

Clean Code

Chapter 4: Comments

Author: Hamed Damirchi

hameddamirchi32@gmail.com

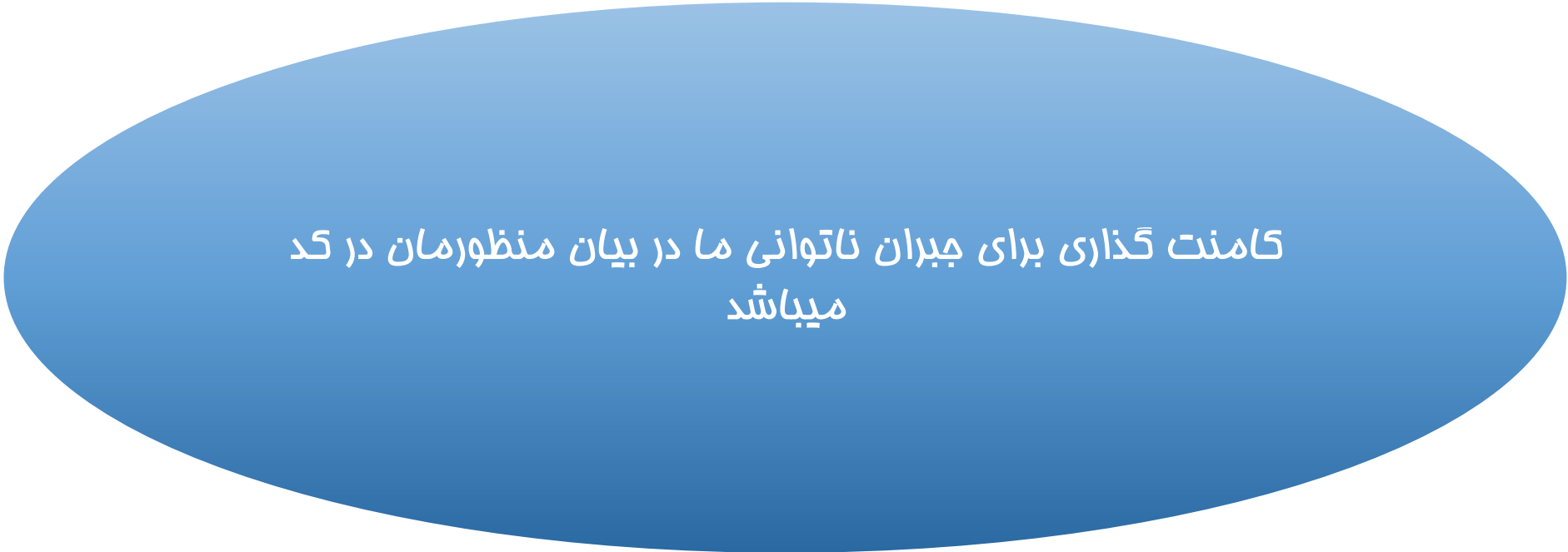
github.com/hamed98

linkedin.com/in/hamed-damirchi-ba4085178/

مقدمه

کد بد را کامنت گذاری نکنید، دوباره بنویسید

- هیچ چیز نمیتواند به اندازه کامنت مناسب کمک کننده باشد
- هیچ چیز نمیتونه به ماژول رو به اندازه کامنت های بیهوده و متعصبانه به هم بریزه
- هیچ چیز نمیتونه به اندازه کامنت های گمراه کننده و اشتباه، خطرناک باشه
- کامنت گذاری لزوما کار درستی نیست همیشه، اگر زبانی که استفاده میکنیم، رسا باشد و برنامه نویس هم توانایی استفاده خوب از آن زبان را داشته باشد، معمولا خیلی کمتر به کامنت گذاری احتیاج میشود و شاید هم اصلا احتیاج نشود



