

***"If it ain't broke, don't fix it"* will never apply to software development.**

by

Fletcher Carson
Executive Director
Omsphere LLC

20 August, 2001

It is *always broken*, and we keep on trying to fix it with new religions. *The Waterfall Approach, Rapid Application Development, Extreme Programming* and dozens of other methodologies have been developed over the years for one reason: **system delivery almost never reflects what was expected by the client, was too late to be of benefit, and/or was not reliable or stable in its operation.**

A constant reminder of this frustrating state of affairs is the often quoted miserable statistics regarding US federal software projects. Just 3% are used as delivered; 7% are used as changed or amended; 20% are abandoned or reworked; 30% paid for but never delivered; **40% delivered and never used**. In the commercial world, we see similar catastrophes that would be hidden by their embarrassed owners if not for the worldwide publicity provided by Internet exposure. Online banks that fall over the first day is just one example.

We keep reinventing how we develop technology. Why is this necessary for such a straight-forward process? Pick a software religion of your choice: talk to the client to determine what they want, write down the requirements either formally or on the back of a napkin, design an architecture that will meet those requirements, build it, test it, deliver it, and wait for applause. What is the breakdown that we simply fail to fix?

The answer is not related to the sophistication or complexity of what is to be delivered, nor in the platforms or tools used to build it, nor in the ability to extract 25 hours of work a day from everyone on the project as it nears completion.

What's left?

The client **for** whom the project exists, **from** whom requirements were extracted (usually under duress), **to** whom it will be delivered, is on vacation until the project is ready for primetime. *The client was never invited to the party.*

The client MUST be coupled to the development of the product throughout its lifecycle. What is missing in the software development methodologies is HOW to successfully link the client to the developers.

It's not through meetings every Tuesday, or software reviews, or executive briefings, or demo milestones. Tried that, been there, done that.

So what's left? Once again that nagging question. What are we missing?

TESTING is the ONLY conceptual and practical vehicle for successfully producing WHAT the client expects, WHEN they expect it, and HOW they expect it to operate. Those are the basic client conditions of satisfaction. *But wait a minute, isn't testing just the way that development proves to the client that they have met the requirements laid out a long time ago when they sat down to build the beast? Exactly! And exactly the problem that has been plaguing mankind since the chip was invented.*

Now, don't forget everything you know about development ...**simply rearrange it with a new mapping exercise, and a new player on the team.** What is proposed is *simply a new articulation of sound principles which most of use in our current development environments.* It changes our interpretation of the architecture and strategies used to develop systems.

Testing has always been the nemesis of development - the beast to battle if you want to make the delivery on time. Prove that you met the letter of the requirement. *"If they wanted blue, why didn't they say so a long time ago? They asked for green, and that's what they're getting because that's what's in the requirements specification. If they want to change it, they'll have to pay for changing it."* Been there, done that, everyone loses, and we invent a new software religion.

Everyone who has ever delivered a project, or had a project deliver something to them, knows this game. And both sides try to use lessons learned to out-manuever the opponent the next time, by seeing how the requirements can be specified in a way to hold the other side responsible for failed expectations. The words "**both sides**" and "**opponent**" were chosen carefully here to reflect the fact that development and the client *are* opponents separated by the testing battlefield.

System testing, functional testing, performance testing, user acceptance testing, load testing. These are all tactical and strategic methods of determining, ultimately, *whether the developers or the client is liable for the inevitable gap in expectations.*

GAME OVER.....NEW ARTICULATION.

Testing is the driving force behind meeting client expectations. Notice that the focus is NOT on proving that requirements were met. **Delivery of ill-specified requirements is a good way to pay the rent for people for the life of the project, but ultimately dashes client hopes for using the technology to effectively compete in the marketplace.**

The focus is on using Testing as the most logical and intelligent vehicle to couple the client to the development from day 1. *Most clients are not able to define their requirements precisely enough to guarantee that the developed product will be what they expected.* To pick on one group of developers who made the most of this flaw, many US military contractors were notorious for low-bidding or under-bidding contracts, knowing that they could ECP (engineering change proposal) the customer to death for **every single change to the requirements specification.** The money was not in delivering what was specified, but in all of the changes that the client made, because no one could think of everything the first time. Been there, seen that, *hated it.*

If the client holds out a vision of technology they need or would like to have to better compete, development assists them in defining exactly what that means in some form of a requirements specification. *However, this time the client is part of the team.* Not just a figurehead or token participant, but a bona fide member of the development team. **They are not there to code it, but to assist those that know how to build it with understanding how the delivered product will be used.** That means that the client must be able to provide feedback often, and early. Testing is the vehicle to perform this feat.

