Chương 12: Sẵn sàng đấu giá

Ở đây chúng tôi viết một test end-to-end để có thể làm cho Sniper đặt bid tại một cuộc đấu giá. Chúng tôi bắt đầu bằng việc giải nghĩa các message trong giao thức đấu giá và phát hiện ra một số lớp trong quá trình đó. Chúng tôi viết những unit test đầu tiên và refactor ra một lớp trợ giúp. Chúng tôi mô tả từng chi tiết của công việc này để trình bày mạch tư duy của chúng tôi lúc đó.

Gia nhập trị trường

Bây giờ, để tiếp tục với lối ẩn dụ khung xương di động, chúng tôi bắt đầu đắp thêm da thịt cho ứng dụng. Hành vi cốt lõi của một Sniper là trong một cuộc đấu giá một item, nó phải đặt một bid cao hơn cho item đó khi có sự thay đổi về giá. Quay lại to-do-list, chúng tôi xem xét lại mấy mục tiếp theo:

- Single item: join, bid, and lose. Khi nhận được giá mới, gửi bid với giá cao hơn giá hiện hành một lượng bằng chênh lệch tối thiểu mà auction quy định. Con số chênh lệch này sẽ được cho trong thông tin cập nhật về giá.
- Single item: join, bid, and win. Phân biệt bidder nào đang giữ phần thắng trong cuộc đấu giá và đừng đấu giá với chính mình.

Chúng tôi biết là còn nhiều nội dung nữa, nhưng đây là một lát cắt mạch lạc về chức năng, nó sẽ cho phép chúng tôi khám phá về thiết kế và cho thấy tiến độ cụ thể của dự án.

Trong một hệ phân tán bất kì tương tự với hệ hiện tại, có rất nhiều những sự cố thú vị và các rắc rối về thời gian, nhưng ứng dụng của chúng tôi chỉ phải xử lý ở phía client của giao thức. Chúng tôi dựa vào giao thức XMPP ở bên dưới để xử lý nhiều vấn đề mà lập trình phân tán thường gặp; cụ thể, chúng tôi trông đợi nó sẽ đảm bảo rằng các message giữa một bidder và một auction sẽ đến đích theo thứ tự mà chúng đã được gửi đi.

Như đã mô tả trong chương 5, chúng tôi bắt đầu một feature mới với một acceptance test. Chúng tôi đã dùng test đầu tiên trong chương trước để giúp bật ra kiến trúc của ứng dụng. Từ đây, chúng tôi có thể dùng các acceptance test để cho thấy tiến độ tăng dần.

Một test cho việc đặt bid

Bắt đầu bằng một test

Mỗi acceptance test chỉ nên chứa vừa đủ requirement. Nếu nhiều quá, lượng chức năng tăng thêm sẽ trở nên khó quản lý. Do đó chúng tôi quyết định rằng test tiếp theo sẽ bổ sung một số thông tin giá. Các bước như sau:

1. Yêu cầu auction gửi một message về giá (Price) cho Sniper.

2. Kiểm tra xem Sniper đã nhận được giá mới và đã phản ứng hay chưa.

3. Kiểm tra xem auction đã nhận được một bid với giá tăng thêm từ Sniper hay chưa.

Để cho test này pass được, Sniper phải phân biệt được giữa các event Price (giá mới) và event Close (đóng auction) được gửi từ auction, hiển thị được giá hiện tại, và tạo ra được bid mới. Chúng tôi sẽ phải mở rộng auction rởm để xử lý được bid. Chúng tôi cũng đã trì hoãn việc cài đặt các chức năng khác sẽ cần đến như: hiển thị khi Sniper thắng đấu giá; chúng tôi sẽ quay lại chuyện này sau. Đây là test mới:

```
public class AuctionSniperEndToEndTest {
  @Test
  public void sniperMakesAHigherBidButLoses() throws Exception {
    auction.startSellingItem();

    application.startBiddingIn(auction);
    auction.hasReceivedJoinRequestFromSniper(); (1)

    auction.reportPrice(1000, 98, "other bidder"); (2)
    application.hasShownSniperIsBidding(); (3)

    auction.hasReceivedBid(1098, ApplicationRunner.SNIPER_XMPP_ID); (4)

    auction.announceClosed(); (5)
    application.showsSniperHasLostAuction();
  }
}
```

Chúng tôi có ba phương thức cần cài để phục vụ test này.

1. Chúng tôi phải đợi cho auction giả nhận được message Join trước khi cho test chạy tiếp. Chúng tôi dùng assertion này để đồng bộ hóa Sniper với auction.
2. Phương thức này yêu cầu auction giả gửi message lại cho Sniper để thông báo rằng giá hiện tại cho item là 1000, mức tăng tối thiểu là 98, và bidder đang giữ phần thắng là người khác (“other bidder.”)
3. Phương thức này yêu cầu ApplicationRunner kiểm tra xem Sniper có đang hiển thị trạng thái rằng nó đang đặt giá (bidding) hay không, sau khi nó đã nhận được message báo giá từ auction.
4. Phương thức này yêu cầu auction giả kiểm tra xem nó đã nhận được bid từ Sniper với giá bằng giá mới nhất cộng với độ chệnh tối thiểu hay chưa. Chúng tôi phải làm nhiều việc hơn một chút vì tầng XMPP tạo một cái tên dài hơn từ id cơ bản, nên chúng tôi định nghĩa một hằng SNIPER_XMPP_ID, mà trong thực tế có giá trị là `sniper@localhost/Auction`.
5. Chúng tôi tái sử dụng logic của test đầu tiên ở chi tiết Sniper lại thua đấu giá

