

- ▶ [subscribe](#)
- ▶ [contact us](#)
- ▶ [submit an article](#)
- ▶ [rational.com](#)
- ▶ [issue contents](#)
- ▶ [archives](#)
- ▶ [mission statement](#)
- ▶ [editorial staff](#)

▶ **What Does "No Time for Requirements" Mean?**

by [Jim Heumann](#)

Requirements Management
Evangelist
Rational Software

Have you ever heard someone say something like the following?

"We don't have time to gather requirements. If we don't start coding right now, we'll never meet our deadline."

Such statements -- which are not uncommon -- represent important clues about the culture and maturity of an individual or a team. When someone says there is no time for requirements, what does that really mean, and what are the implications? Let's take a closer look.

The term *requirement* has many definitions; each method, process, methodologist, and pundit seems to have a unique twist on what a requirement actually is. All would probably agree, however, that requirements should clearly state what it is that a given software project is supposed to build and deliver. If we substitute this simple definition for the term *requirements* in the statement above, then it reads like this:

"We don't have time to figure out what we need to build and deliver. If we don't start coding now..."

Seems unlikely that someone would make that statement out loud, doesn't it? So what is really going on when people say they don't have time to do requirements? And what happens if the requirements don't get done?

What Does It Mean?



I believe that when people make the statement in question, what they really mean is one of the following:

1. "I already know the requirements -- or I can figure them out on the fly."
2. "The requirement specifications I've worked with in the past have been so bad -- I think I'm better off without such things."
3. "I don't really care what we build, as long as we get it done on time (and collect our money)."
4. "This application idea is so new (or so unique) that we just can't pin down requirements in advance."

Each possible interpretation has its own perils, as we shall see.

"I Already Know..."

This is the "kindest" interpretation. The people (or person) in charge of the project feel that they implicitly know what the requirements are or that they can figure them out as they go, while coding. If we dig a little deeper, though, we see that what they really mean is, "My developers know the requirements or can figure them out." In the end, developers are the ones who implement the functionality of any software program; if the specifications about what to build (i.e., the requirements) are not written down and agreed upon, then it is developers who will ultimately make the call on what gets implemented.

What's wrong with that? In many organizations, that is pretty much how they have always done it. Well, one reason why it is no longer acceptable to do things this so-called "old-fashioned way" is that projects today are more complex than they used to be. And they are more complex in two domains: the problem domain and the implementation domain.

To begin with, software has penetrated areas of our lives that we never even imagined it would affect ten years ago: We have software for managing bull breeding, navigating automobiles, detecting fraud in cellular phone use, and for many, *many* other specialized purposes; most developers cannot possibly know or understand the problems to be solved in these particular areas. And as application areas have grown more complex, so has programming. Distribution, concurrency, real-time, interrelated components, new languages, and other issues that developers must consider in order to create a robust system, all place pressure on them to become more and more expert at writing code. Also, these new demands mean that developers have even less time and capacity for figuring out the problem the system needs to solve.

Another reason not to rely on developers to "figure it out on the fly" is that they have an inherent conflict of interest. They have deadlines, sometimes based on their own estimates, sometimes on others'. If you give them a very vague statement of what to do, they will have many choices about how to implement the functionality. It would be natural for them to choose the one that most closely matches their main goal -- which is usually

meeting the deadline.

I once had the job of writing additional code for a GUI (Graphical User Interface) builder that had no facility for the user to specify the color of elements such as push buttons, list boxes, and so on. The stated requirement for the functionality I had to add was "Allow the user to specify color for GUI elements." Now, there are a lot of ways you can allow users to specify a color: You can give them a list of colors in text format; you can present a bunch of boxes in various colors and have them choose; you can show them a color wheel and ask them make a choice; you can ask them to type in numbers between 1 and 256 for red, green, and blue (RGB) values, then combine those to make any color they like. Some options make it easy for the user and harder for the developer; some reverse that relationship.

I am chagrined to admit it now, but I chose to have the user type in RGB values. Why? Because that was the easiest option for *me* to implement, and it allowed me to stay on schedule. The problem was, it was the hardest option for a user. Why do I mention this embarrassing example? Because it shows that forcing developers to make important decisions about user requirements can put those developers in untenable situations and often leads to unsatisfactory results.

"I've Had Past Experiences with Bad Requirements"

Another reason managers may think that gathering requirements is not worthwhile is that they have run up against bad requirements in past projects. Bad requirements come in many forms, but the bottom line is that they either don't provide enough information to really build anything or they provide information that is misleading or contradictory. In such cases, it is true that the project would be better off if everyone just made their best guesses. But remember: Just because you had to work with bad requirements in the past doesn't mean it always has to be that way. Helpful guidelines, techniques, and tools are available to help your team create good and useful requirements.

Good and Bad Requirements

According to the IEEE 830 Documentation Standard for a Software Requirements Specification, the criteria for a high-quality requirements set are as follows:

Unambiguous -- Every statement has one interpretation. Terms are clear and well defined.

Complete -- All significant requirements are included. No items have been left for future definition.

Verifiable -- All features specified as part of the project have a corresponding test to determine whether they have been successfully delivered.

Consistent -- Conflicting terminology, contradictory required actions, and impossible combinations are notably absent.

Modifiable -- Redundancy is absent; index and contents are correct.

Traceable -- Each referenced requirement is uniquely identified.

Correct -- Every stated requirement represents something required of the system to be built. This may sound obvious, but it is surprisingly easy to

include extraneous requirements, or requirements that really pertain to a separate system entirely.

How do you recognize bad requirements? Compare them against the criteria above; if they don't match, then beware.

