# CLEAN CODE

## A handbook of agile software craftsmanship

*(Code sạch – Cẩm nang của lập trình viên)*

*Đơn vị dịch: Google Translate*

*Copy & paste: NQT*

# CHƯƠNG 4

***

# CƠM-MỪN

*“Đừng biến đống code gớm ghiếc của bạn thành comment – hãy viết lại nó”*

*– Brian W. Kernighan and P. J. Plaugher*

Không gì hữu ích bằng một comment được đặt đúng chỗ. Không gì có thể làm lộn xộn đống code của bạn, ngoại trừ những comment ngu xuẩn và dối trá. Và không gì có thể gây nguy hiểm bằng một comment cộc lốc từ đời nào và lại không đúng sự thật.

Các comment không phải là *Bản danh sách của Schindler* (Schindler's List – https://vi.wikipedia.org/wiki/B%E1%BA%A3n_danh_s%C3%A1ch_c%E1%BB%A7a_Schindler). Chúng không tốt hoàn toàn như bạn nghĩ. Các comment, tốt nhất, nên trở thành sự lựa chọn cuối cùng. Nếu ngôn ngữ lập trình của chúng ta có đầy đủ khả năng diễn đạt, hoặc nếu chúng ta có đủ tài năng sử dụng code để thể hiện hết ý định của mình thì chúng ta đã không cần đến những dòng comment.

Việc dùng đúng các comment là một cách để bù đắp cho sự thất bại của chúng ta trong việc thể hiện ý nghĩa của những dòng code. Hãy lưu ý rằng tôi đã dùng từ thất bại. Chính xác là vậy – comment luôn luôn là sự thất bại. Chúng ta phải có chúng vì chúng ta không thể thể hiện hết ý nghĩa của code, và việc sử dụng chúng không phải là nguyên nhân được tán dương.

Vậy nên, khi bạn thấy mình cần viết comment, hãy suy nghĩ kỹ xem liệu có cách nào đó để biến các dòng code thành chính xác những gì bạn muốn thể hiện hay không. Mỗi khi bạn thể hiện chính xác ý nghĩa của code, hãy tự khích lệ mình. Mỗi khi bạn viết comment, bạn nên tự vả vào mặt cho sự thất bại đó .

Tại sao tôi lại thất vọng về các comment? Vì chúng đầy dối trá. Không phải lúc nào cũng vậy, nhưng vấn đề này lại rất thường xuyên xãy ra: Comment càng cũ và càng không liên quan tới code mà nó mô tả, càng có nhiều khả năng nó sai. Lý do rất đơn giản, các lập trình viên không thể bảo trì chúng.

Code của dự án thì thay đổi và phát triển liên tục. Các khối dữ liệu di chuyển từ nơi này đến nơi kia. Những khối đó tách ra và tái lập để tạo nên chương trình. Thật không may, những comment không phải lúc nào cũng theo kịp chúng. Comment thì thường xuyên bị tách ra khỏi code mà nó mô tả và trở nên đứt đoạn với độ chính xác giảm dần. Ví dụ, hãy xem những gì xãy ra với comment này và dòng code mà nó dự định mô tả:

```
MockRequest request;
private final String HTTP_DATE_REGEXP =
 "[SMTWF][a-z]{2}\\,\\s[0-9]{2}\\s[JFMASOND][a-z]{2}\\s"+
 "[0-9]{4}\\s[0-9]{2}\\:[0-9]{2}\\:[0-9]{2}\\sGMT";
private Response response;
private FitNesseContext context;
private FileResponder responder;
private Locale saveLocale;
// Example: "Tue, 02 Apr 2003 22:18:49 GMT"
```

Các biến khác có thể được thêm vào sau đó đặt xen kẽ vào giữa giá trị `HTTP_DATE_REGEXP`. Giá trị `HTTP_DATE_REGEXP` được viết lại và comment của bạn thì tan nát. ☺

Có quan điểm cho rằng các lập trình viên nên có đủ kỷ luật để sửa cả những dòng comment khi họ đã thay đổi code. Tôi đồng ý, họ nên như thế. Nhưng tôi sẽ chọn cách làm cho code rõ ràng dễ hiểu, đến mức nó không cần các comment ngay từ đầu.

Những dòng comment bậy còn tồi tệ hơn là không comment. Chúng lừa gạt và đánh lạc hướng người đọc. Chúng đưa ra những dự tính không bao giờ được thực hiện. Chúng đặt ra những quy tắc cũ, không cần, hoặc không nên được tuân theo nữa.

Chân lý chỉ có thể tìm thấy tại một nơi duy nhất: Code. Chỉ có code mới nói cho bạn biết thật sự những gì nó làm. Nó là nguồn thông tin chính xác duy nhất. Do đó, mặc dù comment đôi khi là cần thiết, nhưng chúng tôi sẽ cố gắng để giảm thiểu nó.

# Đừng dùng comment để làm màu cho code

Một trong những động lực to lớn để viết comment là do code tồi, code tối nghĩa. Chúng ta viết một hàm và chúng ta biết nó khó hiểu, nó vô tổ chức, chúng ta biết nó là một đống hỗ lốn. Vậy nên chúng ta tự nhủ rằng: "Ồ, tốt hơn nên viết comment ở đây!". Không! Tốt hơn bạn nên viết lại code!

Code sáng nghĩa và rõ ràng với ít comment sẽ tuyệt vời hơn so với code tối nghĩa, phức tạp với nhiều comment. Thay vì dành thời gian để viết comment giải thích mớ code hỗ lốn đó, hãy dành thời gian để dọn dẹp nó.

# Giải thích ý nghĩa ngay trong code

Chắc chắn có những lúc code của bạn rất khó để giải thích nó đang làm gì. Thật không may, điều này làm cho nhiều lập trình viên cho rằng code hiếm khi là một cách tốt để giải thích. Điều đó hoàn toàn sai. Bạn muốn nhìn thấy điều gì? Cái này:

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
(employee.age > 65))
```

Hay cái này?

