Chương 13: Sniper đấu giá

Ở đây chúng tôi tách một lớp AuctionSniper và tách ra các lớp nó phụ thuộc. Chúng tôi dùng một cài đặt rỗng của auction để cắm được lớp mới vào phần còn lại của ứng dụng, cho đến khi chúng tôi sẵn sàng gửi lệnh. Chúng tôi dùng một lớp XMPPAuction để nối vòng tương tác với auction house. Chúng tôi tiếp tục đẽo ra các lớp mới từ code hiện hành.

Introducing AuctionSniper

Một lớp mới, với các lớp mà nó phụ thuộc

Ứng dụng của chúng tôi đã nhận các event Price gửi từ auction, nhưng chưa thể giải nghĩa chúng. Chúng tôi cần code thực hiện hai hành động khi phương thức currentPrice() được gọi: gửi một bid giá cao hơn tới auction và cập nhật status tại UI. Chúng tôi có thể mở rộng Main, nhưng lớp này đang trông khá là lộn xộn – nó đã có quá nhiều nhiệm vụ rồi. Có vẻ đã đến lúc tạo thêm cái mà chúng tôi gọi là một "Auction Sniper", component nằm tại trung tâm ứng dụng của chúng tôi. Chúng tôi tạo lớp AuctionSniper. Một số hành vi đáng ra là của nó hiện đang nằm trong Main, và chúng tôi nên bắt đầu với việc tách hành vi đó ra và đưa vào lớp mới – việc này sẽ cần hơi nhiều công sức như chúng ta sẽ thấy.

Với việc một AuctionSniper cần đáp ứng các event Price, chúng tôi quyết định cho nó cài interface AuctionEventListener thay cho Main. Câu hỏi đặt ra là làm gì với UI, nếu ta xem xét di chuyển phương thức này:

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

Liệu có hợp lý hay không khi một đối tượng AuctionSniper biết về chi tiết cài đặt của UI, chẳng hạn như cách dùng Swing thread? Chúng tôi sẽ lại có nguy cơ phá vỡ nguyên lý "single responsibility" lần nữa. Chắc chắn một đối tượng AuctionSniper cần quan tâm đến chiến lược đặt giá và chỉ thông báo các thay đổi về trạng thái bằng ngôn ngữ của *chính mình*.

Giải pháp của chúng tôi là cô lập AuctionSniper bằng cách tạo thêm một quan hệ mới: nó sẽ thông báo cho một SniperListener khi nó thay đổi trạng thái. Interface mới này và unit test đầu tiên cho nó như sau:

```
public interface SniperListener extends EventListener {
  void sniperLost();
}

@RunWith(JMock.class)
public class AuctionSniperTest {
  private final Mockery context = new Mockery();
  private final SniperListener sniperListener = context.mock(SniperListener.class);
  private final AuctionSniper sniper = new AuctionSniper(sniperListener);
  @Test
  public void reportsLostWhenAuctionCloses() {
    context.checking(new Expectations() {{
```

```
      one(sniperListener).sniperLost();
    }});
    sniper.auctionClosed();
  }
}
```

Test nói rằng, khi nhận được một event Close gửi từ auction, Sniper cần báo cáo rằng nó đã thua.

Thông báo lỗi khi test fail như sau:

```
not all expectations were satisfied
expectations:
! expected exactly 1 time, never invoked: SniperListener.sniperLost();
```

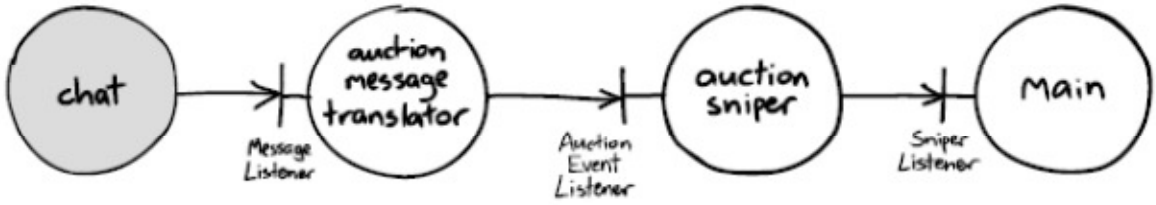
Chúng tôi có thể cho test pass bằng một cài đặt đơn giản:

```
public class AuctionSniper implements AuctionEventListener {
  private final SniperListener sniperListener;

  public AuctionSniper(SniperListener sniperListener) {
    this.sniperListener = sniperListener;
  }

  public void auctionClosed() {
    sniperListener.sniperLost();
  }

  public void currentPrice(int price, int increment) {
    // TODO Auto-generated method stub
  }
}
```

Cuối cùng, chúng tôi lắp AuctionSniper vào hệ thống bằng cách cho Main cài SniperListener.