کامنت گذاری برای جبران ناتوانی ما در بیان منظورمان در کد
میباشد

مقدمه-ادامه

هر کامنت یه جور شکست و ناتوانیه. ناتوانی در بیان اون در کد. اما لازمه چون همیشه نمیشه تو کد منظورمون رو واضح برسونیم. هر وقت خواستی یه کامنت بنویسی، کمی فکر کن بین آیا میتونی کد رو جوری تغییر بدی که احتیاج به کامنت نداشته باشه (یعنی منظورت رو تو کد واضح برسونی)

علت مخالفت نویسنده با کامنت گذاری: چون کد تغییر میکنه و ممکنه عملکرد کد بعدا متفاوت از کامنته بشه. کد ها به مرور زمان تغییر میکنن و کامنت ها نمیتونن همواره تغییرات کد رو دنبال کنن واسه همینه که میگه کامنت خیلی وقت ها دروغ میگه!

```
MockRequest request;  
private final String HTTP_DATE_REGEX =  
    "[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"
```

میبینیم که بین کامنت مرتبط با HTTP_DATE_REGEX به خاطر کدهایی که بعداً اضافه شده با خود متغیر فاصله افتاده.

شاید بگید که برنامه نویس همیشه باید کامنت رو آپدیت نگهداره و دقت کنه
اما بهتره این انرژی صرف تمیز نگه داشتن کد بشه، به طوری که نیاز به کامنت رو از بین ببره.
نداشتن کامنت بسیار بهتر از کامنت های غیر دقیقه
همیشه حقیقت رو میتونی تو خود کد پیدا کنی، اما کامنت ها ممکنه اشتباه باشن.

کامنت ها نمیتونن کد بد رو درست کنن

یکی از علت های رایج کامنت گذاشتن، کثیفی کده!

ما یه ماژولی مینویسیم و میدونیم که گیج کننده و بدون نظمه. پس با خودمون میگیم: بهتره برم کامنت بزارم واسش. نه!، درستش اینه: بهتره برم تمیزش کنم.

یه کد تمیز و expressive بدون کامنت بسیار بهتر از یه کد گیج کننده با کامنت های فراوانه. به جای وقت گذاشتن برای کامنت گذاری روی کد بی نظم، اون وقتو بزار و اون کد رو تمیزش کن

منظور تون رو توی کد برسونید

کدوم یکی بهتره؟

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

Or this?

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

این جور تغییرات چند ثانیه بیشتر زمان نمیبره اما خیلی به تمیز شدن کد کمک میکنه خیلی وقت ها به جای کامنت گذاشتن، میتونی یه تابع درست کنی با اسم مناسب که اون کار رو انجام بده. این طوری دیگه هم کامنت لازم نیست و هم میتونی اون تابع رو چند جا استفاده کنی.

کامنت های خوب

تو این بخش قراره بررسی کنیم چه کامنت هایی خوبن:

1. کامنت های قانونی
2. کامنت های informative
3. کامنت هایی که قصدمون رو توضیح میدن
4. کامنت هایی که شفاف سازی میکنن
5. کامنت های هشدار دهنده راجع به عواقب
6. کامنت های TODO
7. کامنت های نشان دهنده اهمیت
8. javadocs برای API ها

کامنت های قانونی

کامنت هایی هستن که به خاطر استاندارد های کمپانی یا هر دلیل دیگه، باید باشن
مثل کامنت های مرتبط با copyright

```
// 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.
```

اما اگه قراره متن بلندی نوشته بشه، بهتره متن تو یه فایل باشه و تو کامنت به اون فایل ارجاع بدیم

کامنت های informative

در بعضی از مواقع بهتره که اطلاعات پایه رو تو کامنت بنویسیم. مثلا مقدار return تو یه تابع abstract.

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

اما اگه بتونیم اسم تابع رو جوری انتخاب کنیم که به کامنت احتیاج نباشه خیلی بهتره. مثلا تو این مورد بزاریم responderBeingTested.

یا مثلا این جا:

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

البته این جا بهتره کلا این تابع منتقل بشه به یه کلاس که حاوی انواع format convertor های مختلفه.

