NotesOn: Project Management – IT Testing 101

Introduction (V1.3):

The opportunity is just too good to pass up. Absolutely classic. Setting all political discussions aside, after three and a half years and tens to hundreds of millions of dollars (depending on to whom you listen) invested, the release of the "Obamacare" website was an unmitigated disaster. There couldn't be a better poster child example of how not to run a project, of any magnitude. But. In particular. How not to test. Of course it isn't the only such site: Federals, States, Locals, Commercials and Privates have a litany of disastrous Go-Lives to their "credit". To help you prevent such disasters in your IT Life I present the fundamentals of testing.

Ref:

NotesOn: The Four Fundamental Life Cycles of IT

Assumptions:

There are some assumptions that I need to declare before we get into the heart of the core matter of Testing:

1. A successful system requires a solid, well thought out, properly designed database. The database, the data handling if you will, is the foundation for the rest of the system. All of the rest of the system. If your data handling is "messed up" the amount of code required to work around that state is absolutely mind boggling. The reverse is also true. If your data handling scheme is "dialed in", the code logic

surrounding it will be easy to develop and easier to maintain. Poor database designs require an extraordinary amount of testing too.

2. Speaking of design, a proper, well thought out design is wholly dependent upon a thorough requirements phase and complete documentation. You will never achieve the "perfect set" of requirements" on a new system. The best you can hope for is 80%-90%, with 90% being close to a miracle. But. You must make sure that you don't miss any "big chunks" (which, as one reason, can occur by not including all of the stakeholders) during requirements gathering and must not miss anything but the "smallest stuff" during the design phase. Both of these potential pitfalls can be mitigated to a large degree by following the advices in this post.

3. Another big one, which I've mentioned elsewhere, is calling someone a Project Manager who is not.

Background:

This post, actually portions of this post have been laying around on the drafting table, or in a drawer, or "new idea" file folder on the computer for some while. I would chip away at it from time to time (as bandwidth allowed) but I didn't assign a high priority to its completion as I "assumed" most everyone in IT knew how to do testing in IT, at least at some reasonable level of proficiency, whether on a from-scratch system or a COTS (Commercial Off The Shelf) one.

Except. I'm not sure that is the case. Not anymore anyway.

Clearly, within certain segments of the U.S. Federal Government that is not the case. Or within some State Governments. As but one notable example from the past, some years ago California spent a LOT of money (for the time) on a new Department of Motor Vehicles system that failed rather spectacularly.

And how many high profile web based systems have been in the news after being hacked? Divulging tens if not hundreds of millions of personal records, combined, to the less than stellar citizens of the world.

The question is: Why? As in Why such spectacular failures? Not always the only answer, but a primary one is: lacking of testing. More accurately, lack of *proper* testing.

I am reasonably sure that the "Obamacare" website was tested. 6 Point Type tested.

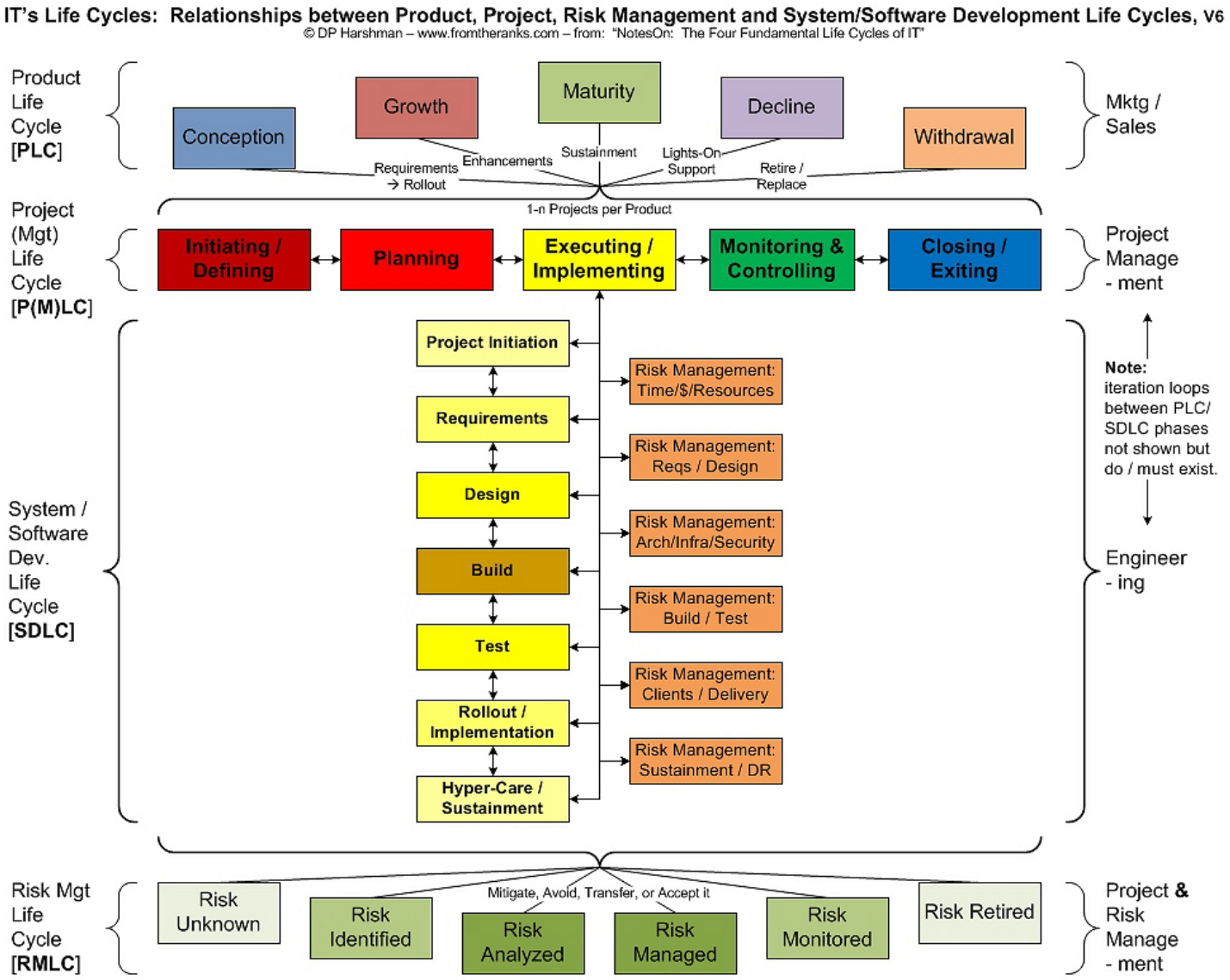
Which is to say: sort of kind of tested. About like some brands of Clam Chowder (a soup) in which a single clam is passed over the cooking pot on its way back into the freezer, to be saved for the next batch.

