.

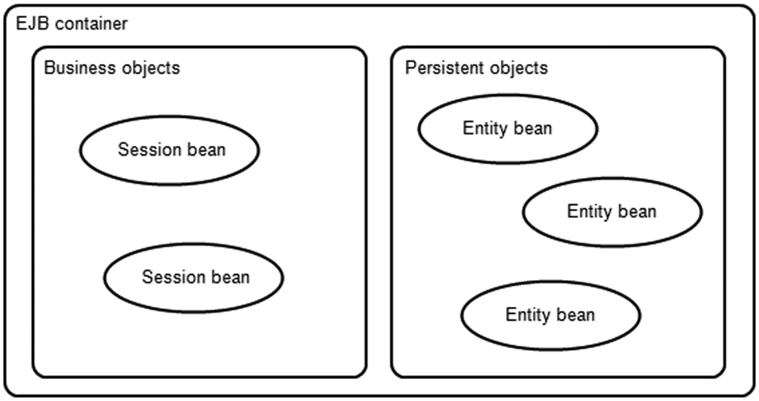

Figure 13.7 The only role of entity beans in a typical EJB-based architecture is persistence.

This has been our playing field for the past several years when it comes to the EJB 2.x specification and persistent objects. Along with the EJB 3 specification, we got a new Persistence API to supersede entity beans. The Persistence API simplifies the way persistent entities are developed, also making quantum leaps in terms of testability. The downside of the improvement is, unfortunately, that all this time "easily testable entity beans" has been an oxymoron.

For the remainder of this chapter, we'll look into how we can test and test-drive entity beans. I'll start by discussing the old EJB 2.x API and what our options are regarding unit testing and test-driven development.[5] Then, we'll look more closely into how the new Persistence API enables our TDD process for developing entity beans.

13.4.1 Testing EJB 2.x entity beans

Entity beans have always been the black sheep of the EJB family. The first versions of the specification were unusable. Even after numerous huge improvements that arrived along with the 2.0 version of the EJB specification, entity beans remained one tough cookie for test-infected developers like yours truly.

Of all types of Enterprise JavaBeans, entity beans have always been the most dependent on container-provided functionality. This was the case with the most-

[5] For a more thorough discussion of the topic of testing EJB 2.x entity beans, consider picking up *JUnit Recipes* by J. B. Rainsberger (Manning Publications, 2005), and refer to the documentation for the MockEJB test harness at www.mockejb.org.

anticipated feature (perhaps a bit ironically, in hindsight) of the EJB 2.0 specification: container-managed persistence.

The traditional bean-managed persistence—where the developer still wrote JDBC code by hand in the lifecycle methods defined by the EJB specification—has never been a problem. This was because testing BMP entity beans doesn't differ from testing any other kind of JDBC code—it's a combination of putting a `DataSource` (mock) into the JNDI tree and verifying that the proper interaction happened between the entity bean and the JDBC interfaces when the test invokes certain lifecycle methods.

We were left with a few options for writing tests for the CMP entity beans, but none supports test-driven development well. Let's take a quick look at these options and their respective advantages and disadvantages from a TDD practitioner's point of view.

Testing inside the container

One of the first and most popular approaches to testing CMP entity beans was to run the tests inside a real EJB container. Frameworks like Jakarta Cactus—an example of which is shown in figure 13.8—extending the JUnit framework were helpful in setting up such a test harness. In practice, we wrote our tests assuming they would be executed in the presence of the real container's JNDI tree, with the real Enterprise JavaBeans deployed to the JNDI tree, and sometimes with the DataSource objects connected to the real database.

Running these in-container tests meant running a build script, which compiled our code, packaged our components, deployed them on the application server, and then invoked the deployed test code remotely. All the remote invocation logic

