The Addison-Wesley Signature Series

A Kent Beck Signature Book

Growing Object-Oriented Software, Guided by Tests

Steve Freeman
Nat Pryce

Praise for *Growing Object-Oriented Software, Guided by Tests*

"The authors of this book have led a revolution in the craft of programming by controlling the environment in which software grows. Their Petri dish is the mock object, and their microscope is the unit test. This book can show you how these tools introduce a repeatability to your work that would be the envy of any scientist."

—*Ward Cunningham*

"At last a book, suffused with code, that exposes the deep symbiosis between TDD and OOD. The authors, pioneers in test-driven development, have packed it with principles, practices, heuristics, and (best of all) anecdotes drawn from their decades of professional experience. Every software craftsman will want to pore over the chapters of worked examples and study the advanced testing and design principles. This one's a keeper."

—*Robert C. Martin*

"Design is often discussed in depth, but without empiricism. Testing is often promoted, but within the narrow definition of quality that relates only to the presence or absence of defects. Both of these perspectives are valuable, but each on its own offers little more than the sound of one hand clapping. Steve and Nat bring the two hands together in what deserves—and can best be described as—applause. With clarity, reason, and humour, their tour de force reveals a view of design, testing, code, objects, practice, and process that is compelling, practical, and overflowing with insight."

—*Kevlin Henney*, co-author of *Pattern-Oriented Software Architecture* and *97 Things Every Programmer Should Know*

"Steve and Nat have written a wonderful book that shares their software craftsmanship with the rest of the world. This is a book that should be studied rather than read, and those who invest sufficient time and energy into this effort will be rewarded with superior development skills."

—*David Vydra*, publisher, testdriven.com

"This book presents a unique vision of test-driven development. It describes the mature form of an alternative strain of TDD that sprang up in London in the early 2000s, characterized by a totally end-to-end approach and a deep emphasis on the messaging aspect of objects. If you want to be an expert in the state of the art in TDD, you need to understand the ideas in this book."

—*Michael Feathers*

"With this book you'll learn the rhythms, nuances in thinking, and effective programming practices for growing tested, well-designed object-oriented applications from the masters."

—*Rebecca Wirfs-Brock*

❶ We call the application through its `main()` function to make sure we've assembled the pieces correctly. We're following the convention that the entry point to the application is a `Main` class in the top-level package. WindowLicker can control Swing components if they're in the same JVM, so we start the Sniper in a new thread. Ideally, the test would start the Sniper in a new process, but that would be much harder to test; we think this is a reasonable compromise.

❷ To keep things simple at this stage, we'll assume that we're only bidding for one item and pass the identifier to `main()`.

❸ If `main()` throws an exception, we just print it out. Whatever test we're running will fail and we can look for the stack trace in the output. Later, we'll handle exceptions properly.

❹ We turn down the timeout period for finding frames and components. The default values are longer than we need for a simple application like this one and will slow down the tests when they fail. We use one second, which is enough to smooth over minor runtime delays.

❺ We wait for the status to change to `Joining` so we know that the application has attempted to connect. This assertion says that somewhere in the user interface there's a label that describes the Sniper's state.

❻ When the Sniper loses the auction, we expect it to show a `Lost` status. If this doesn't happen, the driver will throw an exception.

❼ After the test, we tell the driver to dispose of the window to make sure it won't be picked up in another test before being garbage-collected.

The `AuctionSniperDriver` is simply an extension of a WindowLicker `JFrameDriver` specialized for our tests:

```
public class AuctionSniperDriver extends JFrameDriver {
  public AuctionSniperDriver(int timeoutMillis) {
    super(new GesturePerformer(),
          JFrameDriver.topLevelFrame(
            named(Main.MAIN_WINDOW_NAME),
            showingOnScreen()),
            new AWTEventQueueProber(timeoutMillis, 100));
  }

  public void showsSniperStatus(String statusText) {
    new JLabelDriver(
      this, named(Main.SNIPER_STATUS_NAME)).hasText(equalTo(statusText));
  }
}
```

On construction, it attempts to find a visible top-level window for the Auction Sniper within the given timeout. The method `showsSniperStatus()` looks for the relevant label in the user interface and confirms that it shows the given status. If the driver cannot find a feature it expects, it will throw an exception and fail the test.