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

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

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

Test end-to-end của chúng tôi, cái vốn đã pass giờ vẫn pass, còn cái đang fail giờ vẫn fail ở lỗi cũ. Như vậy là chúng tôi chưa làm hỏng cái gì. Cấu trúc mới được minh họa trong Hình 13.1.



Hình 13.1 Lắp AuctionSniper vào vị trí.

Tập trung, tập trung, tập trung

Một lần nữa, chúng tôi đã nhận ra sự phức tạp quá mức của một lớp và tận dụng nó để tách từ cài đặt khung xương ra một khái niệm mới. Bây giờ, chúng tôi có một đối tượng Sniper với trách nhiệm phản ứng với các event nhận được từ translator. Như bạn đọc sẽ sớm nhận ra, đây là cấu trúc thể hiện được rõ hơn chức năng của code, tạo thuận lợi cho unit test. Chúng tôi cũng thấy rằng phương thức sniperLost() đã rõ nghĩa hơn kiếp trước của nó là auctionClosed(): tên của phương thức đã khớp hơn với nhiệm của phương thức là báo cáo về một cuộc đấu giá thất bại.

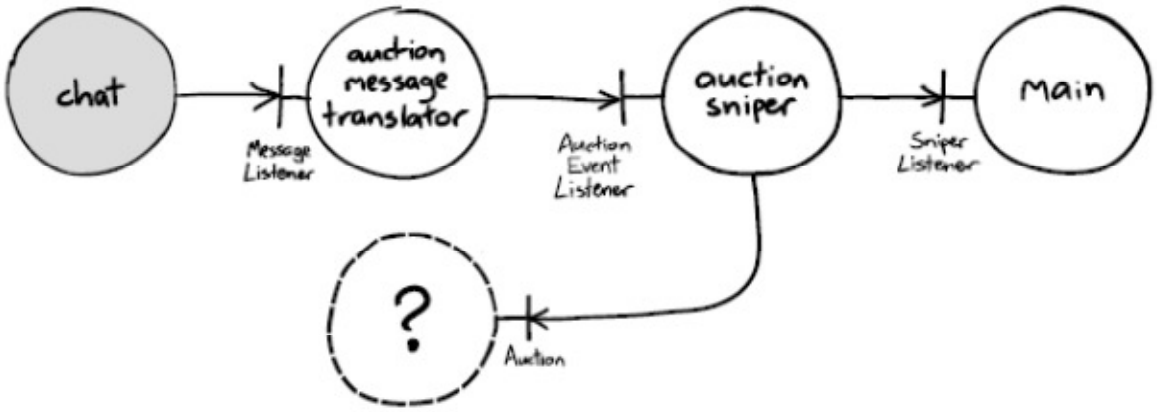
Đó có phải là công việc táy máy tỉa tót tốn thời gian hay không? Rõ ràng là chúng tôi không nghĩ như thế, đặc biệt ở giai đoạn đầu dự án như bây giờ, khi chúng tôi còn đang sắp xếp các ý tưởng. Có những đội phát triển đã bỏ nhiều công sức cho thiết kế hơn là cần thiết. Nhưng kinh nghiệm của chúng tôi là: hầu hết các đội phát triển dành quá ít thời gian cho việc dọn dẹp làm rõ code, và sau đó họ phải trả giá cho việc đó bằng chi phí bảo trì cao. Như chúng tôi đã trình bày vài lần, nguyên lý “single responsibility” là một quy tắc gần tối ưu rất hữu hiệu để phá vỡ những vùng phức tạp. Và các developer không nên e ngại về việc tạo class mới. Chúng tôi thấy rằng Main vẫn làm quá nhiều việc, nhưng chúng tôi chưa rõ tách nó ra như thế nào thì tốt nhất. Chúng tôi quyết định cứ đi tiếp để xem code sẽ dẫn chúng tôi đi đâu.

