Chapter 5 - Formatting

Code formatting is important. It is too important to ignore and it is too important to treat religiously. Code formatting is about communication, and communication is the professional developer’s first order of business.

Vertical Formatting

Vertical Openness Between Concepts

This concept consist in how to you separate concepts in your code, In the next example we can appreciate it.

```
package fitnesse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
      Pattern.MULTILINE + Pattern.DOTALL
      );

  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));
  }

  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
  Pattern.MULTILINE + Pattern.DOTALL);
  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text); match.find(); addChildWidgets(match.group(1));
  }
  public String render() throws Exception { StringBuffer html = new StringBuffer("<b>");
html.append(childHtml()).append("</b>"); return html.toString();
  }
}
```

As you can see, the readability of the first example is greater than that of the second.

Vertical Density

The vertical density implies close association. So lines of code that are tightly related should appear vertically dense. Check the follow example:

```
public class ReporterConfig {

        /**
         * The class name of the reporter listener */
        private String m_className;

        /**
         * The properties of the reporter listener */
        private List<Property> m_properties = new ArrayList<Property>();

        public void addProperty(Property property) { m_properties.add(property);
        }
public class ReporterConfig {
  private String m_className;
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);
  }
}
```

The second code it's much easier to read. It fits in an "eye-full".

Vertical Distance

Variable Declarations. Variables should be declared as close to their usage as possible. Because our functions are very short, local variables should appear at the top of each function,

Instance variables, on the other hand, 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.

There have been many debates over where instance variables should go. In C++ we commonly practiced the so-called scissors rule, which put all the instance variables at the bottom. The common convention in Java, however, is to put them all at the top of the class. I see no reason to follow any other convention. The important thing is for the instance variables to be declared in one well-known place. Everybody should know where to go to see the declarations.

Dependent Functions. If one function calls another, they should be vertically close, and the caller should be above the callee, if at all possible. This gives the program a natural flow. If the convention is followed reliably, readers will be able to trust that function definitions will follow shortly after their use.

Conceptual Affinity. Certain bits of code want to be near other bits. They have a certain conceptual affinity. The stronger that affinity, the less vertical distance there should be between them.

Vertical Ordering

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. *(This is the exact opposite of languages like Pascal, C, and C++ that enforce functions to be defined, or at least declared, before they are used)*

Horizontal Formatting

Horizontal Openness and Density

We use horizontal white space to associate things that are strongly related and disassociate things that are more weakly related. Example:

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

Assignment statements have two distinct and major elements: the left side and the right side. The spaces make that separation obvious.

Horizontal Alignment

```
public class Example implements Base
{
  private   Socket      socket;
  private   inputStream input;
  protected long        requestProgress;

  public Expediter(Socket      s,
                   inputStream input) {
    this.socket =     s;
    this.input  =     input;
  }
}
```

In modern languages this type of alignment is not useful. The alignment seems to emphasize the wrong things and leads my eye away from the true intent.

```
public class Example implements Base
{
  private Socket socket;
  private inputStream input;
  protected longrequestProgress;

  public Expediter(Socket s, inputStream input) {

    this.socket = s;
    this.input = input;
    }
}
```
This is a better approach.

Indentation

The indentation it's important because help us to have a visible hierarchy and well defined blocks.

Team Rules

Every programmer has his own favorite formatting rules, but if he works in a team, then the team rules.

A team of developers should agree upon a single formatting style, and then every member of that team should use that style. We want the software to have a consistent style. We don't want it to appear to have been written by a bunch of disagreeing individuals.