Comments & Formatting

Explain Yourself in Code:

There are certainly times when code makes a poor vehicle for explanation. Unfortunately, many programmers have taken this to mean that code is seldom, if ever, a good means for explanation. This is patently false. Which would you rather see? This:

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

Or this?

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

It takes only a few seconds of thought to explain most of your intent in code. In many cases it's simply a matter of creating a function that says the same thing as the comment you want to write.

TODO Comments:

It is sometimes reasonable to leave "To do" notes in the form of TODO comments.

```
//TODO-MdM these are not needed
// We expect this to go away when we do the checkout model
protected VersionInfo makeVersion() throws Exception
{
      return null;
}
```

TODOs are jobs that the programmer thinks should be done, but for some reason can't do at the moment.

Javadocs in Public APIs:

There is nothing quite so helpful and satisfying as a well-described public API. The javadocs for the standard Java library are a case in point. It would be difficult, at best, to write Java programs without them. If you are writing a public API, then you should certainly write good javadocs for it.

It is just plain silly to have a rule that says that every function must have a javadoc, or every variable must have a comment.

Comments & Formatting

Others:

-> Any comment that forces you to look in another module for the meaning of that comment has failed to communicate to you and is not worth the bits it consumes.

->Don't Use a Comment When You Can Use a Function or a Variable.

->/* Added by Rick */
Source code control systems are very good at remembering who added what, when. There is no need to pollute the code with little bylines.

->Others who see that commented-out code won't have the courage to delete it. They'll think it is there for a reason and is too important to delete. So commented-out code gathers like dregs at the bottom of a bad bottle of wine. We don't have to comment it out any more. Just delete the code. We won't lose it. Promise.

->HTML in source code comments is an abomination, as you can tell by reading the code below. It makes the comments hard to read in the one place where they should be easy to read—the editor/IDE.

Formatting:

The coding style and readability set precedents that continue to affect maintainability and extensibility long after the original code has been changed beyond recognition. Your style and discipline survives, even though your code does not.

Vertical Formatting:

Small files are usually easier to understand than large files are.

We would like a source file to be like a newspaper article. The name should be simple but explanatory. The name, by itself, should be sufficient to tell us whether we are in the right module or not. The topmost parts of the source file should provide the high-level concepts and algorithms. Detail should increase as we move downward, until at the end we find the lowest level functions and details in the source file.

There are blank lines that separate the package declaration, the import(s), and each of the functions. This extremely simple rule has a profound effect on the visual layout of the code. Each blank line is a visual cue that identifies a new and separate concept.

Comments & Formatting

Lines of code that are tightly related should appear vertically dense. For example variable declarations or all imports etc.

Concepts that are closely related should be kept vertically close to each other.

Variables should be declared as close to their usage as possible. Because our functions are very short.

Local variables should appear a the top of each function.

Control variables for loops should usually be declared within the loop statement.

Instance variables should be declared at the top of the class. This should not increase the vertical distance of these variables, because in a well-designed class, they are used by many, if not all, of the methods of the class.

If one function calls another, they should be vertically close, and the caller should be above the callee, if at all possible.

In general we want function call dependencies to point in the downward direction. That is, a function that is called should be below a function that does the calling. This creates a nice flow down the source code module from high level to low level.

Horizontal Formatting:

We should strive to keep our lines short.

We use horizontal white space to associate things that are strongly related and disassociate things that are more weakly related. Consider the following function:

```
private void measureLine(String line) {
        lineCount++;
        int lineSize = line.length();
        totalChars += lineSize;
        lineWidthHistogram.addLine(lineSize, lineCount);
        recordWidestLine(lineSize);
}
```

I surrounded the assignment operators with white space to accentuate them. Assignment statements have two distinct and major elements: the left side and the right side. The spaces make that separation obvious.

On the other hand, I didn't put spaces between the function names and the opening parenthesis. This is because the function and its arguments are closely

related. Separating them makes them appear disjoined instead of conjoined. I separate arguments within the function call parenthesis to accentuate the comma and show that the arguments are separate.

Methods within a class are indented one level to the right of the class. Implementations of those methods are implemented one level to the right of the method declaration. Block implementations are implemented one level to the right of their containing block, and so on.

I can't tell you how many times I've been fooled by a semicolon silently sitting at the end of a while loop on the same line. Unless you make that semicolon visible by indenting it on it's own line, it's just too hard to see.

```
while (dis.read(buf, 0, readBufferSize) != -1)
;
```