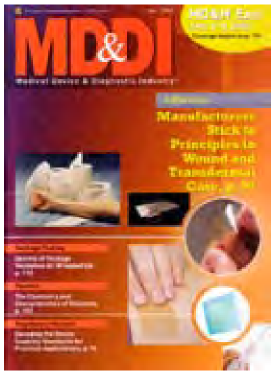
Validating Software for Manufacturing Processes

by David A. Vogel, Ph.D.
Intertech Engineering Associates, Inc.

as published in Medical Device & Diagnostic Industry, May 2006



The software for medical device processes—engineering, quality, regulatory, and so on—must be validated. You don't have to be a software engineer to do it. The software for medical device processes—engineering, quality, regulatory, and so on—must be validated. You don't have to be a software engineer to do it.

Validate it? I just want to use it! Sound familiar? Most companies in the medical device industry understand and accept the need to validate software that is critical to the functioning of a medical device. Perhaps not as widely understood or accepted is the regulatory requirement to validate software that is used to automate any process that is part of a medical device manufacturer's quality system. This broad requirement encompasses manufacturing, engineering, quality, and regulatory functions within the firm.

The responsibility for validating such software often falls to the user of the software, who knows little, if anything, about software validation. Although users may not feel qualified to validate software, it is not necessarily essential to hire software professionals to validate it for them. Non-software engineers can validate many types of software. This article is designed to help non-software engineers understand what validation is, how to go about it, and how to know which validation projects really should be left to software-quality professionals.

Some non-software engineers feel that doing software validation is wasting time. Perhaps they have seen or been part of software testing that simply exercises all the menu commands, and never finds any defects—ever. If validation efforts only include testing, engineers are probably over-looking critical validation activities.

COMPANY PROFILE

Intertech Engineering Associates, Inc.

Address:	100 Lowder Brook Avenue Suite 2500 Westwood, MA 02090 www.inea.com - (781) 801-1100
Industry:	(Electro)Medical Devices
Services:	Assessments Training Consulting Hands-on Engineering
Skills:	Product Design Risk Management Requirements Engineering Electronics Development Software Development Software Verification and Validation Production/Quality System Software Validation

Regulatory Background

FDA's quality system regulation (QSR) that applies to the validation of the software types discussed here is 21 CFR 820.70(i), which addresses automated processes. In addition, 21 CFR Part 11 is the collection of regulations related to electronic records and electronic signatures.

It is useful to look at the regulatory origins to understand what is law and how it differs from the guidance information that FDA produces to interpret the law. The regulation that specifically applies to this software is found in the section on Production and Process Controls, and states

> "*(i) Automated processes.* When computers or automated data processing systems are used as part of production or the quality system, the manufacturer shall validate computer software for its intended use according to an established protocol. All software changes shall be validated before approval and issuance. These validation activities and results shall be documented. 21 CFR 820.70*(i)*"

FDA is actually quite good about producing documents to interpret and elaborate on the federal regulations they are charged with enforcing. The agency issued a software validation guidance in January 2002. This document, "General Principles of Software Validation; Final Guidance for Industry and FDA Staff" (commonly referred to as the GPSV), includes a section (Section 6) that interprets this regulation[1]. The next step is to learn how to apply that interpretation.

What Kinds of Software Must Be Validated?

To answer this question, it's important to understand why the software needs to be validated. There are precise definitions of validation and broadly accepted activities that lead to the conclusion that software is validated. But, when all is said and done, validation activities must confirm that the software does what the user wants it to, and that patients, users, bystanders, the environment, and the medical device company are reasonably well protected from any potential failure of the software.

So, what software needs to be validated other than that which is part of a medical device? It is often tempting to simply conclude that all software should be validated.

What is Required?

As noted earlier, 21 CFR 820.70(i) requires validation of software that automates all or part of any process that is part of the quality system. That software includes the following:

- Software used as part of the manufacturing process (including software embedded in machine tools, statistical process control software, programmable logic controllers [PLCs], and software in automated inspection or test systems).
- Software used in process validation (such as statistical calculation software, spreadsheets etc.).
- Software used in design and development processes (such as CAD software, CAM software, software development tools, software test tools, compilers, editors, code generators, etc.).
- Software used to automate part of the quality process (such as complaint-handling systems, lot-tracking systems, training-database systems, etc.).
- Software used to create, transmit, modify, or store electronic records that are required by regulation.
- Software used to implement electronic signatures for documents required by regulation.