```
if (employee.isEligibleForFullBenefits())
```

Chỉ mất vài giây suy nghĩ để truyền hết thông điệp của bạn vào code. Trong nhiều trường hợp, nó chỉ đơn giản là tạo ra một hàm có tên giống với comment mà bạn muốn viết

# Good Comments

Một số comment là cần thiết hoặc có ích. Chúng ta sẽ xem xet một vài trường hợp mà tôi cho là xứng đáng để bạn bỏ công ra viết. Tuy nhiên, hãy nhớ rằng comment thật sự tốt là comment không cần phải viết ra.

## Comment pháp lý

Đôi khi các tiêu chuẩn mã hóa doanh nghiệp buộc chúng ta phải viết một số comment nhất định vì lý do pháp lý. Ví dụ, tuyên bố bản quyền và quyền tác giả là những điều cần thiết và hợp lý để đưa vào comment khi bắt đầu mỗi source code file.

```
/* Copyright (C) 2003,2004,2005 by Object Mentor, Inc.
 * All rights reserved.
 * Released under the terms of the GNU General Public License version
 * 2 or later.
 */
```

Đừng đưa cả hợp đồng hay những điều luật vào comment. Nếu có thể, hãy tham khảo một vài giấy phép tiêu chuẩn hay tài liệu bên ngoài khác thay vì đưa tất cả những điều khoản và điều kiện vào.

### Comment cung cấp thông tin

Cung cấp thông tin cơ bản với một vài dòng comment đôi khi rất hữu ích. Ví dụ, hãy xem cách mà comment này giải thích về giá trị trả về của một phương thức trừu tượng:

```
// Returns an instance of the Responder being tested.
protected abstract Responder responderInstance();
```

Một comment như thế này, đôi khi, có thể là hữu ích. Nhưng tốt hơn là sử dụng tên của hàm để truyền đạt thông tin nếu có thể. Ví dụ, trong trường hợp này, chúng ta có thể dọn dẹp comment trên bằng cách đặt lại tên hàm thành `responderBeingTested`.

Trường hợp dưới đây có vẻ tốt hơn một chút:

```
// format matched kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile(
  "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

Trong trường hợp này, comment cho chúng ta biết rằng biểu thức được tạo ra khớp với thời gian và ngày tháng, và được định dạng bằng hàm `SimpleDateFormat.format`. Tuy nhiên, nó có thể rõ ràng hơn nếu code này được chuyển sang một lớp đặc biệt chuyển đổi định dạng của ngày và thời gian. Và sau đó, bạn có thể đưa comment trên vào sọt rác.

### Giải thích mục đích

Đôi khi comment không chỉ cung cấp thông tin về những dòng code mà còn cung cấp ý định đằng sau nó. Trong trường hợp sau đây, chúng tôi thấy một comment ghi lại một quyết định thú vị: khi so sánh hai đối tượng, tác giả quyết định rằng anh ta muốn sắp xếp các đối tượng của lớp mình *luôn* cao hơn các đối tượng khác.

```
public int compareTo(Object o)
{
    if(o instanceof WikiPagePath)
    {
        WikiPagePath p = (WikiPagePath) o;
        String compressedName = StringUtil.join(names, "");
        String compressedArgumentName = StringUtil.join(p.names, "");
        return compressedName.compareTo(compressedArgumentName);
```

```
    }
    return 1; // we are greater because we are the right type.
}
```

Dưới đây là một ví dụ tốt hơn. Có thể bạn không đồng tình với cách giải quyết vấn đề của tác giả, nhưng ít ra bạn biết được anh ấy đang cố gắng làm gì

```
public void testConcurrentAddWidgets() throws Exception {
    WidgetBuilder widgetBuilder = new WidgetBuilder(new
Class[]{BoldWidget.class});
    String text = "'''bold text'''";
    ParentWidget parent =
    new BoldWidget(new MockWidgetRoot(), "'''bold text'''");
    AtomicBoolean failFlag = new AtomicBoolean();
    failFlag.set(false);
    //This is our best attempt to get a race condition
    //by creating large number of threads.
    for (int i = 0; i < 25000; i++) {
        WidgetBuilderThread widgetBuilderThread =
        new WidgetBuilderThread(widgetBuilder, text, parent,
                                        failFlag);
        Thread thread = new Thread(widgetBuilderThread);
        thread.start();
    }
    assertEquals(false, failFlag.get());
}
```

> *Race condition* là gì: https://en.wikipedia.org/wiki/Race_condition
> Hoặc link tiếng việt: https://vi.wikipedia.org

### Làm dễ hiểu

Đôi khi bạn cần dùng comment để diễn giải ý nghĩa của các đối số khó hiểu hoặc giá trị trả về, để biến chúng thành thứ gì đó có thể hiểu được. Nhưng tốt nhất vẫn là tìm cách làm cho các đối số hoặc giá trị trả về đó trở nên rõ ràng theo cách của bạn. Nhưng khi nó là một phần của thư viện, hoặc thuộc về một phần code mà bạn không có quyền tùy chỉnh, thì một comment giải thích dễ hiểu có thể có ích trong trường hợp này.

```
public void testCompareTo() throws Exception
{
    WikiPagePath a = PathParser.parse("PageA");
    WikiPagePath ab = PathParser.parse("PageA.PageB");
    WikiPagePath b = PathParser.parse("PageB");
    WikiPagePath aa = PathParser.parse("PageA.PageA");
    WikiPagePath bb = PathParser.parse("PageB.PageB");
```

```
    WikiPagePath ba = PathParser.parse("PageB.PageA");

    assertTrue(a.compareTo(a) == 0); // a == a
    assertTrue(a.compareTo(b) != 0); // a != b
    assertTrue(ab.compareTo(ab) == 0); // ab == ab
    assertTrue(a.compareTo(b) == -1); // a < b
    assertTrue(aa.compareTo(ab) == -1); // aa < ab
    assertTrue(ba.compareTo(bb) == -1); // ba < bb
    assertTrue(b.compareTo(a) == 1); // b > a
    assertTrue(ab.compareTo(aa) == 1); // ab > aa
    assertTrue(bb.compareTo(ba) == 1); // bb > ba
}
```

Dĩ nhiên, khả năng các comment dạng này cung cấp thông tin không chính xác là khá cao. Xem lại các ví dụ trước để thấy rằng việc xác nhận thông tin từ comment khó thế nào. Điều này giải thích lý do tại sao làm cho comment dễ hiểu là cần thiết, mặc dù điều đó đồng nghĩa với sự mạo hiểm. Vậy nên trước khi bạn viết những comment như thế này, hãy chắc chắn rằng không còn cách nào tối ưu hơn, vì sau đó bạn cần quan tâm đến độ chính xác của chúng mỗi khi bạn chỉnh sửa lại code.