Tiền không thực tế - Unrealistic Money

Chúng tôi đang dùng số nguyên để biểu diễn giá tiền (hình dung là cuộc đấu giá đang diễn ra ở Nhật bằng đồng Yên). Trong một hệ thống thực, chúng tôi sẽ định nghĩa một kiểu dữ liệu domain để biểu diễn các giá trị tiền tệ, sử dụng một cài đặt dấu phảy tĩnh (fix-point number). Ở đây, chúng tôi đơn giản hóa biểu diễn để code dễ đọc hơn và vừa cho 1 trang in.

Mở rộng Auction giả

Chúng tôi có hai phương thức cần viết tại FakeAuctionServer để hỗ trợ end-to-end test: reportPrice() phải gửi một message Price qua chat; hasReceivedBid() hơi phức tạp hơn, nó phải kiểm tra xem auction đã nhận được các giá trị đúng từ Sniper hay chưa. Thay vì dịch message đến, chúng tôi xây dựng message được trông đợi và chỉ việc so sánh string. Chúng tôi cũng dùng biểu thức Matcher tại SingleMessageListener để cho FakeAuctionServer được linh hoạt hơn khi định nghĩa dạng message mà nó sẽ chấp nhận. Đây là bản đầu tiên:

```
public class FakeAuctionServer { […]
  public void reportPrice(int price, int increment, String bidder) throws XMPPException {
    currentChat.sendMessage(String.format("SOLVersion: 1.1; Event: PRICE; "
            + "CurrentPrice: %d; Increment: %d; Bidder: %s;", price, increment, bidder));
  }
  public void hasReceivedJoinRequestFromSniper() throws InterruptedException {
    messageListener.receivesAMessage(is(anything()));
  }
  public void hasReceivedBid(int bid, String sniperId) throws InterruptedException {
    assertThat(currentChat.getParticipant(), equalTo(sniperId));
    messageListener.receivesAMessage( equalTo(
                String.format("SOLVersion: 1.1; Command: BID; Price: %d;", bid)));
  }
}

public class SingleMessageListener implements MessageListener { […]
  @SuppressWarnings("unchecked")
  public void receivesAMessage(Matcher<? super String> messageMatcher) throws InterruptedException {
    final Message message = messages.poll(5, TimeUnit.SECONDS);
    assertThat("Message", message, is(notNullValue()));
    assertThat(message.getBody(), messageMatcher);
  }
}
```

Nhìn lại, có một sự mất cân đối giữa hai phương thức "hasReceived". Phương thức cho Join kiểm tra lỏng lẻo hơn nhiều so với phương thức cho Bid, cả về nội dung message cũng như sender của message; chúng tôi sẽ phải nhớ để sau này quay lại chỉnh điểm này. Khi phát triển kiểu tăng dần, chúng ta trì hoãn rất nhiều quyết định, nhưng đôi khi sự nhất quát và cân đối lại hợp lí hơn. Chúng tôi quyết định đưa thêm chi tiết vào hasReceivedJoinRequestFromSniper() trong khi code của nó còn đang dở dang. Chúng tôi cũng tách định dạng message và chuyển chúng vào Main vì chúng tôi sẽ cần đến chúng tại Sniper để tạo các message thô từ đó.

```
public class FakeAuctionServer { […]
  public void hasReceivedJoinRequestFrom(String sniperId) throws InterruptedException {
    receivesAMessageMatching(sniperId, equalTo(Main.JOIN_COMMAND_FORMAT));
  }
  public void hasReceivedBid(int bid, String sniperId) throws InterruptedException {
    receivesAMessageMatching(sniperId, equalTo(format(Main.BID_COMMAND_FORMAT, bid)));
  }
  private void receivesAMessageMatching(String sniperId, Matcher<? super String> messageMatcher)
      throws InterruptedException
  {
    messageListener.receivesAMessage(messageMatcher);
    assertThat(currentChat.getParticipant(), equalTo(sniperId));
  }
}
```

Lưu ý rằng chúng tôi kiểm tra id của Sniper *sau khi* kiểm tra nội dung của message (`receivesAMessage`). Điều này buộc server phải đợi cho đến khi message đã đến nơi, nghĩa là khi đó nó chắc chắn đã nhận một kết nối và thiết lập đối tượng Chat hiện hành. Nếu không test sẽ fail khi kiểm tra id của Sniper quá sớm.