کامنت های توضیح دهنده قصد

کامنت هایی هستند که مربوط به نحوه پیاده سازی نیستند، بلکه توضیح میدن چرا و به چه هدفی این کد نوشته شده.(راجع به چرایی، نه چگونگی)

```
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.
}
```

کامنت کد بالا: توضیح میده چرا وقتی if نقض شد، مقدار ۱ برگردانده میشود.

کامنت های توضیح دهنده قصد-ادامه

```
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());
}
```

کامنت های شفاف کننده

معمولا وقتی استفاده میشن که بخوایم راجع به یه آرگومان مبهم یا خروجی تابع مبهم توضیحی بدیم.

باز هم اگر جا داشته باشه که تو کد منظورمون رو برسونیم حتما باید اون کارو انجام بدیم.

اما گاهی کتابخانه هایی که نصب میکنیم، یا خود توابع زبانی که استفاده میکنیم رو نمیتونیم تغییر بدیم. واسه

همین از کامنت استفاده میکنیم که توضیح بدیم.

کامنت های شفاف کننده-مثال

```
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
}
```

کامنت های شفاف کننده

ریسک اساسی که توی مثال قبل هست، اینه که اشتباه کامنت گذاری کنیم. بنابراین تاجای ممکن باید از این جور کامنت ها پرهیز کنیم و در عوض سعی کنیم تو کد منظورمون رو برسونیم

این بخش بولد شده مهمه! چون در عوض باید این کارو انجام بدیم، نه این که کلا بیخیال تمیز کردن کد و کامنت گذاری بشیم.

کامنت های هشدار دهنده

گاهی اوقات لازمه که به برنامه نویس یه مطلبی رو گوشزد کنیم یا راجع به یه چیزی یه هشدار بدیم. این جور وقتا کامنت گزینه خوبیه.

```
// Don't run unless you
// have some time to kill.
public void _testWithReallyBigFile()
{
    writeLinesToFile(100000000);

    response.setBody(testFile);
    response.readyToSend(this);
    String responseString = output.toString();
    assertSubString("Content-Length: 1000000000", responseString);
    assertTrue(bytesSent > 1000000000);
}
```

کامنت های هشدار دهنده

این کامنتی که این جا اومده بسیار کامنت مناسب و به جاییه:

```
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;  
}
```

این کامنت باعث میشه خواننده متوجه بشه چرا از تابع استاتیک `singleDateFormat` استفاده نشده.

کامنت های TODO

بعضی وقتا برنامه نویس بنا به دلایل مختلف یه کاری که باید انجام بشه رو تو اون لحظه انجام نمیده و میزاره برای بعد. کامنت های TODO این جا استفاده میشن. بعضی از IDE ها فرم خاصی واسه این کامنتا دارن. از جمله دلایل گذاشتن این کامنت:

- یادآوری برای حذف یک فیچر منقضی شده
- درخواست برای یک نفر دیگه برای حل یه مشکل
- درخواست برای انتخاب یه اسم بهتر واسه تابع در آینده
- یادآوری برای انجام تغییراتی که برنامه ریزی شده

این کامنت ها نباید بهونه ای بشه واسه تمیز ننوشتن، یادت باشه: بعدا یعنی هرگز! واسه همین به طور منظم TODO ها رو سرچ کن و اونایی که ممکنه رو برطرف کن

کامنت های نشان دهنده اهمیت

گاهی اوقات یه بخش از کد به ظاهر بودنش مهم نیست، اما در حقیقت اهمیت داره. این جور مواقع میتونیم اهمیت اون تکه کد رو با کامنت نشون بدیم

```
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()));
```

کامنت های API

نوشتن کامنت برای API اهمیت زیادی داره.
مثلا javadoc کامنت های HTML برای ای پی آی تولید میکنه.
اما تو نوشتن کامنت های API موارد گفته شده رو در نظر بگیرید (گمراه کننده نباشه و ...)