When it should have been 26 Point Type TESTED.

So. The following describes how you TEST a system. As in try to break the system test it. Ensure the users can use it test it. Confirm it performs as expected under anticipated loads test it. As in … Well, you get the picture.

Q: Why Test? A: Risk Management Of Course

Do you recall a post I did a while ago called … hmmm .. I have to go back and look up the title … oh yes, "*NotesOn: The Four Fundamental Life Cycles of IT*" (linked above)? There is a diagram in there, the first one, entitled "IT's Life Cycles: Relationships between Product, Project, Risk Management and System/Software Development Life Cycles, v6". I'll bring the diagram forward, but you really should go back and refresh your memory if you have read it or study it in detail if you haven't before going any further:

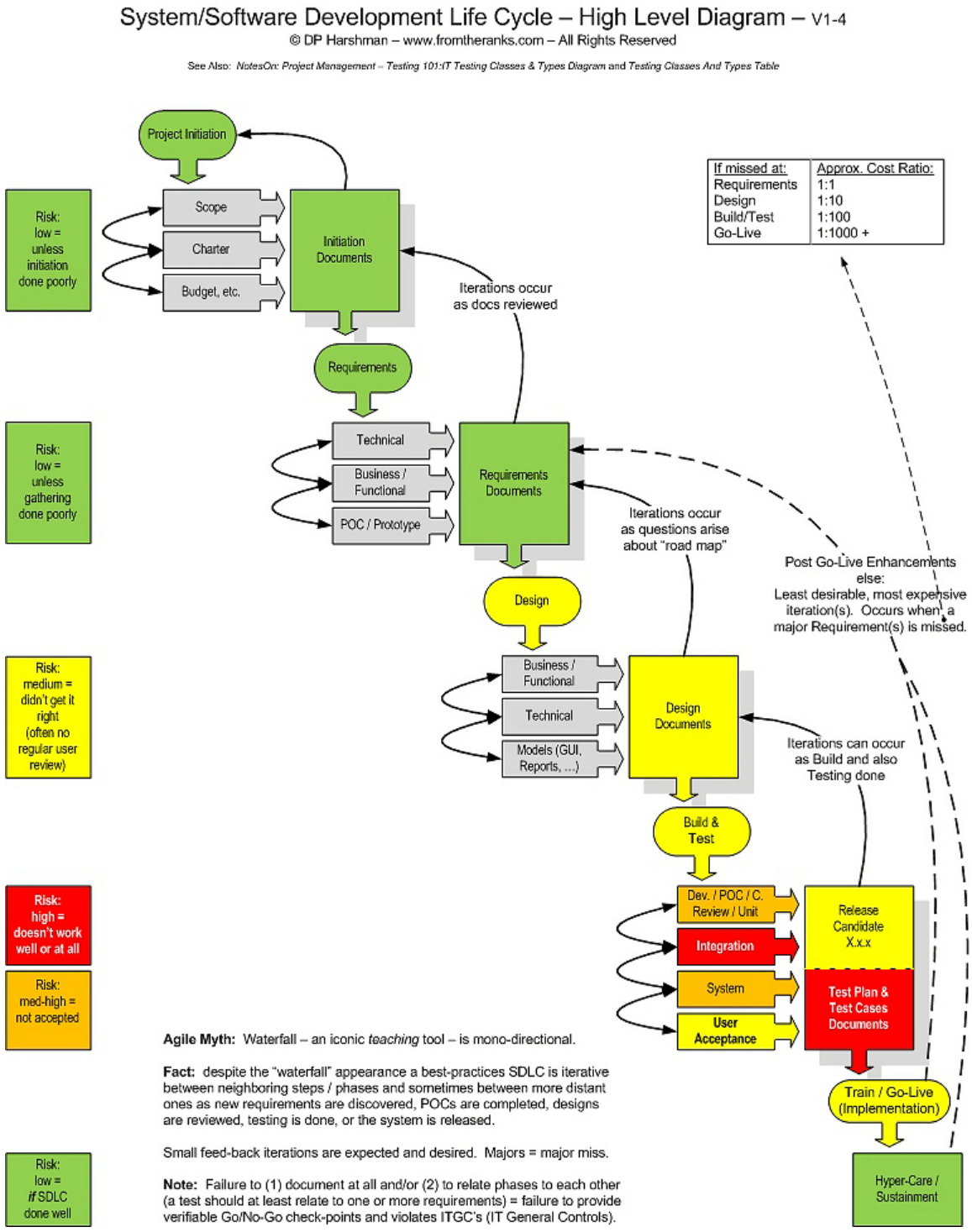


Take a moment to study this diagram. What is important to note in the context of this post is the "Risk Management" objects that are part and parcel of the SDLC steps. The simple truth is that if you ignore, or short cut, these Risk Management steps your system will not perform as desired and may not even run.