I recently came across the following requirement in an actual project requirements document I was reviewing:

"The application must be extremely fast and powerful."

Clearly, this is a poor requirement because "fast" and "powerful" are highly subjective terms, so there is no way to verify whether the system meets the requirement.

Organization and Tools Can Help

In some instances, requirements are not intrinsically bad, but they are unusable because there are so many of them -- and they may be poorly organized. Traditional software requirements specifications often contain thousands of declarative statements that "The system shall..." do something. It is very hard for anybody to understand several thousand things at once. Classifying and organizing the requirements can help. Splitting up the big list into smaller lists in a hierarchy that identifies explicit relationships represents a big step forward in understandability and usability. Writing the functional requirements as a user-oriented "story" can also help. Use cases provide a mechanism to highlight how a given "actor" uses the system and what the system does in response. This goal-oriented approach puts the requirements in context and helps to make them easier to understand.

Tools can help make requirements usable, too. Having the requirements in a specialized requirements database rather than in a collection of unrelated documents makes them much easier to organize, search, and sort. People filling different roles in a software development team can "slice and dice" them any way they choose. A project manager, for example, can use the requirements to help keep the project on track, using the tool to find information about how many requirements have been implemented, how many new ones have been added, and how many have changed during the last week. An architect can search for all of the requirements that are specified as difficult to program and that affect the architecture, so that these can be implemented early in the project in an effort to drive out technical risk. Tools won't help you write good requirements, but they can help you organize them and find the information you need.

"I Don't Really Care"

This is the saddest case. Some people "don't have time" for requirements because they don't really care if they build the right thing. Perhaps they are working on a project that they feel is doomed due to political or organizational problems, so they think that doing good requirements won't help anyway. Or perhaps they have chosen to work on a given project for personal reasons, maybe to get experience in a new technology or

technique. Whatever the reason, if people don't care, then you have a problem that organization and tools can't solve.

"This Application Is So New..."

Software projects aren't always trying to solve well-defined problems with known solutions. Many seek to expand the boundaries of what is possible for computers. And if there is no precedent for your system, it is very hard to get potential users to tell you how they want it to work. In 1978, Dan Bricklin developed VisiCalc, the first electronic spreadsheet, on an Apple II while attending Harvard Business School. There was nothing like it at the time; it is unlikely that he could have gotten requirements from "users," simply because there weren't any. Similarly, consider Napster. It is also unlikely that Shawn Fanning, the 18-year-old Northwestern University dropout who developed Napster in 1999, did extensive requirements gathering before working eighteen hours a day for three months to write the original source code and deploy the site -- yet it was a huge success.

It is sometimes said that such systems are "exploratory software engineering." For their first release, extensive requirements gathering is unlikely to add much value. Subsequent releases, however, are a different story. Once users catch on, they develop certain wants and expectations; developing new features that you think are "cool" may not meet those expectations. Asking users what they need, expressing it clearly, writing it down, organizing it, and using it to design and implement new functionality can help ensure that a good idea grows instead of fizzles.

Lifecycle Implications

When people say they don't have time to do requirements, they are usually thinking only about the first release deadline and not about the implications that omission may have for other concerns throughout the project lifecycle, including schedule, architecture and design, test and performance, and usability issues.

Schedule

If you don't know what you are supposed to do, how do you know when you are done? A prevalent problem for many software projects is late delivery, often caused by "feature creep." Not having control of what is supposed to go into a given release opens the floodgates to ad hoc requests for more features to develop and implement. Having no means to understand the relative value of each request and make informed decisions can easily lead to "biting off more than you can chew" and missing deadlines. With well-defined requirements, you can more easily turn down requests for features that do not further your goals and maintain tighter control of project scope.

Architecture and Design

Without a high-level understanding of the range of functionality that must go into a system, it is easy to create a design that is not adaptable or scaleable. An architecture that meets existing requirements but does not

take future changes into account may end up being hard to modify or extend. Well-defined requirements anticipate modifications and changes, providing a framework for a flexible architecture and overall design.

Test and Performance

If you test software without requirements, then you are merely "making sure the application does what it does" without crashing. Testers can look for GUI elements that don't work, system crashes, and more obvious problems, but in the end they have no means to verify that the program really meets the user's objectives. One simple definition of quality software is that "it does what it is supposed to do." Requirements *specify* what a system is supposed to do, so from very early on in the development lifecycle, you can begin testing to ensure that you are actually creating the workable system your user needs.

Usability

Even if you have functional requirements to design and test against, failures can still occur. If the team doesn't know, for example, how simple the user interface and how fast transaction times must be, how many users the system must support, and who has to maintain the system, then you can easily wind up with an application that provides all of the requested functions but does not solve the problem. By providing information about *non-functional requirements*, you can ensure that your system designers and developers won't spend all of their time implementing the functionality without considering other factors that are vital to the system's success.

Dig a Little Deeper

So now, if you hear people in your organization saying, "We don't have time for requirements," then consider the implications. If your team is inventing something brand new, then you might be able to get away without gathering requirements. But if, like the majority of organizations, you are working in domains for which models and precedents are plentiful, then it may be worth your while to dig a little deeper and find out what those people really mean. Do they intend to let developers make important decisions outside their respective areas of expertise? Do they understand the difference between a good requirement and a bad one? Do they simply not care what system you develop? Do they understand the potential impact on other areas of the project lifecycle if they don't specify requirements? If you understand what is really going on below the surface, you can respond to such statements in a way that can help make your project rather than break it.



For more information on the products or services discussed in this article, please click [here](#) and follow the instructions provided. Thank you!

