An Approach to Good Practices for Design and Development of Software Medical Devices

Matthew Rupert

12/2010

Draft

Here's a dirty little secret:

You know those standard operating procedures that took so long to create and that you forced everyone to read? Of course you do! And assuming you have loyal employees who do what they are asked, the procedures were probably read, boring as they may be. That's the good news. The bad news is that nobody (including the author) remembers exactly what those procedures say.

My first experience working on a software medical device came in 2002. My previous employer, an exciting venture of the dot-com era, failed to provide me the millions of dollars that I expected. Still somewhat fresh out of college, I was hit by reality: Those 100,000 shares of stock weren't going to be worth anything, and I would continue driving a 1998 Dodge Status for a few more years.

Faced with my first humbling experience in the unemployment line, I was eager to accept the first job that came along, and soon enough I found myself working on a blood bank project performing software quality assurance. I'm not sure what was more humbling to an arrogant young software developer: The unemployment line, or being forced to write test scripts. But it was a job, and being one of thousands of software developers looking for work, I took it.

On day one at my new job I learned something bizarre: The software we wrote would be audited and controlled by a set of standards defined by the FDA. "The FDA," I asked, "as in The Food and Drug Administration?" Yep. That FDA. I was told to sit in my cubicle and read something called the CFR, focusing specifically on parts 11 and 820. It was long and boring and strange sounding, and I was annoyed. I yawned incessantly as I forced my way through it.

My next task was to read a bunch of Standard Operating Procedures (SOPs). Upon completion of reading an SOP I took a test answering a few simple questions about the SOP. It was graded and signed and placed in a permanent file somewhere as proof that I knew the SOP. At this point I promptly forgot what the SOP said.

My job was simple. I was to review use cases, write test scripts for those use cases (both manual and automated) and run the scripts. When I finished one task I moved on to the next (that task being

whatever my boss told me to work on). Soon enough the SOPs I had read during my first few weeks of employment were a distant memory. I had no reason to revisit them.

But perhaps I should have. Remember how I signed a test saying that I read and understood the SOP? About 6 months into my employment, it came time for an internal audit of the project. The purpose of this audit was to perform activities similar to those which might occur during an actual FDA audit for a 510k. I was pulled into a room with the auditors and interrogated.

The auditor asked me the first question, “So Matt, when you do your were, how do you know what to do?”

The question seemed odd, but I responded politely, “Uh, well, I guess I do whatever my boss tells me to do.”

Wrong answer.

The auditor continued, “No, what I mean is, how do you know what you are supposed to create?”

I began to squirm a bit, “Um, because my boss tells me what to do… I read the use case and requirements, and write the test.”

This was not going well.

The bizarre questioning continued for an uncomfortably long time, until finally the auditor gave up and told me that I could return to my cubicle. For a brief period of time, I was relieved. Then the project manager caught wind of my inarticulate question and answer session.

I was in trouble.

WHAT WENT WRONG?

So what went wrong during this audit? The obvious and most simple answer is that I wasn’t prepared. Yet, even as I reflect, to this day I’m not entirely sure what the auditor wanted to know. The project manager was irritated with me for not understanding the questioning, yes, but what was the problem here? Why did the project manager assume I didn’t know *how* to do whatever my boss told me to do?

The real problem, of course, was our process. As my bleary eyes could attest, we had stacks of procedures in place. We had the training. Someone had worked very hard to come up with all the “stuff” needed to state how this project would be designed and developed per the requirements of the CFR.

But there was a big piece missing: **Application** of the procedures.

I wasn’t necessarily doing anything wrong. I was just doing whatever my boss told me to do. That’s a good thing, right? And my boss was doing whatever her boss told her to do. Also good, right?

Well, not really. None of us were doing what our very own Standard Operating Procedures, written by us, for us, told us to do. The auditors dinged us for not following our own system! We knew about the CFR, and we knew procedures had to be created in order to satisfy it. But we weren't thinking in very practical terms.

APPLICATION OF STANDARD OPERATING PROCEDURES

21 CFR 820 covers quality system regulation for medical devices (and software medical devices). Also known as the Quality System Regulation outlines Current Good Manufacturing Practice regulations that govern design and development of a software medical device (but you already knew that). It tells us a lot about the controls we need to implement as a part of our quality system, but it doesn't give us many concrete examples or specifics on how to best apply it.

Whether or not you decide you actually need standard operating procedures, i.e., a number of procedures that reside somewhere in some files named "such and such an SOP," is a decision that is up to the organization. Regardless, it is necessary to have certain procedures laid out that explain how all these quality system requirements are actually performed. To that end, I can think of two ways to write standard operating procedures, each of which has its place (and for the purposes of this article, I am writing only in terms of software development projects).