All good Project Managers are constantly aware that Risk Management is a critical part of their Role description. It is not optional. It is not a "nice to do when we have time". It is critical.

What Is An SDLC?

For those of you who cannot rattle off the methodology steps of a basic SDLC, i.e. recite each and their purpose verbatim in your dreams, let's take a few minutes and explore one. Why? Because. If you don't understand what a Software (or System) Development Life Cycle is all about, you will have no clue as to why Risk Management is vital, and, thus, likely no slightest interest in doing any more testing than you have to.



Do you see how it "hangs together"? One logical step, or phase, leading into another? And leading back to the prior if "something isn't right?"

Do you notice the different Risk Levels depending on where you are in the Life Cycle? Or which part? Trust me on this, each assignment is absolutely realistic. The only one that could be raised higher is Requirements, an argument could be made that it should be Yellow instead of Green as, if done inadequately, you will have nothing but problems and headaches.

I also included, in the diagram, visual proof that doing iterations are very much part of the standard SDLC. There is no way any project would ever be successful if the project manager, and team, used a "pass through once" approach. As the Agile folks would like you to believe is the case. Which it is not, of course.

By the way, the small "If missed at" table in the upper right hand corner? That is motivation for investing the proper amount of time in requirements gathering. Finding requirements later on which were missed during the Requirements Gathering phase, because, for instance, "everyone" was in a hurry, will, not may, but will cost you dearly later on. In about the ratios shown.

Q: Why An SDLC? A: Risk Management Of Course

Above I noted that each SDLC phase has Risk Management activities associated with it. Which is true.

But.

Another way to look at it, and perhaps the correct way, is that each Risk Management phase has action steps associated with it that have been gradually developed into a whole workable methodology. In other words, project failures occurred, first, the risks of future failures were identified, second, and then approaches were developed, honed and polished over and over to prevent those risks from occurring in the future.

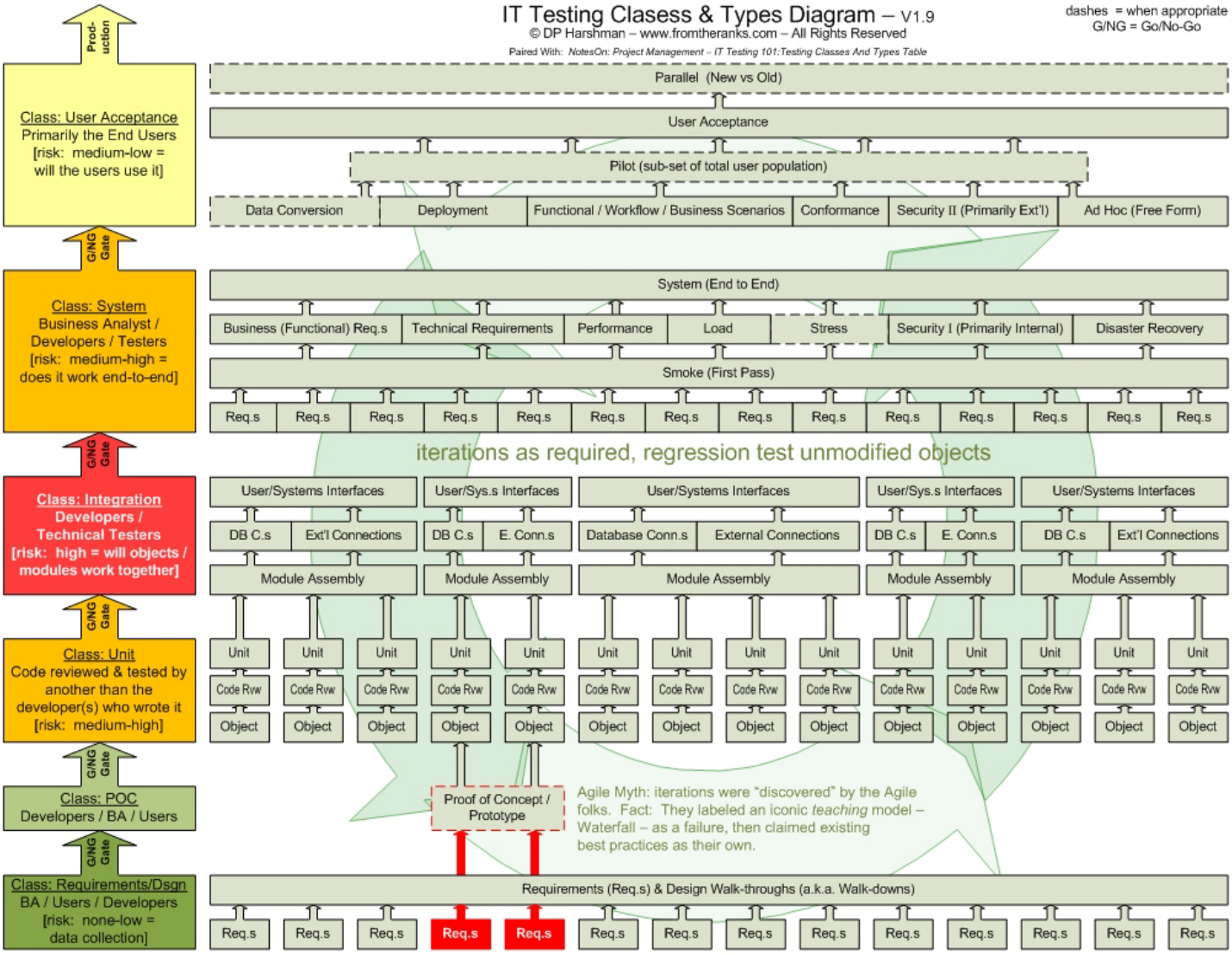
I should note that the SDLC above is simply a topic specific version of development life cycles that have been around for hundreds of years, at least. Architects, engineers, construction folks, city planners, etc., etc. have been following an earlier "incarnation" of the SDLC for a long time. There is more about this in the above referenced post.