Gửi một Bid

Auction Interface

Bước tiếp theo là cho Sniper gửi một bid tới cuộc đấu giá (auction). Vậy Sniper nên liên lạc với ai? Có vẻ không nên mở rộng SniperListener, vì quan hệ giữa Sniper và một SniperListener là để theo dõi chuyện gì đang xảy ra bên trong Sniper, không phải để thực hiện các nhiệm vụ bên ngoài. Nói theo thuật ngữ đã được định nghĩa tại “Object Peer Stereotypes” (tr.52), SniperListener là một notification (nơi nhận thông báo của Sniper), không phải một dependency (cái mà Sniper phụ thuộc vào).

Sau khi thảo luận, chúng tôi quyết định tạo mới Auction: một dạng đối tượng cộng tác (collaborator). Auction và SniperListener đại diện cho hai domain khác nhau trong ứng dụng. Auction là về các giao dịch tài chính, nó nhận bid dành cho các item đang được bán. Còn SniperListener là về các feedback đối với ứng dụng, nó báo cáo các thay đổi về trạng thái của Sniper. Auction là một dependency, vì không có nó thì Sniper không thể hoạt động. Trong khi đó, như chúng tôi đã nói ở trên, SniperListener không như vậy. Việc tạo interface mới làm cho thiết kế trở thành như trong Hình 13.2



Hình 13.2 Tạo thêm interface Auction.

AuctionSniper đặt Bid

Bây giờ chúng tôi đã sẵn sàng đặt bid. Bước đầu tiên là cài hành động phản ứng đối với một event Price. Để bắt đầu, chúng tôi thêm một unit test cho lớp AuctionSniper. Test nói rằng khi nhận được một message Price cập nhật giá, Sniper sẽ gửi một bid cho auction với giá đã được tăng lên. Sniper cũng thông báo với listener của nó rằng nó đang ở trạng thái bidding, nên chúng tôi thêm phương thức sniperBidding(). Chúng tôi đang giả định ngầm rằng đối tượng Auction biết rằng Sniper đang đại diện cho bidder nào, vậy nên Sniper không phải truyền thông tin đó theo bid.

```
public class AuctionSniperTest {
  private final Auction auction = context.mock(Auction.class);
  private final AuctionSniper sniper = new AuctionSniper(auction, sniperListener);
  […]

  @Test
  public void bidsHigherAndReportsBiddingWhenNewPriceArrives() {
    final int price = 1001;
    final int increment = 25;
    context.checking(new Expectations() {{
        one(auction).bid(price + increment);
        atLeast(1).of(sniperListener).sniperBidding();
    }});
    sniper.currentPrice(price, increment);
  }
}
```

Thông báo lỗi khi test fail như sau:

```
not all expectations were satisfied
expectations:
  ! expected once, never invoked: auction.bid(<1026>)
  ! expected at least 1 time, never invoked: sniperListener.sniperBidding()
what happened before this: nothing!
```

Trong khi viết test này, chúng tôi đã nhận ra rằng mình thực ra không quan tâm việc Sniper thông báo cho listener nhiều hơn 1 lần rằng nó đang bidding; đó chỉ là một lệnh cập nhật trạng thái, nên chúng tôi dùng mệnh đề atLeast(1) cho expectation về listener. Mặt khác, chúng tôi có quan tâm rằng bid chỉ được gửi đúng một lần, do đó chúng tôi dùng mệnh đề one() cho expectation cho action. Trong thực tiễn, tất nhiên chúng tôi chắc sẽ chỉ

gọi listener một lần, nhưng điều kiện lỏng này của test diễn đạt được chủ ý của chúng tôi về quan hệ với hai đối tượng này. Chúng tôi cũng dùng mệnh đề atLeast(1) cho các phương thức test khác.

Ta mô tả các giá trị kỳ vọng như thế nào?

Chúng tôi đã đặc tả giá trị kì vọng của bid bằng cách cộng thêm increment vào price. Có các quan điểm khác về việc các giá trị test chỉ nên là các literal với giá trị thực tiếp dễ thấy, hay nên được biểu diễn bằng công thức tính toán mà chúng đại diện. Việc viết các biểu thức tính toán có thể làm cho test dễ đọc hơn nhưng lại có nguy cơ implement lại chính đoạn code cần test. Và trong một số trường hợp, phần tính toán có thể quá phức tạp để có thể tái tạo được trong test. Ở đây, chúng tôi quyết định rằng các tính toán đủ đơn giản để chúng tôi chỉ việc viết thẳng vào test.

