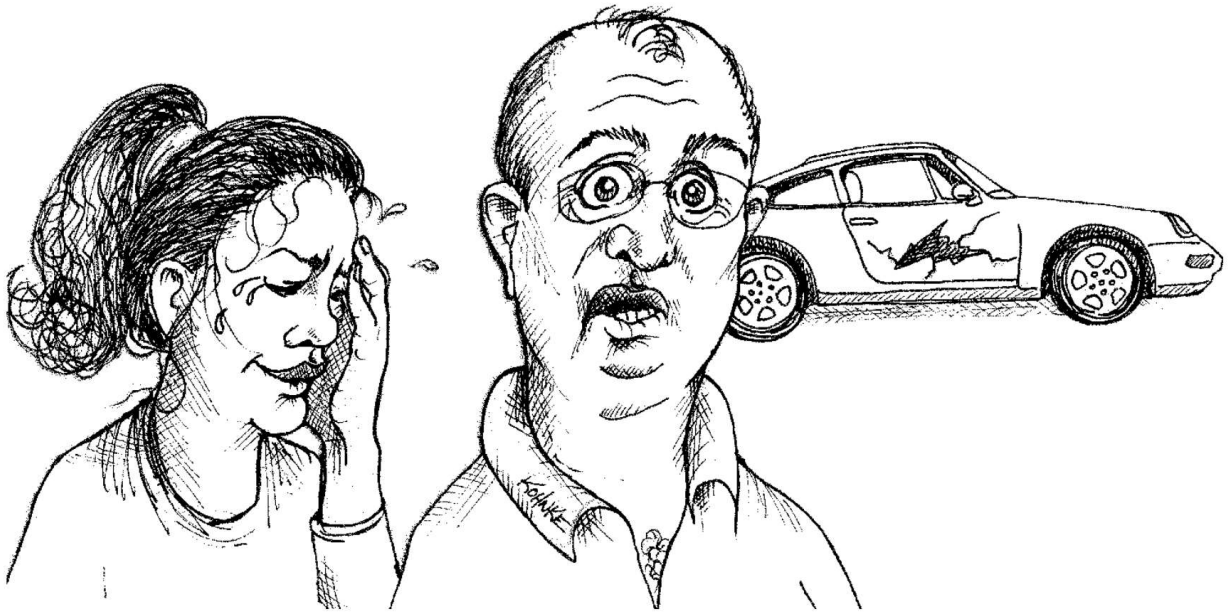
Chaper 07: Xử lý lỗi

Michael Feathers



Có vẻ kỳ lạ khi lại có một phần về xử lý lỗi trong một cuốn sách về mã sạch. Xử lý lỗi chỉ là một trong những việc mà tất cả chúng ta phải làm khi lập trình. Đầu vào có thể bất thường và thiết bị có thể bị lỗi. Nói tóm lại, mọi thứ đều có thể xảy ra sai sót, và khi chúng xảy ra, chúng ta với tư cách là người lập trình có trách nhiệm đảm bảo rằng mã của chúng ta thực hiện những gì nó cần làm.

Tuy nhiên, triển khai với mã sạch phải rõ ràng. Nhiều khi mã bị chi phối hoàn toàn bởi việc xử lý lỗi. Khi tôi nói bị chi phối, tôi không có ý nói rằng xử lý lỗi chỉ là tất cả những gì họ làm. Ý tôi là gần như không thể thấy mã làm gì vì tất cả các lỗi xử lý được rải rác. Xử lý lỗi là quan trọng, *nhưng nếu nó che khuất logic thì đó là sai.*

Trong chương này, tôi sẽ trình bày một số kỹ thuật và cân nhắc mà bạn có thể sử dụng để viết mã vừa rõ ràng vừa mạnh mẽ — mã xử lý lỗi một cách duyên dáng và đúng phong cách.

Sử dụng ngoại lệ thay vì để cho mã xử lý

Trong quá khứ xa xôi, có rất nhiều ngôn ngữ không có ngoại lệ. Trong các ngôn ngữ đó, các kỹ thuật xử lý và báo cáo lỗi bị hạn chế. Bạn đặt cờ lỗi hoặc trả lại mã lỗi mà người gọi có thể kiểm tra. Mã trong Listing 7-1 minh họa những cách tiếp cận này.

