Chương 10: Bộ xương di động

Ở đây, chúng tôi thiết lập môi trường phát triển và viết test end-to-end đầu tiên. Chúng tôi đưa ra một số quyết định về cơ sở hạ tầng để có thể khởi công, xây dựng một bản build. Một lần nữa, chúng tôi lại ngạc nhiên về khối lượng công sức phải bỏ ra cho nó.

Lôi bộ xương ra từ trong rương.

Giờ đã có khái niệm về cái cần phải xây dựng, chúng tôi có thể bắt đầu từ đó và viết unit test đầu tiên?

Chưa đâu.

Công việc đầu tiên là tạo "bộ xương di dộng" mà chúng tôi đã tả ở mục "Đầu tiên, test một bộ xương di động" (trang 32). Một lần nữa, mục đích của bộ xương di động là để giúp ta hiểu về các requirement đủ rõ để có thể đề xuất và thẩm định một kiến trúc hệ thống ở mức phác thảo. Sau này ta vẫn có thể thay đổi quyết định, khi ta đã học được nhiều hơn. Nhưng điểm quan trọng là ta xuất phát từ cái gì đó phản ánh được khung cảnh của giải pháp của ta. Ngoài ra, một điều quan trọng khác là nhờ nó ta có thể đánh giá được cách tiếp cận mà ta đã chọn và test được các quyết định, để sau này ta có thể sửa đổi một cách tự tin.

Đối với hầu hết các dự án, việc phát triển khung xương đòi hỏi một khối lượng công sức đáng ngạc nhiên. Đầu tiên, việc xác định phải làm gì sẽ làm xả ra đủ loại câu hỏi về ứng dụng và vị trí của nó trong thế giới thực. Thứ hai, việc tự động hóa build, package, và deploy vào một môi trường như thật (một khi ta hiểu được môi trường thật nghĩa là sao) sẽ làm xả ra đủ loại câu hỏi về kĩ thuật và tổ chức.

Iteration Zero

Ở hầu hết các dự án Agile, có một giai đoạn đầu tiên khi mà đội phát triển thực hiện phân tích khởi đầu, thiết lập các môi trường vật lí và kĩ thuật, hoặc không thì cũng chuẩn bị bắt đầu. Khi đó đội phát triển không tạo thêm được chức năng nào nhìn thấy được, vì hầu hết công việc nằm ở hạ tầng cơ sở, do đó có thể không hợp lí lắm khi tính đây là một iteration đối với việc lập kế hoạch. Một cách làm thông dụng là gọi bước này là iteration zero: "iteration" vì đội vẫn cần phải tính thời gian cho hoạt động này, và "zero" vì nó xảy ra trước iteration one – khi việc phát triển chức năng bắt đầu. Một nhiệm vụ quan trọng của iteration zero là dùng bộ xương di động để TDD kiến trúc ban đầu.

Tất nhiên, chúng tôi sẽ bắt đầu dựng bộ xương di động bằng việt viết một test.

Test đầu tiên

Bộ xương di động phải phủ được mọi component của hệ thống Auction Sniper: giao diện người dùng, sniping component, và phần liên lạc với một auction server. Lát cắt mỏng nhất mà ta có thể hình dung cho việc test, gạch đầu dòng đầu tiên trong to-do-list của chúng tôi, là Auction Sniper có thể gia nhập một cuộc đấu giá rồi đợi nó kết thúc. Lát cắt này nhỏ đến mức chúng tôi thậm chí không cần quan tâm đến việc gửi bid; chúng tôi chỉ muốn biết rằng hai bên có thể liên lạc với nhau và rằng chúng tôi có thể test hệ thống từ bên ngoài (thông qua GUI và việc đẩy vào một vài event như thể event từ auction server bên ngoài. Một khi