---

jMock Expectation không phải được kiểm tra theo thứ tự

Đây là test đầu tiên của chúng tôi với nhiều hơn một expectation, cho nên chúng tôi sẽ nhấn mạnh rằng thứ tự khai báo các expectation không cần phải khớp với thứ tự các phương thức được gọi trong code. Nếu thứ tự gọi là cái quan trọng, các expectation cần có thêm mệnh đề sequence, xem thêm tại Appendix A

Cài đặt để làm cho test pass rất đơn giản.

```
public interface Auction {
  void bid(int amount);
}

public class AuctionSniper implements AuctionEventListener { […]
  private final SniperListener sniperListener;
  private final Auction auction;

  public AuctionSniper(Auction auction, SniperListener sniperListener) {
    this.auction = auction;
    this.sniperListener = sniperListener;
  }

  public void currentPrice(int price, int increment){
    auction.bid(price + increment);
    sniperListener.sniperBidding();
  }
}
```

Đặt bid thành công với AuctionSniper

Bây giờ chúng tôi phải gắn AuctionSniper vào ứng dụng. Phần dễ làm là hiển thị trạng thái bidding, phần hơi khó hơn là gửi bid cho auction. Việc đầu tiên là làm cho code compile. Chúng tôi cài phương thức mới sniperBidding vào lớp Main, và để tránh phải để code quá lâu trong tình trạng không compile, chúng tôi truyền cho AuctionSniper một cài đặt rỗng của Auction.

```
public class Main implements SniperListener { […]
  private void joinAuction(XMPPConnection connection, String itemId) throws XMPPException {
    Auction nullAuction = new Auction() {
         public void bid(int amount) {}
    };
```

```
    disconnectWhenUICloses(connection);

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

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

Vậy cái gì nằm trong cài đặt Auction? Nó cần truy nhập vào chat để nó có thể gửi một message bid. Để tạo chat, constructor của nó cần một translator, translator cần một Sniper, và Sniper cần một Auction. Chúng tôi đang có một quan hệ phụ thuộc vòng tròn (dependency loop) mà chúng tôi cần phá vỡ.

Nhìn lại thiết kế, có một đôi chỗ mà chúng tôi có thể can thiệp, nhưng té ra rằng API của ChatManager gây hiểu nhầm. Nó không đòi hỏi MessageListener tạo một đối tượng Chat, mặc dù phương thức createChat() có hàm ý như vậy. Theo ngôn ngữ của chúng tôi, MessageListener là một notification; chúng tôi có thể truyền null vào khi tạo Chat và rồi bổ sung một đối tượng MessageListener sau.

Diễn đạt chủ ý tại API

Chúng tôi có thể phát hiện rằng mình có thể truyền MessageListener null chỉ là vì chúng tôi có mã nguồn của thư viện Smack. Chi tiết này không rõ ràng khi đọc API, có lẽ vì tác giả muốn ép người dùng làm cho đúng, và không rõ vì sao ai đó lại muốn một đối tượng Chat không có listener.

Bây giờ chúng tôi có thể cấu trúc lại phần cài đặt kết nối và dùng Chat để gửi bid.

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

    final Chat chat = connection.getChatManager().createChat(auctionId(itemId, connection), null);
    this.notToBeGCd = chat;

    Auction auction = new Auction() {
        public void bid(int amount) {
          try {
            chat.sendMessage(String.format(BID_COMMAND_FORMAT, amount));
          } catch (XMPPException e) {
            e.printStackTrace();
          }
        }
    };
    chat.addMessageListener(new AuctionMessageTranslator(new AuctionSniper(auction, this)));
    chat.sendMessage(JOIN_COMMAND_FORMAT);
  }
}
```

Cài đặt rỗng - Null Implementation

Chúng tôi dùng cài đặt rỗng để làm cài đặt tạm thời, được thêm vào để lập trình viên có thể cho công việc tiến triển ở chỗ khác và sẽ quay lại thay bằng cài đặt với đầy đủ.