Double-Entry Values

Chúng tôi đang dùng cùng một hằng để tạo message Join và để kiểm tra nội dung của nó. Bằng cách chỉ dùng một cấu trúc, chúng tôi loại bỏ được sự trùng lặp về code, và thể hiện được trong code mối liên hệ giữa hai đầu của hệ thống (đầu gửi và đầu nhận message). Mặt khác, chúng tôi đang tự gây nguy cơ làm cho cả hai bị sai và không có test để bắt lỗi nội dung bất hợp lệ. Trong trường hợp này, code đơn giản đến mức cài thế nào cũng

được, nhưng câu trả lời sẽ kém chắc chắn hơn khi phát triển cái gì đó phức tạp hơn, chẳng hạn như tầng persistence. Chúng ta có nên dùng cùng một framework để đọc và ghi các giá trị hay không? Có thể đảm bảo rằng nó không chỉ đọc kết quả từ cache? Hay rằng các giá trị được ghi đúng? Chúng ta có nên viết mấy query tới thẳng CSDL để cho chắc chắn không?

Câu hỏi cốt lõi là: chúng ta nghĩ là mình đang test cái gì vậy? Ở đây, chúng tôi nghĩ rằng các tính năng liên lạc quan trọng hơn, rằng các message đủ đơn giản để chúng tôi có thể dựa vào các hằng String, và rằng chúng tôi muốn có khả năng tìm các đoạn code liên quan đến định dạng message tại IDE. Các developer khác có thể có các kết luận khác và là các kết luận đúng cho dự án của họ.

Chúng tôi điều chỉnh end-to-end test để khớp với API mới, quan sát test fail, và sau đó thêm chi tiết cho Sniper để làm cho test pass.

```
public class AuctionSniperEndToEndTest {
  @Test
  public void sniperMakesAHigherBidButLoses() throws Exception {
    auction.startSellingItem();

    application.startBiddingIn(auction);
    auction.hasReceivedJoinRequestFrom(ApplicationRunner.SNIPER_XMPP_ID);

    auction.reportPrice(1000, 98, "other bidder");
    application.hasShownSniperIsBidding();

    auction.hasReceivedBid(1098, ApplicationRunner.SNIPER_XMPP_ID);

    auction.announceClosed();
    application.showsSniperHasLostAuction();
  }
}

public class Main { […]
  private void joinAuction(XMPPConnection connection, String itemId) throws XMPPException {
    Chat chat = connection.getChatManager().createChat(
        auctionId(itemId, connection),
        new MessageListener() {
          public void processMessage(Chat aChat, Message message) {
            SwingUtilities.invokeLater(new Runnable() {
              public void run() {
                ui.showStatus(MainWindow.STATUS_LOST);
              }
            });
          }
        });
    this.notToBeGCd = chat;
    chat.sendMessage(JOIN_COMMAND_FORMAT);
  }
}
```

Test fail ngoài dự tính - A Surprise Failure

Cuối cùng, chúng tôi viết thêm phương thức kiểm tra tại ApplicationRunner để có thể chạy test lần đầu tiên. Cài đặt rất đơn giản: chúng tôi chỉ thêm một hằng status và chép nội dung từ một phương thức có sẵn.

```
public class ApplicationRunner { […]
  public void hasShownSniperIsBidding() {
```

```
    driver.showsSniperStatus(MainWindow.STATUS_BIDDING);
  }

  public void showsSniperHasLostAuction() {
    driver.showsSniperStatus(MainWindow.STATUS_LOST);
  }
}
```

Chúng tôi đang trông đợi kết quả gì đó về việc thiếu một cái text trong status label. Nhưng thay vào đó, chúng tôi nhận được lỗi này:

```
java.lang.AssertionError:
Expected: is not null
    got: null
  […]
  at auctionsniper.SingleMessageListener.receivesAMessage()
  at auctionsniper.FakeAuctionServer.hasReceivedJoinRequestFromSniper()
  at auctionsniper.AuctionSniperEndToEndTest.sniperMakesAHigherBid()
  […]
```

Và dòng này tại error stream:

```
conflict(409)
  at jivesoftware.smack.SASLAuthentication.bindResourceAndEstablishSession()
  at jivesoftware.smack.SASLAuthentication.authenticate()
  at jivesoftware.smack.XMPPConnection.login()
  at jivesoftware.smack.XMPPConnection.login()
  at auctionsniper.Main.connection()
  at auctionsniper.Main.main()
```

Sau một lúc tìm hiểu, chúng tôi nhận ra chuyện gì đã xảy ra. Cái test thứ hai mới được thêm đã cố kết nối với server bằng chính account và tên resource mà test đầu tiên đã dùng. Cũng như Southabee's On-Line, server đã được cấu hình để từ chối nhiều connection cùng mở, do đó test thứ hai fail vì server nghĩ rằng test thứ nhất vẫn đang kết nối. Khi chạy thực, ứng dụng của chúng tôi sẽ không sao vì chúng tôi dừng toàn bộ tiến trình (process) khi kết thúc, việc này sẽ tự động cắt kết nối. Còn bây giờ, chúng tôi đã mắc bẫy sự thỏa hiệp nhỏ của chính mình (về việc chạy ứng dụng tại một thread mới thay vì tại một tiến trình mới). Việc đúng đắn cần làm bây giờ là thêm một lệnh call back để disconnect client khi đóng cửa sổ, để ứng dụng sẽ tự dọn dẹp khi nó kết thúc:

```
public class Main { […]
  private void joinAuction(XMPPConnection connection, String itemId) throws XMPPException {
    disconnectWhenUICloses(connection);
    Chat chat = connection.getChatManager().createChat(
    […]
    chat.sendMessage(JOIN_COMMAND_FORMAT);
  }
  private void disconnectWhenUICloses(final XMPPConnection connection) {
    ui.addWindowListener(new WindowAdapter() {
      @Override public void windowClosed(WindowEvent e) {
        connection.disconnect();
      }
    });
  }
}
```