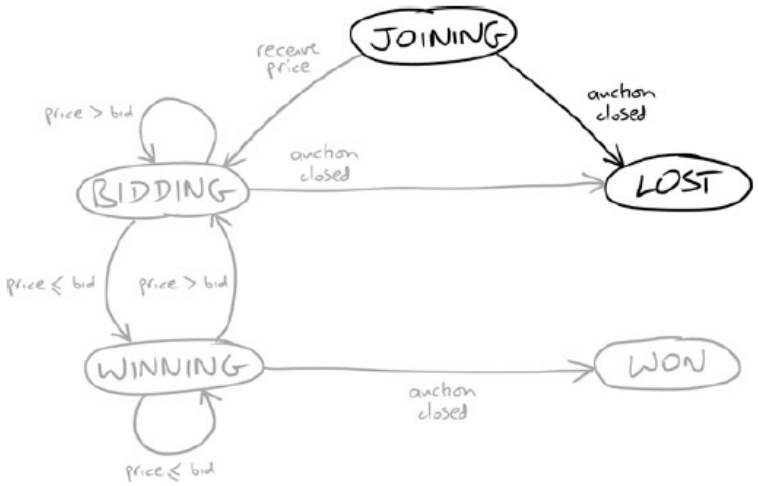
kịch bản đó chạy được, chúng tôi sẽ có cái nền móng vững chắc để có thể xây trên đó các feature còn lại mà khách hàng muốn.

Chúng tôi muốn bắt đầu bằng việc viết một test như thể production code phục vụ nó đã có sẵn, và sau đó sẽ điền thêm vào đó bất cứ cái gì cần thiết để làm cho nó chạy được. Làm giật lùi từ test giúp chúng ta tập trung vào cái mà ta muốn hệ thống làm được (what), thay vì bị cuốn vào sự phức tạp của việc làm thế nào để nó làm được (*how*). Do đó, đầu tiên, ta viết một test để mô tả các chủ ý của ta càng rõ càng tốt, trong phạm vi khả năng diễn đạt của ngôn ngữ lập trình đang dùng. Sau đó, ta xây dựng cơ sở hạ tầng để hỗ trợ ta test hệ thống theo cách ta muốn, thay vì viết test sao cho phù hợp với một cơ sở hạ tầng có sẵn. Việc này thường chiếm phần không nhỏ các nỗ lực ban đầu của ta, vì có rất nhiều việc để mà chuẩn bị. Một khi cơ sở hạ tầng đã sẵn sàng, ta có thể cài feature để test pass.

Phác thảo về test mà chúng tôi muốn làm như sau:

1. Khi một cuộc đấu giá đang bán một item
2. Và một Auction Sniper đã bắt đầu gia nhập auction đó
3. Thì auction sẽ nhận được một yêu cầu gia nhập (Join request) mà Auction Sniper đó gửi
4. Khi một auction tuyên bố rằng nó đã kết thúc,
5. Thì Auction Sniper sẽ hiển thị rằng nó đã thua cuộc.

Nội dung đó mô tả một transition trong máy trạng thái Hình 10.1. Chúng tôi cần dịch nó thành cái gì đó chạy được. Chúng tôi dùng JUnit làm test framework vì nó đã quen dùng và được hỗ trợ rộng rãi. Chúng tôi cũng cần các cơ chế để điều khiển ứng dụng và cuộc đấu giá mà ứng dụng đang giao tiếp với. Các dịch vụ test tại Southabee's On-Line không dễ dàng và miễn phí. Chúng tôi phải đặt trước và trả tiền cho mỗi lần, việc này không thực tế nếu chúng tôi muốn liên tục chạy test. Chúng tôi sẽ cần đến một dịch vụ đấu giá rởm mà chúng tôi có thể điều khiển từ các test để chúng có hành vi như thật – hoặc ít nhất cũng giống như những gì chúng tôi nghĩ về những thứ thật, cho đến khi chúng tôi có cơ hội test ở môi trường thật. Cuộc đấu giá rởm này sẽ được làm đơn giản hết mức có thể. Nó sẽ chỉ kết nối với một XMPP message broker, nhận lệnh từ Sniper đang được test, và cho phép test gửi event trở lại. Chúng tôi không cài lại nguyên Southabee's On-Line, mà chỉ cài vừa đủ hỗ trợ các kịch bản test.