The Fake Auction

A `FakeAuctionServer` is a substitute server that allows the test to check how the Auction Sniper interacts with an auction using XMPP messages. It has three responsibilities: it must connect to the XMPP broker and accept a request to join the chat from the Sniper; it must receive chat messages from the Sniper or fail if no message arrives within some timeout; and, it must allow the test to send messages back to the Sniper as specified by Southabee's On-Line.

Smack (the XMPP client library) is event-driven, so the fake auction has to register listener objects for it to call back. There are two levels of events: events *about* a chat, such as people joining, and events *within* a chat, such as messages being received. We need to listen for both.

We'll start by implementing the `startSellingItem()` method. First, it connects to the XMPP broker, using the item identifier to construct the login name; then it registers a `ChatManagerListener`. Smack will call this listener with a `Chat` object that represents the session when a Sniper connects in. The fake auction holds on to the chat so it can exchange messages with the Sniper.

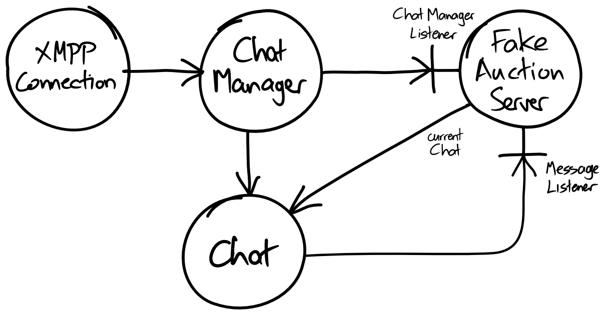

Figure 11.1 *Smack objects and callbacks*

So far, we have:

```
public class FakeAuctionServer {
  public static final String ITEM_ID_AS_LOGIN = "auction-%s";
  public static final String AUCTION_RESOURCE = "Auction";
  public static final String XMPP_HOSTNAME = "localhost";
  private static final String AUCTION_PASSWORD = "auction";

  private final String itemId;
  private final XMPPConnection connection;
  private Chat currentChat;

  public FakeAuctionServer(String itemId) {
    this.itemId = itemId;
    this.connection = new XMPPConnection(XMPP_HOSTNAME);
  }

  public void startSellingItem() throws XMPPException {
    connection.connect();
    connection.login(format(ITEM_ID_AS_LOGIN, itemId),
                     AUCTION_PASSWORD, AUCTION_RESOURCE);
    connection.getChatManager().addChatListener(
      new ChatManagerListener() {
        public void chatCreated(Chat chat, boolean createdLocally) {
          currentChat = chat;
        }
      });
  }

  public String getItemId() {
    return itemId;
  }
}
```

A Minimal Fake Implementation

We want to emphasize again that this fake is a minimal implementation just to support testing. For example, we use a single instance variable to hold the chat object. A real auction server would manage multiple chats for all the bidders—but this is a fake; its only purpose is to support the test, so it only needs one chat.

Next, we have to add a `MessageListener` to the `chat` to accept messages from the Sniper. This means that we need to coordinate between the thread that runs the test and the Smack thread that feeds messages to the listener—the test has to wait for messages to arrive and time out if they don't—so we'll use a single-element `BlockingQueue` from the `java.util.concurrent` package. Just as we only have one `chat` in the test, we expect to process only one message at a time. To make our intentions clearer, we wrap the queue in a helper class `SingleMessageListener`. Here's the rest of `FakeAuctionServer`:

```
public class FakeAuctionServer {
  private final SingleMessageListener messageListener = new SingleMessageListener();

  public void startSellingItem() throws XMPPException {
    connection.connect();
    connection.login(format(ITEM_ID_AS_LOGIN, itemId),
                     AUCTION_PASSWORD, AUCTION_RESOURCE);
    connection.getChatManager().addChatListener(
      new ChatManagerListener() {
        public void chatCreated(Chat chat, boolean createdLocally) {
          currentChat = chat;
          chat.addMessageListener(messageListener);
        }
      });
  }

  public void hasReceivedJoinRequestFromSniper() throws InterruptedException {
    messageListener.receivesAMessage(); ❶
  }

  public void announceClosed() throws XMPPException {
    currentChat.sendMessage(new Message()); ❷
  }

  public void stop() {
    connection.disconnect(); ❸
  }
}

public class SingleMessageListener implements MessageListener {
  private final ArrayBlockingQueue<Message> messages =
                            new ArrayBlockingQueue<Message>(1);

  public void processMessage(Chat chat, Message message) {
    messages.add(message);
  }

  public void receivesAMessage() throws InterruptedException {
    assertThat("Message", messages.poll(5, SECONDS), is(notNullValue())); ❹
  }
}
```