کامنت های بد

حالا میخوايم کامنت های بد رو بگيم. جایی که اکثر کامنت ها توی این بخش میفتن!
کامنت های بد معمولا بهانه ای برای کثیف نوشتن کده!
انواع کامنت های بد:

کامنتی که زیر لب واسه خودت میگی! (mumbling)

اگه تصمیم گرفتی یه کامنت بنویسی، وقت بزار و با دقت کلمات رو بنویس
مثلا تو این کامنت، نویسنده واسه خودش یه چیزی گفته اما یا عجله داشته یا واسش خیلی مهم نبوده که دقیق بنویسه.

```
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
    }
}
```

اگر نویسنده کد این جا خواسته که با نوشتن یه کامنت، خودش رو راحت کنه و بخش catch رو خالی بزاره، اشتباه کرده!

اگر اینو نوشته که بعدا بیاد و درستش کنه، اشتباهی بزرگتر کرده!

واسه این که بفهمیم واسه چی اینو نوشته، باید بقیه کد رو بخونیم و بفهمیم داستان چیه

*: هر کامنتی که واسه فهمیدنش مجبور بشی یه ماژول دیگه رو بخونی تا بفهمی منظورش چیه، یک کامنت بسیار بی ارزشیه!

گامنت های زائد

مثال:

Listing 4-1

waitForClose

```
// Utility method that returns when this.closed is true. Throws an exception
// if the timeout is reached.
public synchronized void waitForClose(final long timeoutMillis)
throws Exception
{
    if(!closed)
    {
        wait(timeoutMillis);
        if(!closed)
            throw new Exception("MockResponseSender could not be closed");
    }
}
```

تو این مثال خوندن کامنت بیشتر از خوندن کد طول میکشه!

این کامنت هیچ اطلاعات بیشتری نسبت به خود کد بهمون نمیده.

یه مثال دیگه چون طولانیه تو اسلاید نیوردم، Listing4-2 صفحه ۶۱ کتاب. (دقت کن به این معنی نیست که

نباید داکيومنت واسه API نوشته میشد، بلکه کامنت های فعلی اشتباهن و باید یه چیز بهتر نوشته میشد)

کامنت های گمراه کننده

گاهی اوقات برنامه نویس کامنتی مینویسه که دقیق اتفاقی که تو کد میفته رو توضیح نمیده(ناقصه). و این باعث گمراهی خواننده میشه.

مثلا تو Listing4-1 اتفاقی که تو کد میفته، اینه که به محض این که `isClosed=true` میشه تابع ریترن نمیکنه، بلکه یه مدت زمانی میگذره و بعد ریترن میشه.

این اتفاق به خوبی تو کامنت توضیح داده نشده. بنابراین برنامه نویس ممکنه زمان طولانی رو صرف کنه تا متوجه بشه چرا برنامه داره کند کار میکنه (چون به محض `isClosed=true` شدن ریترن نمیشه)

کامنت های داکیومنت اجباری

این حالت، حالتی که هر متغیر و تابع API بخوایم کامنتی بزاریم که بدیهیه و احتیاج نداره، این کار باعث شلوغ شدن کد میشه.

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);
}
```

کامنت های روزنامه وار (لاگ کردن تغییرات)

فقط میشه گفت ☺ !

البته قدیم یه توجیهی داشت این جور کامنا، چون ابزار های کنترل سورس مثل گیت نبودن. اما الان اشتباه محضه.

