alodar systems™

Integration Tip

WHITE PAPER

How to Analyze Requirements

Most project teams know the importance of gathering requirements and analyzing them before starting design and implementation. A good set of requirements is the foundation upon which a successful project is built. Ill-formed requirements lead to misunderstandings, rework, and, potentially, total project failure.
But what is a "good" requirement? Certainly, like all the other elements of project documentation, it must be clear and succinct. But in addition, a good requirement is testable, assignable, and cost-effective.

Testability

We all probably have been on projects with requirements that read something like "The user interface shall be easy to use." Ease of use is desirable to be sure. And if the final product is not, the project team will certainly hear about it.

But stating this desire as a requirement gives no real guidance on how to proceed. During the course of design and implementation there is no objective way of measuring how well the team is satisfying it. It is just verbiage that gets in the way and creates expectations that might not be realized.

If a requirement is not testable, it is not a requirement. It has to be fixed. It is as simple as that.

If you encounter an untestable requirement during requirement analysis, you can fix it in one of two ways. First, you can rework it so that it does become testable. User interface requirements are particularly hard in this regard of course, but even something like "Eighty percent of a test group shall rate ease of use at least a 4 on a scale of 1 to 5" is superior to "The interface shall be easy to use". The reworking does not guarantee a great interface but it is something that the project team can work towards during development, testing, and user acceptance.

The second way to handle an untestable requirement is to move it to the goals or objectives section of the requirement document. You do not want to get rid of the desire all together so the thought does need a home --- but not as a requirement.

Assignability

Eventually the final product of the project is tested against the list of (hopefully) testable requirements. Often it is discovered that not only do some requirements fail because of flaws but also because, oops, the functionality was just never implemented.

Good integration practice, of course, is to ensure that each and every requirement is assigned to an element of the design. This can be a multi-step process in which requirements are first assigned to the top-level elements such as the Financial or the Customer Relationship Management (CRM) applications. Then within each element they are further assigned to sub-elements, continuing down to the purview of the single individual responsible for a portion of the design or implementation. A rigorous application of assignment helps ensure that no requirement is forgotten.

Even with rigorous assignment, requirements can still "fall between the cracks". One of the main reasons this happens is that some requirements, as written, are applicable at a high level and no designer or developer ownership is ever assigned. For example, "The response time from pressing the Enter key until the screen updates shall be less than one second" is a testable requirement. The problem is that the delay consists of input verification on a client, network messaging, application server and servlet processing, backend application processing, network messaging again, and finally client presentation. Each of these pieces of the design might be the responsibility of a different individual. There is no ownership of the requirement.

The solution to this problem is to create what are called *derived requirements.* Derived requirements are children of a parent requirement. If all of them are satisfied than the parent requirement is satisfied as well. In the example, the derived requirements could be “Input verification shall be less than .1 second” and so on for the rest with the total for all of the pieces summing to 1 second.

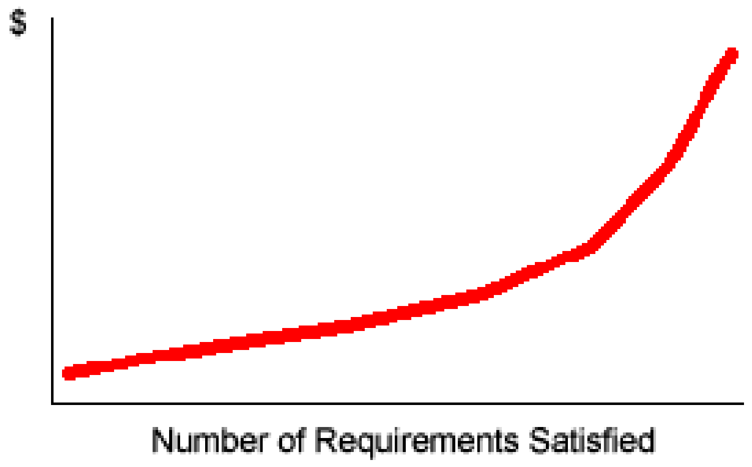
Specifying the allocated delays is a hard problem. A project team often can only guess. But by doing so, timing requirements can be assigned to individual elements. Every designer or developer knows his responsibility. During testing the allocated times can be shifted as needed. The key point is that no requirement falls between the cracks.

Cost-Effectiveness

Not all requirements are created equal. Some are easy to implement, such as “Name, street, city, zip, and state shall be displayed”. Others are complex, hard to implement and hard to test, such as “When the system loading exceeds 80%, the backup process shall be initiated which copies all in-memory records to disk”.

Although it is impossible to do this in reality, consider the thought experiment in which you assigned the actual project cost to design, develop, and test every single requirement. Now sort these costs from the least to the most expensive and sum them up, that is, create a graph of the cost of satisfying one requirement, then two, then three, and so on up until all of the requirements are satisfied, including the most costly ones. The curve you get would look something like that in the figure.

Most of the requirements are satisfied reasonably. A few, however, contribute a disproportionate amount to overall project cost. A cost-effective project only implements requirements up to where the curve begins to rise sharply. In real life we do not have such a quantitative curve, but usually we do have a gut feel for which the hard requirements are.



You can do one of three things with these requirements. First, review just how important they are. They indeed might not be worth the cost. Second, move them to goals or objectives. Keep them around but not as requirements. Third, prototype them before beginning the major development effort. Often prototyping provides design insights that greatly reduce implementation costs.

Testable, assignable, cost-effective -- the marks of a good requirement. Checking for them is an essential part of requirement analysis.