The first approach is to tailor a number of procedures in a way that is specific to the project. In this way we can handle changing technologies and environments. Over time, as things change, it will undoubtedly be necessary to revisit all of the SOPs. An even bigger problem is that this approach doesn't lend itself to a "one size fits all" set to SOPs, and they become a procedural and documentation nightmare. This approach is fine for a small environment with only a few concurrent software projects with similar environments, but it won't scale well for more extensive corporate needs.

A more practical approach, in my humble opinion, is to create SOPs so that they are environment and technology agnostic. This way, at least for all of the software medical devices that are being designed and developed, we do have a set of procedures that make sense and remain relevant for more general and longer lasting usage. This leads to a different approach to handling software project management. These high-level SOPs, of course, are not very pragmatic on their own (and this is a good thing!). They need another layer so that they may be applied in some practical way.

In this second approach, we no longer technically constrained by the procedures that we worked so hard to put in place. No longer do we lean toward using a single version control system, database, ticketing system or tool simply because it would be too difficult to refine our SOPs. Our SOPs no longer state what we must use; they simply state what we must do. If any SOP ever prevents efficient software design and development by prohibiting the ability to utilize changing technologies and practices, it is a bad SOP.

Over the years I've worked with many good developers who view "process" as a dirty word. I don't blame them. All too often we let our processes become so cumbersome that they don't serve any real

purpose other than to frustrate those who are forced to work within their confines. In such a situation, the processes are sidestepped, rendering them useless.

So how do we make a good process? How do we make use of SOPs in a way that serves rather than restricts productivity and good design?

The answer is this: We let our standard operating procedures serve as guidelines and we create work instructions on a per-project level that explain the actual (and practical) use of those guidelines. This is done (at least as comprehensively as possible) during our project planning. It is at the project level that we state which corporate-wide SOPs apply to this project and *how* we will apply them. This means that for a given SOP we must have one or several work instructions explaining how we implement them, what applications are used, how to use those applications and when to use them. Where a design control SOP states "versioning control is used for source code," our project plan (or configuration management plan) states "Subversion is used for source code version control. The repository for project X resides at…"

With so much involved, hopefully it is clear why the project plan should include detailed work instructions to help us make sense and real application of our procedures.

Remember my story about being interviewed during the audit? Sure, I had read the SOPs, but I didn't really know how they applied. This is because the SOPs (in this case) provided a high level of design control, but the practical application lived only in the heads of those who happened to be using the tools for the project. And even when, by luck, and SOP was followed, it was only because the approach being taken happened to be in line with a general guidance and not with the specifics of any work instruction. Those SOPs provided little more than lip service to the CFR.

It all comes down to this: A standard operating procedure is useless if it has no concrete application. Without an understandable and usable means of application, an SOP is just an annoying piece of paper that will cause headaches down the road.

Our work instructions point us to the specifics on how and when to use Redmine, Subversion and Hudson (continuous integration builds, later in this article). Essentially, the standard operating procedures can be thought of as a framework, and the work instructions the implementation. And these work instructions should allow us to use the technology available to make our processes not only applicable, but effective and helpful. We mustn't think of these procedures as a roadblock to our work—On the contrary, they should make our work more efficient. If they do not, we are going about things in the wrong way.

ONE APPROACH

After that long introduction, my desire is to write about one approach to handling design and development activities of a software project. This is an approach that works well for any software

project, not just one in which a 510K is being pursued. Given the fact that the biggest needs are for good communication and tracing, perhaps we shouldn't be surprised that the open source community has taught us a thing or two about team collaboration and communication on software project. It is this community that has learned a great deal about pulling together work from a community across the global to create high quality software.

When it comes to a regulated project we are very concerned with traceability. We want to know that use cases, business rules, requirements, documents, code revisions, hazards and tests can be traced backward and forward. Despite having some excellent technology available (for free!), many companies insist on manual documentation of such tracing. This creates a mess of documentation that is difficult (if not impossible) to maintain and likely to contain errors. You're just begging an auditor to find a problem! If it is your full time job to edit and verify these documents, you may want to stop reading now.

Here goes: You don't need to hire someone to work full time to shuffle through this mess of documentation! Sure, you may need a technical writer, but there is plenty of technology available to make this aspect of design and development easy. There are tools out there already that, when used properly, can make software project management from the highest level to the most detailed tracing downright fun.

WHY ISN'T EVERYONE DOING IT THIS WAY THEN?

1. There is a perception that using 3rd party tools is difficult within the confines of the CFR.

   "But how do we validate all this software?" you ask, "The CFR requires us to validate all the tools we use." Show me where. What the CFR states is that we must show intended usage of our software and show that the tools we use work the way we think they should work for our needs. Simple.