Those are the types of software that the regulation requires to be validated. If a device company is using software to automate a process that is required by FDA, it is essential to show that the software accurately, reliably, and consistently meets the requirements for its intended use.

Does that mean you need to do it simply because FDA says so? At the simplest level, yes. But why is FDA so interested in how software works? FDA isn't so interested in the software itself as it is in the processes that the software is automating. FDA wants to be sure those processes are accurate, reliable, and consistent.

If FDA is interested in a company's processes, shouldn't the company also be interested? If software validation reduces the risk of a failure that could ultimately result in patient harm or jeopardize the integrity of other quality systems, then why not require software validation to reduce the risk of other, non-regulated functions? Wouldn't it be nice to reduce the risk of software failure that could disable your company's e-mail for a week, or shut down a production line for hours at a time, or delay deliveries of raw materials, or lose track of accounts receivable? Shouldn't the company be as concerned about these functions as FDA is about those that are regulated?

The point is that software validation is not just a regulatory nuisance; it is fast becoming a necessity for the device industry's increasingly software-controlled environments.

What Software Should Non-Software Engineers Validate?

Non-software engineers should be able to validate most software categorized as off-the-shelf or embedded. As its name implies, off-the-shelf software is purchased for a specific purpose, such as CAD software, compilers, or calibration-tracking software.

Embedded software (or firmware) is software that is part of a machine tool or instrument. Sometimes it may not be obvious that an instrument is designed with software embedded in the design. Certainly, instruments with graphic user interfaces are based on embedded software. Other instruments or tools with simpler user interfaces may power up with a splash display that briefly communicates the version of embedded software that is controlling the display. A large machine tool may include many microprocessor-controlled subsystems (and thus use embedded software). It may take some effort to even identify how many software items are included in some instruments and tools. PLCs can, in general, be treated like embedded software systems.

For all but the simplest custom software (software written for a specific purpose that is unique to a company), validation should probably be left to software development and validation professionals. Spreadsheets, macros, batch files, and similar items created in house for specific purposes should all be treated like custom software, but those are usually small and simple enough that they can be validated by non-software engineers.

Of course, the distinction is not always that clear. There are combinations of the above classifications. For example, many off-the-shelf software packages require custom software elements in order to do anything useful. There are also custom software systems that include some subelements that are either off-the-shelf, custom developed internally, or custom developed externally. Non-software engineers can participate in the validation of these complex systems by focusing on the system-level validation for intended use, while leaving some of the more-technical verification testing activities for the software development and validation professionals.

To understand why the type of software makes any difference in determining who might be capable of validating it, it is important to understand the following:

- The tools available to validate the software in the state it is presented.
- The assumptions the engineer can make about the state of the software when it is presented for validation.
- The objective to keep defects from getting into the software, to find defects that are already in the software, or to protect the company from defects that one simply assumes are in the software.

What is Validation, Anyway?

First, it helps to understand what validation is not:

- Validation is not synonymous with testing.
- Validation and verification are not interchangeable terms.

- Software verification and validation (SV&V) activities are not simply testing.

FDA's definition of validation is a good one: "Confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses, and that the particular requirements implemented through software can be consistently fulfilled."[1] Note the absence of the word *test* from this definition. Testing may be one of the means that is used to provide the objective evidence that the requirements implemented in software can be fulfilled, but it is not the only means, nor is it sufficient alone.

Validation comprises all activities appropriate for an engineer to come to a reasonable conclusion that a given piece of software reliably meets the requirements for its intended use. Some of those activities are verification activities. For example, for custom-developed software, verifying that each software requirement is represented in the design is a verification activity. That activity has provided an additional increment of confidence that the user's needs (as represented in the software requirements) will be implemented because it was verified that they are properly represented in the design. (This provides just partial confidence; there is still plenty that can go wrong between design and final implementation.)

Testing, too, can be a verification activity. It verifies that the documented design is properly implemented.

Many activities can contribute to the conclusion that software has been validated. Requirements management, design reviews, and defect tracking, as well as unit, integration, and system-level testing are all techniques available to software professionals during development. Many of these techniques help prevent defects from getting into the software during development. Risk management, change control, life cycle planning, system-level testing, and output verification are well within the grasp of non-software professionals. These techniques are focused on identifying any defects that are in the software, preventing defects from appearing later in the life cycle, and planning for the inevitability that defects will be discovered once the software is used.

