**Thợ lành nghề #6: Một lần không đủ (Dịch vụ Socket 1)**

Ngày 16 tháng 9 năm 2002.

Sự kiện ngày hôm qua làm tôi mệt lả cả người. Jerry và tôi giải quyết xong vấn đề tạo dãy thừa số nguyên tố bằng cách tuồn qua mỗi lần một kiểm thử tí hon. Ðây là một cách giải quyết vấn đề kỳ lạ nhất mà tôi từng thấy nhưng nó lại làm việc ngon lành hơn giải pháp nguyên thủy của tôi.

Tôi đi lại bất định trong hành lang, ngẫm nghĩ đến chuyện này mãi. Tôi chẳng còn nhớ đến bữa tối nay ở đâu nữa. Tôi ngủ thiếp đi sớm hơn ngày thường và chiêm bao về loạt kiểm thử bé tí kia.

Sáng nay khi tôi trình diện Jerry, gã nói:

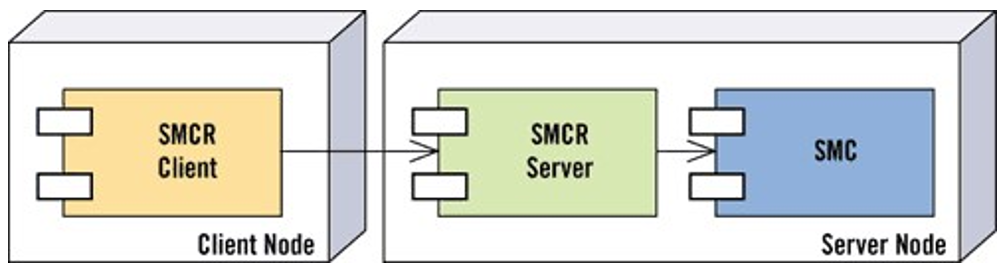
“Chào Alphonse. Mày đã sẵn sàng cho một chương trình *thật* chưa?”

“Ông thừa biết như thế! Thích quá, vâng, tôi sẵn sàng! Tôi quá mệt mấy cái trò thử nghiệm này lắm rồi.”

“Tốt lắm! Tụi mình có một chương trình gọi là SMC dùng để biên dịch ngữ pháp của máy trạng thái hữu hạn (finite state machine grammar) vào môi trường Java. Ông C muốn tụi mình biến chương trình ấy thành một dịch vụ trên mạng.”

“Ý ông là sao?”, tôi hỏi.

Jerry xoay qua bản phác thảo rồi bắt đầu vừa giảng giải vừa minh hoạ.

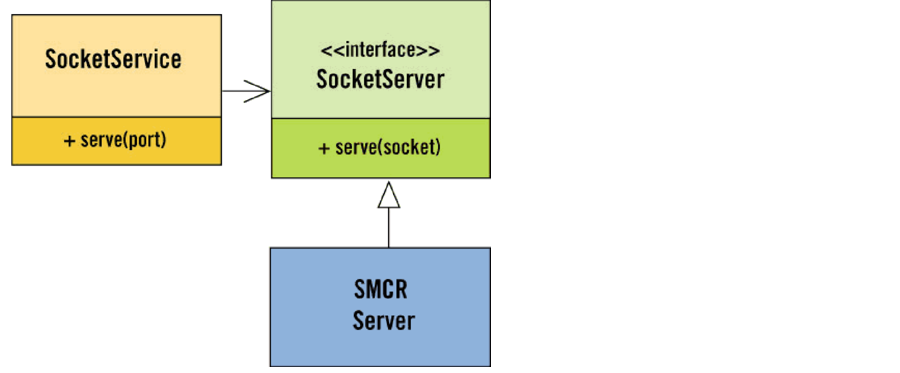


“Mình sẽ viết hai chương trình. Một cái gọi là SMCR Client và cái kia gọi là SMCR Server. Người dùng nếu muốn biên dịch ngữ pháp của máy trạng thái hữu hạn sẽ gọi SMCR Client cùng với tên của tập tin muốn biên dịch. SMCR Client sẽ gửi tập tin đó đến một máy đặc biệt nơi SMCR Server đang hoạt động. SMCR Server sẽ gọi trình biên dịch SMC và gửi kết quả biên dịch về cho SMCR Client. SMCR Client sẽ lưu các dữ kiện này vào thư mục của người dùng. Ðối với người dùng, cơ chế này không khác gì họ đang dùng SMC trực tiếp.”