   The CFR doesn't exist so that we may engineer software poorly. Nor does it exist to tie our hands from using the best new tools available. It exists to tell us how to do it right.

2. Open source software cannot be trusted.

   This objection can only be addressed in a separate article! All I can say to this is that we have learned repeatedly that widely-used open source software is often better and just as well supported as its closed-source counterpart. One need only look as far as GNU for proof (www.gnu.org).

3. Technologies change. How do I know Subversion will be used in 10 years?

   You don't, and it may not be. But for everyone out there using other legacy systems (Visual Sourcesafe anyone?), we can't let this be a concern. The technology may change, but our servers can be maintained and archived for as long as necessary.

4. We don't have the time for such overhead or IT support.

   At the risk of sounding cliché, you don't have time to not do this. A little setup and thought up front streamlines the process and mitigates the risk of serious problems down the road.

   If you ever find yourself wondering how to update tracing long after the completion of a software requirement, document change or test, your procedures have failed you. And if you don't have a practical approach to applying your procedures, this WILL happen at some point in the project. Sure, there is some upfront cost to be considered, but it comes with greater efficiency throughout the project lifecycle.

THE TOOLS

When it comes to a setup such as the one I am about to describe, there is no "one size fits all" approach. The tools used must be evaluated with consideration to the environment, project and corporate needs. So while the tools I am about to list will no doubt work for a wide range of needs, there are many other tools that are worthy alternatives (Trac, Git, Mercurial, etc.).

The main tools in my example are:

- Subversion – http://subversion.tigris.org/ - Version control (of everything!)
- Redmine – http://www.redmine.org/ - Ticketing and issue tracking system
- Hudson – http://hudson-ci.org/ - Continuous Integration builds

Other tools that I will mention include:

- TortoiseSVN - http://tortoisesvn.tigris.org/
- PMD - http://pmd.sourceforge.net/
- Cobertura - http://cobertura.sourceforge.net/
- Findbugs - http://findbugs.sourceforge.net/

The tools I've listed above are ones that I am very familiar with and like. They serve the needs of a software project (regulated or non-regulated) very well when integrated, they are widely used and supported and they integrate very nicely. Best of all, being open-source, they are all free (and written by some of the best software developers in the world). For use in the typical corporate environment, each tool can use LDAP authentication. As far as environment restrictions go, each of the tools can be run on Windows, Linux, Solaris and MAC OS X.

VERSION CONTROL

The earliest phase of any software project is the planning phase. At this stage, people involved with the project have meetings and discuss some very high level needs. There are probably some presentations and documents that are created. Project management plans have not been developed, but they should be thought about. And as I stated previously, we begin creation of our work instructions (the application of our SOPs) in this stage.

The design history file (DHF) of a project must contain all of the historical "stuff" that goes into the project, so even at this early stage it is necessary to decided on a version control system and create a repository. Sure, there may be no tracing involved yet, but this early "stuff" should still be kept around In the DHF. Because the earliest phases of the software project result in outputs that are to be included in the DHF, it is necessary determine the version control tool and establish the version control repository early on (for the sake of this article, I assume a basic understanding of version control/revision control systems).

PROJECT TRACING

Once again: Tracing. Tracing is everything, and Subversion, with its changesets, lends itself extremely well to integration with other tools used throughout the project. When used with your issue tracking software every issue can be linked directly with a set of items in the repository that are related to addressing and resolving that issue. With a click of the mouse we can see a list of all the project file modifications related to a single issue.

I recommend using a single version control system and repository for all of the "stuff" that goes into a project. This means that project management plans, documents, presentations, code, test data and results should all go into the same repository for a project. If documents are stored in one repository (or in a different version control system altogether) and software code is stored in a different repository, we lose much of the project traceability that we could have.

(Note: When placing binaries in a version control system there is no merge path as there is with text file source code. This means it is good practice for team members, when editing documents, to place a strict lock on the file while editing. This can be done in Subversion. Strict file locking allows others to be notified that another user is currently working on a file.)

While a clear benefit of this approach is the fact that all of the "stuff" of a project is associated with the same repository, some may view this as a problem with the setup. I suggest this setup only because I am thinking specifically in terms of an FDA regulated software product in which it is beneficial to relate all elements of the project in a single traceable repository. In this setup documentation will be versioned (and tagged) along with project source code, and this may or may not be desirable depending on project needs.