### Các cảnh báo về hậu quả

Đôi khi nó rất hữu ích để cảnh báo các lập trình viên khác về hậu quả xảy ra. Ví dụ, đây là một comment giải thích tại sao test case này lại bị tắt:

```
// Don't run unless you
// have some time to kill - Đừng chạy hàm này, trừ khi mày quá rảnh
public void _testWithReallyBigFile() {
    writeLinesToFile(10000000);
    response.setBody(testFile);
    response.readyToSend(this);
    String responseString = output.toString();
    assertSubString("Content-Length: 1000000000", responseString);
    assertTrue(bytesSent > 1000000000);
}
```

Tất nhiên là ngày nay, chúng tôi tắt các test case bằng cách sử dụng thuộc tính `@Ignore` với một chuỗi giải thích thích hợp: `@Ignore ("It takes too long to run")`. Nhưng trước khi JUnit 4 xuất hiện, việc đặt một dấu gạch dưới vào trước tên hàm là một quy tắc rất phổ biến. Các comment đồng thời được xem như là cách đánh dấu để lập trình viên chú ý đến cảnh báo của hàm hơn.

Đây là một ví dụ đau khổ khác:

```
public static SimpleDateFormat makeStandardHttpDateFormat()
{
    //SimpleDateFormat is not thread safe,
    //so we need to create each instance independently.
```

```
    SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM yyyy
HH:mm:ss z");
    df.setTimeZone(TimeZone.getTimeZone("GMT"));
    return df;
}
```

Có thể còn nhiều cách tốt hơn. Tôi đồng ý. Nhưng comment trong trường hợp này là hoàn toàn hợp lý, nó sẽ ngăn được một số lập trình viên ham hố sử dụng một phương thức khởi tạo tĩnh.

### TODO comments

Đôi khi việc để lại các dòng comment dạng `//TODO` là điều cần thiết. Trong trường hợp dưới đây, comment dạng TODO giải thích tại sao hàm này lại là hàm suy biến, và tương lai của hàm sẽ như thế nào:

```
// TODO-MdM these are not needed - Hàm này không cần thiết
// We expect this to go away when we do the checkout model
// Nó sẽ bị xóa khi chúng tôi thực hiện mô hình thanh toán
protected VersionInfo makeVersion() throws Exception
{
    return null;
}
```

Những comment dạng TODO là những công việc mà lập trình viên cho rằng nên được thực hiện, nhưng vì lý do nào đó mà họ không thể thực hiện nó ngay lúc này. Nó có thể là một lời nhắc để xóa một hàm không dùng nữa, hoặc yêu cầu người khác xem xét một số vấn đề: đặt lại một cái tên khác tốt hơn, lời nhắc thay đổi code của hàm khi kế hoạch của dự án thay đổi,.. Nhưng dù TODO có là gì đi nữa, nó chắc chắn không phải là lý do để bạn quăng đống code ẩu, code bừa vào dự án.

Ngày nay, hầu hết các IDE đều cung cấp các tính năng đặc biệt để định vị các comment TODO, do đó bạn không cần lo việc bỏ quên/lạc mất nó. Tuy vậy, bạn sẽ không muốn code của bạn bị lấp đầy bởi TODO. Vậy nên hãy thường xuyên để mắt tới chúng và dọn dẹp chúng ngay khi có thể.

### Khuếch đại

Comment có thể được dùng để khuếch đại tầm quan trọng của một cái gì đó có vẻ không quan trọng:

```
String listItemContent = match.group(3).trim();
// the trim is real important. It removes the starting
// spaces that could cause the item to be recognized
// as another list.
new ListItemWidget(this, listItemContent, this.level + 1);
return buildList(text.substring(match.end()));
```

### Javadocs in Public APIs

Không có gì hữu ích và tuyệt vời bằng một public API được mô tả tốt. Các javadoc của thư viện chuẩn của Java là một trường hợp điển hình. Sẽ rất khó để viết các chương trình Java mà thiếu chúng.

Nếu đang viết một public API, chắc chắn bạn nên viết javadoc tốt cho nó. Nhưng hãy ghi nhớ những lời khuyên còn lại của chương này, các javadoc có thể là một cú lừa, hoặc mang lại phiền toái như bất kỳ comment nào khác .

## Bad Comments

Đa số các comment rơi vào thể loại này. Chúng thường được sử dụng như cái cớ cho việc viết code rởm hoặc biện minh cho các cách giải quyết đầy thiếu sót, các giá trị không đầy đủ so với cách lập trình viên nghĩ về nó.

### Độc thoại

Quăng vào một comment chỉ vì bạn thấy *thích*, hoặc chỉ vì quá trình xử lý cần đến nó, điều đó được gọi là *hack* (giải quyết vấn đề không theo cách thường mà dùng thủ thuật, đường tắt,...). Nếu bạn quyết định viết comment, hãy dành thời gian cho nó để đảm bảo đó là comment tốt nhất mà bạn có thể viết.