The End-to-End Tests Pass

Bây giờ các test end-to-end đã pass: Sniper có thể thua khi không bid lần nào, và thua sau khi đặt một bid. Chúng tôi có thể gạch thêm một mục từ to-do-list. Nhưng XMPP exception trong test mới chỉ được xử lý bằng cách bắt và in ra. Bình thường, chúng tôi coi đây là cách xử lý rất tồi (very bad practice), nhưng chúng tôi muốn nhìn thấy test pass và muốn đưa được một cấu trúc vào trong code – và chúng tôi biết rằng các test end-to-end đằng nào cũng sẽ fail nếu có trục trặc khi gửi message. Để đảm bảo là mình sẽ không quên, chúng tôi thêm một mục vào to-do-list để ghi nhớ rằng phải tìm một giải pháp tốt hơn, Hình 13.3.

Hình 13.3. Tiến thêm một bước.

Dọn dẹp cài đặt

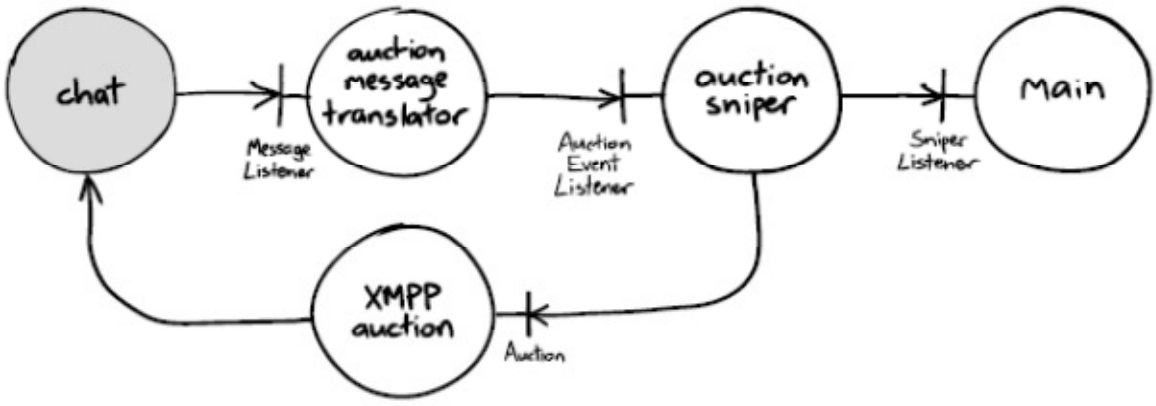
Tách lớp XMPPAuction

Các test end -to-end đã pass, nhưng chúng tôi chưa xong việc vì cài đặt mới còn lộn xộn. Chúng tôi thấy rằng nội dung của joinAuction() dính đến nhiều domain: quản lí chat, gửi bid, tạo sniper, v.v.. Chúng tôi cần dọn dẹp. Để bắt đầu, chúng tôi thấy rằng mình đang gửi lệnh auction từ hai level khác nhau, tại top level và từ bên trong Auction. Việc gửi lệnh tới một cuộc đấu giá nghe có vẻ như loại công việc mà đối tượng Auction của chúng tôi nên làm, do đó đóng gói chúng vào với nhau có vẻ là việc hợp lí. Chúng tôi thêm một phương thức mới vào interface Auction, mở rộng cài đặt anonymous, và tách nó ra thành một lớp nested (tạm như vậy). Chúng tôi cần một cái tên cho lớp này. Điểm nổi bật của cài đặt này cho interface Auction là nó dựa trên cơ sở hạ tầng gửi và nhận message. Vậy nên chúng tôi gọi lớp mới là XMPPAuction.

```
public class Main implements SniperListener { […]
  private void joinAuction(XMPPConnection connection, String itemId) {
    disconnectWhenUICloses(connection);

    final Chat chat = connection.getChatManager().createChat(auctionId(itemId, connection), null);
    this.notToBeGCd = chat;

    Auction auction = new XMPPAuction(chat);