Subversion is superior to many of its predecessors because of its introduction to "changesets." A changeset provides a snapshot in time of the entire project repository. When documents, presentations or source code are changed and committed to the repository, a new changeset number is created. Now at any time in the future we can checkout all items in the repository tree as of that changeset. When asked what version of a product something was changed in we can pull everything relevant to the project at the point of that change. No longer do we have to tag or label our repository in order to revisit a particular instance in time (although Subversion still allows tagging). Every single commit to the repository effectively results in a "tag."

This is not to say that tagging is no longer useful. On the contrary, it remains very useful. All software releases, included internal releases, should be tagged (and our work instructions should tell us when and how to perform tagging).

Another advantage of Subversion is that, unlike some of its predecessors, it allows for the control and history of directories as well as files (including file and directory name changes). The most commonly used predecessor to Subversion, CVS, did not maintain a version history of a file or directory was renamed. Subversion can handle the renaming of any version controlled object.

RELEASING SOFTWARE

When software is released it is typically given some kind of version number (e.g., 1.0). This is good, but it doesn't tell us the specifics of what went into that build. It's a good idea to include the Subversion changeset number somewhere in the release so that we always know EXACTLY what went into the build. I would include a build.xml (or build.prop) file somewhere that includes the version number of the release, the Subversion changeset number and the date of the build. As far as the last two values, these can (and should) be generated automatically by your build scripts.

As far as actually using Subversion, within Linux/Unix all commands are available from the command line. When working in Windows, I really like using TortoiseSVN. It integrates with Windows File Explorer, showing icons that indicate the status of any versioned file. It also provides a nice interface for viewing file differences (even differences of your Word documents!) and repository history.

TICKETING AND ISSUE TRACKING

For too long we've thought of our ticketing systems as "bug trackers." One of the previously most popular open source tools, Bugzilla, even used the word "bug" in its name. But issue tracking does not need to imply that it is only useful for tracking software defects. On the contrary! It can be used for everything in software design and development, from addressing documentation needs to capturing software requirements to handling software defect reporting.

(On this note, I would like to add something that may require an entire article. I think it might be best to get away from using standard documents for the capture of software use cases, requirements, hazards and so on. By capturing everything related to a software project in our issue tracking tool we can leverage the power of a tool like Trac or Redmine to enhance team collaboration and project tracing. But I won't bite off more than I can chew right now.)

When I first began writing all of these thoughts of mine down I was going to use Trac as my example (http://trac.edgewall.org/). Trac is a great tool, but there's something better now, and that something is Redmine (http://www.redmine.org/).

The principal shortfall of Trac is the fact that it doesn't lend itself well (at all) to handling multiple projects. One installation of Trac can be integrated with only single Subversion repository, and the ticketing system can only handle a single project. I still like the way Trac can be used to group tickets into sprints, but using subprojects in Redmine, a similar grouping can be achieved.

In the past, if a tool was used at all, we used Bugzilla or Clearquest to handle our issue tracking. These tools were very good at the time, but they did not integrate well with other tools, nor did they include features like wikis, calendars or Gantt charts. (Admittedly, I have not used Clearquest in many years, so I really have no idea whether or not it has since addressed some of these needs.)

So what's so great about Redmine?

1. The power of the wiki.

   Your documents have all of your project management details, work instructions, use cases, requirements and so on. Again, I think this information can be placed into the wiki, but that may be a step that not everyone is comfortable taking.

   That said, all developer setup, lessons learned and other informal notes can be placed in the wiki. One time I spent nearly 4 days tracking down a very strange defect. By the time I finally figured it all out I had learned a lot about a very strange issue that others were surely to encounter. I created a wiki page explaining the issue.

   Another power feature of the wiki is the fact that with Redmine (and Trac) we can link not only to other wiki pages, but to tickets (issues), projects, sub-projects and Subversion changesets. Again, more tracing. Nice.

2. Subversion Integration

   With Subversion and Redmine integrated I can link back and forth between the two. Those work instructions explaining to the team how we will make use of our procedures should explain that no ticket can be closed without a link to a Subversion changeset (unless, of course, the ticket is rejected).

   Redmine can be configured to search for keywords in your Subversion changeset commit. For example, if I am checking in several files that address issue #501, I might put a comment like this:

   "Corrected such and such. This fixes #501."

   We can configure Redmine to look for that word "fixes" and use it. Redmine may use that word as a flag to close the ticket and link to the changeset that was created when I did that commit. Likewise, when we view your Subversion history, we will see "#501" attached to the changeset as a link to the ticket. The tracing works both ways… Beautiful!

3. Multiple project handling (and integration with different Subversion repositories)

This is major reason why I (and others) switched to Redmine. Trac was great, but it only handled a single project. Redmine, with its handling of multiple projects, and be used corporate-wide for all development, and each project can be tied to a different Subversion repository.