In layman's language, validation simply gets down to answering the following questions:

- What are you counting on the software to do?
- What makes you think that the software is working?
- Can you tell when it is not working?
- What will you do about it if and when the software fails?
- What can accidentally cause the software to fail?

Rote exercising of each menu item in a software application doesn't fully address any of the above points. It does provide objective evidence. Unfortunately, it may not be objective evidence that the software meets the requirements of the intended use, or that those needs will be consistently fulfilled.

Validation Step-by-Step

For the types of software that non-software engineers can easily validate, the validation process consists of five fundamental components:

- Life cycle planning.
- Identification of requirements for intended use.
- Identification and management of risk.
- Change control.
- Testing.

Define a life cycle for the software by itemizing the phases that the software will go through

Documentation of the activities that support these components provides the evidence that the software will meet the requirements for its intended use consistently.

Life Cycle Planning. A software life cycle is a description of the phases that software goes through from the initial concept that software might be used to automate a process through the acquisition, installation,

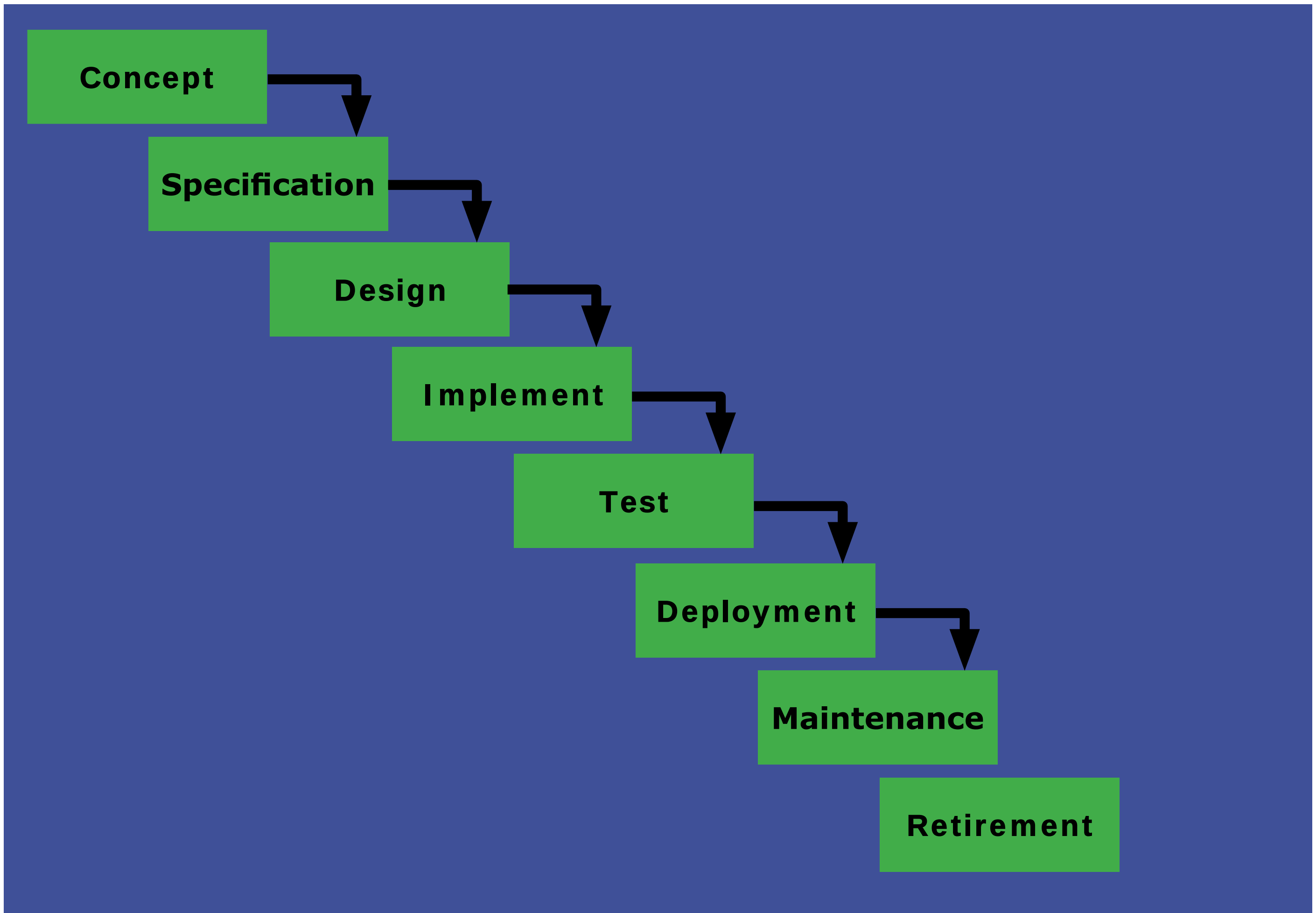


Figure 1.
Example waterfall life cycle for developed software

maintenance, and eventual retirement of the software. The actual phases may differ from category to category, or from company to company. Software techies have various software development life cycles that are widely described in the literature. Many of those life cycles do not apply to the types of software that are considered within the scope of this article, because they are mostly concerned with the development-related phases of the life cycle.

It is helpful to consider why this is important. The basic concept is to think about the software's life within the organization and to plan activities at appropriate phases that will contribute to the company's confidence that the software will meet the user needs consistently.

Remember that validation means that the software meets the requirements for its intended use consistently. Consequently, it is appropriate to review the validation components at all or at least several phases of the life cycle to ensure that the assumptions made early in the life cycle are still applicable later in the life cycle.

There is no single life cycle model that fits all types of software used to automate parts of a quality system. The life cycle of a spreadsheet is very different from the life cycle of a complaint-handling system that will be deployed in a hundred locations worldwide. Even the life cycle of a single-use spreadsheet is different from the life cycle of a spreadsheet template that could be used by a number of people. The considerations within each life cycle phase are different depending on the software item, its intended use, and its intended users. This is not boilerplate. It does take some thought, but that thought can establish the foundation of validation activities for the life of the software. This kind of activity addresses the *consistency* requirement of the definition of validation.

Define a life cycle for the software by itemizing the phases that the software will go through. For each phase, detail the activities to be performed to support the remaining validation components. This becomes the validation plan for the software. Document it. File it. Follow it.