Dưới đây là một ví dụ tôi tìm thấy trong FitNesse, comment này có thể hữu ích. Nhưng tác giả đã vội vàng hoặc đã không quan tâm nhiều đến nó. Cách anh ta độc thoại làm cho người đến sau cảm thấy hoang mang:

```
public void loadProperties()
{
    try
    {
        String propertiesPath = propertiesLocation + "/" +
PROPERTIES_FILE;
        FileInputStream propertiesStream = new
FileInputStream(propertiesPath);
        loadedProperties.load(propertiesStream);
    }
    catch(IOException e)
    {
        // No properties files means all defaults are loaded
    }
}
```

Comment trong khối `cacth` trên nghĩa là gì? Hẳn là nó có ý nghĩa gì đó đối với tác giả, nhưng lại không được diễn giải đầy đủ. Rõ ràng, nếu `IOExeption` xãy ra, điều đó có nghĩa là không có thuộc tính file, và trong trường hợp này các thuộc tính mặc định sẽ được tải. Nhưng, cái gì sẽ tải thuộc tính

mặc định? Những thuộc tính mặc định đó đã được tải trước khi gọi `loadProperties()`? Hay chúng được tải khi `loadedProperties.load(propertiesStream)` phát sinh Exception? Hay chúng được tải sau khi gọi `loadProperties()` ? Có phải tác giả chỉ đang cố làm hài lòng chính bản thân anh ta về việc bỏ trống khối `catch`? Hoặc, kinh khủng hơn – tác giả đã dùng comment như một dấu hiệu, để sau này quay lại và viết các đoạn code tải các thuộc tính mặc định vào khối `catch`?

Cách duy nhất là kiểm tra code của hệ thống để tìm hiểu những gì xảy ra. Bất kỳ comment nào khiến bạn phải lục tung code của hệ thống lên để tìm hiểu thì comment đó không truyền tải tốt thông tin cho bạn, và nó không xứng đáng với số bit nó chiếm trong mã nguồn.

### Các comment thừa thải

Listing 4-1 cho thấy một hàm đơn giản với các comment ở đầu hoàn toàn thừa thải. Comment này, có lẽ, còn làm người đọc mất thời gian hơn so với việc đọc code của hàm.

Listing 4-1
**waitForClose**

```
// Utility method that returns when this.closed is true. Throws an
exception
// if the timeout is reached.
// Phương thức return khi this.closed là true, phát sinh ngoại lệ nếu
hết thời gian chờ
public synchronized void waitForClose(final long timeoutMillis)
 throws Exception
{
    if(!closed)
    {
        wait(timeoutMillis);
        if(!closed)
            throw new Exception("MockResponseSender could not be
                                 closed");
    }
}
```

Comment này nhằm mục đích gì? Nó chắn chắn không cung cấp nhiều thông tin hơn code. Nó không diễn giải cho code, không cung cấp mục đích hoặc lý do. Nó không dễ hiểu hơn code. Sự thật là, nó ít chính xác hơn code nhưng lại lôi cuốn người đọc chấp nhận sự thiếu chính xác đó thay cho hiểu biết thật sự.

Bây giờ hãy xem xét vô số comment vô ích và dư thừa của javadocs trong Listing 4-2 được lấy từ Tomcat. Những comment này chỉ để làm lộn xộn và làm code thêm khó hiểu. Chúng tồn tại nhưng không vì mục đích cung cấp thông tin mà chỉ khiến mọi thứ tệ hơn. Tôi chỉ chỉ ra cho bạn được một vài dòng đầu tiên, còn nhiều hơn nữa nếu bạn xem qua hết module này.

Listing 4-2

**ContainerBase.java (Tomcat)**

```
public abstract class ContainerBase
 implements Container, Lifecycle, Pipeline,
 MBeanRegistration, Serializable {
    /**
     * The processor delay for this component.
     */
    protected int backgroundProcessorDelay = -1;
    /**
     * The lifecycle event support for this component.
     */
    protected LifecycleSupport lifecycle = new LifecycleSupport(this);
    /**
     * The container event listeners for this Container.
     */
    protected ArrayList listeners = new ArrayList();
    /**
     * The Loader implementation with which this Container is
     * associated.
     */
    protected Loader loader = null;
    /**
     * The Logger implementation with which this Container is
     * associated.
     */
    protected Log logger = null;
    /**
     * Associated logger name.
     */
    protected String logName = null;
    /**
     * The Manager implementation with which this Container is
     * associated.
     */
    protected Manager manager = null;
    /**
     * The cluster with which this Container is associated.
     */
    protected Cluster cluster = null;
    /**
     * The human-readable name of this Container.
     */
    protected String name = null;
    /**
     * The parent Container to which this Container is a child.
     */
    protected Container parent = null;
```

```
    /**
     * The parent class loader to be configured when we install a
     * Loader.
     */
    protected ClassLoader parentClassLoader = null;
    /**
     * The Pipeline object with which this Container is
     * associated.
     */
    protected Pipeline pipeline = new StandardPipeline(this);
    /**
     * The Realm with which this Container is associated.
     */
    protected Realm realm = null;
    /**
     * The resources DirContext object with which this Container
     * is associated.
     */
    protected DirContext resources = null;
```

### Các comment sai sự thật

Đôi khi, với mục đích hoàn toàn trong sáng, một lập trình viên diễn đạt ý định của anh ta trong comment, nhưng nó lại không đủ chính xác. Hãy dành một ít thời gian để xem lại comment dư thừa và sai lệch trong Listing 4-1.

Bạn có phát hiện ra comment ở Listing 4-1 sai lệch như thế nào chưa? Phương thức không `return` *khi* `this.closed` là `true`, nó return *nếu* `this.closed` là `true`. Mặt khác, nó giả vờ chờ một khoản thời gian và sinh ra một Exception *nếu* `this.closed` vẫn chưa là `true`.

Một chút thông tin sai lệch, ẩn nấp trong một comment khó đọc hơn cả code mà nó diễn giải. Điều này có thể khiến một lập trình viên khác gọi hàm này và mong muốn nó `return` ngay khi `this.closed` trở thành `true`. Sau đó anh ta phải debug chương trình để tìm câu trả lời cho việc nó thực thi quá chậm.