Giờ thì chúng tôi nhận được lỗi test mà mình đang trông đợi, vì Sniper chưa có cách gì để bắt đầu đặt bid.

```
java.lang.AssertionError:
Tried to look for...
    exactly 1 JLabel (with name "sniper status")
    in exactly 1 JFrame (with name "Auction Sniper Main" and showing on screen)
    in all top level windows
and check that its label text is "Bidding"
```

```
but...
    all top level windows
    contained 1 JFrame (with name "Auction Sniper Main" and showing on screen)
    contained 1 JLabel (with name "sniper status")
    label text was "Lost"
  […]
  at auctionsniper.AuctionSniperDriver.showsSniperStatus()
  at auctionsniper.ApplicationRunner.hasShownSniperIsBidding()
  at auctionsniper.AuctionSniperEndToEndTest.sniperMakesAHigherBidButLoses()
```

Phát triển từ ngoài vào trong – Outside-In Development

Thông báo lỗi ở trên xác định đích cho công việc coding tiếp theo. Nó nói với chúng tôi về cái đích mà chúng tôi phải ngắm đến – chúng tôi chỉ việc điền code vào cài đặt cho đến khi test đó pass.

Cách tiếp cận của chúng tôi đối với TDD là bắt đầu đi từ event bên ngoài (cái thứ kích hoạt hành vi ta đang muốn cài) vào trong code, mỗi lần một đối tượng, cho đến khi gặp một hiệu ứng nhìn thấy được (chẳng hạn một message được gửi hay một log entry được ghi) – dấu hiệu cho thấy rằng ta đã đi đến đích. Mỗi test end-to-end cho ta thấy những điểm cuối của quá trình đó, để ta có thể khám phá con đường ta đi qua vùng trung gian.

Trong những mục sau, chúng tôi sẽ xây dựng các lớp cần thiết cho việc cài đặt Auction Sniper. Chúng tôi sẽ đi chậm, theo sát các quy tắc TDD, để cho thấy quy trình làm việc chạy như thế nào. Trong các dự án thực, đôi khi chúng tôi thiết kế trước một chút để có cảm giác về bức tranh toàn cảnh, nhưng trong hầu hết các tình huống thì chúng tôi thực sự thực hiện những gì được mô tả ở đây. Cách làm này mang lại kết quả đúng đắn và buộc chúng tôi phải hỏi đúng câu cần hỏi.

Chú ý hết cỡ vào chi tiết?

Chúng tôi đã bắt được lỗi xung đột resource vì cấu hình server của chúng tôi khớp với cấu hình của Southabee's On-line, do may mắn hoặc do kinh nghiệm. Chúng tôi có thể đã dùng một cấu hình khác cho phép các kết nối mới được bắt đầu từ các kết nối cũ, điều này sẽ dẫn đến kết quả là test vừa rồi pass, nhưng đi kèm một message gây hiểu nhầm của thư viện Smack tại error stream về xung đột đã xảy ra. Điều này có thể không sao trong khi phát triển, nhưng nó kèm theo nguy cơ Sniper sẽ bắt đầu gặp sự cố khi chạy thực.

Làm sao ta có thể hy vọng bắt lỗi được tất cả các lựa chọn cấu hình trong toàn bộ hệ thống? Ở một mức độ nào đó, chúng ta không thể. Và điều này nằm tại trung tâm nhiệm vụ của tester chuyên nghiệp. Cái ta có thể làm là cố gắng thử được càng nhiều phần của hệ thống càng tốt và thử càng sớm càng tốt, và lặp đi lặp lại công việc này. Chúng ta có thể giúp chính mình đương đầu với toàn bộ sự phức tạp của hệ thống bằng cách gìn giữ chất lượng cao của các component trong hệ thống và bằng cách liên tục cố gắng đơn giản hóa chúng. Nếu việc này nghe có vẻ đòi hỏi chi phí cao, ta hãy xem xét chi phí của việc tìm và sửa một lỗi như lỗi vừa kể trong một hệ thống đang chạy thực với công suất sử dụng cao.

AuctionMessageTranslator

Tách ra một lớp mới

Cổng vào của chúng tôi đối với Sniper là nơi ta nhận một message từ auction thông qua thư viện Smack: nó là event kích hoạt hành vi tiếp theo mà chúng tôi muốn cài đặt. Trong thực tế, điều đó có nghĩa rằng chúng tôi cần đến một lớp cài đặt interface MessageListener để gắn vào Chat. Khi lớp này nhận được một message thô từ

auction, nó sẽ dịch message ra thành cái gì đó đại diện cho một event của auction, và event đó cuối cùng sẽ kích hoạt một hành động của Sniper, dẫn đến việc xảy ra một thay đổi tại UI.