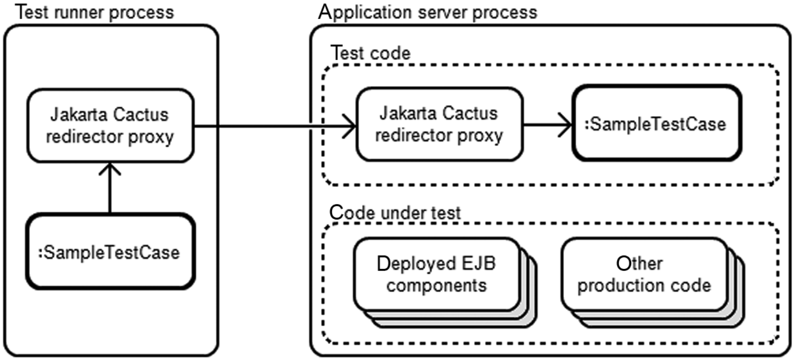

Figure 13.8 In-container tests using Jakarta Cactus. We run two copies of a `TestCase`. The client-side copy only talks to the server-side redirector proxy, and the server-side copy executes the test logic on request.

was handled by the framework (Jakarta Cactus, for example), and from the build script's perspective, the output was identical to any regular JUnit test run.

Listing 13.21 shows an example of a Jakarta Cactus-based unit test that assumes it will be running inside the real container.

Listing 13.21 Jakarta Cactus test assuming real runtime environment

```
import javax.naming.*;
import org.apache.cactus.*;

public class ConverterTest extends ServletTestCase {

    public void testEntityBeanOnServerSide() throws Exception {
        Context ctx = new InitialContext();
        ProductHome home =
                (ProductHome) ctx.lookup("java:comp/ejb/Product");
        Collection manningBooks = home.findByVendor("Manning");
        assertEquals(1, manningBooks.size());
        Product first = (Product) manningBooks.get(0);
        assertEquals("TDD in Action", first.getName());
    }
}
```

Connect to real JNDI tree

The problem with this approach is two-fold: It requires a complex infrastructure (figure 13.8), and it's slow. Waiting for half a minute for the components to be deployed doesn't promote running our tests after each tiny change, which is what we'd like to be able to do. As application servers are slowly starting to get their hot-deployment features working reliably, this becomes less of a problem compared to recent history, where each deployment had to include restarting the server—taking even more time.

The fact that using a full-blown container slows our test cycle significantly naturally leads us to consider a lightweight alternative: an embedded container.

Using a lightweight embedded container

Along with providing a mock implementation of the JNDI tree, the MockEJB framework provides a lightweight, embeddable EJB container. Although not a full-blown, specification-compliant implementation of the EJB specification, MockEJB's container implementation provides all the necessary bits for sufficiently unit- or integration-testing our EJB 2.x components.

Listing 13.22 shows the steps for deploying an entity bean to MockEJB's container.

Listing 13.22 Deploying an EJB 2.x entity bean with MockEJB

```
EntityBeanDescriptor descriptor =
        new EntityBeanDescriptor(
            "ejb/Product",
            ProductHome.class,
            Product.class,
            ProductBean.class);

MockContextFactory.setAsInitial();
Context context = new InitialContext();

MockContainer mockContainer = new MockContainer(context);
mockContainer.deploy(descriptor);
```

❶ Create deployment descriptor

❷ Initialize and connect to mock JNDI tree

❸ Create Mock-Container and deploy bean

What we do in listing 13.22 is simple. The first step is to ❶ create a descriptor object to act as a substitute for the standard XML deployment descriptor (ejb-jar.xml). The information needed includes the JNDI name with which the container should bind the bean's home interface to the JNDI tree, the home and business interfaces of the bean, and the bean class. Next, we ❷ initialize MockEJB's own JNDI implementation and connect to it by creating a new `InitialContext` with the default constructor. Finally, we ❸ create a new `MockContainer` instance on top of the `InitialContext` and deploy any bean descriptors we need. From here on, we can look up the entity bean's home interface from the JNDI tree as usual. If we want to test two or more beans together, we create a suitable descriptor for each and deploy them.

The `MockContainer` also gives us an `EntityDatabase` to which we can manually add entity instances, and the `MockContainer` takes care of returning the entity with a matching primary key from the `EntityDatabase` when that particular home interface's finders are invoked, for example. Furthermore, an interceptor framework is built into the `MockContainer`, which lets us intercept calls to deployed beans and their home interfaces. This interceptor framework is needed, for instance, for allocating created entity bean instances with unique primary keys.

Being able to deploy the entity beans without a slow, full-blown container is a clear advantage over using the real thing. The only major downside for this approach has been that dealing with the interceptors isn't typically a walk in the park. The fact that the `MockContainer` isn't the same thing as the real EJB container doesn't matter much—we're mostly interested in getting the business logic in place, and these tests give us enough confidence in that regard.