    chat.addMessageListener( new AuctionMessageTranslator(new AuctionSniper(auction, this)));
    auction.join();
  }

  public static class XMPPAuction implements Auction {
    private final Chat chat;
    public XMPPAuction(Chat chat) {
      this.chat = chat;
    }

    public void bid(int amount) {
      sendMessage(format(BID_COMMAND_FORMAT, amount));
    }

    public void join() {
      sendMessage(JOIN_COMMAND_FORMAT);
    }

    private void sendMessage(final String message) {
      try {
        chat.sendMessage(message);
      } catch (XMPPException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Chúng tôi bắt đầu nhìn thấy một mô hình domain rõ ràng hơn. Dòng auction.join() thể hiện chủ ý của chúng tôi rõ ràng hơn cài đặt chi tiết trước đây, khi code gửi một string tới chat. Thiết kế mới được mô tả trong Hình 13.4 và chúng tôi nâng XMPPAuction thành một lớp ở top-level.

Chúng tôi vẫn thấy joinAuction() không rõ ràng, và chúng tôi muốn kéo các chi tiết liên quan đến XMPP ra khỏi Main. Nhưng chúng tôi chưa sẵn sàng cho việc đó. Một điểm nữa cần nhớ.



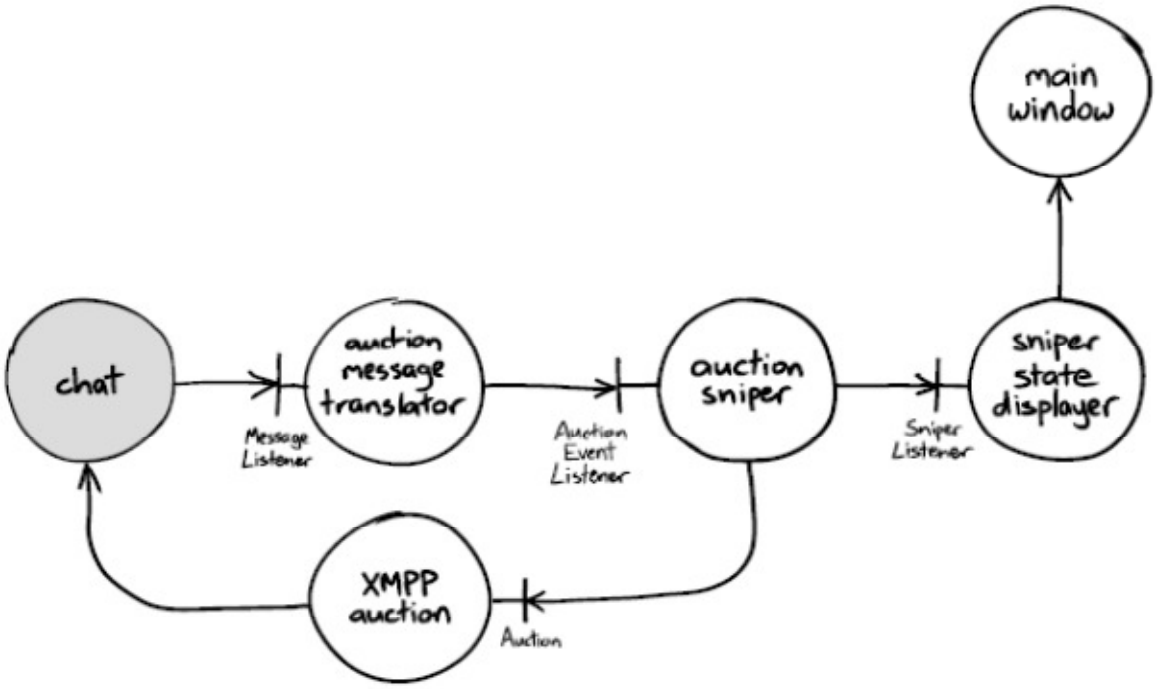
Hình 13.4 Hoàn thiện chu trình với XMPPAuction

Extracting the User Interface

Các hoạt động khác trong lớp Main đang cài đặt UI và hiển thị trạng thái hiện tại để đáp ứng các event nhận được từ Sniper. Chúng tôi không thực sự thoải mái với tình trạng Main cài SniperListener; một lần nữa, nó có cảm giác như là các trách nhiệm khác nhau bị trộn lẫn lộn (khởi động ứng dụng và đáp ứng event). Chúng tôi quyết định tách hành vi kiểu SniperListener vào một lớp nested nằm trong Main, với cái tên tốt nhất mà chúng tôi có thể nghĩ ra là SniperStateDisplayer. Lớp mới này là cầu nối giữa hai domain: nó dịch Sniper event thành một biểu diễn mà Swing có thể hiển thị, điều đó bao hàm làm việc với cơ chế threading của Swing. Chúng tôi lắp một đối tượng của lớp mới vào AuctionSniper.