“OK, tôi nghĩ là tôi đã hiểu vấn đề.” Tôi nói. “Nghe khá đơn giản.”

“Nó khá đơn giản thật.” Jerry đáp. “Nhưng đụng đến socket lúc nào cũng thú vị hơn một chút.”

Chúng tôi ngồi xuống máy tính và, như thường lệ, sẵn sàng để viết kiểm thử đầu tiên. Jerry suy nghĩ một lúc rồi trở lại bản phác thảo và phác hoạ ra một biểu đồ như sau:



“Ðây là ý nghĩ của tao về SMCR Server.” Gã nói. “Chúng ta sẽ đặt mã quản lý socket vào lớp **SocketService**. Lớp này sẽ đón và quản trị các truy cập từ bên ngoài vào. Khi **serve(port)** được gọi, nó sẽ tạo một dịch vụ socket với port đã ấn định và bắt đầu tiếp nhận truy cập. Bất cứ khi nào có một truy cập xảy ra nó sẽ tạo một thread mới và chuyển giao nhiệm vụ điều tác sang method **serve(socket)** thuộc **SocketServer**. Với cách đó, mình tách mã quản trị socket ra khỏi phần mã mình muốn dùng để thao tác các dịch vụ khác.”

Không biết được liệu cách làm này có hiệu quả không, tôi chỉ gật đầu. Rõ ràng gã có lý do để nghĩ như thế. Tôi chỉ theo đuôi mà thôi.

Kế tiếp gã viết cái test như sau:

public void testOneConnection() throws Exception {

SocketService ss = new SocketService();

ss.serve(999);

connect(999);

ss.close();

assertEquals(1, ss.connections());

}

“Việc tao làm ở đây có tên là ***Intentional Programming*** (lập trình có chủ định). Jerry nói. “Tao đang gọi tới đoạn mã trong lúc nó chưa tồn tại. Cách làm này để diễn đạt chủ ý của mình về mã nguồn sẽ ra sao, làm việc như thế nào.”

“OK.” Tôi đáp. “Ông tạo cái **SocketService** rồi ông chỉ định nó tiếp nhận các truy cập trên port 999. Kế tiếp có vẻ như ông truy cập vào dịch vụ mới vừa được tạo ra trên port 999. Cuối cùng ông đóng **SocketService** và xác nhận rằng nó có một truy cập.”

“Ðúng như thế.” Jerry xác nhận.

“Nhưng làm sao ông biết **SocketService** sẽ cần các phương thức **connections**?”

“Ô, có lẽ không cần nó. Tao chỉ đặt nó ở đó để có thể kiểm thử nó.”

“Như vậy không phí sao?” Tôi dặm hỏi.

Jerry nghiêm khắc nhìn tôi và trả lời: “Không có gì làm cho việc viết một kiểm thử được dễ dàng lại là phung phí cả Alphonse. Tụi tao thường thêm các phương thức và các lớp, đơn giản để làm cho các lớp dễ kiểm thử hơn.”

Mặc dù, không khoái cái phương thức **connnections()** nhưng tôi cứ làm thinh.

Chúng tôi chỉ viết vừa đủ phương thức khởi tạo **SocketService** và các phương thức **serve**, **close** và **connect** để có thể biên dịch. Các phương thức này đều trống nên khi chúng tôi chạy thử, cái test bị hỏng như dự đoán.

Kế tiếp Jerry viết phương thức **connect** như một phần của lớp kiểm thử.

private void connect(int port) {

try {

Socket s = new Socket(&quot;localhost&quot;, port);

s.close();

} catch (IOException e) {

fail(&quot;could not connect&quot;);

}

}

Khi chạy kiểm thử này ta có báo lỗi như sau:

“**testOneConnection: could not connect”**

Tôi nói: “Nó hỏng vì không thể tìm ra port 999 ở đâu hết, đúng không?”

“Ðúng vậy!” Jerry trả lời. “Nhưng chuyện đó dễ thôi. Ðây, sao mày không sửa nó đi?”

