Preventing the most-probable errors in design

Robert M. Poston, Editor
Software Standards

The many unique facets of an evolving software product’s design make it interesting to work on. But software design is also perplexing because the evolving product continually compels engineers to reset directions throughout a project. Software designers are often, in fact, redesigners because they go back and again to make improvements in their design when they see a better way. This behavior stems from the fact that designers can’t know everything about their design on day one of a project.

Error-prone phase. There are several widely accepted software design goals in our industry, and most designers start a project with them in mind. The trouble is that these traditional design goals soon conflict with product requirements.

On the first day of a project, the designer supposes that “Of course, I want to make this product easy to modify and understand.” It is not until several days later that design details have evolved enough for the designer to see the effects of another requirement on the emerging modular conception.

An ease-of-use requirement demands a 0.05-second response time. Can the design goal (modularity) and the product requirement (speed) both be satisfied considering that interfaces between the newly designed modules eat up considerable time?

So you can see that a designer’s knowledge and understanding of a software product changes with time. This evolution presents lots of opportunity for errors.

Most-probable errors list. And what do you do about those errors? Those of you following this series know one answer: Compile and use a most-probable errors list. The July and September issues of Software Standards tell how to develop and use that list.

Designers benefit from both compiling and using this homegrown list. To compile a most-probable errors list, some analysis must be made: What mistakes does your organization make in design? Which few cause the most trouble? Many designers will recall the 80/20 rule from Pareto distribution that says that 20 percent of errors cause 80 percent of problems.

After the list is compiled, designers see in a formal document what mistakes are most likely to play havoc with their designs. This awareness helps designers prevent those mistakes in the future. A most-probable errors list doesn’t take long to make, and many design teams notice productivity increases soon after it is used.

New design standard. Designers have to record their design information if they want to communicate it to others or use it themselves later. This information is recorded in a document the IEEE calls a software design description. Recently, the IEEE published a standard called IEEE Recommended Practice for Software Design Descriptions (Std 1016-1987) that describes a software design description. The table of contents for the description in the new standard is illustrated in Figure 1.

Sample list. Jack Barnard of AT&T in Denver chaired the IEEE working group that developed the 1016 standard. Because Barnard brought lots of design experience to the standard, I wanted to know what would be on his most-probable errors list. He agreed to make a list and partition it into three categories of missing, wrong, and extra information (see Table 1). Many of the errors listed will be immediately familiar to designers.

Barnard can put the 1016 standard to work right away to help prevent these errors from recurring, and he can use other error deterrents with the standard to keep even more mistakes out of future designs.

Missing information. The first error on the list is “missing a function.” This might be called a double-trouble error because it is an error that probably first appeared in the requirements specification phase. Had it been eliminated there, it wouldn’t be listed as an error in design. Many design errors are really errors carried forward from requirements, so it is imperative to minimize errors in requirements information (see Software Standards in September) before concentrating on design errors.

Sometimes, though — even with a first-rate requirements specification that lists all necessary functions — you can still miss a function in the design description. To prevent this, you can use a technique called tracing. You can create a table to show where a function is defined in requirements information where it is incorporated into the design description. If there’s an entry in both the requirements and the design columns, every function in the design has been accounted for.

You can easily construct a tracing table by using almost any popular word processor and a few operating system commands with the 1016 design descrip-
tion. In the table of contents (Figure 1), paragraph 3.1.1 is called "Module 1 description." That is where functions to be implemented in that module should be listed. Suppose you are defining your functions using an editor running on an MS-DOS PC. Position the first occurrence of every function name on a line by itself and precede the name with two unique symbols. For example,

```plaintext
^#function_w
any text
^#function_a
```

With function names entered this way in the software design-description file SDD.TXT, you can build your design traceability column with one MS-DOS command line as follows:

```plaintext
find ^#*<a:\sdd.txt | sort > dsf_list.txt
```

The file DSF_LIST.TXT now contains the information for the design column in the traceability table. Here is how the file contents should appear:

```plaintext
function_a
function_w
```

The same function can be used on the requirements specification (use IEEE Std 830) to produce the requirements column for the table. Because you used the DOS Sort command, the entries will be sorted alphabetically. You can print the output files and compare them to discover missing functions. On some operating systems, this comparison can be automated, but MS-DOS’s Comp command is insufficient to do so.

To prevent the second item on the list, "missing interface definition," return to the standard. It is hard to miss Section 5 (titled "Interface Description") in the table of contents. For every module listed in Section 3, there must be an interface description in Section 5.

"Missing data definitions" is the third error on the most-probable errors list. Subsection 3.3 in the standard 1016 outline provides a home for data definitions. Data completeness can be checked two ways: (1) Every piece of data named elsewhere in the design description (such as in Subsection 3.1 or Section 5) must be defined in Subsection 3.3 and (2) software reviews must be conducted.

The fourth error is "missing logic descriptions." Logic descriptions or detailed designs originate in the design phase. The 1016 standard shows you how to prevent this error when it says that every module identified in the architectural or high-level design (Section 3) must have a corresponding detailed design in Section 6.

Wrong information. In the wrong-information category, the same information components (function, interface descriptions, and so on) appear again, but this time the errors are not of missing information but of wrong information. Wrong information usually boils down to ambiguous or inconsistent descriptions.

You can prevent errors of ambiguity or inconsistency in design information exactly the same way you do in requirements information. Use terms with one and only one, definition, and use tables where information is stored once, and only once.

Extra information. A piece of information defined in Subsection 3.3 but not named in any other part of the design description is probably extra information. A module that has a detailed design defined in Section 6 but not named in Section 3 is probably noise. An interface defined in Section 5 but not referenced in Section 3 should be suspect.

Join me next issue for how to prevent the most-probable errors in software testing. As always, I welcome contributions to the series.

---

Table 1. Most-probable errors in design (as compiled by Jack Barnard).

<table>
<thead>
<tr>
<th>Category</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Missing information</td>
<td>Function description</td>
</tr>
<tr>
<td></td>
<td>Interface description</td>
</tr>
<tr>
<td></td>
<td>Data description</td>
</tr>
<tr>
<td></td>
<td>Logic description</td>
</tr>
<tr>
<td></td>
<td>Dependency description (control, data, or time)</td>
</tr>
<tr>
<td>Wrong information</td>
<td>Ambiguous or inconsistent function description</td>
</tr>
<tr>
<td></td>
<td>Ambiguous or inconsistent interface description</td>
</tr>
<tr>
<td></td>
<td>Ambiguous or inconsistent data description</td>
</tr>
<tr>
<td></td>
<td>Ambiguous or inconsistent logic description</td>
</tr>
<tr>
<td></td>
<td>Ambiguous or inconsistent dependency description</td>
</tr>
<tr>
<td>Extra information</td>
<td>Defined but not used function description</td>
</tr>
<tr>
<td></td>
<td>Defined but not used interface description</td>
</tr>
<tr>
<td></td>
<td>Defined but not used data description</td>
</tr>
<tr>
<td></td>
<td>Defined but not used logic description</td>
</tr>
<tr>
<td></td>
<td>Defined but not used dependency description</td>
</tr>
</tbody>
</table>

---

Figure 1. Software design description table of contents (from IEEE Std 1016-1987).