```
public class Main { // doesn't implement SniperListener
  private MainWindow ui;

  private void joinAuction(XMPPConnection connection, String itemId) {
    disconnectWhenUICloses(connection);
    final Chat chat = connection.getChatManager().createChat(auctionId(itemId, connection), null);
    this.notToBeGCd = chat;

    Auction auction = new XMPPAuction(chat);
    chat.addMessageListener(
        new AuctionMessageTranslator(
            connection.getUser(),
            new AuctionSniper(auction, new SniperStateDisplayer())));
    auction.join();
  }
  […]

  public class SniperStateDisplayer implements SniperListener {
    public void sniperBidding() {
     showStatus(MainWindow.STATUS_BIDDING);
    }

    public void sniperLost() {
      showStatus(MainWindow.STATUS_LOST);
    }

    public void sniperWinning() {
      showStatus(MainWindow.STATUS_WINNING);
    }

    private void showStatus(final String status) {
      SwingUtilities.invokeLater(new Runnable() {
          public void run() { ui.showStatus(status); }
      });
    }
  }
}
```

Hình 13.5 cho thấy ta đã tối giản Main đến mức nó không còn phải tham gia việc chạy ứng dụng (để dễ đọc, chúng tôi đã bỏ qua WindowAdapter với nhiệm vụ đóng connection). Nó giờ chỉ còn một việc là tạo các component khác nhau và lắp ghép chúng với nhau. Chúng tôi đã coi MainWindow làm một lớp ngoại vi để đại diện cho Swing framework.



Hình 13.5 Extracting SniperStateDisplayer

Dọn dẹp Translator

Cuối cùng, chúng tôi cũng thực hiện được lời hứa quay lại với AuctionMessageTranslator. Chúng tôi bắt đầu với việc giảm những đoạn code râu ria (noise) bằng cách bổ sung các constant và static import, và một số phương thức helper để giảm trùng lặp code.

Sau đó chúng tôi thấy rằng phần lớn code thao tác ánh xạ name/value và khá là mang tính chất quy trình. Chúng tôi có thể cải thiện thiết kế bằng cách trích ra một inner class, AuctionEvent, để đóng gói công việc mở gói nội dung message. Chúng tôi yên tâm là mình có thể refactor lớp này một cách an toàn, vì nó được unit test bảo vệ.

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

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

  public void processMessage(Chat chat, Message message) {
    AuctionEvent event = AuctionEvent.from(message.getBody());

    String eventType = event.type();
    if ("CLOSE".equals(eventType)) {
      listener.auctionClosed();
    } else if ("PRICE".equals(eventType)) {
      listener.currentPrice(event.currentPrice(), event.increment());
    }
  }
  private static class AuctionEvent {
    private final Map<String, String> fields = new HashMap<String, String>();
    public String type() { return get("Event"); }
```

```
    public int currentPrice() { return getInt("CurrentPrice"); }
    public int increment() { return getInt("Increment"); }
    private int getInt(String fieldName) {
      return Integer.parseInt(get(fieldName));
    }
    private String get(String fieldName) { return fields.get(fieldName); }
    private void addField(String field) {
      String[] pair = field.split(":");
      fields.put(pair[0].trim(), pair[1].trim());
    }
    static AuctionEvent from(String messageBody) {
      AuctionEvent event = new AuctionEvent();
      for (String field : fieldsIn(messageBody)) {
        event.addField(field);
      }
      return event;
    }
    static String[] fieldsIn(String messageBody) {
      return messageBody.split(";");
    }
  }
}
```

Đây là một ví dụ của "breaking out" mà chúng tôi đã trình bày tại "Value Types" (tr.59). Có vẻ không hiển nhiên lắm, nhưng AuctionEvent là một giá trị: nó bất biến và không có gì khác biệt giữa hai event với nội dung giống nhau. Cải tiến lần này tách rời các mối quan tâm bên trong AuctionMessageTranslator: top level làm việc với các event và listener, và đối tượng inner làm nhiệm vụ dịch cú pháp string.

Đóng gói các kiểu collection

Mặc dù Java generics tránh được việc phải đổi kiểu object, chúng tôi vẫn có thói quen đóng gói cả các kiểu thông dụng, chẳng hạn các collection, bên trong các lớp của chính chúng tôi. Chúng tôi cố gắng sử dụng ngôn ngữ của bài toán hiện đang giải quyết, thay vì ngôn ngữ của các cấu trúc Java. Trong hai phiên bản processMessage() của chúng tôi, bản đầu tiên có nhiều code râu ria về chuyện tra cứu và dịch cú pháp các giá trị. Bản thứ hai được viết bằng ngôn ngữ của các event đấu giá, do đó rút ngắn khoảng cách ngữ nghĩa giữa domain và code – code diễn đạt nội dung domain được rõ ràng hơn.