It's time to move on to the successor for the legacy entity beans: the EJB 3 specification and the new Persistence API. The good news is that virtually everything about the new specification is significantly easier to unit-test—let alone test-drive—and that includes persistent objects!

13.4.2 Test-driving EJB 3 entities

The new Java Persistence API, introduced as part of the EJB 3 specification, is a complete revamp as far as entity beans are concerned. The old EJB 2.x-style entity beans are still supported in the name of backward compatibility, but it's safe to say the new API will take the throne faster than we can say "dependency injection and annotations." This is because of the new API's approach of making the entity beans plain old Java objects and letting the developer define database-mapping using annotations rather than an external configuration file.[6]

Before stretching our test-driven muscles again, let's look at what EJB 3 entities consist of.

Benefits of being 100% Java

Before, we had CMP entity beans with abstract setters and getters, BMP entity beans littered with JDBC code, JNDI lookups all over the place, and a bunch of mandatory lifecycle methods doing nothing. The new Persistence API gets rid of that. Persistent objects are plain old Java objects with certain annotations used for indicating persistent fields, dependencies, and so forth. One of the welcome changes is the fact that persistent fields don't require a pair of getters and setters when using field injection. For example, consider the persistent entity in listing 13.23, representing a user in our system.

Listing 13.23 Example of an EJB 3 entity

```
import javax.persistence.*;

@Entity                         ❶ Declare
@Table(name = "USERS")            persistent
public class User {               class

    @Id                                                 ❷ Declare
    @GeneratedValue(strategy = GenerationType.AUTO)       primary
    private Integer id;                                   key field

    private String username;
    private String password;
```

[6] Although in some cases the external configuration file has its merits, the vast majority of projects will find that embedding the object-relational mapping annotations into the source code provides more bang for the buck.

```
    public Integer getId() {
        return id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public boolean matchesPassword(String password) {
        return this.password.equals(password);
    }
}
```

❸ **Provide only accessors we need**

This code is all we need to introduce a persistent entity into our system. The class-level annotations ❶ tell the EJB container the plain old Java class being defined should be persisted and instances of the class should be stored in a database table named `USERS`. (By default, the class name is used—in this case, `User`.)

The EJB 3 specification supports both field- and property-based (setter/getter) persistent fields. We can use either of these, but only one at a time for a specific class. In ❷ we're tacking a couple of persistence annotations on the private field `id` to declare `id` as the primary key field; we're further specifying a mechanism for the container to auto-generate the primary key for new instances. By doing this, we've committed to using field-based persistence; all nontransient members of the class (whether declared using the `transient` modifier or the `@Transient` annotation) will be considered persistent fields.

With this approach, we're free to ❸ define only those setter and getter methods in our persistent class that our application needs—no need to add a getter or a setter to satisfy a contract defined in the specification. For example, we don't want to expose a setter for the `User` class's primary key, but we do want to give read-only access to it. Similarly, we might not want to provide a getter for the `User` object's password but rather provide a `matchesPassword` method that compares the given password with the value of the `private` field.

If that's all there is to EJB 3 entities, then how does test-driving them differ from test-driving plain old Java code? That's the thing—it doesn't. Entities in EJB 3 are plain old Java objects! We could test-drive the annotations into our entity class (relationship annotations such as @OneToMany, @ManyToMany, and so on come to mind) and have automated tests for checking that certain rules are followed (such as not mixing the field- and property-based persistence within a class hierarchy). There's another caveat, however.

Although we can instantiate our entity classes with the new operator and operate on the instances just as we can with any other Java objects, I haven't seen how we save, find, update, and remove these entities to and from the database—which is what I'll talk about next.

Introducing the EntityManager API

We aren't likely to want to test setters and getters directly (with the possible exception of @Transient accessor methods that might have additional behavior), but we probably need to find, create, update, and delete our entity beans somewhere in our application.

In the EJB 3 specification, all CRUD[7] operations are performed using the EntityManager API—specifically the javax.persistence.EntityManager interface. The EntityManager is a generic facility for performing CRUD operations on any persistent entity class, as illustrated in figure 13.9. Listing 13.24 shows some of the methods available on the EntityManager interface.