Chúng ta đã có một lớp như vậy trong Main – nó là một lớp anonymous và trách nhiệm của nó không rõ ràng lắm:

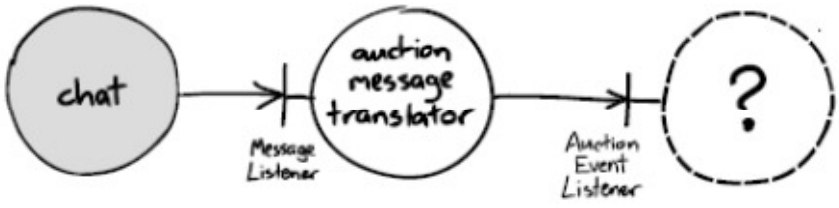
```
new MessageListener() {
  public void processMessage(Chat aChat, Message message) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        ui.showStatus(MainWindow.STATUS_LOST);
      }
    });
  }
}
```

Một cách không tường minh, đoạn code trên nhận một message Close (loại duy nhất mà chúng tôi có cho đến nay) và cài đặt phản ứng của Sniper. Chúng tôi muốn làm cho tình huống này tường minh hơn và bổ sung tính năng. Bắt đầu bằng việc nâng lớp anonymous thành một lớp đứng độc lập ở top-level, có nghĩa rằng nó cần một cái tên. Từ mô tả ở đoạn trước, chúng tôi chọn từ "translate" (dịch) và gọi lớp mới là AuctionMessageTranslator, vì nó sẽ *dịch* các *message* nhận được từ *auction*.

Rắc rối ở đây là lớp anonymous hiện tại đang dùng đến trường ui của Main. Ta sẽ phải gắn cái gì đó vào lớp mới để nó có thể phản ứng với một message. Việc dễ thấy nhất là truyền MainWindow vào cho nó, nhưng chúng tôi không vui vẻ gì với việc tạo quan hệ phụ thuộc vào một UI component. Điều đó gây khó khăn cho unit test, vì ta sẽ phải kiểm tra trạng thái của một component đang chạy tại thread của các event Swing.

Quan trọng hơn nữa, một quan hệ phụ thuộc như vậy sẽ phá với nguyên tắc "single responsibility": trách nhiệm mở gói các message thô từ auction đã khá là đủ việc cho một lớp, nếu thêm việc phải biết cách biểu diễn trạng thái của Sniper nữa thì quá nhiều. Như chúng tôi đã viết tại "Designing for Maintainability" (tr. 47), chúng tôi muốn giữ sự tách biệt giữa các mối quan tâm khác nhau (separation of concerns).

Do các điều kiện đó, chúng tôi quyết định rằng lớp AuctionMessageTranslator mới sẽ giao việc xử lý các event đã được giải nghĩa cho một lớp cộng tác, ta sẽ biểu diễn lớp này bằng một interface ActionEventListener; ta có thể truyền một đối tượng cài interface đó vào construction của AuctionMessageTranslator. Chúng tôi còn chưa biết cái gì nằm trong interface này, và cũng chưa bắt đầu nghĩ về cài đặt của nó. Mối quan tâm ngay bây giờ của chúng tôi là làm sao để translator chạy được; phần còn lại cứ để từ từ. Đến đây, thiết kế trông giống như trong Hình 12.1 (các kiểu thuộc về các framework bên ngoài, chẳng hạn Chat, được tô màu):



Hình 12.1: Lớp AuctionMessageTranslator.

Unit Test đầu tiên

Chúng tôi bắt đầu với loại event đơn giản hơn. Như ta đã thấy, một event loại Close không giữ giá trị gì – nó chỉ là một trigger đơn giản. Khi translator nhận được một message Close, chúng tôi muốn nó gọi listener của mình cho đúng cách.

Và vì đây là unit test đầu tiên, chúng tôi sẽ xây dựng nó rất chậm để thể hiện rõ được quy trình (sau này chúng tôi sẽ đi nhanh hơn). Bắt đầu với tên của phương thức test. JUnit dùng cơ chế riêng để nhặt ra các phương thức test, do đó ta có thể viết tên dài tùy ý và mô tả kỹ tùy ý vì ta không bao giờ phải tự gọi chúng trong code. Test đầu tiên nói rằng khi nhận được một message Close thô, translator sẽ bảo với bất cứ cái gì đang nghe rằng auction đã đóng.

```
package test.auctionsniper;

public class AuctionMessageTranslatorTest {
   @Test
   public void notifiesAuctionClosedWhenCloseMessageReceived() {
     // nothing yet
   }
}
```

Đặt các test vào một package khác

Chúng tôi có thói quen đặt test vào trong một package khác, không phải chính package chứa code mà chúng test. Chúng tôi muốn đảm bảo rằng chúng tôi đang lái code từ các giao diện public của chúng, như một client bất kì, thay vì mở cổng sau từ trong cùng một package để test.

Bước tiếp theo là thêm action mà nó sẽ kích hoạt hành vi mà chúng ta muốn test- trong trường hợp này là gửi một message Close.

```
public class AuctionMessageTranslatorTest {
  public static final Chat UNUSED_CHAT = null;
  private final AuctionMessageTranslator translator = new AuctionMessageTranslator();

  @Test
  public void notifiesAuctionClosedWhenCloseMessageReceived() {
    Message message = new Message();
    message.setBody("SOLVersion: 1.1; Event: CLOSE;");
    translator.processMessage(UNUSED_CHAT, message);
  }
}
```

Dùng null khi giá trị tham số không quan trọng

UNUSED_CHAT là một cái tên có nghĩa cho một hằng có giá trị null. Chúng tôi truyền nó vào trong processMesssage() thay cho một đối tượng Chat thực, vì khó tạo đối tượng Chat – constructor của nó có phạm vi package và để tạo được đối tượng ta sẽ phải điền cả một chuỗi các đối tượng mà nó phụ thuộc. Và đằng nào chúng tôi cũng chẳng cần đến nó cho chức năng hiện tại. Do đó chúng tôi chỉ truyền một giá trị null để thỏa mãn đòi hỏi của trình biên dịch,nhưng dùng một hằng có tên để làm rõ ý nghĩa của nó.