Listing 7-1
DeviceController.java

```
public class DeviceController {
    ...
    public void sendShutDown() {
        DeviceHandle handle = getHandle(DEV1);
        // Kiểm tra trạng thái của thiết bị
        if (handle != DeviceHandle.INVALID) {
            // Lưu trạng thái thiết bị vào record field
            retrieveDeviceRecord(handle);
            // Nếu không bị treo, hãy tắt
            if (record.getStatus() != DEVICE_SUSPENDED) {
                pauseDevice(handle);
                clearDeviceWorkQueue(handle);
                closeDevice(handle);
            } else {
                logger.log("Thiết bị bị treo. Không thể tắt");
            }
        } else {
            logger.log("Xử lý không hợp lệ cho: " + DEV1.toString());
        }
    }
```

```
    ...
}
```

Vấn đề với những cách tiếp cận này là chúng gây lộn xộn cho người gọi. Người gọi phải kiểm tra lỗi ngay sau cuộc gọi. Thật không may, người gọi rất dễ quên. Đối với trường hợp này, tốt hơn là bạn nên ném một ngoại lệ khi bạn gặp lỗi. Mã sạch hơn. Logic của nó không bị che khuất bởi việc xử lý lỗi.
Listing 7-2 hiển thị mã sau khi chúng ta đã chọn đưa ra các ngoại lệ trong các phương pháp có thể phát sinh lỗi.

Listing 7-2
DeviceController.java (with exceptions)

```
public class DeviceController {
    ...
    public void sendShutDown() {
        try {
            tryToShutDown();
        } catch (DeviceShutDownError e) {
            logger.log(e); }
        }

    private void tryToShutDown() throws DeviceShutDownError {
        DeviceHandle handle = getHandle(DEV1);
        DeviceRecord record = retrieveDeviceRecord(handle);
        pauseDevice(handle);
        clearDeviceWorkQueue(handle);
        closeDevice(handle);
    }

    private DeviceHandle getHandle(DeviceID id) {
        ...
        throw new DeviceShutDownError("Xử lý không hợp lệ cho: " + id.toString());
        ...
    }
    ...
}
```

Để ý xem nó sạch hơn bao nhiêu. Đây không chỉ là vấn đề thẩm mỹ. Mã này tốt hơn vì hai mối quan tâm không bị chồng chéo, thuật toán **ShutDown** và **xử lý lỗi** giờ đã được tách biệt. Bạn có thể xem xét từng vấn đề đó và hiểu chúng một cách độc lập.

Trước tiên viết khối Try-Catch-Finally của bạn

Một trong những điều thú vị nhất về các ngoại lệ là chúng xác định phạm vi trong chương trình của bạn. Khi bạn thực thi mã trong phần **try** của câu lệnh **try-catch-finally**, bạn đang nói rằng việc thực thi có thể xẩy ra lỗi tại bất kỳ thời điểm nào và sau đó mã trong **catch** sẽ được thực thi.

Theo một cách nào đó, các khối **try** giống như các giao dịch. **catch** đảm bảo để chương trình của bạn ở trạng thái không bị gián đoạn, bởi bất kể điều gì xảy ra trong **try**. Vì lý do này, bạn nên bắt đầu bằng câu lệnh **try-catch-final** khi bạn viết mã, nó có thể đưa ra các ngoại lệ. Điều này giúp bạn xác định người dùng mã đó sẽ mong đợi điều gì khi xẩy ra lỗi với mã được thực thi trong **try**.