Additionally, a single project can have multiple sub-projects. This gives us the flexibility to use sub-projects for sprints, specific branched versions and so on.

4. Hudson Integration

   With Hudson integrated I don't have to leave the wiki to see how my CI builds look. Not only that, I can link to a specific CI build from any page within the wiki or ticketing system.

5. Full configurability

   Everything can be configured in Redmine. Yes, EVERYTHING. We can even configure the flow of tickets.

USE THE ISSUE TRACKING SYSTEM FOR ALL PROJECT DESIGN AND DEVELOPMENT

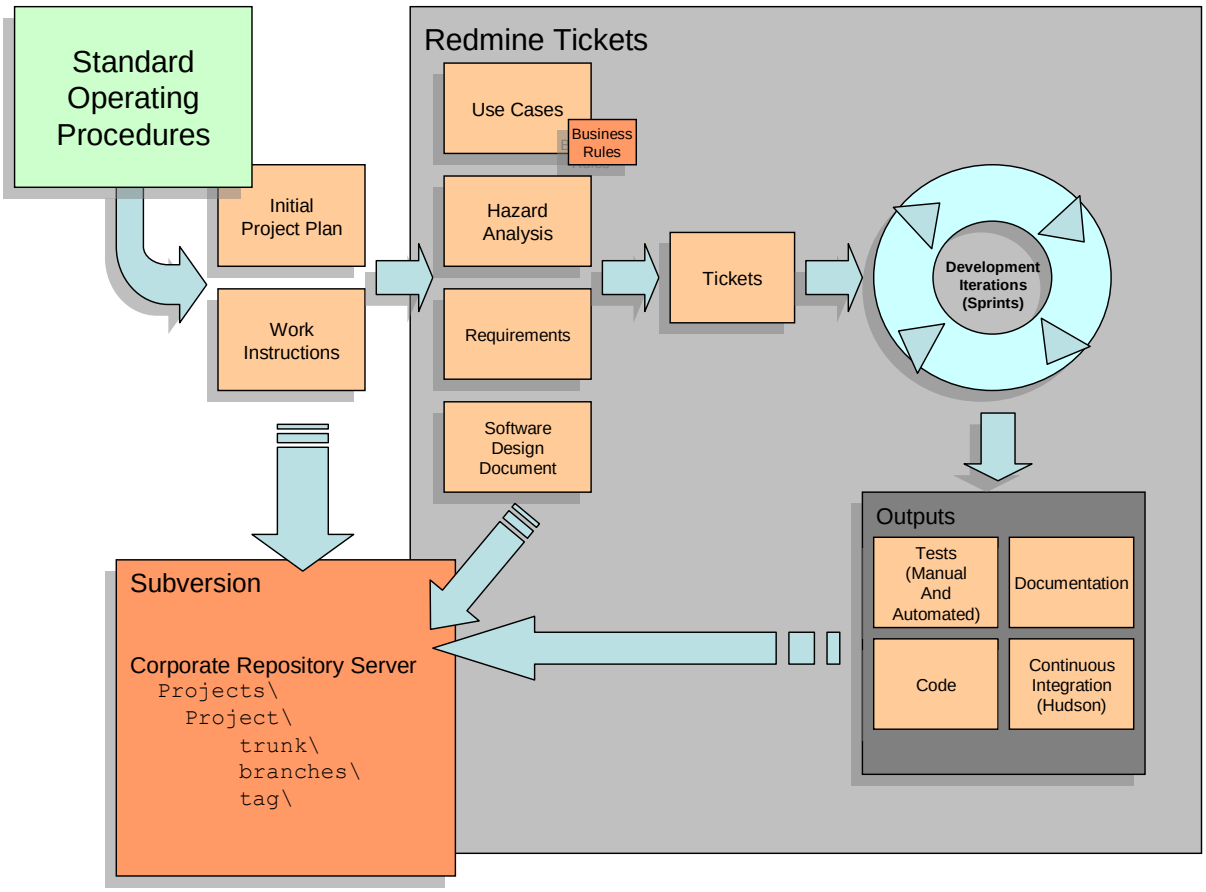
I propose that it is not enough to simply leave functional requirements in the software requirements specification document. This does not provide sufficient tracing, nor does it provide a clear path from idea to functional code. Here are the steps that I suggest:

1. All requirements and software design items are entered as tickets. For now they are simply high level tickets with no "child tickets."
2. The development team, organized by a lead developer, breaks down each high level ticket into as many child tickets as necessary. Using the ticketing system, we set up relationships so that the parent ticket (the requirement itself) cannot be closed until all child tickets are completed. (Note: It may be a good idea to require corresponding unit tests with each ticket.)
3. Hazards (and I'm not doing to bother an explanation of hazard and risk analysis in this article) are mitigated by a combination of documentation, requirements and tests. We can leverage our ticketing system to capture our hazards and provide tracing in much the same way as with requirements. This does not remove the need for a traceability matrix, but it does enhance our ability to create and maintain it. (As a side note, I think it would be great to use the Redmine wiki for use cases, requirements, hazard analysis, software design documents and traceability matrices, thereby allowing for linking within, but this may be a hard sell for now).