Requirements Identification. This is not as hard as it sounds. What are the intended uses of the software? Itemize them in sentences or short paragraphs. For each intended use, define the requirements for the software to adequately meet that intended use. Use quantifiable, verifiable language to define the requirements. The following is inadequate: "The software shall control the temperature of the chamber to whatever the operator sets it to and shall get to that temperature as quickly as possible." This is more like it: "The software shall control temperature of the chamber with a resolution of 0.2°C and an accuracy of ±0.4°C. The software shall operate over a range of 37-120°C. The software shall drive the chamber to heat at a minimum rate of 10°C per minute. The software shall not allow the temperature to overshoot the set point temperature by more than 0.5°C anywhere in the operating range."

In planning requirements activities, identify the requirements early in the conceptual phases of the life cycle. Review and revise the requirements as you evaluate competing software packages.

Even in post-deployment maintenance phases, those responsible for validation should also review and revise the intended uses and requirements for the software. Later upgrades and maintenance releases of the software may introduce new features that will change the intended use (and therefore the requirements) for the software. Account for this in the life cycle validation planning to indicate the need to review requirements in the maintenance phase.

Risk Analysis & Management. Risk analysis is predicting, quantifying, evaluating, and controlling risks associated with the use of the software. Risk management is the identification and design of methods to detect software failures and to prevent, correct, or mitigate the damage caused by such failures.

The risk component of validation should be factored in at several phases of the life cycle too. In the early conceptual phase, engineers can predict what risks may be present from the use of the software. In later phases, as more is known about the software and the system or process it controls, individual failure modes may be identifiable. At all phases, those responsible for the software should consider what kinds of risk control might be put in place to reduce the risk of harm from failure of the software.

If control measures depend on the software to detect hazardous situations and to take appropriate action these requirements of the software should be targets of testing in later phases of the software life cycle.

For example, consider software to control a sterilizer. Without knowing anything about the requirements for the software, or how it is implemented, one can readily appreciate that there is a risk that a software failure might result in parts not being fully sterilized. In later life cycle phases, the analysis of risks gets more detailed, and it begins to recognize specific failure modes that might result in non-sterilized parts. The software may not run the sterilizer long enough. The sterilizer mechanism may become ineffective (blown fuses, out of sterilizing chemicals, occluded input lines, occluded drains, etc.). Will the automating software detect these situations and will it function properly in each case? If not, control measures should be identified to ensure safe operation.

If control measures depend on the software to detect hazardous situations and to take appropriate action, these requirements of the software certainly should be targets of testing in later phases of the software life cycle.