❶ The test needs to know when a `Join` message has arrived. We just check whether *any* message has arrived, since the Sniper will only be sending `Join` messages to start with; we'll fill in more detail as we grow the application. This implementation will fail if no message is received within 5 seconds.

❷ The test needs to be able to simulate the auction announcing when it closes, which is why we held onto the `currentChat` when it opened. As with the `Join` request, the fake auction just sends an empty message, since this is the only event we support so far.

❸ `stop()` closes the connection.

4. The clause `is(notNullValue())` uses the Hamcrest *matcher* syntax. We describe `Matchers` in "Methods" (page 339); for now, it's enough to know that this checks that the Listener has received a message within the timeout period.

The Message Broker

There's one more component to mention which doesn't involve any coding—the installation of an XMPP message broker. We set up an instance of Openfire on our local host. The Sniper and fake auction in our end-to-end tests, even though they're running in the same process, will communicate through this server. We also set up logins to match the small number of item identifiers that we'll be using in our tests.

A Working Compromise

As we wrote before, we are cheating a little at this stage to keep development moving. We want all the developers to have their own environments so they don't interfere with each other when running their tests. For example, we've seen teams make their lives very complicated because they didn't want to create a database instance for each developer. In a professional organization, we would also expect to see at least one test rig that represents the production environment, including the distribution of processing across a network and a build cycle that uses it to make sure the system works.

Failing and Passing the Test

We have enough infrastructure in place to run the test and watch it fail. For the rest of this chapter we'll add functionality, a tiny slice at a time, until eventually we make the test pass. When we first started using this technique, it felt too fussy: "Just write the code, we know what to do!" Over time, we realized that it didn't take any longer and that our progress was much more predictable. Focusing on just one aspect at a time helps us to make sure we understand it; as a rule, when we get something working, it stays working. Where there's no need to discuss the solution, many of these steps take hardly any time at all—they take longer to explain than to implement.

We start by writing a build script for *ant*. We'll skip over the details of its content, since it's standard practice these days, but the important point is that we always have a single command that reliably compiles, builds, deploys, and tests the application, and that we run it repeatedly. We only start coding once we have an automated build and test working.

At this stage, we'll describe each step, discussing each test failure in turn. Later we'll speed up the pace.

First User Interface

Test Failure

The test can't find a user interface component with the name `"Auction Sniper Main"`.

```
java.lang.AssertionError:
Tried to look for...
    exactly 1 JFrame (with name "Auction Sniper Main" and showing on screen)
    in all top level windows
but...
    all top level windows
    contained 0 JFrame (with name "Auction Sniper Main" and showing on screen)
  […]
  at auctionsniper.ApplicationRunner.stop()
  at auctionsniper.AuctionSniperEndToEndTest.stopApplication()
  […]
```

WindowLicker is verbose in its error reporting, trying to make failures easy to understand. In this case, we couldn't even find the top-level frame so JUnit failed before even starting the test. The stack trace comes from the `@After` method that stops the application.

Implementation

We need a top-level window for our application. We write a `MainWindow` class in the `auctionsniper.ui` package that extends Swing's `JFrame`, and call it from `main()`. All it will do is create a window with the right name.

```
public class Main {
  private MainWindow ui;

  public Main() throws Exception {
    startUserInterface()
  }

  public static void main(String... args) throws Exception {
    Main main = new Main();
  }

  private void startUserInterface() throws Exception {
    SwingUtilities.invokeAndWait(new Runnable() {
      public void run() {
        ui = new MainWindow();
      }
    });
  }
}
```