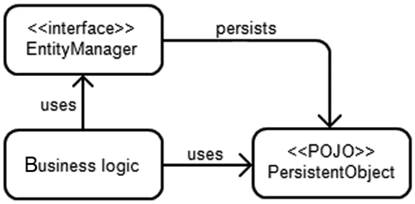

Figure 13.9 The EntityManager's role is to persist plain old Java objects representing persistent entities.

Listing 13.24 Some of the essential methods on the EntityManager interface

```
package javax.persistence;

public interface EntityManager {

    // methods for creating finder queries using EJB-QL, SQL, or
    // using a named query configured externally
    Query createQuery(String ejbqlString);
    Query createNativeQuery(String sqlString);
    Query createNamedQuery(String name);

    // method for finding an entity by its class and primary key
    T find(Class<T> entityClass, Object primaryKey);

    // methods for saving, loading, deleting, and updating entities
```

[7] CRUD (create, read, update, delete) refers to the common database operations we perform with persistent objects.

```
        void persist(Object entity);
        void refresh(Object entity);
        void remove(Object entity);
        T merge(T entity);

        // method for synchronizing the persistence context to database
        void flush();

        // some less common methods omitted for brevity...
    }
```

Components wanting to manipulate persistent objects need to use the EntityManager interface to do so. This simplifies our life because the EntityManager interface is almost without exception injected into the component using it. Even if this isn't the case, setting up a fake JNDI tree isn't impossible, although it's more involved.

With this in mind, let's try some test-driving. How about a UserManagerBean that can do all sorts of things with persistent User objects?

Test-driving with the EntityManager

We're developing a UserManagerBean. Say we want it to be a stateless session bean. What behavior do we want from the UserManagerBean? We need to be able to locate a User object by its username, so we'll start from there.

We'll need a mock implementation of the EntityManager interface, and we decide to use EasyMock. Starting to sketch the interaction we expect the UserManagerBean to initiate toward the EntityManager, we find we also need a mock object for the javax.persistence.Query interface, and we need a User object. After organizing our sketch and adding the invocation of the UserManagerBean, we arrive at listing 13.25, which is a compact test regardless of the behavioral mocking of the EntityManager and Query interfaces.

Listing 13.25 Testing for expected usage of the EntityManager API

```
import javax.persistence.*;
import org.junit.*;
import org.laughingpanda.beaninject.Inject;
import static org.easymock.EasyMock.*;

public class UserManagerBeanTest {

    private final String username = "bob";

    @Test
    public void findingUserByUsername() throws Exception {
        EntityManager em = createMock(EntityManager.class);
        Query q = createMock(Query.class);
        User user = createDummyUser(username);

        expect(em.createNamedQuery("findUserByUsername"))
            .andReturn(q); |#2
        expect(q.setParameter("username", username)).andReturn(q);
        expect(q.getSingleResult()).andReturn(user);

        replay(em, q);

        UserManagerBean userManager = new UserManagerBean();
        Inject.bean(userManager).with(em);
        Assert.assertEquals(user, bean.findByUsername(username));

        verify(em, q);
    }

    // helper methods omitted for brevity
}
```

Create mocks for EntityManager and Query

Expect bean to named query ❶

Inject mock EntityManager using Bean Inject library ❷

By ❷ using the open source Bean Inject library[8] to inject our mock `EntityManager` into the `UserManagerBean`, we state that the `UserManagerBean` should have a field or setter method for an `EntityManager`. We also state that when `findByUsername()` is called, ❶ the `UserManagerBean` should use a named query `findUserByUsername` and populate a query parameter named `username` with the username we passed in. It's simple once we get used to EasyMock's expect-and-return syntax.

This was a trivial read-only scenario, but it proved that test-driving EJB 3 entities is no different from testing regular Java classes. Don't you love dependency injection? We'll wrap up with a quick summary of what we've seen in this chapter, but by all means continue driving more functionality into the `UserManagerBean` if you're following along and feeling the thrill!

13.5 Summary

That was a rough ride through the various types of Enterprise JavaBeans and the two current versions of the specification—not to mention that the two specifications seem to have almost nothing in common. You made it this far and should pat yourself on the back. Let's recap the main topics covered.