4. Not all requirements are functional code requirements. Many are documentation and/or quality requirements. These should be capture in the same ticketing system. Use the ability of the system to label the *type* of a ticket to handle the fact that there are different categories. By doing this, even documentation requirements are traceable in our system.

I'm not suggesting that the tickets will be locked down this early. Not for a moment! Tickets are created, closed and modified throughout project design and development process. Our project plan (created before we started writing code) explains to us which tickets need to be done when, focusing more on the highest level tickets. That said, I find it best to use some sort of iterative approach (and allow the development team to use sub-iterations, or "sprints").

The diagram below shows how everything may be incorporated into the ticketing system throughout all stages of software development. I propose that any activity that results in a design output should first be captured as a ticket. (My goal here is not to describe process, rather, it is to illustrate the fact that our ticketing system can and should be used for all project activities. Whatever your software design and development process, RUP, Agile, traditional waterfall, etc., the ticketing system can be used throughtout.)



CONTINUOUS INTEGRATION BUILDS

To answer the first question, before it is even asked: Yes, a software project should have continuous integration (CI) builds. This goes for projects with a large team as well as projects with a small team. While the CI build is very useful for a large group to collaborate there is tremendous value even for the smallest team. Even a software engineer working alone can benefit from a continuous integration build.

What's so great about CI builds (using Hudson)?

1. A build can be traced to a Subversion changeset, and therefore to our ticketing system (and entire process).

   Okay, so I'm starting to sound repetitive, but tracing is a good thing, and with this setup I can trace all over the place! With all of the process in place, from our standard operating procedures to work instructions to use cases and requirements to tickets to changests, how

do we know that the team members working on a project are actually following procedure? The CI environment, at least with regard to the ongoing development of code, gives us a single point of overview for all other activities. From here we can see changesets, tickets, build status and test coverage. Using the proper add-ons, we can even gain insight into the quality of the code that is being developed.

To pull this of, of course, we need to use our ticketing system wisely. With Redmine, and just about any good ticketing system, we can capture elements of software requirements and software design as parent tickets. These parent tickets have one to many sub-tickets, which themselves can have sub-tickets. A parent ticket may not be closed or marked complete until a child ticket is completed.

2. Immediate feedback

   At its most basic level, Hudson only does one thing: It runs whatever scripts we tell it to. The power of Hudson is that we can tell it to run whatever we want, log the outcome, keep build artifacts, run third party evaluation tools and report on results.

   With Subversion integration, Hudson will display the changeset relevant to a particular build. It can be configured to generate a build at whatever interval we want (nightly, hourly, every time there is a code commit, etc.).

   Personally, every time I do any code commit of significance, one of the first things I do is check the CI build for success. If I've broken the build I get to work on correcting the problem (and if I cannot correct the problem quickly, I roll my changeset out so that the CI build continues to work until I've fixed the issue).

   Hudson can be configured to email team members on build results, and it may be useful to set it up so that developers are emailed should a build break.

3. A central location for builds

   The build artifacts of your project, in general, should not be a part of the repository (there are exceptions to this rule). Build artifacts belong in your continue integration build tool, where they can be viewed and used by anyone on the team who needs them. These artifacts include test results, compiled libraries and executables.

   Too often a released build is created locally on some developer's machine. This is a serious problem, because we have no good way of knowing what files were actually used to create that build. Was there a configuration change? A bug introduced? An incorrect file version?

While developers have good reason to generate and test build locally, a formal build used for testing (or, more importantly, release) much never be created locally. Never.

Build artifacts are not checked in to the source control repository for a number of reasons, but the biggest reason is because we never want to make assumptions about the environment in which those items were build. The build artifacts should instead remain in our CI environment, where we know the conditions within which the build was generated.

Also, because these builds remain easily accessible and labeled in the CI build environment, any team member can easily access any given build. In particular, it may become necessary to use a specific build to recreate an issue in a build that has been released for internal or external use. Because we know the label of the build (the version number given to it) as well as the repository changeset number of the build (because our build and install scripts include it), we know precisely which build to pull from the CI build server to recreate the necessary conditions.

