Technical Explorations

by Keith R. Bennett

Applying User Interface Design to Source Code

Nov 28, 2008 • keithrbennett

The same user interface guidelines that make for an outstanding software product apply to source code as well. It turns out that programmers actually have human minds after all.

Without effective and efficient presentation, a product's content and functionality can be missed and misunderstood. This principle applies way beyond the realm of software products. It also applies to traffic signals, automobiles, elevators, remote controls, exercise equipment, and airline cockpits, to name a few.

A good user interface designer understands and exploits the way the human mind perceives and learns. Good designs translate the essence of the product into a presentation that makes the best use of the toolset of the human mind.

Who, Me?

Amazingly, many software developers believe that we as a group somehow transcend the constraints and limitations of human perception, learning, and thought. According to them, that which is necessary for the common masses of "users" does not apply to us, for we are the high priesthood of software engineers, masters of the virtual universe.

Source Code

What am I talking about? Source code, that medium with which we spend so many of our working hours. Although source code is used as input to a compiler, it is also used by humans; that is, us.

If you doubt that principles of user interface design apply to source code, consider a ridiculously extreme exaggeration: a program's source code written as a single line of text with obfuscated names. The compiler would have no problem with this. However, what would it be like to maintain, debug, or enhance such a beast?

Therefore, I propose the following:

Principle #1: In source code, presentation is important.

A good presentation can offer cues and clues that assist the reader in quickly learning and understanding the content; a bad presentation is an obstacle rather than an aid.

Individuals have differing degrees of sensitivity to presentation. Those who have no problem transcending lack of organization and visual cues find it difficult to understand those of us who do. This can cause tension among colleagues with differing attitudes and expectations. Which brings me to…

Principle #2: Human minds differ greatly.

Brains are not like other human organs, which function basically alike, regardless of the individual. The variations among human minds are huge. If you doubt this, then consider the fact that some people risk their lives to save their fellow man, whereas others celebrate their suffering and destruction.

Variations of the mind apply to learning and intelligence as well. Contrary to the implication of the use of "intelligence tests", there are many types of intelligence. Some people can memorize large amounts of data;

others excel at inferring patterns and solving problems; others can read people and know how to get the best out of them; and still others are expert at matters of the heart. Although the amount of respect and value accorded these skills by human societies differs, these are all valuable types of intelligence.

What does this have to do with software engineering?

The failure to recognize, respect, and exploit differences among individuals causes much damage in the workplace. When workers are treated as identical automatons, the result is higher cost, less success, and poor morale.

For example, within our field, some people are good at scanning a problem and crafting a quick and dirty solution; others need more time to thoroughly study the problem domain, but learn it more thoroughly and produce a higher quality solution. Which of the two types should be assigned to develop a prototype? How about a library used by many that must be robust and efficient?

While this at first may appear to be an unfortunate burden, in fact the contrary is true. An astute manager can define and assign tasks in such a way as to exploit each team member's strengths. The result can be morale, timeliness, economy, and project success that far exceeds that derived from the automaton approach. Indeed, the stunning success of the human race in dominating the planet could not have occurred without specialization.

Hopefully you now agree that presentation is important even in source code. Below are a handful of concrete principles and techniques I find helpful toward this end.

Using Spaces as Separators

Almost all natural (human) languages use some kind of space to separate words. The Thai language does not. When reading Thai, one must know from the context where one word ends and another begins. This imposes an extra burden on the reader.

My United States passport displays the passport number as nine consecutive digits, without any separators. Reading this number is difficult and error prone. In contrast, telephone numbers in the U.S. are expressed as (999) 999-9999 rather than 9999999999, and postal zip codes are expressed as 99999-9999 rather than 999999999. The reason for this should be obvious. Therefore, it is also better in source code to use spacing in such a way that optimizes readability.

```
for(int i=0;i<max&&;(!foo)&&(!bar);i++)
```

end

is better as:

```
for(int i = 0; i < max && (! foo) && (! bar); i++)
```

…or some variation thereof.

Right Margins

Text should rarely if ever extend beyond the right margin of the target reader's page. Compelling the reader to scroll horizontally to read source code results in unnecessary delay and invisibility of some of the nearby source code. In other words, all source code that is part of the visible lines of the display should be visible at any one time. The right margin should be determined by the media you intend to support; if you are using low resolution displays or printed output, your margin would be smaller than if source code will only be read from high resolution displays. If you are working with a team member with especially poor eyesight, then using a shorter maximum line length will be very helpful to that person.

Minimize Scope and Distance of Local Variables

Define variables in the narrowest scope possible. If you must have a long method, then consider dividing it into blocks of code with local variables defined inside them. This communicates more information to the reader, namely, that the variables will not be used outside of the block. Doing this also yields the benefit of making refactoring opportunities (that is, method extraction) more obvious.

Be Aggressive in Dividing Large Classes into Smaller Ones

Not many software engineers would argue that objects are bad and we should go back to using procedural languages. Yet I've seen in code lots of missed opportunities for clarification by failing to use classes more.