Okay? Does that make sense? With some hands-on experience you'll have a better understanding of why the above SDLC model is used. As opposed to, say, Agile type approaches where one grinds and grinds and grinds and grinds out small packets of deliverables (hopefully), potentially forever.

Agile is okay for maintenance and minor enhancements I suppose. But don't EVER use it on major projects … such as the "Obamacare" web site. You are doomed to failure (at the very least by horrific to toxic cost overruns) if you do. Because, in part, due to the lack of a proper testing regimen (i.e. an incomplete, often brushed off, testing routine).

The Testing Classes & Types Diagram:

Speaking of testing, below is a diagram that lays out the Testing Classes and the types of testing within each. Read from the bottom up. Take your time. Print it out and carry it in your portfolio or tape it to your wall. Just leave the copyright and website data on please. [For full sized posters of this diagram please contact me via the "Contact Us" page.]



Did you notice the iterative circle in the background? Testing is not a "one shot" deal. If you test and find something is "broken" you fix and re-test. You don't "maybe fix" and move on.

The object of testing is to make sure that we are delivering what the user has requested … and can use!

Remember our IT motto? "Helping them do "it" better, whatever "it" is"?

If it doesn't work … if it blows up the first time they try to log in … if it seriously under-performs … if it …

Well, again, you get the idea.

Behind The Diagram:

The diagram is nice, it can be a great visual reminder of all of the testing steps that your project needs to account for (not all projects require all steps but all do need to have most of them done and signed off on all the time) but some "behind the diagram" data would probably help.

Below is a spreadsheet laid out in reverse order, i.e. top-down, which should provide you with more detail; in what each test means, common names for the tests (more than once the same test has been called something else by someone else), and some rules of the road for each. I've broken up my master table into sections and included key descriptions below each.

If you have questions, or you feel I have "completely missed the boat" on your favorite test (as in not including its name) please let me know. By the way, the shading refers to the Risk Levels noted in the above diagram.

Requirements:

Testing Classes & Types Table V1-4

Paired with: *NotesOn: Project Management - IT Testing 101: IT Testing Classes & Types Diagram*

What To Test	Test Class	Common Name(s)	Participants	Comments	Done
Are the requirements complete?	Requirements / Design	Requirements Gathering & Review	Entire team of Developers, the BA, the PM and the Stakeholders (End Users)	You need to, must, walk-down **all** of the requirements gathered (business and technical) from top-to-bottom, side-to-side, corner-to-corner. Ensure there are no major misses, no "chunks" omitted, no processes ignored, no exceptions to the normal business rules are left out in the cold, etc.	
The basic concepts and work flows of the future system.	Proof Of Concept (POC)	Prototyping	Business Analyst, End Users, Developer(s)	While not a test, strictly speaking, when called for Prototyping does test the shared vision of the future system via "look and feel" (Graphical User Interface, or GUI) models. The models represent what the system should "look like" but in draft form, with virtually no business logic (or code) behind the mock-ups. Prototyping is optional but may be done on both custom and COTS systems (especially those which are configuration based).	
Does this new method, new approach, new technique work?	Proof Of Concept (POC)	Proof Of Concept	Developer(s), Business Analyst (BA), End Users	When called for, build sufficient code pieces, and assemble new hardware if appropriate, to test the concept and prove it works before continuing project.	

Technically, Requirements Gathering is not a "Testing Activity". Except it is. Or parts of it. Here's a few questions that you must ask yourself as and after you talk to the executives, managers, entry clerks, folks on the front line, … in other words while you are walking in their shoes in an effort to do it right:

1. On the surface, does it all make sense? If not, your notes and diagrams have failed the test.

2. Can you, from end to end, *think* with it? If not, your efforts are incomplete and have failed the test.

3. As you review your notes and diagrams and so on, by yourself, with your team, and with the users, are there gaps? Does the user often say "Oh, that reminds me!" and so on? Then you are not finished.

Prototyping/Proof Of Concept:

Both are forms of testing, even if they are not technically considered to be so. If you have a new idea, or wish to use a new technology, or are unsure of a possible solution then … you do a prototype or a proof of concept. They are virtually synonymous and often used interchangeably.

If the prototype fails, if the proof of concept never gets off the ground? Then that test failed and you have to come up with a Plan B. Or maybe Plan C. Or. You have completely misfired on your gathering and there is some part of the requirements that you do not yet fully understand. Do not brush off this step if there is any uncertainty, even the least little bit in the back of your skull. Prove it out. First. Don't wait for later when you have likely locked yourself into the Plan, when it is too late and too expensive to change course. Make sense?