Từ trước giờ tôi chưa bao giờ viết mã cho socket nên không biết phải làm gì tiếp. Jerry chỉ tôi đến phần **ServerSocket** trong Javadocs. Các ví dụ ở đây xem ra rất đơn giản nên tôi viết thêm các phương thức của**SocketService** như sau:

import java.net.ServerSocket;

public class SocketService {

private ServerSocket serverSocket = null;

public void serve(int port) throws Exception {

serverSocket = new ServerSocket(port);

}

public void close() throws Exception {

serverSocket.close();

}

public int connections() {

return 0;

}

}

Chạy phần mã này báo: **“testOneConnection: expected: <1> but was: <0>”**

“À ha!” Tôi nói: “Nó tìm ra port 999. Quá đã! nhưng mình cần đếm số lần truy cập!”

Nên tôi đổi **SocketService** class như sau:

import java.io.IOException;

import java.net.ServerSocket;

import java.net.Socket;

public class SocketService {

private ServerSocket serverSocket = null;

private int connections = 0;

public void serve(int port) throws Exception {

serverSocket = new ServerSocket(port);

try {

Socket s = serverSocket.accept();

s.close();

connections++;

} catch (IOException e) {

}

}

public void close() throws Exception {

serverSocket.close();

}

public int connections() {

return connections;

}

}

Nhưng đoạn mã này không chạy, nó cũng chẳng báo lỗi. Khi tôi chạy phần test, nó bị treo.

“Chuyện gì vậy cà?” Tôi thắc mắc.

Jerry mỉm cười. “Thử xem mày có thể mò ra không Alphonse. Dò thử đi.”

“OK, xem thử. Cái kiểm thử gọi **serve** để tạo ra socket và tiếp tục gọi **accept**. Ồ! **accept** không trả về cho đến khi nó có được một truy cập, và vì **serve** không hề trả lại nên mình không hề có cơ hội gọi **connect**.”

Jerry gật đầu. “Vậy thì mày định sửa nó thế nào Alphonse?”

Tôi nghĩ ngợi một chút. Tôi cần gọi hàm **connect** sau khi gọi **accept** nhưng khi mình gọi **accept** nó không trả về cho đến khi mình gọi **connect**. Nhìn qua thì có vẻ không thể được.

“Không phải là không được đâu Alphonse.” Jerry cất tiếng. “Mày chỉ cần tạo ra một thread.”

Tôi lại ngẫm nghĩ thêm một chút nữa. Ðúng rồi, tôi có thể đặt phần gọi cho việc tiếp nhận truy cập trong một thread khác rồi mới bắt lấy thread đó và gọi bước truy cập.

Tôi nói: “Tôi biết lý do tại sao ông nói tạo mã nguồn cho socket thú vị hơn một chút rồi đó.” và tôi thay đổi đoạn mã như sau:

private Thread serverThread = null;

public void serve(int port) throws Exception {

serverSocket = new ServerSocket(port);

serverThread = new Thread(

new Runnable() {

public void run() {

try {

Socket s = serverSocket.accept();

s.close();

connections++;

} catch (IOException e) {

}

}

}

);

serverThread.start();

}

“Sử dụng lớp lồng nhau nặc danh hay lắm đó Alphonse.” Jerry nói.

“Cám ơn.” Tôi cảm thấy sung sướng khi được gã khen. “Nhưng e nó tạo một chùm đuôi khỉ ở cuối cái hàm.”

“Mình sẽ tái cấu trúc nó sau, đầu tiên cứ chạy cái kiểm thử đi đã.”

Cái kiểm thử chạy ổn nhưng Jerry có vẻ đăm chiêu, như thể gã vừa bị ai nói dối.

“Chạy cái kiểm thử lần nữa xem Alphonse.”

Tôi vui vẻ nhấn nút run và cái kiểm thử lại chạy ngon lành.

“Lần nữa.” Gã nói.

Tôi nhìn gã một giây xem thử gã có đùa không. Rõ ràng gã không đùa. Mắt gã dán chặt trên màn hình như thể gã đang săn lùng “drebin”. Thế nên tôi nhấn nút một lần nữa và thấy:

**“testOneConnection: expected:<1> but was:<0>”**

“Chờ một chút!” tôi rú lên. “Không thể nào!”

“Ồ, có thể chớ.” Jerry nói. “Tao đã đợi nó xảy ra.”

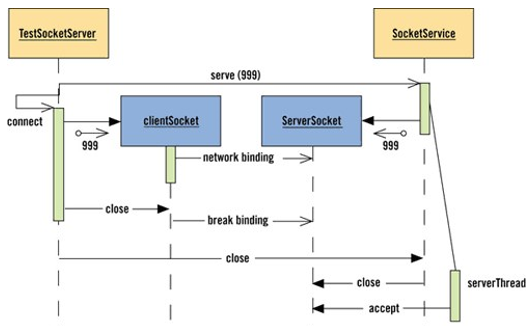
Tôi nhấn nút liên tục. Trong mười lần có đến ba lần hỏng. Không biết tôi có loạn trí không?