Xuất phát từ interface MessageListener, chúng tôi tạo một cài đặt khung cho AuctionMessageTranslator:

```
package auctionsniper;
```

```
public class AuctionMessageTranslator implements MessageListener {
    public void processMessage(Chat chat, Message message) {
      // TODO Fill in here
    }
}
```

Tiếp theo, chúng tôi muốn một lệnh kiểm tra xem việc dịch message có xảy ra hay không – nó sẽ fail vì chúng tôi chưa cài gì cả. Chúng tôi đã quyết định rằng chúng tôi muốn translator thông báo cho listener của nó khi xảy ra event Close, do đó chúng tôi mô tả hành vi đó trong test như sau:

```
@RunWith(JMock.class)
public class AuctionMessageTranslatorTest {
  private final Mockery context = new Mockery();
  private final AuctionEventListener listener = context.mock(AuctionEventListener.class);
  private final AuctionMessageTranslator translator = new AuctionMessageTranslator();

  @Test
  public void notifiesAuctionClosedWhenCloseMessageReceived() {
    context.checking(new Expectations() {{
      oneOf(listener).auctionClosed();
    }});
    Message message = new Message();
    message.setBody("SOLVersion: 1.1; Event: CLOSE;");
    translator.processMessage(UNUSED_CHAT, message);
  }
}
```

Phần đánh đậm là điểm quan trọng nhất trong test – tuyên bố của chúng tôi về hiệu ứng cần phải có của translator đối với môi trường của nó. Dòng đó nói rằng khi chúng tôi gửi một message thích hợp tới translator, chúng tôi trông đợi là nó sẽ gọi tới phương thức auctionClosed() của listener đúng một lần.

Chúng tôi nhận được kết quả test fail nói rằng chúng tôi chưa cài hành vi mình cần:

```
not all expectations were satisfied
expectations:
  ! expected once, never invoked: auctionEventListener.auctionClosed()
what happened before this: nothing!
  at org.jmock.Mockery.assertIsSatisfied(Mockery.java:199)
  […]
  at org.junit.internal.runners.JUnit4ClassRunner.run()
```

Phần quan trọng trong thông báo lỗi là:

```
expected once, never invoked: auctionEventListener.auctionClosed()
```

nó nói cho chúng tôi biết rằng chúng tôi chưa gọi listener như đáng ra đã phải làm. Chúng tôi cần làm 2 việc để pass được test này. Thứ nhất, chúng tôi cần nối translator và listener để chúng có thể liên lạc với nhau. Chúng tôi quyết định truyền listener vào constructor của translator; cách này đơn giản và đảm bảo được rằng translator sẽ được set-up với một listener.

Phần setup của test trông thế này:

```
public class AuctionMessageTranslatorTest {
  private final Mockery context = new Mockery();
  private final AuctionEventListener listener = context.mock(AuctionEventListener.class);
  private final AuctionMessageTranslator translator = new AuctionMessageTranslator(listener);
```

Việc thứ hai cần làm là gọi phương thức auctionClosed(). Thực ra đó là tất cả những gì chúng ta cần làm để pass test này, vì chúng ta chưa xác định bất kì hành vi nào khác.

```
public void processMessage(Chat chat, Message message) {
  listener.auctionClosed();
}
```

Test pass. Trông có vẻ như là gian lận, vì thực ra chúng tôi chưa mở gói message nào. Những gì chúng tôi đã làm là xác định các mảnh ghép và đặt chúng vào chỗ của nó – giải quyết dần từng mảnh chức năng, những mảnh này sẽ phải tiếp tục chạy đúng khi chúng tôi thêm những mảnh mới.

Đơn giản hóa Test Setup

Bạn đọc có thể đã nhận ra rằng tất cả các field trong test đều có dạng final. Như đã mô tả trong chương 3. JUnit tạo riêng cho mỗi phương thức test một đối tượng của lớp test, do đó, các field được tạo lại cho từng phương thức test. Chúng tôi khai thác điểm này bằng cách để càng nhiều field dạng final càng tốt, và khởi tạo chúng tại constructor, việc này sẽ loại bỏ mọi quan hệ phụ thuộc thành vòng tròn. Đôi khi chúng ta không thể làm như vậy mà phải trực tiếp gắn các quan hệ phụ thuộc, tuy nhiên ta có thể làm được trong đa số các trường hợp.

Closing the User Interface Loop

Giờ chúng tôi đã có sự khởi đầu của component mới, chúng tôi có thể gắn nó vào Sniper để đảm bảo chúng tôi không đi quá xa khỏi vùng code chạy được. Trước đây, Main đã update UI của Sniper, nên bây giờ ta bắt nó implement AuctionEventListener và chuyển chức năng tới phương thức auctionClosed() mới.