Unit Tests:

Testing Classes & Types Table V1-4

Paired with: *NotesOn: Project Management - IT Testing 101: IT Testing Classes & Types Diagram*

What To Test	Test Class	Common Name(s)	Participants	Comments	Done
Does the code appear to comply with requirements	Unit	Code Review	A Developer(s) other than the one(s) who wrote it.	The code should be clean, commented and easily understood by someone familiar with the programming language. And it should, on paper, meet the design requirements.	
Does the executable object take the input (event or data), translate or transform it and produce the desired output (event or data)?	Unit	Technical and Limited Functional Test (a.k.a. Black Box)	A Developer other than the one(s) who wrote it.	Without knowing the internals, does the single object, piece of code, work? As designed and per the requirements? Define the "expected" results before the test then compare to the actual results. This is the most basic basic testing there is. If it doesn't work at the unit level …	
Does the object "talk to", communicate with, another or other objects as it should?	Unit	Communication Test	A Developer other than the one(s) who wrote it.	May require a "harness", an artificial environment, to do the test, or sufficient objects, or chunks of code, around it so as to act as if it were embedded in the module.	
Does the UI control(s) -- button, link, menu item -- function as required and designed?	Unit	UI Controls	A Developer other than the one(s) who wrote it.	If the control doesn't work at the basic unit level it certainly won't work at the integration, or system, level. Controls and data validity are two of the most obvious test failure points for users.	
Does the object interact with the data source as required -- insert, update, delete, query.	Unit	Data Test	A Developer other than the one(s) who wrote it.	To state the obvious, not all data sources are alike, not all connections to data sources work the same, so test, always. And test for proper error messaging when the connection isn't present or drops.	
Does the database object, such as a stored procedure, function as required and designed.	Unit	Database Objects Test	Database developer(s) other than the one(s) who wrote it	There are more objects than just stored procedures in a database that need to be exercised to ensure compliance; could be triggers, could be custom data types, and so forth, plus the tables themselves.	

Unit Tests are the first level of technical team testing and are conducted on the lowest level components (objects) of the system to ensure they work per their requirements. Each object's actual test results must match its expected results; which must be declared before the test … not afterwards. The combined individually discrete objects comprise the foundation of the system and if one or more don't work the system won't work as required or desired. Though the name suggests otherwise, there are multiple facets of the Unit

Test, as per the table above, and, where appropriate, each is done to prove the object is worthy. Please don't skip those that apply because it may seem inconvenient. Not doing so will cost a great deal more later on.

Code Review: included in the Unit Test Class is "Code Review". The team which skips passed this, and assumes that all developers always get it right, is setting itself up for "issues" down the road. Here are some "rules of the road" for doing code reviews, and why to do code reviews:

1. Does the developer insist his/her code is fine and it's a waste of time to walk through it? If so, this is a minor red flag. Could be a confidence issue. Could be they didn't fully understand the requirements and are "afraid" to admit it. Could be they "did it their way", which is to say maybe they didn't follow your coding standards (if you have any) and perhaps they tossed in a few extra "bells and whistles".

2. Does the code scan easily? Can it be easily understood by someone familiar with the language? If not, this is a definite red flag. Worst case scenario is that it takes an "Enigma Code/Decoder" machine from WWII to interpret the code, to make sense of the code. More likely is that the developer has "short handed" their variables and/or nested logic within logic within logic within logic, trying to jam it all into one or two lines until it is nearly indecipherable. Maybe it actually works. But. The next developer who comes along to maintain or enhance it is going to waste a LOT of time trying to figure out, trying to interpret, the previous developer's work.

3. Is the code commented? Are the requirements, the reason for building the object, included? At least briefly? Is the basic logic of the code object commented? At least that much? If not, when even that developer comes back six months later to "tweak it" they'll have to figure out, all over again, what they did and why. Comments don't take up space in compiled objects so there's no "space saving" value for not doing it. Comments don't take "a lot of time" either. Developers hate commenting but … tough. They're paid to do a job professionally.

Integration:

Testing Classes & Types Table V1-4

Paired with: *NotesOn: Project Management - IT Testing 101: IT Testing Classes & Types Diagram*

What To Test	Test Class	Common Name(s)	Participants	Comments	Done
Do all of the objects in each module "talk to each other" from end to end.	Integration	Module Assembly Test	The entire Team of Developers; often the BA; possibly the testing Team.	This is the module's "smoke test". What you want to know is: does it blow up (in essence catch on fire) when you start it up for the first time in its entirety.	
Does the discrete module meet each of its requirements.	Integration	Module Functional Test	A Developer other than the one(s) who wrote it; experienced tester preferred	Functional testing shows up under System testing as well.	
Does the discrete module "talk to" its data source correctly -- inserts, deletes, updates, queries.	Integration	Database Connection Test	A Developer other than the one(s) who wrote it and the DBA (Database Admin.)	A basic test to ensure the module properly finds and connects to the data source with the proper login/password and proper permissions and privileges (i.e. authentication and authorization).	
Does the discrete module "talk to" other modules &/or hardware components per its requirements.	Integration	External Connection Test	A Developer other than the one(s) who wrote it.	Again, determine expected results, before the test, and compare the actual results to expected.	
Does the discrete module handle data transformations per its requirements.	Integration	Data Test, a.k.a. Data Transformation Test	A Developer other than the one(s) who wrote it.	This step is the next level above Database Connections, once you have the module talking to the data source, is it processing / transforming the data correctly.	
Does the module meet the UI requirements and design specs.	Integration	Basic UI Test	Developer(s), Business Analyst(s), End Users	Often involves a "walk through" with the end users. Catches "that's what I said but not what I meant" issues (better early than late). A tedious test as it checks navigation, internal to the module work flows, all data entry rules, default rules, min-max range limits, "garbage character" filtering rules, etc.	