```
* 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);
```

کامنت های نویزی

این جور کامنت ها رو چشم خودکار نادیده میگیره و فقط باعث شلوغی کد میشه

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

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

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

کامنت های نویزی-ادامه

```
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!
        }
    }
}
```

کامنت های نویزی-ادامه

تو کد قبل، کامنت اول اوکیه، چون اطلاع میدی که `catch` اول کاملاً نرماله و ریکوئست متوقف شده. اما کامنت دوم چی؟ معلومه برنامه نویس چون خسته شده و `try/catch` های تو در تو دیده اینو نوشته! اما درستش چیه؟ باید محتوای `Exception` آخری رو به تابع جدا میکرد:

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


کامنت های نویزی ترسناک!

یه سری کامنت های دیگه خیلی نویزی و سمّی ان! مثلاً:

این کد از یه پروژه اوپن سورس معروف برداشته شده

وقتی نویسنده به کامنتش اهمیت نمیده چرا باید خواننده واسش مفید باشه؟

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

اگه میتونید از تابع یا متغیر استفاده کنید، از کامنت استفاده نکنید

مثال:

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

بهبتره این کد رو به شکل زیر بازنویسی کنیم. این طوری از متغیر استفاده کردیم و اسم متغیر بیان گر منظور ما هست و دیگه احتیاجی به کامنت نیست.

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

البته در ابتدا اگه اولی رو بنویسیم اشکال نداره، اما مهمه که در ریفکتور های منظمی که انجام میدیم تبدیلیش کنیم به دومی.

نشانگر های موقعیت

منظور از نشان گر های موقعیت چنین چیز هاییه:

// Actions //////////////////////////////////////

موقعیت های نادری هست که این جور کامنت ها مفید باشه، به طور کلی این جور کامنت ها باعث شلوغ شدن کد میشن و بهتره استفاده نشه. مخصوصا اون اسلش های آخر
پس با دقت و به دفعات کم از این بنر ها کامنت ها استفاده کنید. در غیر این صورت این هم میره تو بخش نویز ها و مغز خواننده اتوماتیک نادیده میگیره.

کامنت های نشان دهنده انتهای اسکوپ

استفاده از این جور کامنت ها تو تابع های طولانی شاید بد نباشه، اما تو تابع های کوچیک قطعا باعث شلوغ شدن بیشتر کد میشه.

هر وقت خواستی از این جور کامنت ها استفاده کنی، به جاش تابع رو به تابع های کوچکتر تبدیل کن.

```
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
}
```

ثبت نویسندہ تو کامنت

`/* Added by Rick */`

به جای این کار، از ابزار های کنترل سورس کد (مثل گیت) استفاده کنید. چون خودکار ثبت میکنه چه کسی و چه زمانی چه چیزی رو نوشته.

کامنت کردن کد

وقتی یه بخشی از کد رو کامنت میکنیم، خواننده نمیتونه بفهمه چرا این کامنت شده. آیا پاک کنه؟ نکنه؟ آیا نویسنده کامنت کرده که بعدا از کامنت در بیاره؟ آیا یادش رفته حذف کنه این بخشو؟ اصلا تکلیف خواننده با این بخش کامنت شده چیه؟

```
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()));
```

قبلا که ابزار هایی مثل گیت نبود، یه توجیهی داشت این جور کامنتا. اما الان میدونیم که با ابزار هایی مثل گیت، کد قبلی از بین نمیره و میتونیم برگردونیم. بنابراین کد رو با خیال راحت حذف کنید، اگه لازم شد میشه برگردوند.

کامنت های HTML

این جور کامنت ها که معمولا برای API ها ساخته میشن، باید توسط ابزار داکيومنت ساز (مثل javadoc برای جاوا) ساخته بشن. نه این که خودت دستی کامنت HTML ای بسازی.