Quy tắc thực dụng của chúng tôi là: cố gắng hạn chế việc truyền qua lại các kiểu generic (ngoặc <>). Cụ thể, đối với các collection, chúng tôi xem đó là một dạng lặp code (duplication). Đó là dấu hiệu rằng có một khái niệm domain nên được tách thành một kiểu dữ liệu.

Trì hoãn quyết định

Có một kĩ thuật mà đến đây chúng tôi đã dùng vài lần, đó là bổ sung một cài đặt rỗng của một phương thức (hoặc một lớp) để đi bước tiếp theo. Việc này giúp chúng tôi tập trung vào nhiệm vụ trước mắt thay vì bị kéo sang việc suy nghĩ về nhóm chức năng quan trọng tiếp theo. Ví dụ, cài đặt rỗng của Auction cho phép chúng tôi lắp một quan hệ mới tìm thấy vào một unit test mà không bị sa đà vào các vấn đề về xử lý trao đổi message. Điều đó còn có nghĩa rằng chúng tôi có thể dừng lại để suy nghĩ về các quan hệ phụ thuộc giữa các đối tượng của mình mà không phải chịu áp lực của tình trạng code chưa compile.

Giữ cho code ở tình trạng compile được

Chúng tôi cố gắng giảm thiểu những khoảng thời gian mà code không compile được, bằng cách chỉ thay đổi từng bước nhỏ. Khi gặp lỗi compile, ta không thể chắc chắn các thay đổi của ta đang ảnh hưởng bao xa, vì trình biên dịch không thể cho ta biết. Điều đó có nghĩa chúng tôi không thể check-in vào repository – việc mà chúng tôi muốn làm càng thường xuyên càng tốt. Càng có nhiều đoạn code đang dở dang, càng có nhiều thứ chúng ta phải chú ý hơn, và chính việc này lại thường có nghĩa rằng chúng ta sẽ tiến càng chậm hơn. Một trong những phát kiến vĩ đại nhất của TDD là chúng ta có thể tổ chức sao cho các bước phát triển là rất nhỏ.

Thiết kế dần dần lộ ra - Emergent Design

Điều chúng tôi hy vọng có thể trở nên rõ ràng trong chương này là cách mà chúng tôi nuôi dưỡng cho một thiết kế phát triển dần lên từ một khởi đầu có vẻ không hứa hẹn gì. Đại loại, có hai công việc mà chúng tôi thực hiện luân phiên: thêm tính năng mới và ngắm nghía dọn dẹp phần code được tạo ra. Giai đoạn dọn dẹp có vai trò cốt tử, nếu không có nó, chúng tôi sẽ có kết cục là một đống code hổ lốn không thể bảo trì được. Nếu có đoạn nào chưa rõ nên dọn dẹp thế nào, chúng tôi sẽ tạm hoãn đoạn đó, nhưng chắc chắn sẽ dành thời gian quay lại khi có thể. Trong lúc đó, chúng tôi giữ cho code sạch sẽ hết mức có thể, tiến theo từng bước nhỏ, và dùng các kĩ thuật như cài đặt rỗng (null implementation) để giảm thiểu thời gian code không dịch được.

Hình 13.5 cho thấy chúng tôi đã xây dựng một lớp đệm xung quanh core implementation, "bảo vệ" nó khỏi các quan hệ phụ thuộc với bên ngoài. Chúng tôi cho đây là giải pháp tốt. Nhưng điều thú vị ở đây là: chúng tôi tiến đến giải pháp này từng bước một, bằng cách tìm các tính năng trong các lớp mà chúng hoặc là nên đi với nhau hoặc không nên đi với nhau. Tất nhiên, chúng tôi có chịu ảnh hưởng của kinh nghiệm làm việc với những dự án tương tự. Nhưng chúng tôi rất cố gắng đi theo những gì code hiện tại đang nói, thay vì áp các quan niệm và kinh nghiệm có sẵn. Đôi khi, khi chúng tôi làm theo cách này, chúng tôi thấy rằng domain dẫn chúng tôi đi theo những hướng rất đáng kinh ngạc.