This class of technical team testing is done to verify that all modularized unit level objects (logic pools) communicate properly within their container, then each with one another, then externally with other systems, and then, finally, function collectively as expected. Note: end-to-end is officially done during System and UA

Testing but full Integrations often do a basic front-to-back as all the dots are connected for the first time. Test targets typically include:

- Module to module data transfer (inbound and outbound)
- Module to database/data source (inbound and outbound)
- Security – authentication and authorization
- Access – module / data availability constraints
- User Interfaces – display and navigation and transition between modules
- Housekeeping of global and local variables, temp files, temp tables, etc.
- Etc.

Database / Data Integrity: this is called out separately though it is part of that testing (and part of Unit at a lower level), as the technical team must ensure the "data flows", i.e. that all database related logic works as required and designed and to ensure the database itself properly manages the data it stores, i.e. that all relationships and all update and insert and delete rules are correctly enforced. Again, do not brush this off in the early testing stages hoping that it will all work properly later on. Building a system is not too dissimilar to building a house. You can't put the roof on until the foundation and framing are soundly in place.

System:

Testing Classes & Types Table V1-4

Paired with: *NotesOn: Project Management - IT Testing 101: IT Testing Classes & Types Diagram*

What To Test	Test Class	Common Name(s)	Participants	Comments	Done
Do the related objects (i.e. the assembled modules), act as an integral whole without major faults.	System	Smoke Test	Developers and Business Analyst; possibly testers too. (Users typically not involved in this test.)	A test to determine if the components "hang together" without "blowing up" or leaking smoke. Term taken from old pipe and electrical tests, i.e. can you push smoke into the pipes without leaks, or plug the device into a power source and not have it blow up.	
Test the technical / foundational aspects of the system.	System	Technical Requirements Test	Testing Team and Developers	Particularly in custom applications (but not exclusively) there is a technical requirements document that covers architecture, performance, UI, naming standards, and the like. Test it all.	
Test each user requirements. Name the expected result(s) and then compare actual result(s).	System	Business Requirements (a.k.a. Functional) Test	Business Analyst, Testers, with Developers standing by to answer questions.	The entire business requirements document(s) is "walked down" and checked off against the Build. All of it. Note: If the customer is buying a "commercial off the shelf" (COTS) system, with or without customization, this is where they start their testing regimen against their requirements.	
Test against pre-designed business scenarios.	System	Scenario Test I	End users and Testing Team.	Goes beyond basic Business Requirements. Scenarios are created by stringing a number of requirements together into logical steps that mirror business processes, a.k.a. work flows. "Test scripts" should be written and followed by the testers.	
Test against performance requirements.	System	Performance Test (sometimes includes aspects of load and stress testing too.)	Testing Team	Briefly, does the system meet the (usually user) specified performance requirements, are responses meeting normal period usage/user load expectations and the like. If not where is the system bottlenecking.	
Test against peak load requirements.	System	Load Test (sometimes called or grouped under Performance Testing)	Testing Team.	In brief, will the entire system (hardware, software and network components) handle the anticipated typical end user demand (e.g. 6,000 concurrent users), meet expected peak (maximum) demand with other workload requirements with adequate response times.	

I couldn't figure out a way to "clip" this entire section of the table in one piece so I had to break it in two:

Testing Classes & Types Table V1-4

Paired with: *NotesOn: Project Management - IT Testing 101: IT Testing Classes & Types Diagram*

What To Test	Test Class	Common Name(s)	Participants	Comments	Done
Test against beyond peak load requirements.	System	Stress Test (sometimes called or grouped under Performance Testing)	Testing Team.	Sometimes omitted, if deemed as optional, this is intended to determine the beyond expected peak volume which the entire system can tolerate before "something breaks" and, if it does, "what breaks and when".	
Test all internal security requirements.	System	Security Test I	Testing Team with Security Team (internal and possibly external)	Look for any and all internal gaps between Security's requirements and the proposed release version, including end user rights and privileges (authentication and authorization), and segregation of duties (SOx controls).	
Test the disaster recovery plan.	System	Disaster Recovery	Full team (DBAs, Network, Developers, Infrastructure, ...)	All systems should have a recovery plan of some form; sophistication depends on its criticality. But. Whatever the DR tier/level make sure it works; before Go-Live.	
Test existing features after changes added.	System	Regression Test	Testing Team.	Regression testing is done if changes were made to an existing system or portion thereof. Makes sure the new changes haven't broken existing features or prior, unmodified, requirements. Often the test is focused on "likely candidates" that may be affected by the change but at times this approach is insufficient. (definitely iterative)	

Much of this class of testing is conducted by the entire team (techs and business SMEs) to verify that individual modules and their processes at least adequately meet expectations, i.e. meet both technical (aka non-functional) and functional (business / end user) requirements. Test targets include:

- User interfaces – data display, data edit, proper event handling, and system navigation
- Business rules enforcement, including business cycle management (ex: month end close)
- Work flows – routing, approvals, disapprovals, etc.
- Document management
- Multi-systems integration tests
- Database adds/deletes/updates (yes, you test this again)
- Database process logic via stored procedures, triggers, etc.
- Exceptions, i.e. proper exception handling.

Performance Testing: There are times when all three of the following types are “mushed together”, considered to be the same thing, under the heading of Performance Testing, i.e. no real distinction is made at all between them. That can be okay as long as (a) the types are understood and (b) a thoughtful determination is made, based on specific known data, that Stress testing, for example, is not required, i.e. there is no anticipated risk left unchallenged if one decides not to attempt to break the system. Learn these:

Performance: does the system meet the (usually user) specified performance requirements, are responses meeting normal period expectations. If not where is the system bottlenecking. Is it a code problem? Is it a poor database design? Is the server(s) being used overloaded already? Is … An experienced team will have preconceived expectations as to how well the entire system should perform, they will assume or have a benchmark (ex: sub-2 second response to a user input or request) against which to test their projected typical demands on the system.