### Các comment bắt buộc

Thật điên rồ khi cho rằng tất cả các hàm đều phải có javadoc, hoặc mọi biến đều phải có comment. Những comment như vậy chỉ làm rối code, đưa ra những lời bịa đặt, và ủng hộ việc gây nhầm lẫn và vô tổ chức.

Ví dụ, javadoc được yêu cầu cho mọi hàm dẫn đến sự kinh khủng khiếp như Listing 4-3. Tình trạng lộn xộn này không giúp ích được gì mà ngược lại,nó chỉ làm xáo trộn code và ngầm tạo ra những cú lừa khác,...

Listing 4-3

```
/**
 *
 * @param title The title of the CD
 * @param author The author of the CD
 * @param tracks The number of tracks on the CD
 * @param durationInMinutes The duration of the CD in minutes
 */
public void addCD(String title, String author,
 int tracks, int durationInMinutes) {
    CD cd = new CD();
    cd.title = title;
    cd.author = author;
    cd.tracks = tracks;
    cd.duration = duration;
    cdList.add(cd);
}
```

### Các comment nhật ký

Đôi khi mọi người thêm một comment vào đầu module mỗi khi họ chỉnh sửa nó. Những comment này tích lũy như một loại nhật ký lưu lại mọi thay đổi đã từng được thực hiện. Tôi đã thấy một số module với hàng trăm dòng như thế này:

```
/*
 * Changes (from 11-Oct-2001)
 * --------------------------
 * 11-Oct-2001 : Re-organised the class and moved it to new package
 * com.jrefinery.date (DG);
 * 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate
 * class (DG);
 * 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
 * class is gone (DG); Changed getPreviousDayOfWeek(),
 * getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
 * bugs (DG);
 * 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
 * 29-May-2002 : Moved the month constants into a separate interface
 * (MonthConstants) (DG);
 * 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG);
 * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
 * 13-Mar-2003 : Implemented Serializable (DG);
 * 29-May-2003 : Fixed bug in addMonths method (DG);
 * 04-Sep-2003 : Implemented Comparable. Updated the isInRange javadocs (DG);
 * 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);
 */
```

Ngày xửa ngày xưa, chúng tôi có lý do chính đáng để tạo và cập nhật các nhật ký này khi bắt đầu một module. Ở thời điểm đó, các hệ thống quản lý mã nguồn (source control) chưa xuất hiện và chúng tôi phải thực hiện điều này. Nhưng ngày nay, những nhật ký này chỉ làm module lộn xộn hơn. Chúng nên được loại bỏ hoàn toàn.

### Các comment gây nhiễu

Đôi khi bạn gặp các comment không cung cấp thông tin gì ngoài sự phiền phức, chúng lặp lại một vấn đề hiển nhiên và không cung cấp thêm thông tin mới

```
/**
 * Default constructor.
 */
protected AnnualDateRule() {
}
```

Rìa-lý? Còn cái này:

```
/** The day of the month. */
private int dayOfMonth;
```

Và sau đó là sự thừa thãi này:

```
/**
 * Returns the day of the month.
 *
 * @return the day of the month.
 */
public int getDayOfMonth() {
    return dayOfMonth;
}
```

Những comment này phiền đến mức chúng tôi phải học cách bỏ qua chúng. Khi chúng tôi đọc code, mắt chúng tôi chỉ lướt qua chúng. Cuối cùng, các comment bắt đầu sai lệch khi code xung quanh chúng thay đổi, còn chúng thì không – ai lại để ý đến chúng chứ.

Comment đầu tiên trong Listing 4-4 có vẻ hợp lý, nó giải thích tại sao khối `catch` được bỏ qua. Nhưng comment thứ hai chỉ đơn thuần là gây nhiễu cho người đọc. Rõ ràng lập trình viên này đã rất mệt mỏi với việc viết các khối `try/catch` và anh ta đang trút giận.

Listing 4-4
**startSending**

```
private void startSending()
{
    try
    {
        doSending();
```

```
    }
    catch(SocketException e)
    {
        // normal. someone stopped the request.
    }
    catch(Exception e)
    {
        try
        {
            response.add(ErrorResponder.makeExceptionString(e));
            response.closeAll();
        }
        catch(Exception e1)
        {
            //Give me a break! – Đưa tao ra khỏi đây!
        }
    }
}
```

Thay vì trút giận vào một comment vô giá trị và gây bối rối cho người đọc, lập trình viên nên nhận ra rằng vấn đề của anh ta có thể được giải quyết bằng cách tái cấu trúc lại phần code đó. Anh ta nên sử dụng năng lượng của mình cho việc chuyển khối `try/catch` đó thành một hàm riêng biệt, như trong Listing 4-5

Listing 4-5
**startSending (refactored)**

```
private void startSending()
{
    try
    {
        doSending();
    }
    catch(SocketException e)
    {
        // normal. someone stopped the request.
    }
    catch(Exception e)
    {
        addExceptionAndCloseResponse(e);
    }
}
private void addExceptionAndCloseResponse(Exception e)
```

```
{
    try
    {
        response.add(ErrorResponder.makeExceptionString(e));
        response.closeAll();
    }
    catch(Exception e1)
    {
    }
}
```

Tránh việc viết comment gây nhiễu bằng quyết tâm làm sạch code. Bạn sẽ thấy điều đó làm cho bạn trở thành một lập trình viên đẳng cấp hơn, và hạnh phúc hơn.

### (Lại là) các comment gây nhiễu – nhưng đáng sợ hơn

Các javadoc cũng có thể trở nên phiền phức. Mục đích của các javadoc sau là gì (từ một thư viện mã nguồn mở nổi tiếng)? Trả lời: Không gì cả. Chúng chỉ là những comment dư thừa, bị sao chép lại như một khao khát cung cấp tài liệu cho lập trình viên.

```
/** The name. */
private String name;
/** The version. */
private String version;
/** The licenceName. */
private String licenceName;
/** The version. */
private String info;
```

Xem những comment trên một lần nữa, bạn có thấy lỗi cắt-dán không? Nếu tác giả đã không chú ý đến comment khi viết (hoặc dán) chúng, sao chúng ta có thể trông mong chúng giúp ích được chúng ta?