Hình 10.1: Một Sniper tham gia rồi sau đó thua

Việc điều khiển ứng dụng Sniper phức tạp hơn. Chúng tôi muốn test vận hành được ứng dụng ở mức càng gần với end-to-end càng tốt, để chứng tỏ rằng phương thức main() khởi tạo ứng dụng đúng cách và rằng các component thực sự làm việc với nhau. Điều đó có nghĩa rằng chúng tôi cần bắt đầu bằng cách làm việc từ các khía cạnh thấy được từ bên ngoài (ở đây là giao diện người dùng) thay vì thực tiếp kích hoạt các đối tượng domain của ứng dụng. Chúng tôi cũng muốn nội dung test càng rõ ràng càng tốt về cái đang được test, test được viết trên cơ sở mối quan hệ giữa một Sniper và cuộc đấu giá của nó, cho nên chúng tôi sẽ che hết phần code rối rắm dành cho việc thao tác với Swing trong một lớp ApplicationRunner. Chúng tôi sẽ bắt đầu bằng việc viết test như thể tất cả phần code mà nó cần đã có sẵn, rồi về sau sẽ điền cài đặt vào.

```
public class AuctionSniperEndToEndTest {
  private final FakeAuctionServer auction = new FakeAuctionServer("item-54321");
  private final ApplicationRunner application = new ApplicationRunner();

  @Test public void sniperJoinsAuctionUntilAuctionCloses() throws Exception {
    auction.startSellingItem();  // Step 1
    application.startBiddingIn(auction);  // Step 2
    auction.hasReceivedJoinRequestFromSniper(); // Step 3
    auction.announceClosed();  // Step 4
    application.showsSniperHasLostAuction();  // Step 5
  }
  // Additional cleanup
  @After public void stopAuction() {
    auction.stop();
  }
  @After public void stopApplication() {
    application.stop();
  }
}
```

Chúng tôi áp dụng quy ước đặt tên như sau: Nếu một phương thức kích hoạt một event để dẫn lái test, tên của nó sẽ là một lệnh, chẳng hạn startBiddingIn(). Nếu một phương thức khẳng định rằng cái gì đó đáng ra phải xảy ra, tên của nó sẽ có dạng mô tả; ví dụ, showsSniperHasLostAuction(), nó sẽ ném một exception nếu ứng dụng không hiển thị trạng thái đấu giá là đã thua. JUnit sẽ gọi hai phương thức stop() say khi test chạy xong để dọn dẹp môi trường run-time.

Khi viết test, một trong các giả thuyết của chúng tôi là mỗi FakeAuctionServer được buộc vào một item cho trước. Điều này khớp với cấu trúc chúng tôi đã định : Southabee's On-Line tổ chức nhiều cuộc đấu giá, mỗi cuộc chỉ bán đúng một item.

Mỗi lúc một domain - One Domain at a Time

Ngôn ngữ của test này quan tâm đến các cuộc đấu giá và Sniper, không đả động gì về các tầng message hay các hay component trong UI. Việc giữ cho ngôn ngữ được nhất quán giúp ta hiểu được cái gì là quan trọng đối với test này. Hiệu ứng phụ của nó còn là ta sẽ không phải sửa test khi phần cài đặt liên quan đến message hay UI thay đổi, mà nhất định là nó sẽ thay đổi.

Vài lựa chọn ban đầu

Giờ ta phải làm cho test pass, việc sẽ đòi hỏi nhiều công chuẩn bị. Ta cần tìm hoặc viết bốn component: một XMPP message broker, một aution rởm mà có thể liên lạc bằng XMPP, một GUI testing framework, và một test harness có thể đáp ứng được kiến trúc đa luồng, không đồng bộ của chúng ta. Ta cũng phải đưa project vào hệ thống version control với một quy trình tự động build/deploy/test. So sánh với việc unit test một lớp,

có rất nhiều việc phải làm, nhưng toàn là việc quan trọng. Ngay cả ở mức cao như hiện nay, việc viết test vẫn dẫn hướng cho phát triển (TDD) hệ thống. Khi làm việc với test end-to-end đầu tiên, ta sẽ phải đưa ra một số quyết định về cấu trúc, chẳng hạn như sắp xếp package ra sao và deploy như thế nào.

Đầu tiên là package, ta sẽ cần một XMPP message broker để cho phép ứng dụng nói chuyện với nhà đấu giá rởm của ta. Sau khi tìm hiểu, chúng tôi quyết định dùng Openfire và thư viện Smack. Chúng tôi cũng cần một high-level test framework mà có thể làm việc với cả Swing và Smack, cả hai đều đa luồng và event-driven. Chúng tôi chọn WindowLicker, nó hỗ trợ cách tiếp cận không đồng bộ mà chúng tôi cần cho các test. Lắp ghép lại, chúng tôi có cơ sở hạ tầng như trong hình 10.2:

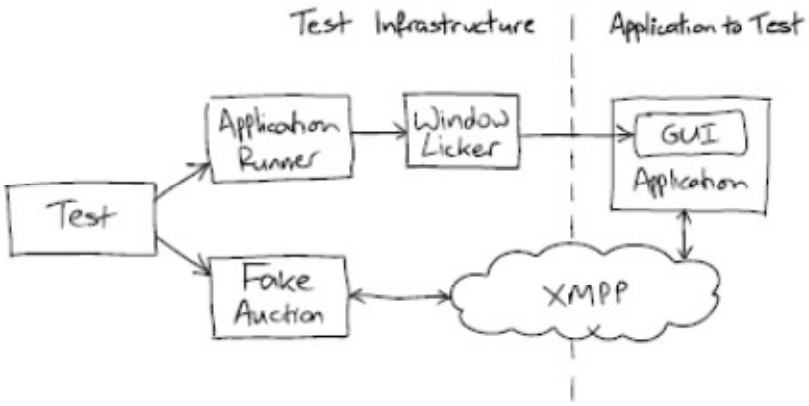


Figure 10.2 *The end to end test rig*

End-to-End Testing

End-to-end testing cho các hệ thống event-based, chẳng hạn như Sniper, phải chấp nhận sự không đồng bộ. Các test chạy song song với ứng dụng và không biết chính xác khi nào ứng dụng sẵn sàng hoặc chưa sẵn sàng. Điều này không giống với unit test, khi mà một test điều khiển trực tiếp một đối tượng trong cùng một thead và do đó có thể kiểm tra trực tiếp trạng thái và hành vi của đối tượng.

Một test end-to-end không thể ngó vào bên trong ứng dụng đích, nó phải đợi để phát hiện một số hiệu ứng nhìn thấy được từ ngoài, chẳng hạn như UI thay đổi hoặc một thông tin mới xuất hiện trong log.

Kĩ thuật thông dụng là kiểm tra đi kiểm tra lại xem hiệu ứng đã xảy ra chưa và coi như thất bại nếu hiệu ứng không xảy ra trong một giới hạn thời gian cho trước. Còn có một sự phức tạp nữa là khi ứng dụng đích phải ổn định đủ lâu sau event thì test mới bắt được kết quả. Một test không đồng bộ đợi kiểm tra một giá trị mà chỉ thoáng hiện trên màn hình rồi biến mất sẽ là một test quá không đáng tin cậy để có thể dùng cho quy trình build tự động, do đó một kĩ thuật thông dụng là kiểm soát ứng dụng và đi theo kịch bản. Tại mỗi bước, test đợi một assertion được thỏa mãn rồi mới gửi event để đánh thức ứng dụng cho bước thứ hai. Chương 14 thảo luận đầy đủ về việc test hành vi không đồng bộ.

Tất cả những điều này làm cho end-to-end testing chậm hơn và chập chờn hơn (có khi nó fail chỉ vì mạng hôm nay chậm), do đó có thể cần phải phân tích lí do khi test fail. Có những đội phát triển quy ước rằng các test về thời gian phải fail vài lần liên tục thì mới được thông báo là fail. Điều này khác với việc unit test thì lúc nào cũng phải pass. Trong trường hợp của chúng tôi, cả Swing và cơ sở hạ tầng trao đổi message đều

không đồng bộ, do đó việc dùng WindowLicker (framework đợi kiểm tra giá trị) để lái Sniper đã đáp ứng được tính chất không đồng bộ cố hữu của các test end-to-end của chúng tôi:

Sẵn sàng

Bạn đọc có thể đã nhận ra rằng chúng tôi đã bỏ qua một điểm: test đầu tiên của chúng tôi không hẳn là end-to-end. Nó không dùng đến dịch vụ đấu giá thực, vì không dễ mà có điều kiện dùng. Một phần quan trọng của kĩ năng TDD là đánh giá nên đặt giới hạn ở đâu cho cái cần test và làm cách nào để cuối cùng sẽ test hết. Trong trường hợp này, chúng tôi phải bắt đầu với một dịch vụ đấu giá rởm dựa theo tài liệu của Southabee's On-Line. Tài liệu đó có thể đúng có thể sai, nên chúng tôi sẽ ghi lại việc này như là một rủi ro đã nhận ra trong kế hoạch dự án, và lập kế hoạch cho việc test lại với server thực ngay khi chúng tôi có đủ chức năng để thực hiện một giao dịch thực sự - kể cả nếu chúng tôi thành ra mua một cái chân nến xấu xí (nhưng rẻ tiền) tại một cuộc đấu giá thực. Chúng tôi phát hiện được sự không nhất quán nào càng sớm thì càng có ít code phải sửa do đã dựa trên sự hiểu nhầm đó, và càng có nhiều thời gian dành cho việc sửa đám code đó.

Chúng tôi nên bắt tay vào việc.