4. Unit tests are run over and over (and over)

Developers should do whatever they can to keep the CI build from breaking. Of course, this doesn't always work well. I've broken the CI build countless times for a number of reasons:

- I forgot to add a necessary library or new file
- I forgot to commit a configuration change
- I accidentally included a change in my changeset
- It worked locally but not when built on the CI build server (this is why the CI build server should, as much as possible, mimic the production environment)
- Unit tests worked locally but no on the CI build server because of some environmental difference

If not for a CI build environment with good unit tests, such problems would only be discovered at a later time or by some other frustrated team member. In this regard, the CI build saves us from many headaches.

The most difficult software defects to fix (much less, find) are the ones that do not happen consistently. Database locking issues, memory issues and race conditions can result in such defects. These defects are serious, but if we never detect them how can we fix them?

It's a good idea to have unit tests that go above and beyond what we traditionally think of as "unit tests," and go several steps further, automating functional testing). This is another one of those areas where team members often (incorrectly) feel that there is not sufficient time to do all the work.

With Hudson running all of these tests ever time a build is performed we sometimes encounter situations where a test fails suddenly for no apparent reason. It worked before, an hour ago, and there has not been any change to the code, so what did it fail this time? Pay attention, because this WILL happen.

5. Integration with other nice-to-haves, such as Findbugs, PMD and Cobertura

   Without going into too much detail, there are great tools out there that can be used with Hudson to evaluate the code for potential bugs, bad coding practices and testing coverage. These tools definitely come in handy. Use them.

   (Someone recently asked me what PMD is an acronym for. Considering the fact that I was recommending its use, you would think I would know the answer. It turns out I didn't know the answer because it isn't an acronym for anything.)

AN EXAMPLE

I'll end with a simple example. We've kicked off a project, reviewed the SOPs that will guide the project and created work instructions to apply those SOPs to our project. We've set up our repository (Subversion), our ticketing system (Redmine) and our CI environment (Hudson) and linked them all together. We've completed and captured our use cases, business requirements, software requirements, software design document and initial hazard analysis. Now what?

Let's start with an example use case. This use case, along with all project use cases, resides in our repository (of course).

```
Use Case 1: User authentication and login
Project: Project X
Overview: This use case describes the authentication and login
requirements for users of System X.
Business Rules:
…
Scenario:
…
Requirements:
…
Risks and Hazards:
…
```

From this use case we derived a number of requirements (and perhaps a few hazards were identified). Here's a single requirement (pulled from our requirements document) that may have come from this use case. (Note: If we name our requirements in an intelligent way, such that the requirement ID includes the use case number from which it came, the tracing from requirement to use case is obvious.)

Requirement ID	Description
1.0.1.2	User authentication shall be required for all users of the system.

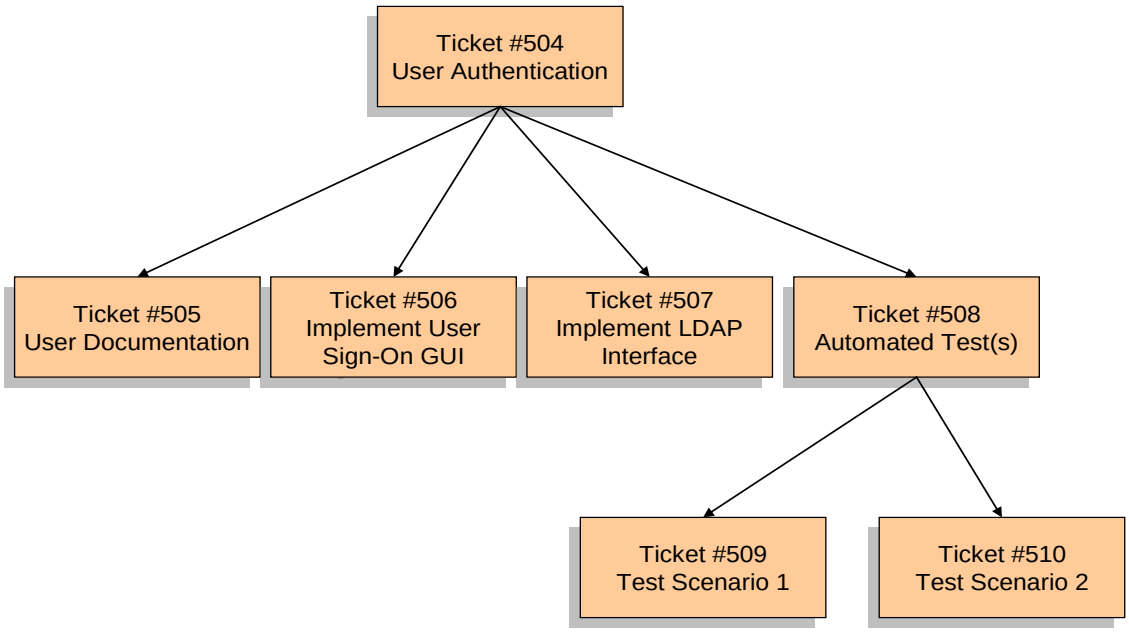
Now that's a fairly bad requirement alone, in that it really doesn't say much about how authentication is handled, but we're just dealing with examples here.