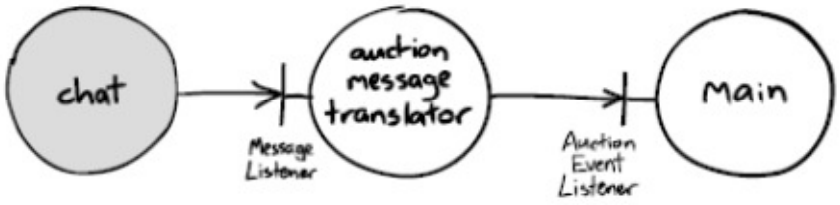
```
public class Main implements AuctionEventListener{ […]
  private void joinAuction(XMPPConnection connection, String itemId) throws XMPPException {

    disconnectWhenUICloses(connection);

    Chat chat = connection.getChatManager().createChat(
        auctionId(itemId, connection),
        new AuctionMessageTranslator(this));
    chat.sendMessage(JOIN_COMMAND_FORMAT);
    notToBeGCd = chat;
  }

  public void auctionClosed() {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
          ui.showStatus(MainWindow.STATUS_LOST);
        }
    });
  }
}
```

Cấu trúc hiện giờ giống như trong Hình 12.2.



Hình 12.2: Giới thiệu lớp AuctionMessageTranslator.

Chúng ta đã đạt được gì?

Tại bước bé xíu này, chúng tôi đã tách một feature của ứng dụng ra thành một lớp riêng. Chức năng đó giờ đã có một cái tên và có thể làm unit test cho nó. Chúng tôi cũng đã làm cho lớp Main đơn giản đi một chút, giờ nó không cần quan tâm đến việc dịch nội dung các message từ auction. Điều này chưa phải cái gì to tát, nhưng bạn đọc sẽ thấy rằng khi Sniper phát triển, cách tiếp cận này sẽ giúp chúng tôi giữ cho code được sạch sẽ và mềm dẻo, với việc từng component có trách nhiệm rõ ràng và quan hệ giữa chúng cũng rõ ràng mạch lạc.

Mở gói một message về giá - Unpacking a Price Message

Thêm các kiểu event cho message.

Chúng tôi chuẩn bị bổ sung kiểu message thứ hai từ auction – message để cập nhật giá hiện tại. Sniper cần phân biệt giữa hai loại, nên chúng tôi dẫn lại định dạng message trong chương 9 của Southabee's On-line như sau:

```
SOLVersion: 1.1; Event: PRICE; CurrentPrice: 192; Increment: 7; Bidder: Someone else;
SOLVersion: 1.1; Event: CLOSE;
```

Đầu tiên, chúng tôi thử mô hình hóa các message này dưới dạng các lớp, nhưng chúng tôi không thấy rõ về hành vi của chúng, nên chúng tôi từ bỏ ý định này. Chúng tôi quyết định bắt đầu từ một giải pháp đơn giản.

Unit test thứ hai

Việc đưa event Price vào test thứ hai sẽ buộc chúng tôi phải dịch cú pháp (parse) message nhận được. Test này có cấu trúc giống test trước, nhưng nhận được một xâu input khác và trông đợi việc gọi một phương thức khác của listener. Một message Price chứa thông tin chi tiết về bid mới nhất. Ta cần mở gói message, lấy các chi tiết đó và truyền cho listener. Do đó, chúng ta đưa chúng vào signature của phương thức mới currentPrice(). Test có nội dung như sau:

```
@Test
public void notifiesBidDetailsWhenCurrentPriceMessageReceived() {
  context.checking(new Expectations() {{
    exactly(1).of(listener).currentPrice(192, 7);
  }});
  Message message = new Message();
  message.setBody("SOLVersion: 1.1; Event: PRICE; CurrentPrice: 192; Increment: 7; Bidder: Someone else;");
  translator.processMessage(UNUSED_CHAT, message);
}
```

Để compile được, chúng tôi bổ sung một phương thức cho listener; việc này chỉ tốn có một cú gõ phím tại IDE

```
public interface AuctionEventListener {
  void auctionClosed();
  void currentPrice(int price, int increment);
}
```

Test fail.

```
unexpected invocation: auctionEventListener.auctionClosed()
expectations:
   ! expected once, never invoked: auctionEventListener.currentPrice(<192>, <7>)
what happened before this: nothing!
   […]
   at $Proxy6.auctionClosed()
```

```
   at auctionsniper.AuctionMessageTranslator.processMessage()
   at AuctionMessageTranslatorTest.translatesPriceMessagesAsAuctionPriceEvents()
   […]
   at JUnit4ClassRunner.run(JUnit4ClassRunner.java:42)
```

Nội dung quan trọng lần này là:

```
unexpected invocation: auctionEventListener.auctionClosed()
```

có nghĩa là code trong khi test chạy đã gọi nhầm phương thức: auctionClosed(). Vị trí xảy ra lỗi đã quá rõ ràng, chúng tôi chỉ việc sửa.