### Đừng dùng comment khi bạn có thể dùng hàm hoặc biến

Hãy xem xét đoạn code sau:

```
// does the module from the global list <mod> depend on the
// subsystem we are part of?
if (smodule.getDependSubsystems().contains(subSysMod.getSubSystem()))
```

Chúng có thể được viết lại mà không cần comment:

```
ArrayList moduleDependees = smodule.getDependSubsystems();
String ourSubSystem = subSysMod.getSubSystem();
if (moduleDependees.contains(ourSubSystem))
```

Có thể tác giả đã viết comment trước (tôi đoán thế), và sau đó viết code để hoàn thành nội dung của comment. Tuy nhiên, tác giả nên tái cấu trúc code, như tôi đã làm, để comment được xóa mà không ảnh hưởng đến thông điệp của đoạn code đó.

### Đánh dấu lãnh thổ

Thỉnh thoảng vài lập trình viên thích đánh dấu một vị trí trong tệp mã nguồn. Ví dụ, gần đây tôi đã thấy thứ này trong một chương trình tôi đang xem xét:

```
// Actions ////////////////////////////////
```

Rất hiếm khi các hàm có cùng chức năng được tập hợp bên dưới một câu miêu tả như thế này. Nhưng nói chung, chúng đang bừa bộn nên cần phải loại bỏ, đặc biệt là cả tá dấu slash ở cuối câu.

Theo cách nghĩ này, một comment đánh dấu sẽ gây chú ý mạnh nếu bạn không thường xuyên nhìn thấy chúng. Vì vậy, hãy hạn chế dùng, và chỉ dùng nếu chúng mang lại lợi ích đáng kể. Nếu bị lạm dụng, chúng sẽ bị người đọc cho vào vùng "phiền phức" và nhanh chóng bị bỏ qua.

### Các comment kết thúc

Đôi khi các lập trình viên sẽ đưa ra các comment cụ thể về việc đóng dấu ngoặc nhọn, như trong Listing 4-6. Mặc dù điều này có thể có ý nghĩa với các hàm dài và có nhiều cấu trúc lồng nhau, nhưng chúng lại thường được dùng trong các hàm nhỏ, hoặc các hàm mà lập trình viên thấy thích. Vì vậy lần tới nếu muốn đánh dấu kết thúc một khối lệnh phức tạp, hãy thử chia nhỏ hoặc rút ngắn hàm của bạn.

Listing 4-6
**wc.java**

```
public class wc {
    public static void main(String[] args) {
        BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
        String line;
        int lineCount = 0;
        int charCount = 0;
        int wordCount = 0;
        try {
            while ((line = in.readLine()) != null) {
                lineCount++;
                charCount += line.length();
                String words[] = line.split("\\W");
```

```
                wordCount += words.length;
            } //while
            System.out.println("wordCount = " + wordCount);
            System.out.println("lineCount = " + lineCount);
            System.out.println("charCount = " + charCount);
        } // try
        catch (IOException e) {
            System.err.println("Error:" + e.getMessage());
        } //catch
    } //main
}
```

### Thuộc tính và dòng tác giả

```
/* Added by Rick */
```

Hệ thống quản lý mã nguồn ghi nhớ rất tốt việc ai đã thêm gì, thêm khi nào,.. Không cần phải làm ô nhiễm code bằng những dòng tác giả. Bạn có thể nghĩ rằng những comment như vậy sẽ hữu ích, người khác sẽ biết ai đã làm việc với phần code đó. Nhưng thực tế là chúng có xu hướng ở lại đó trong nhiều năm, và ngày càng ít chính xác, và ít liên quan.

Nhắc lại lần nữa, hệ thống quản lý mã nguồn là nơi tốt hơn để lưu những thông tin này.

### Comment-hóa code

Comment-hóa code (chuyển code thành comment) là một trong những thói quen khiến lập trình viên dễ bị ghét nhất. Đừng làm việc đó!

```
InputStreamResponse response = new InputStreamResponse();
response.setBody(formatter.getResultStream(),
formatter.getByteCount());
// InputStream resultsStream = formatter.getResultStream();
// StreamReader reader = new StreamReader(resultsStream);
// response.setContent(reader.read(formatter.getByteCount()));
```

Những người khác khi thấy comment dạng này sẽ không đủ dũng cảm để xóa nó. Họ nghĩ rằng nó nằm đó vì một lý do nào đó, và chắc hẳn là nó quan trọng.

Hãy xem xét đoạn code dưới đây:

```
this.bytePos = writeBytes(pngIdBytes, 0);
//hdrPos = bytePos;
writeHeader();
```

```
writeResolution();
//dataPos = bytePos;
if (writeImageData()) {
    writeEnd();
    this.pngBytes = resizeByteArray(this.pngBytes, this.maxPos);
}
else {
    this.pngBytes = null;
}
return this.pngBytes;
```

Tại sao lại có hai dòng code bị chuyển thành comment? Chúng có quan trọng không? Có phải chúng để lại một lời nhắc cho một số thay đổi sắp xảy ra? Hay chúng chỉ là một dòng comment đã tồn tại từ nhiều năm trước và đơn giản là không ai thèm dọn dẹp?

Đã từng có một thời gian, vào những năm 60, khi những comment dạng này có ích. Nhưng hiện tại hệ thống quản lý mã nguồn đang và sẽ tiếp tục hoạt động tốt. Hệ thống quản lý mã nguồn sẽ nhớ code cho tôi và bạn, và chắc chắn không có lý do gì để comment-hóa một dòng code. Hãy xóa dòng code đó, chúng ta sẽ không bị mất nó. Tôi thề đấy.