Sometimes software controls only one component of a larger process. Later operations, inspections, or cross-checks in the process may verify the output of the software-driven component of the process. This is one of the best risk control measures for software failure, and it results in solid validation of the software. The surrounding process is verifying every output of the software throughout the life cycle of the software. This is much more confidence boosting than a week of testing once in the life cycle of the software. In fact, this type of thinking, with appropriate documentation of the rationale, can even reduce the amount of testing required.

One component of risk is the likelihood that a failure can occur and result in harm. At a very high level, it is important to consider the pedigree of the software to assess the likelihood of failure. This is why custom-developed software has so many more validation activities associated with the requirements, design, and development phases of the software development life cycle. These activities provide a level of assurance that the design and development processes were conducive to producing high-quality software. If software is downloaded freeware or shareware, the pedigree is unknown, and the likelihood of failure is unknown and must be assumed to be high.

Many more checks and balances or testing should be considered for high-risk software. If software is purchased from a reputable supplier that is known to have quality software used for similar purposes and known to have a large user base, an assumption of low risk of failure can be rationalized.

Change Control and Configuration Management. At some point—prior to deployment of the software—the software item is considered to be validated for its intended use. How do you make sure the intended use, and thus the state of validation, doesn't change? That is what must be addressed in the change-control activities in the later phases of the software's life cycle. Software professionals usually refer to these activities in their configuration management plans. Configuration management also includes many other activities related to the development of software. For the types of software considered in this article, change control is the most important component of configuration management. The points to consider include the following:

- How is the validated configuration of the software item identified? Document the version, build, or time-and-date stamp of the software.
- What else is needed for the software to operate? Identify any other software that is required for the operation of the validated software item. Record the versions of any of these collateral software items. For example, if an engineer is validating a spreadsheet, it is essential to record the version of spreadsheet validated (probably the time-and-date stamp of the spreadsheet file), and to record the version information for the underlying spreadsheet application program (e.g., Excel 2003, build 11.6560.6568, service pack 2). Identify which associated hardware and operating system version levels were part of the validated configuration.
- Who is responsible for determining when the software can change? This is *change control*. How will changes to the software be controlled? Someone should be identified as responsible for deciding when the software changes, and for revalidating the software after it changes.
- What should be done to revalidate the software when a change is made? Revalidating means more than retesting. Requirements and risks need to be reevaluated to be sure they haven't changed with any new features or other changes to the software. Maintenance-phase changes to the software should be

viewed as their own mini life cycles, as almost all validation activities of each life cycle phase should be reviewed, revised, and supplemented to adequately validate the new software.

Testing. Testing is really a risk control measure. Risk combines the severity of harm resulting from a failure with the likelihood of the failure. Testing can reduce the likelihood of failure, thus reducing risk. The level of reduction, of course, depends on the quality of the testing. Furthermore, because the likelihood of failure is unknown before the test, and because an engineer likely does not have a good quantitative measure of how much testing reduces the likelihood of failure, it leaves an engineer with little to measure how much the testing has lowered the risk. All that is known is that some testing is probably better than no testing; and more testing is probably better than less testing.

So what can be done to increase the value of testing? First of all, use all that great thought that went into risk analyses and risk management plans. If a company has risk controls in place to prevent, detect, correct, or mitigate failures in the process that is automated by software, it is imperative to test them. Be sure they really do prevent, detect, correct, or mitigate. FDA's GPSV guidance repeatedly calls for validation effort commensurate with complexity and risk. Focusing testing on making sure risk control measures are effective is perhaps the best use of a test budget that is commensurate with risk.

Next, focus test efforts on areas of complexity because that's where defects are likely to be found. Look for complex error conditions to make sure the software deals with them properly. For example, in many software-driven instruments, power failure and recovery handling are often fruitful areas of testing simply because they are often implemented as afterthoughts. The conditions are complex, difficult to predict, and difficult to simulate. On the production floor, however, power failure is a fact of life. Machines can destroy valuable product or simply self-destruct because the software designers didn't anticipate the software starting with the machine in an unexpected state. Similarly, user error or intentional misuse of the software is often not predicted by the software developer and consequently may not be handled properly by the software.

Check for conditions that could cause problems such as pressing two buttons at the same time, stuck inputs, out-of-range input values. Perform operations in different sequences to ensure that the software functions properly in each case. Testing functionality in which defects are suspected (i.e., error guessing) is testing budget well spent. Conversely, exercising menu commands (which probably have been exercised missions of times by other users) seldom yields new defects. The best test is one that finds a new defect.