Phiên bản đầu tiên của chúng tôi còn thô, nhưng nó đã qua được test.

```
public class AuctionMessageTranslator implements MessageListener {
  private final AuctionEventListener listener;

  public AuctionMessageTranslator(AuctionEventListener listener) {
    this.listener = listener;
  }

  public void processMessage(Chat chat, Message message) {
    HashMap<String, String> event = unpackEventFrom(message);
    String type = event.get("Event");
    if ("CLOSE".equals(type)) {
      listener.auctionClosed();
    } else if ("PRICE".equals(type)) {
      listener.currentPrice(Integer.parseInt(event.get("CurrentPrice")),
                            Integer.parseInt(event.get("Increment")));
    }
  }

  private HashMap<String, String> unpackEventFrom(Message message) {
    HashMap<String, String> event = new HashMap<String, String>();
    for (String element : message.getBody().split(";")) {
      String[] pair = element.split(":");
      event.put(pair[0].trim(), pair[1].trim());
    }
    return event;
  }
}
```

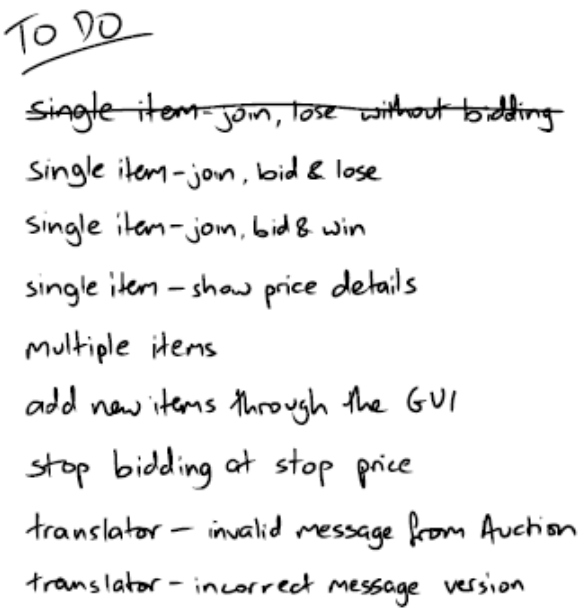
Cài đặt này tách thân message thành một tập của các cặp key/value, chúng được hiểu thành một event của auction để có thể thông báo cho AuctionEventListener. Chúng tôi cũng phải sửa FakeAuctionServer để gửi một event Close thật, thay vì một message rỗng như hiện nay, nếu không thì các test end-to-end sẽ fail một cách oan uổng.

```
public void announceClosed() throws XMPPException {
  currentChat.sendMessage("SOLVersion: 1.1; Event: CLOSE;");
}
```

Chạy test end-to-end lần nữa, chúng tôi được nhắc nhở rằng mình đang làm việc với feature đặt bid. Các test cho thấy status label của Sniper vẫn đang hiển thị trạng thái Joining thay vì Bidding.

Phát hiện ra các việc cần làm khác

Phần code đã pass được unit test, nhưng vẫn còn thiếu cái gì đó. Code giả định rằng message có cú pháp đúng và version đúng. Với thực tế là message sẽ đến từ bên ngoài hệ thống, chuyện này nghe hơi rủi ro, do đó chúng tôi cần thêm một số việc xử lí lỗi. Chúng tôi không muốn phá vỡ luồng công việc hiện tại (đang làm cho các feature chạy được), nên chúng tôi thêm mục xử lý lỗi vào trong to-do-list để sau này sẽ quay lại (Hình 12.3)



Hình 12.3. Bổ sung nhiệm vụ xử lý lỗi.

Chúng tôi cũng lo ngại về việc translator chưa được đủ rõ ràng về nhiệm vụ của nó: việc parse cú pháp và phát lệnh xử lý còn đang trộn lẫn vào nhau. Chúng tôi note lại để ghi nhớ rằng mình sẽ dọn dẹp lớp này ngay sau khi pass được acceptance test hiện tại – sắp rồi.

Kết thúc công việc

Hầu hết công việc trong chương này là để xác định chúng tôi muốn nói gì và nói điều đó như thế nào: Chúng tôi viết một test end-to-end để mô tả hành vi mà Sniper cần có; chúng tôi viết những cái tên unit test dài ngoằng để mô tả nhiệm vụ của một lớp; chúng tôi tách lớp mới để tách rời các khía cạnh khác nhau của chức năng; và chúng tôi viết khá nhiều phương thức nhỏ để giữ cho mức độ trừu tượng của từng layer của code được nhất quán. Nhưng trên hết, chúng tôi viết một cài đặt thô để chứng tỏ rằng chúng tôi biết cách làm cho code thực hiện được requirement, và sau đó chúng tôi sẽ refactor – việc của chương tiếp theo.

Chúng tôi muốn nhấn mạnh rằng "first-cut" của code (lát cắt đầu tiên để đắp vào bộ xương) chưa xong. Chúng tôi mới chỉ sắp xếp được các ý tưởng của mình và đảm bảo rằng đã xếp được các mảnh vào chỗ của nó, nhưng code còn chưa diễn đạt được chủ ý một cách sạch sẽ rõ ràng. Nếu không dọn dẹp, sự lộn xộn đó sẽ tích lũy dần và làm giảm năng xuất làm việc. Giống như làm đồ mộc mà không đánh giấy ráp – cuối cùng ai đó sẽ bị dằm đâm vào tay.