Làm sao chương trình lại giở trò như vậy?

“Làm sao ông biết được vậy Jerry? Ông có liên hệ gì đến sấm truyền Aldebran hả?”

“Không, tao có viết loại mã này trước đây nên biết đôi điều cần dự phỏng. Mày có thể giải thích rõ chuyện gì xảy ra không? Suy nghĩ cho thấu đáo và kỹ càng đó.”

Ðúng là đau đầu nhưng tôi bắt đầu ráp từng phần lại với nhau. Tôi đến bản phác thảo và vẽ ra:



Khi đã phác thảo xong, tôi giải thích kịch bản cho Jerry. “**TestSocketServer** gửi thông điệp **serve(999)** đến **SocketService**. **SocketService** tạo **ServerSocket** và **serverThread** rồi trả về. Sau đó **TestSocketServer** gọi **connect,** đoạn mã đã tạo nên client socket. Hai socket này hẳn đã tìm thấy nhau bởi vì chúng ta không nhận được lỗi ‘could not connect’. **ServerSocket** hẳn đã tiếp nhận truy cập nhưng có lẽ **serverThread** chưa có cơ hội để chạy. Và trong khi **serverThread** bị khóa, hàm **connect** đóng client socket lại. Kế tiếp **TestSocketServer** gởi thông điệp đóng đến **SocketService** và đoạn mã này đóng **serverSocket**. Khi **serverThread** có cơ hội gọi hàm **accept** thì server socket đã đóng mất.”

“Tao nghĩ mày đúng đó.” Jerry nói. “Hai biến cố – tiếp nhận và đóng – thiếu đồng bộ và hệ thống này dễ hỏng với các trình tự xảy ra. Cái này mình gọi là trường hợp dồn đuổi (race condition). Chúng ta phải bảo đảm thắng cuộc đuổi chạy này.”

Chúng tôi quyết định thử nghiệm giả thuyết của tôi bằng cách đưa vào các lệnh in ra màn hình trong khối ‘catch’ sau khi **accept** được gọi. Hẳn vậy, trong mười lần kiểm thử, chúng tôi thấy thông điệp này ba lần.

Jerry hỏi tôi: “Thế thì làm sao mình cho kiểm thử đơn vị chạy đây?”

“Theo tôi nghĩ, dường như cái kiểm thử không thể chỉ mở client socket rồi đóng lại ngay lập tức.”

Tôi đáp “Nó cần phải đợi bước tiếp nhận.”

Gã nói: “Mình có thể đợi 100ms trước khi đóng client socket.”

“Ừa, chắc là được nhưng code hơi xấu.” Tôi trả lời.

“Hãy xem thử mình làm cho nó chạy được hay không cái đã rồi sẽ tính chuyện tái cấu trúc sau.”

Nên tôi thay đổi phương thức **connect** như sau:

private void connect(int port) {

try {

Socket s = new Socket(&quot;localhost&quot;, port);

try {

Thread.sleep(100);

} catch (InterruptedException e) {

}

s.close();

} catch (IOException e) {

fail(&quot;could not connect&quot;);

}

}

Phần thay đổi cho kết quả kiểm thử 10 trên 10.

“Gớm thật.” Tôi nói. “Khi mình đối phó với nhiều thread thì phải dè chừng trường hợp dồn đuổi (race condition). Nhấn nút kiểm thử nhiều lần là một thói quen tốt nên tập.”

“Hên là mình khám phá ra nó trong mấy cái kiểm thử.” Tôi nói. “Không thì khó mà kiếm ra nó sau khi hệ thống đã chạy.”

Jerry chỉ gật đầu.

**Bài tiếp:** [Thợ lành nghề #7: Một lần không đủ (Dịch vụ Socket 2)](https://codegym.vn/blog/2018/08/05/tho-lanh-nghe-7-mot-lan-khong-du-dich-vu-socket-2/)

**Bài trước:** [Thợ lành nghề #5: Bước nhỏ](https://codegym.vn/blog/tho-lanh-nghe-5-buoc-nho/)

**Tác giả:** Robert C. Martin

**Người dịch:** Hoàng Ngọc Diêu (conmale)