### HTML comment

Đưa HTML vào comment là một hành động xúc phạm lập trình viên. Như cách mà bạn thấy bên dưới. Nó làm comment khó đọc trực tiếp nhưng lại dễ đọc hơn trên trình soạn thảo hoặc IDE khác. Nếu các comment được trích xuất bởi công cụ và hiển thị như một trang web, việc tô vẽ cho comment bằng HTML là trách nhiệm của công cụ đó, chứ không phải của lập trình viên.

```
/**
 * Task to run fit tests.
 * This task runs fitnesse tests and publishes the results.
 * <p/>
 * <pre>
 * Usage:
 * &lt;taskdef name=&quot;execute-fitnesse-tests&quot;
 * classname=&quot;fitnesse.ant.ExecuteFitnesseTestsTask&quot;
 * classpathref=&quot;classpath&quot; /&gt;
 * OR
 * &lt;taskdef classpathref=&quot;classpath&quot;
 * resource=&quot;tasks.properties&quot; /&gt;
 * <p/>
 * &lt;execute-fitnesse-tests
 * suitepage=&quot;FitNesse.SuiteAcceptanceTests&quot;
 * fitnesseport=&quot;8082&quot;
```

```
 * resultsdir=&amp;quot;${results.dir}&amp;quot;
 * resultshtmlpage=&amp;quot;fit-results.html&amp;quot;
 * classpathref=&amp;quot;classpath&amp;quot; /&amp;gt;
 * </pre>
 */
```

### Thông tin phi tập trung

Nếu bắt buộc phải viết một comment, hãy đảm bảo rằng nó giải thích cho phần code gần nó nhất. Đừng cung cấp thông tin của toàn bộ hệ thống trong một comment cục bộ. Xem xét ví dụ dưới đây: Dù thừa thải nhưng nó cung cấp thông tin về cổng mặc định. Tuy nhiên hàm này lại hoàn toàn không có quyền kiểm soát cổng mặc định. Comment không mô tả cho hàm, mà mô tả một phần nào đó trong hệ thống. Tất nhiên, không có gì đảm bảo comment này sẽ được thay đổi khi code mà nó mô tả thay đổi.

```
/**
 * Port on which fitnesse would run. Defaults to <b>8082</b>.
 *
 * @param fitnessePort
 */
public void setFitnessePort(int fitnessePort)
{
    this.fitnessePort = fitnessePort;
}
```

### Quá nhiều thông tin

Đừng đưa các cuộc thảo luận hoặc mô tả không liên quan vào comment của bạn. Comment dưới đây được lấy ra từ một module có chức năng mã hóa và giải mã base64. Một người nào đó có thể đọc code này mà không cần những thông tin phức tạp từ comment.

```
/*
 RFC 2045 - Multipurpose Internet Mail Extensions (MIME)
 Part One: Format of Internet Message Bodies
 section 6.8. Base64 Content-Transfer-Encoding
 The encoding process represents 24-bit groups of input bits as
output
 strings of 4 encoded characters. Proceeding from left to right, a
 24-bit input group is formed by concatenating 3 8-bit input groups.
 These 24 bits are then treated as 4 concatenated 6-bit groups, each
 of which is translated into a single digit in the base64 alphabet.
 When encoding a bit stream via the base64 encoding, the bit stream
 must be presumed to be ordered with the most-significant-bit first.
 That is, the first bit in the stream will be the high-order bit in
```

```
 the first 8-bit byte, and the eighth bit will be the low-order bit
in
 the first 8-bit byte, and so on.
 */
```

### Mối quan hệ khó hiểu

Sự kết nối giữa comment và code mà nó mô tả nên rõ ràng. Nếu bạn gặp khổ sở khi phải viết comment, ít nhất bạn cũng muốn người khác nhìn vào comment và hiểu những gì nó đang nói về code.

Xem xét ví dụ sau:

```
/*
 * start with an array that is big enough to hold all the pixels
 * (plus filter bytes), and an extra 200 bytes for header info
 * bắt đầu với một mảng đủ lớn để chứa tất cả các pixel
 * (cộng với một số byte của bộ lọc), và thêm 200 byte cho tiêu đề
 */
 this.pngBytes = new byte[((this.width + 1) * this.height * 3) +
200];
```

Một byte của bộ lọc nghĩa là gì? Nó có liên quan gì đến +1 không? Hay *3? Hay cả hai? Tại sao lại là giá trị 200 mà không phải giá trị khác? Mục đích của comment là để giải thích code, không phải giải thích chính nó. Thật đáng tiếc khi comment lại cần một lời giải thích cho riêng mình.

### Comment làm tiêu đề cho hàm

Các hàm ngắn không cần nhiều mô tả. Đặt một cái tên đủ tốt cho hàm thường tốt hơn việc đặt một comment ở trước hàm đó.

### Javadocs trong code không công khai

[...] Việc tạo các trang javadoc cho các lớp và các hàm trong hệ thống thường không hữu ích, và thủ tục bổ sung các comment không khác gì sự lộn xộn và làm mất thời gian.

### Ví dụ

Tôi đã viết module trong Listing 4-7 cho *XP Immersion* đầu tiên. Nó được dự định là một ví dụ về phong cách viết code xấu và comment xấu. Kent Back sau đó đã tái cấu trúc code này lại dưới hình thức dễ chịu hơn trước mặt hàng tá sinh viên. Sau đó tôi đã điều chỉnh chúng để phù hợp với quyển sách *Agile Software Development, Principles, Patterns, and Practices* và bài viết *Craftsman* đầu tiên đã được xuất bản trên tạp chí *Software Development*.

Điều tôi cảm thấy thú vị trong module này là, đã có lúc nhiều người trong chúng tôi xem nó là một *tài liệu tham khảo hay*. Còn bây giờ chúng ta xem nó là một mớ lộn xộn. Hãy xem có bao nhiêu vấn đề về comment mà bạn có thể tìm thấy.