Development should not begin by coding a design. It should begin by developing the tests that will be used to prove the system works. Sound backwards?

Let's take an example of an online bank. A web front-end that talks to a mid-tier containing a database and processing logic, and a back-end legacy mainframe where products are handled (credit determined, loans approved, etc). Traditionally, teams begin working on different vertical parts of the system, because it is too large in both effort and technology to build the

entire horizontal system with one team. One or more front-end teams, mid-tier teams to handle databases and batch processing, and back-end teams to handle Loan and Mortgage processing. Then after the *First Build* is defined, all of the teams try to get their code to compile and work together, like gluing 5-story buildings together, except that the ceiling heights were different in each building. Tweak, glue, band-aids, paper clips, and voila...it can walk. (*Not ready for testing yet, because the client is not getting this beta version...it was simply there as something on which to build the next set of functionality, which the client WILL get*). After the Second Build, which pushes the schedule, it's given to testing to demonstrate that all of the requirements the developers implemented actually work. Makes perfect sense. **Oh yes, and remember that since development went over schedule and budget, testing will have half the time and half the resources to test a new system.** Happy client?

Testing intelligently changes this. The system is built horizontally, not vertically. The system will eventually be delivered by showing, for example, that Joe Customer can open a loan using the web screen front-end, have his credit checked, and be approved before he logs out. The client knows that this is what they want the system to do. So the test procedures for testing this scenario can be built BEFORE the code is developed. Why? *Because the definitions of the screens, the interfaces between systems (APIs), and the processing of the information flow must (or should) be known before coding begins.* **Therefore, it is possible to construct a test harness that utilizes the APIs to push data into the system, gather output, and determine if this is what the client thought the system would look like.**

Now development builds horizontally, so that the entire system is in place before vertical development takes place. *Place ONE field on the web screen for the customer to enter his name. Build the software that will connect the systems and allow this one field to get to the mid-tier where it will be placed into the database as the only data field in the system. Build the interfaces that will allow this field of information to get to the back-end. Build the software to capture this field in the backend, and process it. Build the interfaces necessary to export this field to external vendors to print documents, perform credit checks, etc.* The result is a system that is completely operational, but does almost nothing.

However, the system can now be tested end to end, by entering the customer name, determining if it ended up in the mid-tier in the right place, and got out the backend to the external vendors in the correct format. Performance testing and load testing can be accomplished also. A baseline of getting a simple transaction through the system can be

ascertained, *and a load test can determine how much the interfaces and databases can handle before the system crashes.* The client can see how the system will work, get a feel of how fast it is, whether it can produce results under a projected load. They will know whether it seems that the architecture being developed is a reflection of how they see the system working operationally, and how it will be maintained.

Building testing into development from day 1 also provides a powerful mechanism to make Testing and Development partners in this process rather than playing their traditional adversarial roles. Testing can produce test harnesses based upon the APIs, and provide them to the developers. **The developers now know what the criteria are for acceptance by Testing, and ultimately by the client.** For interfaces that are low level, the developers can use these same tools to build harnesses to ensure that their code works from gozinta to gozouta (low level APIs). *Each defined interface can be simulated to isolate the local testing from the need to have all other systems on-line.*

For example, the external interface to a vendor that produces documentation to be sent to the customer, can be simulated based on the API specification, and then functionally tested - without trying to test the other system at the same time. **Simulation of interfaces allows the developers to have complete control over their testing, without coordinating execution with the other systems. It also allows testers to perform any type of testing independent of connected systems.**

Last, but not least, it allows the client to see how the system will perform, and look when it is finished. This allows the client to make changes very early in the process, and manage the expectations of the delivered product.