Special Situations: 100% Verifiable Output

In certain situations, the output of a software-driven machine tool or software-driven process may be 100% verifiable. For example, consider a software-driven production instrument that crimps connectors onto a wire lead. The pull strength and conductivity of the lead are tested by a quality control (QC) test on every lead that is produced by the machine. In this case, the output of the software-driven machine is 100% verified by the QC tests on every lead ever produced. This is a much better validation of the output of the machine than any software testing executed at a snapshot in time would ever produce.

Does the software still need to be validated? The answer is yes. Again, validation is not synonymous with testing. The analysis that would lead one to ask this question is, in fact, a validation activity. To ask the question implies that one has evaluated the intended use and has combined that with a risk management plan to check the machine output for the safety-critical attributes of pull strength and conductivity. Intended use and risk management are validation activities.

Now consider how changes to the software are controlled, and how the validation state of the software would need to be reevaluated when the software changes. Again, this change control or configuration management is a validation activity.

Testing in this example may be greatly reduced. If it is concluded that any possible software malfunction that could affect product quality would be detected in the QC test, then software testing for those malfunctions can be greatly reduced or eliminated. Some testing may still be recommended for operator safety functions (such as emergency stops and safety interlocks), security functions, power fail and recovery functions, etc.

Note that the testing is focused on functions related to intended use and safety, not on trying to reverse engineer the detailed software requirements that are then verified in numerous and lengthy tests. The validation is the collection of all of the activities that lead to the conclusion that the software is fit for use. Documentation of the activities, the resulting logic, and any test results becomes the validation package. Take credit for it if you do it by documenting it.

Conclusion

Keep two key points in mind. First, and engineer does not need to be a software guru to validate some types of software for their intended uses. Second, software validation is not synonymous with software testing. Software validation is thinking rationally and systematically about the use of the software throughout its life cycle. Validation is establishing controls for ensuring the correct operation, detection capabilities for improper operation, backup plans for what happens if the software fails, and yes, some testing to ensure that the software and the backup plans perform as desired.

Reference

[1] "General Principles of Software Validation: Final Guidance for Industry and FDA Staff" (Rockville, MD: FDA, 2002).

ABOUT THE AUTHOR:

David Vogel is the founder and president of Intertech Engineering Associates, Inc.

Dr. Vogel was a participant in a joint AAMI/FDA workgroup to develop a standard for Critical Device Software Validation which was subsequently included in the IEC 62304 Software Lifecycle Standard. He was also a participant on the joint AAMI/FDA workgroup to develop a Technical Information Report (TIR) for Medical Device Software Risk Management. Currently, Dr. Vogel is a member of the AAMI/FDA workgroup developing a TIR on Quality System Software Validation.

A frequent lecturer for workshops and seminars on topics related to medical device development and validation, Dr. Vogel also is the author of numerous publications and holds several patents.

Dr. Vogel received a bachelor's degree in electrical engineering from Massachusetts Institute of Technology. He earned a master's degree in biomedical engineering, a master's degree in electrical and computer engineering, and a doctorate in biomedical engineering from the University of Michigan.

Intertech Service Offerings:

Risk Analysis and Management
Software Design and Development
Electronic Design and Development
Requirements Development and Management
Documentation and Traceability
Verification and Validation
Evaluations, Reviews, Inspections
Planning
Project Management
Compliance Consulting and Training
Manufacturing and Quality System Software Validation

Leverage INTERTECH's expertise to:

Reduce Project Risk
Shorten Time to Market
Cut Development and Test Cost
Assure Quality Products

ABOUT INTERTECH:

Intertech Engineering Associates has been helping medical device manufacturers bring their products to market since 1982. Through a distinct top-down service model, Intertech offers high-level consulting and hands-on engineering. By balancing technical expertise and practical business experience, we support clients through all phases of product development. While we do make your job easier, Intertech exists not to replace but to partner with clients to help balance the concerns of quality, time and cost.

With considerable experience in FDA regulatory compliance, our time-tested development process can anticipate and solve problems inexpensively on the planning board rather than through costly solutions later in the development, test, or post-deployment phases. By using deliberate processes, Intertech ensures an improvement in quality and can build client expertise.

Call us today for more information or a free consultation at 781.801.1100

INTERTECH Engineering Associates, Inc.
100 Lowder Brook Drive Suite 2500 Westwood, MA 02090 USA

www.inea.com - info@inea.com - Tel: (781) 801-1100 - Fax: (781) 801-1108