Hãy xem một ví dụ. Chúng ta cần viết một số mã truy cập tệp và đọc một số đối tượng được **serialized**. Chúng ta bắt đầu với **unit test** cho thấy rằng chúng ta sẽ nhận được một ngoại lệ khi tệp không tồn tại:

```
@Test(expected = StorageException.class)
public void retrieveSectionShouldThrowOnInvalidFileName() {
    sectionStore.retrieveSection("Tập tin không hợp lệ");
}
```

Việc kiểm tra thúc đẩy chúng ta tạo ra hàm này:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
    // Fake return cho đến khi chúng ta có một triển khai thực sự
    return new ArrayList<RecordedGrip>();
}
```

Thử nghiệm của chúng ta không thành công vì nó không có ngoại lệ. Tiếp theo, chúng ta thay đổi triển khai để nó cố gắng truy cập một tệp không hợp lệ. Thao tác này đưa ra một ngoại lệ:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
    try {
        FileInputStream stream = new FileInputStream(sectionName);
    } catch (Exception e) {
        throw new StorageException("Lỗi truy xuất", e);
    }
    return new ArrayList<RecordedGrip>();
}
```

Chúng ta hiện đã vượt qua bài kiểm tra vì đã phát hiện ra ngoại lệ. Tại thời điểm này, chúng ta có thể tham khảo lại. Chúng ta có thể thu hẹp loại ngoại lệ mà chúng ta **catch** được để phù hợp với loại thực sự được đưa ra từ phương thức khởi tạo của **FileInputStream**: **FileNotFoundException**:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
    try {
        FileInputStream stream = new FileInputStream(sectionName);
        stream.close();
    } catch (FileNotFoundException e) {
        throw new StorageException("Lỗi truy xuất”, e);
    }
    return new ArrayList<RecordedGrip>();
}
```

Bây giờ chúng ta đã xác định phạm vi với cấu trúc **try-catch**, chúng ta có thể sử dụng TDD để xây dựng phần còn lại của logic mà chúng ta cần. Logic đó sẽ được thêm vào giữa quá trình tạo **FileInputStream** và **close**, và có thể giả vờ rằng không có gì sai.

Cố gắng viết các bài kiểm tra buộc xảy ra các trường hợp ngoại lệ, sau đó thêm hành vi vào trình xử lý của bạn để điều chỉnh các bài kiểm tra của bạn. Điều này sẽ khiến bạn phải xây dựng phạm vi giao dịch của khối **try** trước và sẽ giúp bạn duy trì bản chất giao dịch của phạm vi đó.

Sử dụng Unchecked Exceptions

Cuộc tranh luận đã kết thúc. Trong nhiều năm, các lập trình viên Java đã tranh luận về lợi ích và mối quan hệ của các **checked exceptions**. Khi các **checked exceptions** được giới thiệu trong phiên bản Java đầu tiên, chúng dường như là một ý tưởng tuyệt vời. Hàm của mọi phương thức sẽ liệt kê tất cả các ngoại lệ mà nó có thể chuyển cho trình gọi của nó. Hơn nữa, những ngoại lệ này là một phần của loại phương pháp. Mã của bạn thực sự sẽ không biên dịch nếu hàm không khớp với những gì mã của bạn đã viết.

Vào thời điểm đó, chúng ta nghĩ rằng các **checked exceptions** là một ý tưởng tuyệt vời; và tất nhiên, chúng có thể mang lại *một số* lợi ích. Tuy nhiên, rõ ràng là giờ đây chúng không còn cần thiết để phát triển phần mềm. C# không có các **checked exceptions** và bất chấp mọi nỗ lực, C++ cũng không. Python hay Ruby cũng vậy. Tuy nhiên, có thể viết phần mềm bằng tất cả các ngôn ngữ này. Do đó, chúng ta phải **quyết định - thực sự - liệu** các **checked exceptions** có xứng đáng với cái giá của chúng hay không.