Listing 4-7

**GeneratePrimes.java**

```
/**
 * This class Generates prime numbers up to a user specified
 * maximum. The algorithm used is the Sieve of Eratosthenes.
 * <p>
 * Eratosthenes of Cyrene, b. c. 276 BC, Cyrene, Libya --
 * d. c. 194, Alexandria. The first man to calculate the
 * circumference of the Earth. Also known for working on
 * calendars with leap years and ran the library at Alexandria.
 * <p>
 * The algorithm is quite simple. Given an array of integers
 * starting at 2. Cross out all multiples of 2. Find the next
 * uncrossed integer, and cross out all of its multiples.
 * Repeat untilyou have passed the square root of the maximum
 * value.
 *
 * @author Alphonse
 * @version 13 Feb 2002 atp
 */
import java.util.*;
public class GeneratePrimes
{
    /**
    * @param maxValue is the generation limit.
    */
    public static int[] generatePrimes(int maxValue)
    {
        if (maxValue >= 2) // the only valid case
        {
            // declarations
            int s = maxValue + 1; // size of array
            boolean[] f = new boolean[s];
            int i;
            // initialize array to true.
            for (i = 0; i < s; i++)
            f[i] = true;
            // get rid of known non-primes
            f[0] = f[1] = false;
            // sieve
```

Listing 4-7
**GeneratePrimes.java**

```
            int j;
            for (i = 2; i < Math.sqrt(s) + 1; i++)
            {
                if (f[i]) // if i is uncrossed, cross its multiples.
                {
                    for (j = 2 * i; j < s; j += i)
                    f[j] = false; // multiple is not prime
                }
            }
            // how many primes are there?
            int count = 0;
            for (i = 0; i < s; i++)
            {
                if (f[i])
                count++; // bump count.
            }
            int[] primes = new int[count];
            // move the primes into the result
            for (i = 0, j = 0; i < s; i++)
            {
                if (f[i]) // if prime
                primes[j++] = i;
            }
            return primes; // return the primes
        }
        else // maxValue < 2
        return new int[0]; // return null array if bad input.
    }
}
```

Trong Listing 4-8, bạn có thể thấy một phiên bản được tái cấu trúc của module trên. Chú ý rằng việc sử dụng comment được hạn chế tối đa. Chỉ có hai comment trong toàn bộ module và chúng đều hợp lý.

Listing 4-8
**GeneratePrimes.java**

```
/**
```

Listing 4-8

**GeneratePrimes.java**

```
 * This class Generates prime numbers up to a user specified
 * maximum. The algorithm used is the Sieve of Eratosthenes.
 * Given an array of integers starting at 2:
 * Find the first uncrossed integer, and cross out all its
 * multiples. Repeat until there are no more multiples
 * in the array.
 */
public class PrimeGenerator
{
    private static boolean[] crossedOut;
    private static int[] result;
    public static int[] generatePrimes(int maxValue)
    {
        if (maxValue < 2)
            return new int[0];
        else
        {
            uncrossIntegersUpTo(maxValue);
            crossOutMultiples();
            putUncrossedIntegersIntoResult();
            return result;
        }
    }
    private static void uncrossIntegersUpTo(int maxValue)
    {
        crossedOut = new boolean[maxValue + 1];
        for (int i = 2; i < crossedOut.length; i++)
            crossedOut[i] = false;
    }
    private static void crossOutMultiples()
    {
        int limit = determineIterationLimit();
        for (int i = 2; i <= limit; i++)
            if (notCrossed(i))
                crossOutMultiplesOf(i);
    }
    private static int determineIterationLimit()
    {
```

Listing 4-8

**GeneratePrimes.java**

```
        // Every multiple in the array has a prime factor that
        // is less than or equal to the root of the array size,
        // so we don't have to cross out multiples of numbers
        // larger than that root.
        double iterationLimit = Math.sqrt(crossedOut.length);
        return (int) iterationLimit;
    }
    private static void crossOutMultiplesOf(int i)
    {
        for (int multiple = 2*i;
                multiple < crossedOut.length;
                multiple += i)
            crossedOut[multiple] = true;
    }
    private static boolean notCrossed(int i)
    {
        return crossedOut[i] == false;
    }
    private static void putUncrossedIntegersIntoResult()
    {
        result = new int[numberOfUncrossedIntegers()];
        for (int j = 0, i = 2; i < crossedOut.length; i++)
            if (notCrossed(i))
                result[j++] = i;
    }
    private static int numberOfUncrossedIntegers()
    {
        int count = 0;
        for (int i = 2; i < crossedOut.length; i++)
            if (notCrossed(i))
                count++;
        return count;
    }
}
```

Thật dễ để chỉ ra comment đầu tiên là dư thừa vì nó đọc rất giống hàm `generatePrimes`. Tuy nhiên, tôi nghĩ rằng comment giúp người đọc dễ tiếp cận thuật toán, vì vậy tôi có xu hướng để lại nó.

Comment thứ hai chắc chắn là cần thiết. Nó giải thích lý do đằng sau việc sử dụng giá trị căn bậc hai làm giới hạn vòng lặp. Tôi không thể tìm thấy tên biến đơn giản hay cách thức khác để làm rõ điểm này. Mặt khác, việc sử dụng căn bậc hai có thể để lại vài vấn đề. Tôi có tiết kiệm được nhiều thời gian khi dùng giá trị căn bậc hai làm giới hạn cho vòng lặp hay không? Thời gian tiết kiệm được có nhiều hơn thời gian tính căn bậc hai hay không?

Hmm, thật đáng để suy nghĩ. Sử dụng căn bậc hai làm giới hạn vòng lặp đem lại thỏa mãn cho máu hacker trong tôi, nhưng tôi không nghĩ người khác cũng có máu đó. Một comment ở vị trí đó sẽ giúp người đọc dễ dàng hiểu nó hơn.

## Tham khảo

**[KP78]:** Kernighan and Plaugher, *The Elements of Programming Style*, 2d. ed., McGrawHill, 1978.