Next, a project may (it should) have a software design document. This document is where more details specific to the software design are offered. This document is written by technical people for technical people, explaining design patterns, database considerations, libraries and other specific technical requirements. This software design document may have specific line items regarding authentication implementation (realms, LDAP, etc.).

To capture the requirement in our ticketing system we create a new ticket. This ticket will serve as the "parent" to all other implementation details related to it.

Ticket #	Created By	Description
504	Joe Project Manager	User authentication shall be required for all users of the system.
Updates:		
Priority:	Moderate	
Due Date:	12/21/2012	
Assigned To:	Joe Project Manager	
Dependencies:	505, 506, 507, 508	

This ticket alone doesn't have much meaning. But when we look at its dependencies, we see that there is work to be done. Perhaps ticket #505 is some user documentation need, and #506 has to do with LDAP. Using the ability of our ticketing system to create dependencies we can capture all of our requirements, track their relationship to implementation (including design, development and documentation), organize the flow of work and trace the entire process. Ticket # 405, our requirement, cannot be closed until all the dependencies are addressed.



Of course there isn't anything magical about using an issue tracking system in this way. We've all been doing this for years. What is unique to this approach is the fact that we are taking the issue tracing a step further by including user requirements, using a single repository, and tracing through out continuous integration. No longer must we treat the documentation, development, design and testing needs as a part of separate systems. By using our ticketing system to handle all project design and development outputs, we maintain a history record of the entire process and of the relationships throughout. Because of this, we have simplified what was previously a mess of project documentation maintenance. Our traceability matrix is no longer some separate element that must be created and validated after the fact—It is an integral part of our process.

Another great benefit of capture requirements as tickets comes in the form of continuous testing. If you are fortunate enough to have a dedicated tester or testers, this approach provides the QA team the resources needed to test the product not just at the end of a development phase, but throughout. I recall one project I worked on very early in my career in which I became quite annoyed by the QA team. Every day it seemed they would write new tickets in our system, flagging them as "defects," indicating that some feature of our system was not working. That feature was not working because it was not yet developed! Had the requirements been included in our ticketing system from the highest to lowest level of functional need, such time wasters would have never been introduced.

Another important aspect to having tickets organized in such a manner is that having a large number of detailed tickets leads to a much more granular view into project and requirement completeness at any given time. It is much easier for team members to estimate progress when tickets are broken down in such a way that they are manageable and easily accomplished in a reasonable amount of time. Likewise, progress reports to management have much greater meaning.

Finally, and this may be the greatest benefit, by organizing our tickets in this way we achieve something slightly less obvious but at least as important: A more rewarding work environment. Taking on a single very large task can seem daunting, if not impossible. It is difficult to know where or how to start, and, as a developer, it can be difficult to get motivated to work on something that doesn't seem to have an end in sight. By breaking a project down into many smaller tasks we give ourselves a real way to organize and assign tickets and use sprints within larger iterations to reach end goals. As a contributor to any team, it certainly feels rewarding to close a ticket and say "I completed something today!"

CONCLUSION

In my career I've been in roles where "process" is annoying and seems only to limit productivity. I've also experienced the opposite, where "process" is not a dirty word, rather, it is so appropriately tied to daily activities that it seems, at the risk of sounding rosy, enjoyable! This is the type of process, a very real application of a quality system, that will lead to a highly effective use of these guidances that we call "standard operating procedures."

Revisiting the story I shared at the introduction of this article, armed with work instructions to make our quality system an integral part of our daily process, I would have been much more prepared to respond to the auditor interrogation. That is important, no doubt, but 21 CFR part 820 was not written so that individual contributors may appease auditors. It is there so that those implementing a quality system may figure out the best practices for their environment and needs.

If our goal is only to achieve a 510k we are missing the point. Our goal should be, first and foremost, writing quality medical device software that does not harm or injure patients. The 510k is secondary. Okay, admittedly I'm idealizing a bit. We all know the 510k has to happen to release our controlled software, yes. But if we strive to make our quality systems useful in a very practical way, getting the 510k will be a piece of cake.