Load: technical testing conducted to ensure that system wide response times meet expected maximum, peak, demand and workload requirements. To be successful the application logic, the database design, the access medium (client-server or web or mobile), the server architecture and the network must be sized for peak and "tuned" to work together. Typically the test load is applied gradually, and increased periodically while key indications of system performance are monitored. Though it does stress the system it is not a stress test as you are *not* trying to break it (though, if it does you have learned something). This type includes:

- Navigation and event request response times under max expected loads (ex: sub 2 second response to load customer's record)
- Transaction/Data retrieval and update response times under heaviest network traffic periods (ex: sub 2 second response time to post completed invoice during busiest time of day)
- Largest report printing threshold (e: no more than 5 minutes to run Aging report for the month)
- Simulation of maximum expected peak user count, all or most of whom are hitting the server as hard as is realistic for them to do (think *Holiday Super Discount Sale!* Loads)

A real life example is taking a system supposedly architected for 6,000 concurrent users and, using a load testing tool (a piece of software that can simulate 1 to 'n' users doing one or multiple tasks), discovering it blows up (or crashes if you prefer) at 600. I've seen this happen. More than once.

Stress: technical testing conducted to ensure the entire system continues to run and reaction times are acceptable under anticipated maximum, or greater, workloads. The *entire* system (hardware, software, databases, application servers, network servers, network pipes) is "pushed to its limits" to determine where resources bottle-neck due to, for example, inadequate memory, insufficient network bandwidth, CPU usage max'd (ex: pegged at 100%), data requests are I/O bound (too many outbound and/or inbound demands for data or the requests are being backlogged), etc. The goal is to either break the system (and if so what and when) or to prove the system will not crash and burn under heavy or heavier than worst-case demands. Any component of the entire system is fair game – database servers, web servers, firewalls, network switches and routers, app servers, load balancers, clusters if any, ...

Scenario: your average expected is 600 concurrent users at any given moment but at peak times the demand may rise to greater than 6,000 and the customer orders may well jump from 50 every five minutes to 500 every five minutes. What happens? What is the threshold beyond which the system falls to its knees (or worse), temporarily or permanently.

You can't afford to design and build for an infinite number of users and an endless list of oddball Use Cases but you may want to find the failure point and set throttles that will keep the system up and running.

Regression: technical and user testing conducted to ensure the functions and features of the previous release that were not (should not have been) touched by the update(s) or bug fix(es) still function and perform as before. There are two ways (at least) to do regression testing: (1) re-do/re-run every test script that was run on the previous release against the new release candidate to ensure all requirements (functional and technical) are still met, or (2) re-run every test script against a copy of the previous release – this confirms that the previous release is not broken and that the test scripts are valid – and then run it against a copy of the new release candidate. Do not add the test scripts for the new features, if there are any. Note: this is one activity that should be done iteratively at all levels of testing (Unit on up) to help ensure the "new" didn't break the "old", though more often than not it is only done (if then) before final User Acceptance Testing.

User Acceptance:

Testing Classes & Types Table V1-4

Paired with: *NotesOn: Project Management - IT Testing 101: IT Testing Classes & Types Diagram*

What To Test	Test Class	Common Name(s)	Participants	Comments	Done
Validated testing of any/all data conversion steps.	System / User Acceptance (UAT)	Data Conversion Test	Technical Team, Test Team	Could be considered to be in boundary area between System and UAT testing; verify that all data conversion is accurate. May require multiple iterations to "get it right".	
Test the roll-out, a.k.a. deployment of the system.	System / User Acceptance (UAT)	Roll-out / Go-Live / Deployment Test	Technical Team	Could be considered to be in boundary area between System and UAT testing; often repeated multiple times to ensure the final install / upgrade checklist is 100% accurate. Continues until it *is* 100%.	
Test the entire system against business usage scenarios.	User Acceptance (UAT)	Scenarios Test II	Business Analyst, End Users.	A final look at the system from end to end, using prior written test scripts plus any new ones to cover (implemented) change requests.	
Test against known regulations.	User Acceptance (UAT)	Conformance	Business Analyst, End Users.	Testing conducted to ensure that specific controls (S-Ox, PII, HIPPA, etc.) are met or required regulations (EPA, Health & Safety, OSHA, ...) are adequately addressed.	
Test the overall security in keeping with its data security ranking.	User Acceptance (UAT)	Security Test II	Technical and Security Teams and External Security Consultants	As part of the final UAT round, test internal and external access (firewalls and iDMZ/eDMZ zones), virus and intrusion detection and prevention settings, "hackability" (often tested by external consultants), etc. Critical for Confidential and Restricted systems but should be done for all security classifications (i.e. Public and Internal too).	
Test the entire system however the users wish.	User Acceptance (UAT)	Ad Hoc Test (a.k.a. Free Form a.k.a. Destruction)	End users and possibly the Test Team.	Users do whatever they wish. The intent is to try to break it, from simple garbage in/garbage out, to min-max range tests to doing something "unexpected" or "out of sequence" to ... As long as it is legal, safe and doesn't destroy property virtually anything is fair.	