Live a little. Be a big spender. Use classes with reckless abandon wherever they would be helpful and appropriate. Economizing on the number of classes is not very important, as long as each class has a legitimate purpose of existence and is not an incorrect substitute for composition.

Sometimes I'll be working on a class, and notice that it has grown a bit large. If I scrutinize it, I realize that several methods and variables have much in common, but do not share in performing the main mission of the class. For example, I might be working on a TableModel to display the most recently opened files in a Swing application, and need code that manages the "Most Recently Used" (aka "MRU" list). This MRU list belongs in a class of its own. This provides the following benefits:

- the more cohesive classes are much easier to test.
- the more aggressive encapsulation reduces the number of interactions, making the code simpler.
- the new class may perform a general function that can be reused elsewhere.

Viewing this code in a different light now, one might notice opportunities to simplify it by retrofitting it with third party (e.g. Jakarta Commons) code that is pretested and richer in function.

Wikipedia has an excellent elaboration on this concept of cohesion at http://en.wikipedia.org/wiki/Cohesion_(computer_science).

Use Spacing to Communicate Relationships

A larger separation in meaning should be represented by a larger separation in whitespace. For example, it is common within a method to skip a line for readability. Therefore, I suggest skipping at least *two* lines between method definitions. This makes it easier for the reader to see that it is a method boundary, and can be especially helpful when viewing an entire file. Although this is not as important when viewing a file in an IDE, many developers do not use IDE's, and they would not be that helpful when the source code is printed out.

Line continuations should be indented at least twice as much as indentations expressing logical levels. For example:

```
if (some_really_longgggggggggg_condition &&
        another_really_longgggggggggg_condition) {
    handleIt();
}
```

The degree of indentation signals to the reader whether the line is a continuation or a deeper level of nesting.

Maximize the Signal to Noise Ratio

Anything that appears in the source code should communicate enough meaning to justify its presence. The reader should be able to assume that text exists for a good reason.

For example, the comment below is useless because it adds nothing, and worse than useless because it wastes the reader's time in the process:

```
// gets the foo and puts it into myFoo:
String myFoo = getFoo();
```

Avoid Code Duplication

In addition, avoid code duplication. Copying a several line long method and changing a single line is usually a disservice to the project and to your colleagues. If you haven't already read it, pick up a copy of Martin Fowler's excellent book, "Refactoring: Improving the Design of Existing Code", and internalize it. Other types of duplication are elaborated in "The Pragmatic Programmer", an excellent book by Andy Hunt and Dave Thomas. They refer to this concept as "DRY" (Don't Repeat Yourself).

Be Precise

Be precise. If the way you have expressed something could be interpreted in ways other than your intention, refine it so that it can't. If a colleague asks you to clarify what an identifier means, consider the question a strong clue that the name may need to be improved.

Name Precision and Quality as a Function of Scope and Visibility

The degree to which a name should be precise and accurate depends on the scope of the name's use. For example, the name of a public class should be precise and informative. This is because it may be difficult or even impossible for the reader to consult the source code for clarification. The name may be used far from the source code that would explain it, and that code may not even be available.

On the other hand, the control variable of a two-line loop iterating over an array can be as general as "i". The scope is limited to so small an area that the meaning can be easily inferred from the code right adjacent to it.

Self Documenting Code

Prefer self documenting code to code plus documentation. This can be helpful in simplifying logic for the reader. Complex boolean expressions can be simplified by using intermediate boolean variables with human readable names. This self-documenting code is superior to inscrutable code with supplementary comments because comments tend to get stale (i.e. not get updated) when the code is changed.

Avoid Verbosity

This is an example of the signal to noise ratio issue. Consider the two methods below that express the same thing:

```
boolean isOddVerbose(int n) {

    boolean odd;
    if (n % 2 == 0) {
        odd = false;
    } else
        odd = true;
    }
    return odd;
}

>boolean isOddConcise(int n) {
    return (n % 2) != 0;
}
```

The reader has far less to parse, and the eyes have far less distance to travel, in the concise method.

Another way of making code clearer and more concise is through the use of the ternary operator (? and :), an often underutilized feature of many programming languages. Consider the two alternative representations of getFoo() below:

```
public Foo getFooVerbose(Bar bar) {
    Foo foo;
    if (bar != null) {
```

```
        foo = Foo.X;
    } else {
        foo = Foo.Y;
    }
    return foo;
}

public Foo getFooConcise(Bar bar) {
    return (bar != null) ? Foo.X : Foo.Y;
}
```

While the ternary operator may take a little getting used to, once it is familiar it's a handy tool that's hard to do without.

Conclusion

I hope I have persuaded you that good presentation in source code is not merely a frill, but rather an essential component of effective communication with your colleagues. Above are just a handful of concrete suggestions of how to do this. I look forward to reading your suggestions as well.

About Me

I'm a Sun Certified Java Programmer with over twenty years experience in software development using a wide variety of languages, tools, and operating systems. Currently I live and work in Reston, Virginia, building Bennett Business Solutions, a small consulting company focusing on Ruby software development.

Published with GitHub Pages