[8] http://www.laughingpanda.org/mediawiki/index.php/Bean_Inject.

We started by considering the fundamental issue that has plagued test-infected EJB developers for several years: the problem of managed objects being too tightly coupled to the services provided by the container and the real container being too heavy to be included as part of our unit test harness. Having stated the problem, we set out to discuss the different types of EJB components, beginning from the session bean.

First, we found it isn't difficult to let our unit tests simulate the EJB container and the way it manages our bean class's lifecycle. We also determined that it might make sense to write automated tests to verify the implicit correctness of our bean class—it defines the methods and annotations required by the specification.

Soon we stumbled on the problem of how EJB components have traditionally acquired their dependencies by actively performing JNDI lookups. We devised three strategies for tackling this problem: faking the JNDI tree with a mock implementation, extracting the lookup operations into simple getter methods we can override in our tests, and making the former easier to do in our tests by adopting a poor-man's implementation of dependency injection.

Next, we turned our attention to the major asynchronous components of the EJB specification: message-driven beans and the Timer service. We came up with a need for a message-driven bean and proceeded to test-drive the `onMessage` method's implementation into place. For the Timer service, we implemented a scenario where a `User` entity set a timer upon creation and expired its password when the timer timed out. Once again, we found there's nothing special to test-driving timers—we need a flexible mock-object framework like EasyMock, and we're all set.

We spent the remainder of the chapter discussing entity beans. First, we talked about the few options available for testing EJB 2.x entity beans, focusing on container-managed persistence. After reviewing our options of running tests inside a real container and using an embedded, lightweight container, we delved into the new Persistence API in EJB 3. We discovered that entities in EJB 3 are plain old Java objects and thus a no-brainer to test-drive. We also discovered that the `EntityManager` interface takes care of performing the persistence operations for our entity beans, so we proceeded to test-drive a session bean that uses the EntityManager API to locate persistent entities. Again, it wasn't much different from test-driving regular Java classes.

Having plowed through this chapter at a pretty fast pace, we now know that test-driven EJB development isn't an oxymoron. Even if you didn't have much background in Enterprise JavaBeans, I'm sure you have an idea of where to start if you find yourself on the new EJB project at work next Monday.

JAVA

TEST DRIVEN

Lasse Koskela

In test-driven development, you first write an executable test of what your application code must do. Only then do you write the code itself and, with the test spurring you on, improve your design. In acceptance test-driven development (ATDD), you use the same technique to implement product features, benefiting from iterative development, rapid feedback cycles, and better-defined requirements. TDD and its supporting tools and techniques lead to better software faster.

Test Driven brings under one cover practical TDD techniques distilled from several years of community experience. With examples in Java and the Java EE environment, it explores both the techniques and the mindset of TDD and ATDD. It uses carefully chosen examples to illustrate TDD tools and design patterns, not in the abstract but concretely in the context of the technologies you face at work. It is accessible to TDD beginners, and it offers effective and less-well-known techniques to older TDD hands.

What's Inside

- Hands-on examples of how to test-drive Java code
- How to avoid common TDD adoption pitfalls
- ATDD development and the Fit framework
- How to test Java EE components—Servlets, JSPs, and Spring Controllers
- How to handle tough issues like multithreaded programs and data-access code

Lasse Koskela, a methodology specialist at Reaktor Innovations in Finland, has coached dozens of teams in agile methods and practices such as test-driven development.

For more information, code samples, and to purchase an ebook visit www.manning.com/TestDriven

"... very engaging writing style. I was blown away!"

—Michael Feathers
Consultant, Object Mentor

"Full of hard-won lessons that take years to learn on your own."

—Laurent Bossavit
Consultant, 2006 Gordon Pask Award Winner

"Strengthen your quality safety-net with the TDD ideas in this book!"

—Christopher Haupt
Principal Consultant
Mobirobo LLC

"... a well-spring of up-to-date information and practices."

—Jason Rogers
Software Engineer
RiskMetrics Group, Inc.

"... it's much better on TDD than other books I've read ... including a wonderful job in providing TDD philosophy."

—Dave Corun
Architect, Social Solutions

ISBN-13: 978-1932394856
ISBN-10: 1932394850
54499
9 781932 394856

manning

$44.99 / Can $53.99