```
/**
 * Task to run fit tests.
 * This task runs fitnesse tests and publishes the results.
 * <p/>
 * <pre>
 * Usage:
 * <taskdef name="execute-fitness-tests"
 *   classname="fitnesse.ant.ExecuteFitnesseTestsTask"
 *   classpathref="classpath" />
 * OR
 * <taskdef classpathref="classpath"
 *   resource="tasks.properties" />
 * <p/>
 * <execute-fitness-tests
 *   suitepage="FitNesse.SuiteAcceptanceTests"
 *   fitnesseport="8082"
 *   resultsdir="{results.dir}"
 *   resultshtmlpage="fit-results.html"
 *   classpathref="classpath" />
 * </pre>
 */
```

اطلاعات غیر لوکال (غیر محلی) در کامنت گذاشتن

موقع کامنت گذاشتن، دقت کن که کامنت تماما درباره قطعه کدی باشه که نزدیکشه (کدی که بلافاصله بعدش ظاهر میشه)

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

کامنت بالا، علاوه بر این که کامنت زائیدیه (چون اطلاعات خاصی نمیده)، مقدار **default** رو هم مشخص کرده. در حالی که میدونیم این مقدار دیفالت، یه جای دیگه کد داره تنظیم میشه و آوردنش تو این جا، اشتباهه، چون تابع زیرش هیچ کنترلی روی مقدار دیفالت نداره. هیچ تضمینی هم نیست که وقتی مقدار دیفالت عوض شد، این جا تو کامنت هم عوض بشه.

اطلاعات خیلی زیاد در کامنت

قرار دادن اطلاعات زیاد (مثلا تاریخچه یا توضیحات غیر مرتبط) تو کامنت اشتباهه.

مثلا کامنت زیر واسه یه مازولی که قرار بود صرفا تست کنه که آیا عمل دیکد و انکد base64 درست انجام میشه یا نه گذاشته شده. به جز خط اول (شماره RFC) احتیاجی به مابقی کامنت که برای خیلی ها غیر قابل فهمه، نیست.

/*

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.

*/

کامنت های غیر واضح

ارتباط بین کامنت و قطعه کد، باید شفاف باشد. حداقل ویژگی که یه کامنت باید داشته باشد اینه که خواننده با خوندن کامنت و کد مرتبط باهاش، متوجه بشه که کامنت دقیقا داره چی میگه.

مثلا اگه گفتید مشکل کامنت زیر چیه؟!

```
/*  
 * start with an array that is big enough to hold all the pixels  
 * (plus filter bytes), and an extra 200 bytes for header info  
 */  
this.pngBytes = new byte[((this.width + 1) * this.height * 3) + 200];
```

مشکلش اینه که ما با خوندن کامنت هم باز احتیاج به یه کامنت داریم که خود این کامنتو توضیح بده!

مثلا **filter byte** چیه که تو کامنت بهش اشاره شده؟ آیا اون $1 +$ میشه؟ یا اون ضربدر ۳؟ آیا هر پیکسل یه بایته؟ اون ۲۰۰ چیه؟

یادت باشه هدف از کامنت اینه که توضیح دهنده کدی باشه که نمیتونه خودش رو توضیح بده.

📄 Function Header

توابع کوچک معمولاً احتیاجی به توضیح ندارند، چون اگر اصول رعایت بشه، یک کار انجام میدن و اون یک کار تو اسم تابع میاد. بنابراین احتیاجی به توضیح این که تابع داره چی کار میکنه نیست. انتخاب یه اسم خوب برای تابع خیلی بهتر از نوشتن یه کامنت واسه توضیح کار تابعه.

نوشتن javadoc برای کدهای غیر پابلیک

همونقدر که نوشتن جاواداک (داکیونت API) برای API های public خوبه، نوشتنش برای کدی که قرار نیست public بشه، خوب نیست.

مثلا برای کلاس ها و توابع داخلی (private) نباید استفاده بشه.

مثال

یه مثال خیلی خوب از کامنت های بد و خوب (و کلا کد بد و خوب) تو صفحه ۷۱ کتاب هست (صفحه ۱۰۲ پی دی اف)

متنی که بعد مثال نوشته شده مهمه. میگه تو این کد دومیه کامنت اول رو میشه راجع بهش بحث کنیم که آیا ضروریه بودنش یا نه. اما کامنت دوم که میگه باید تا رادیکال اون عدد بریم واسه پیدا کردن عدد های اول، مهمه و باید باشه. چون اگر نباشه خیلی ها ممکنه گیج شن چرا تا رادیکال پیش رفتیم.