Định nghĩa lớp Exception theo nhu cầu của người gọi

Giá bao nhiêu? Giá của các trường hợp **checked exceptions** là vi phạm nguyên tắc Mở/Đóng. Nếu bạn ném một **checked exceptions** từ một phương thức trong mã của bạn và lệnh **catch** ở ba mức trên, *bạn phải khai báo ngoại lệ đó trong signature của mỗi phương thức giữa bạn và lệnh **catch**. Điều này có nghĩa là một thay đổi ở cấp thấp của phần mềm có thể buộc phải thay đổi signature ở nhiều cấp cao hơn. Các mô-đun đã thay đổi phải được xây dựng lại và triển khai lại, mặc dù không có gì họ quan tâm đến đã thay đổi.

Xem xét hệ thống phân cấp gọi của một hệ thống lớn. Các hàm ở trên cùng gọi các hàm bên dưới chúng, gọi tiếp các hàm khác bên dưới chúng, vv... Bây giờ, giả sử một trong những hàm cấp thấp nhất được sửa đổi theo cách mà nó phải đưa ra một ngoại lệ. Nếu ngoại lệ đó được chọn, thì hàm phải thêm một mệnh đề **throws**. Nhưng điều này có nghĩa là mọi hàm gọi hàm đã sửa đổi cũng phải được sửa đổi để bắt được ngoại lệ mới hoặc để nối mệnh đề **throws** thích hợp vào chữ ký của nó. vv... Kết quả thực là một loạt các thay đổi hoạt động theo cách của chúng từ mức thấp nhất của phần mềm đến mức cao nhất! Tính năng đóng gói bị phá vỡ bởi vì tất cả các hàm trong đường dẫn của một lần ném phải biết về chi tiết của ngoại lệ cấp thấp đó. Vì mục đích của các ngoại lệ là cho phép bạn xử lý lỗi ở khoảng cách xa, thật đáng tiếc khi các **checked exceptions** phá vỡ tính đóng gói theo cách này.

checked exceptions đôi khi có thể hữu ích nếu bạn đang viết thư viện quan trọng: Bạn phải **catch** chúng. Nhưng trong phát triển ứng dụng nói chung, chi phí phụ thuộc lớn hơn lợi ích mà nó đem lại.

Cung cấp ngữ cảnh có ngoại lệ

Mỗi ngoại lệ mà bạn đưa ra phải cung cấp đủ ngữ cảnh để xác định nguồn và vị trí của lỗi. Trong Java, bạn có thể lấy dấu vết ngăn xếp từ bất kỳ ngoại lệ nào; tuy nhiên, dấu vết ngăn xếp không thể cho bạn biết mục đích của hoạt động không thành công.

Tạo thông báo lỗi đầy đủ thông tin và chuyển chúng cùng với các ngoại lệ của bạn. Đề cập đến hoạt động không thành công và loại lỗi. Nếu bạn đang logging ứng dụng của mình, hãy chuyển đủ thông tin để có thể ghi lại lỗi trong lần **catch** của bạn.

Xác định các loại ngoại lệ theo nhu cầu của người gọi

Có nhiều cách phân loại lỗi. Chúng ta có thể phân loại chúng theo nguồn của chúng: Chúng đến từ thành phần này hay thành phần khác? Hoặc loại của chúng: Chúng bị lỗi thiết bị, lỗi mạng, hoặc lỗi lập trình? Tuy nhiên, khi chúng ta xác định các lớp ngoại lệ trong một ứng dụng, mối quan tâm quan trọng nhất của chúng ta là **cách chúng được bắt**.

Hãy xem một ví dụ về phân loại ngoại lệ kém. Đây là câu lệnh **try-catch-finally** cho lệnh gọi thư viện của bên thứ ba. Nó bao gồm tất cả các ngoại lệ mà các cuộc gọi có thể ném ra:

```
ACMEPort port = new ACMEPort(12);
try {
    port.open();
} catch (DeviceResponseException e) {
    reportPortError(e);
    logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
    reportPortError(e);
    logger.log("Unlock exception", e);
} catch (GMXError e) {
    reportPortError(e);
    logger.log("Device response exception");
} finally {
    ...
}
```

Tuyên bố đó chứa đựng rất nhiều sự trùng lặp và chúng ta không nên ngạc nhiên. Trong hầu hết các tình huống xử lý ngoại lệ, công việc mà chúng ta thực hiện tương đối chuẩn, bất kể nguyên nhân thực tế là gì. Chúng ta phải ghi lại một lỗi và đảm bảo rằng chúng ta có thể tiếp tục.

Trong trường hợp này, bởi vì chúng ta biết rằng công việc chúng ta đang làm gần như giống nhau bất kể ngoại lệ xảy ra là gì, chúng ta có thể đơn giản hóa mã của mình đáng kể bằng cách gói API mà chúng ta đang gọi và đảm bảo rằng nó trả về một loại ngoại lệ chung:

```
LocalPort port = new LocalPort(12);
try {
    port.open();
} catch (PortDeviceFailure e) {
    reportError(e);
    logger.log(e.getMessage(), e);
} finally {
    ...
}
```

Lớp **LocalPort** của chúng ta chỉ là một trình bao bọc đơn giản giúp bắt và dịch các ngoại lệ được ném bởi lớp **ACMEPort**:

```
public class LocalPort {
    private ACMEPort innerPort;
    public LocalPort(int portNumber) {
        innerPort = new ACMEPort(portNumber);
    }
    public void open() {
        try {
            innerPort.open();
        } catch (DeviceResponseException e) {
            throw new PortDeviceFailure(e);
        } catch (ATM1212UnlockedException e) {
            throw new PortDeviceFailure(e);
        } catch (GMXError e) {
            throw new PortDeviceFailure(e);
        }
    }
    ...
}
```

Các trình gói như chúng ta đã xác định cho **ACMEPort** có thể rất hữu ích. Trên thực tế, gói các API của bên thứ ba là một phương pháp hay nhất. Khi bạn bọc một API của bên thứ ba, bạn giảm thiểu sự phụ thuộc của mình vào nó: Bạn có thể chọn chuyển sang một thư viện khác trong tương lai mà không cần thay đổi nhiều. Gói cũng giúp bạn dễ dàng bắt chước các cuộc gọi của bên thứ ba hơn khi bạn đang kiểm tra mã của riêng mình.

Một lợi thế cuối cùng của gói là bạn không bị ràng buộc với các lựa chọn thiết kế API của một nhà cung cấp cụ thể. Bạn có thể xác định một API mà bạn cảm thấy thoải mái. Trong ví dụ trước, chúng ta đã xác định một loại ngoại lệ duy nhất cho lỗi thiết bị **port** và nhận thấy rằng chúng ta có thể viết mã sạch hơn nhiều.

Thường thì một lớp ngoại lệ duy nhất là tốt cho một vùng mã cụ thể. Thông tin được gửi với ngoại lệ có thể phân biệt các lỗi. Chỉ sử dụng các lớp khác nhau nếu đôi khi bạn muốn bắt một ngoại lệ và cho phép ngoại lệ khác đi qua.

Xác định dòng chảy bình thường

Nếu bạn làm theo lời khuyên trong phần trước, bạn sẽ có một sự tách biệt tốt giữa logic nghiệp vụ và việc xử lý lỗi của bạn. Phần lớn mã của bạn sẽ bắt đầu trông giống như một thuật toán không trang trí. Tuy nhiên, quá trình thực hiện việc này đẩy khả năng phát hiện lỗi ra rìa chương trình của bạn. Bạn bọc các API bên ngoài để bạn có thể đưa ra các ngoại lệ của riêng mình và bạn xác định một trình xử lý phía trên mã của mình để bạn có thể đối phó với bất kỳ tính toán nào bị hủy bỏ. Hầu hết thời gian đây là một cách tiếp cận tuyệt vời, nhưng có một số lúc bạn có thể không muốn nó xẩy ra.

Hãy xem một ví dụ. Dưới đây là một mã khó hiểu tính tổng các chi phí trong ứng dụng thanh toán:

```
try {
    MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
    m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
    m_total += getMealPerDiem();
}
```

Trong nghiệp vụ này, nếu các bữa ăn tiêu tốn nhiều chi phí, chúng sẽ trở thành một phần của tổng số. Nếu không, nhân viên sẽ nhận được một khoản tiền **công tác phí** cho ngày hôm đó. Ngoại lệ làm lộn xộn logic. Sẽ tốt hơn nếu chúng ta không phải giải quyết trường hợp đặc biệt? Mã của chúng ta sẽ trông đơn giản hơn nhiều. Nó sẽ trông như thế này:

```
    MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
    m_total += expenses.getTotal();
```

Chúng ta có thể làm cho mã đơn giản như vậy không? Nó chỉ ra rằng chúng ta có thể. Chúng ta có thể thay đổi **ExpenseReportDAO** để nó luôn trả về một đối tượng **MealExpense**. Nếu không có chi phí bữa ăn, nó trả về đối tượng **MealExpense** trả về **công tác phí** là tổng của nó:

```
public class PerDiemMealExpenses implements MealExpenses {
    public int getTotal() {
        // trả lại công tác phí mặc định
    }
}
```

Đây được gọi là MẪU TRƯỜNG HỢP ĐẶC BIỆT [Fowler]. Bạn tạo một lớp hoặc cấu hình một đối tượng để nó xử lý một trường hợp đặc biệt cho bạn. Khi bạn làm như vậy, mã khách hàng không phải đối phó với các hành vi đặc biệt. Hành vi đó được gói gọn trong đối tượng trường hợp đặc biệt.

Đừng trả về Null

Tôi nghĩ rằng bất kỳ cuộc thảo luận nào về xử lý lỗi cũng nên đề cập đến những việc làm mà dẫn đến lỗi. Đầu tiên trong danh sách: **trả về null**. Tôi không thể đếm được số lượng các ứng dụng mà tôi đã thấy trong đó gần như mọi dòng đều là lệnh kiểm tra **null**. Đây là một mã ví dụ:

```
public void registerItem(Item item) {
    if (item != null) {
        ItemRegistry registry = peristentStore.getItemRegistry();
        if (registry != null) {
            Item existing = registry.getItem(item.getID());
            if (existing.getBillingPeriod().hasRetailOwner()) {
                existing.register(item);
            }
        }
    }
}
```

Nếu bạn làm việc trong một cơ sở mã với mã như thế này, nó có thể không tệ đối với bạn, nhưng nó thật sự là sự thiếu trách nhiệm! Khi chúng ta trả về **null**, về cơ bản chúng ta đang tự tạo ra công việc cho chính bản thân mình đồng thời cũng để cho người gọi hàm phải giải quyết thêm vấn đề. Thử tưởng tượng người gọi hàm quên kiểm tra **null** một lần, ứng dụng ngay lập tức sẽ xẩy ra lỗi và tất nhiên không ai mong muốn điều đó cả.

Bạn có nhận thấy là không có kiểm tra **null** trong dòng thứ hai của câu lệnh **if** lồng nhau đó không? Điều gì sẽ xảy ra trong khi chạy nếu **persistentStore** null? Chúng ta đã có một **NullPointerException** trong khi chạy và ai đó đang bắt **NullPointerException** ở cấp cao nhất, hoặc họ không làm vậy. Dù thế nào thì nó cũng rất tệ. Vậy chính xác thì bạn nên làm gì để đáp lại một **NullPointerException** được ném ra từ sâu bên trong ứng dụng của bạn?

Có thể dễ dàng nói rằng vấn đề với đoạn mã trên là nó thiếu kiểm tra **null**, nhưng trên thực tế, vấn đề là nó xảy ra *quá nhiều*. Nếu bạn muốn trả về **null** từ một phương thức, hãy xem xét việc ném một ngoại lệ hoặc trả về một đối tượng ĐẶC BIỆT để thay thế. Nếu bạn đang gọi một phương thức trả về **null** từ một API của bên thứ ba, hãy xem xét gói phương thức đó bằng một phương thức ném ngoại lệ hoặc trả về một đối tượng đặc biệt.

Trong nhiều trường hợp, các đối tượng trường hợp đặc biệt là một biện pháp khắc phục dễ dàng. Hãy tưởng tượng rằng bạn có mã như thế này:

```
List<Employee> employees = getEmployees();
if (employees != null) {
    for(Employee e : employees) {
        totalPay += e.getPay();
    }
}
```

Bây giờ, **getEmployees** có thể trả về **null**, nhưng có nhất thiết phải vậy không? Nếu chúng ta thay đổi **getEmployees** để nó trả về một danh sách rỗng (danh sách không có phần tử nào) thay vì trả về **null**, chúng ta có thể xóa mã kiểm tra **null**:

```
List<Employee> employees = getEmployees();
for(Employee e : employees) {
    totalPay += e.getPay();
}
```

May mắn thay, Java có **Collections.emptyList()** và nó trả về một danh sách bất biến được xác định trước mà chúng ta có thể sử dụng cho mục đích này:

```
public List<Employee> getEmployees() {
    if( .. không có employees .. )
```

```
        return Collections.emptyList();
    }
```

Nếu bạn viết mã theo cách này, bạn sẽ giảm thiểu cơ hội xuất hiện **NullPointerExceptions** và mã của bạn sẽ sạch hơn.

Đừng truyền Null

Trả về **null** từ các phương thức là không tốt, nhưng truyền **null** vào các phương thức thì còn tệ hơn. Trừ khi bạn đang làm việc với một API yêu cầu bạn truyền vào **null**, còn đâu bạn nên tránh truyền **null** trong mã của mình bất cứ khi nào có thể.

Hãy xem một ví dụ để biết tại sao. Đây là một phương pháp đơn giản để tính toán một số liệu cho hai điểm:

```
public class MetricsCalculator {
    public double xProjection(Point p1, Point p2) {
        return (p2.x – p1.x) * 1.5;
    }
    ...
}
```

Điều gì xảy ra khi ai đó truyền **null** làm đối số cho hàm này?

```
calculator.xProjection(null, new Point(12, 13));
```

Tất nhiên, chúng ta sẽ nhận được một **NullPointerException**.
Vậy, làm thế nào chúng ta có thể sửa chữa nó? Chúng ta có thể tạo một loại ngoại lệ mới và **throw** nó:

```
public class MetricsCalculator {
    public double xProjection(Point p1, Point p2) {
        if(p1 == null || p2 == null) {
            throw InvalidArgumentException("Invalid argument for MetricsCalculator.x
        }
        return (p2.x – p1.x) * 1.5;
    }
}
```

Tốt hơn chưa? Nó có thể tốt hơn một chút so với **NullPointerException**, nhưng hãy nhớ rằng, chúng ta phải xác định một trình xử lý cho **InvalidArgumentException**. Người xử lý phải làm gì? Có bất kỳ hướng đi nào tốt hơn không? Có một sự thay thế khác. Chúng ta có thể sử dụng một tập hợp các **assertions** để xác nhận:

```
public class MetricsCalculator {
    public double xProjection(Point p1, Point p2) {
        assert p1 != null : "p1 should not be null";
        assert p2 != null : "p2 should not be null";
        return (p2.x - p1.x) * 1.5;
    }
}
```

Đó là một tài liệu tốt, nhưng nó không giải quyết được vấn đề. Nếu ai đó truyền vào giá trị **null**, chúng ta sẽ vẫn gặp lỗi trong khi chạy chương trình.
Trong hầu hết các ngôn ngữ lập trình, không có cách nào tốt để đối phó với giá trị **null** do người gọi vô tình truyền qua. Cách tiếp cận hợp lý là cấm truyền vào **null** theo mặc định. Khi bạn làm như vậy, bạn có thể viết mã với nhận định rằng giá trị **null** trong danh sách đối số đầu vào là dấu hiệu của một vấn đề và kết thúc với ít lỗi hơn.

Phần kết luận

Mã sạch có thể đọc được, nhưng nó cũng cần phải mạnh mẽ. Đây không phải là những mục tiêu xung đột. Chúng ta có thể viết mã sạch và mạnh mẽ nếu chúng ta thấy việc xử lý lỗi là một mối quan tâm riêng biệt, một thứ có thể xem là độc lập với logic chính của chúng ta. Ở mức độ mà chúng ta có thể làm được điều đó, chúng ta có thể lập luận về nó một cách độc lập và chúng ta có thể đạt được những bước tiến lớn trong khả năng bảo trì mã của chúng ta.

Thư mục

[Martin]: Agile Software Development: Principles, Patterns, and Practices, Robert C. Martin, Prentice Hall, 2002.