Again, this section of the table grew too long with the adjustments so I split it in two:

Testing Classes & Types Table V1-4

Paired with: *NotesOn: Project Management - IT Testing 101: IT Testing Classes & Types Diagram*

What To Test	Test Class	Common Name(s)	Participants	Comments	Done
Test the entire system on a limited basis.	User Acceptance (UAT)	Pilot Test (a.k.a. Conference Room Pilot or CRP)	Limited pool of end users, usually SMEs.	A small group, or a select office or region, tests the system in the "real world" before it is rolled out to everyone.	
Test the new system against the prior system / process.	User Acceptance (UAT)	Parallel Test	End users.	If there is an old one, both the old and the new system are running at the same time. It is double work for the users but it provides them the "warm and fuzzy" certainty that the new matches or exceeds the capabilities and results of the old.	
Will the users sign off on the system.	User Acceptance (UAT)	User / Stakeholders Sign Off (Last Go/No-Go Gate)	Key end users.	If the users won't sign off i.e. won't accept the system, there are still issues. Bluntly, the system has not passed its final test and is not acceptable (as long as the Release candidate meets all original requirements and approved change requests).	
"Field" testing of the entire system under real-time conditions.	"Field" or "Real World"	Post Go-Live (a.k.a. Roll-Out, Release or Final Release)	All Users	A "Hyper Care" Team is set up to quickly respond to reported bugs/issues; minimum 30 days, best 90. H/C Team sets priorities with user's steering group, some bugs/issues added to the future enhancements "wish list".	
Note 1: change control is not shown but is a key element of all development, testing and release phases.					
Note 2: above test steps can be and often are iterative … but only until the system is acceptable and accepted.					

User Acceptance testing is conducted by the users before release into production as the final “testing gate”. The users confirm all elements of the system are working as requested/expected: look and feel, navigation, displays, edit rules, configuration, performance, work flows, business rules, etc. UA (better known as UAT) is done against the approved requirements, not against “wish lists” for future features. They may not do every single element listed in the above table, though they are certainly welcome to contribute their time and energy to them all if they wish, but they should be made aware of the results of all of them.

Following are a few of the key aspects of the overall User Acceptance Class of testing:

Conformance: user testing conducted to ensure that specific controls (S-Ox, PII, HIPPA, etc.) are met or required regulations (EPA, Health & Safety, OSHA, …) are adequately addressed. [Personally, I often include these tests as part of Functional as they too are system requirements. However. There may be occasions when a stakeholder, or auditor perhaps, wishes to verify that the critical controls are in place and then confirm compliance with the regulatory aspects, in which case these test cases could be extracted and run separately.]

Workflow / Scenarios: user testing conducted to ensure that all routing, approval, disapproval and escalation processes built into the system work as required. All workflows and scenarios are done against written test cases. My tool of choice is Microsoft’s TFS but there are others out there. If nothing else, though, use a

spreadsheet where test case numbers are matched up against the applicable requirements' numbers (and don't forget that work flows are requirements). Don't "wing it".

Ad Hoc (a.k.a. Free Form or Break-me): user testing conducted by the users, completely un-scripted and often un-supervised to help drive out the "engineer's logic" from the system (i.e. "If you do it exactly the way I wrote it, it works"). They may, and should, attack navigation, go after date entry edit rules, event sequence rules, yes/no/cancel logic, performance killing processes (to use our prior example, posting invoices while requesting aging reports), etc. End users have found more bugs by not doing what they were "supposed to" do than anyone could ever recall or would ever admit to.

Pilot: user testing conducted on a limited set of users as a near final user acceptance step. Typically done with complex systems and/or where the new system is introducing significant changes in how the business does business. In these events Organizational Change Management (OCM) steps are required to achieve full acceptance of the new system and its new business processes and the Pilot test is one way to work the "OCM Bugs" out of the OCM Program before introducing it to the entire company.

Parallel: user testing conducted at the same time between the old system and the new system, or the "old version" of the system and the "new version" of the system, to ensure, primarily, that the business functionality and data handling produce identical results. For instance if the users have an old General Ledger system, which they know produces correct and accurate results they will continue to run it while inserting the same data sets into the new General Ledger system. The end results, i.e. the financial reports, should at least be identical numerically if not in formatting.

There are probably at least another dozen terms (ex: black box) but if you use the above as a guide and test what needs to be tested, what you're not sure of, in a methodical way, and non-methodical too sometimes, you will obtain a workable system; not necessarily 100% perfect but workable, one that helps do "it" better.

Root Cause:

So why do major projects fail? Miserably? With the "Obamacare" site but one example? As noted above under "Assumptions", possibly poor requirements gathering, likely poor design methodology (both of which speak to a poor system development life cycle, quite possibly an "Agile" style), but most certainly ***abysmal*** testing protocols.

No question. No doubt. None at all. Because ***true*** testing would have led straight back to poor development and build protocols, which probably would have led the experienced Project Manager or Manager back to poor design which possibly would have led the well seasoned Auditor back to poor requirements. Simple.

Summary:

Does this make sense? Is it now obvious to those of you, who may not have known how to Test, that the "Obamacare" website (healthcare.gov, at the moment) "team" did NOT thoroughly test? Could not have thoroughly tested?

An "excuser" gave as their reason for failure that it is one of the most complicated websites ever envisioned. Horse-puckies. I can envision a LOT of sites that would take three and a half years (or more) and a hundred million, or more, tax payer dollars which could be a LOT more complicated. And, because I and my teams would apply standard SDLC methodologies, including Testing, when done, when it was Go-Live time, it would, ***minor*** glitches aside, work.

If anyone in the Federal Government truly wants to know "Why?" look first at the development methodology in "use" and then the testing protocols not being followed. [By the way, a usual edict behind this type of result is "The rubber must meet the road, NOW!"] As I said in the beginning, the "Obamacare" website is an absolute text book case on how "not" to build a system.

Hope this helps,

DP Harshman