Testing Under Pressure

Rob Sabourin

If you are involved in testing software, then I've probably shared your pain. I have tested under pressure, without much time, and with severe resource limitations. I have managed to successfully test in turbulent projects within many different planned or agile lifecycles. In order to succeed, I stick to my core values and apply five key principals:

1. Purposeful testing, begin with the end in mind
2. Active context listening
3. Flexible decision making workflows
4. Ruthless triage of requirements, testing and bugs
5. Always know the last best build

Here are examples of how I have applied each principal in real projects.

Begin with the End in Mind, Next Generation Rendezvous

The first principal, "Purposeful testing, begin with the end in mind", is about understanding how to answer software engineering's fundamental question, "How do we know we are finished?" I start with an understanding of the project purpose to focus development and testing efforts.

It was a "death march" project. The fate of the company was in play. Over two hundred existing customers received government grants to purchase Rendezvous. Rendezvous had to be operational by a fixed date, or clients would be penalized and forced to return their grants.

Development had been going on for well over a year with a team of five programmers, a development lead, and product manager. The development and testing teams were set up and staffed. They were expected to collaborate using an iterative feature-driven development strategy progressively adding features to the product and delivering the work in progress to the testing team. The trouble was that the development team could not even get one build ready to deliver to the testing team. How could we test something that developers couldn't even build?

I discovered some interesting characteristics of the project. The Next Generation Rendezvous product was being developed as a general purpose product. The previous version of Rendezvous was a collection of over two hundred different custom configurations. Customers had their own private code base. They paid for the customizations on a case by case basis. This was a lucrative development model. The customers loved it. The developers loved it. Developers were treated as if they were heroic knights in shining armour. But now, the product management goal was to develop a single product for the entire market.

Next Gen Rendezvous had hundreds of menus, dialogues, and controls. Testing and development were building the system on a feature basis. Due to the unstable code and the feature-based test focus, it would have been impossible to meet the mandated delivery dates.

I studied the project from many perspectives. I met dozens of people familiar with Rendezvous. I spoke with many customers, operators, support staff, trainers, and other subject matter experts.

I refocused the development and testing efforts. I introduced a usage scenario-based testing approach. I identified twenty seven critical Rendezvous usage scenarios. I set up a small test team comprised of a test guru and several subject matter experts with field experience. Weekly builds were delivered to the testing team. Testing of each build focused on walking through the twenty seven usage scenarios with sanitized real data. Progressively, the developers stabilized the code on a scenario basis. The testers were able to identify critical bugs that blocked users from getting their jobs done.

A month ahead of the deadline, we had pilot installations at several clients. We successfully released Next Gen Rendezvous on time. In subsequent releases, developers removed superfluous menus, dialogues, and controls. We simplified the product, creating a graceful general purpose application.

By focusing on how the customer used Rendezvous, testing was able to drive product convergence and thus tame a crucial death march project.

How Active Context Listening Saved the Communiqué Project

The second principal, “Active context listening”, helps me understand how shifting business, technical, and organizational context drivers influence testing priorities. Active context listening is a dynamic and continuous process. I identify opportunities to offer value. I have been able to change testing from a bottleneck into a project’s critical success factor.

The testing team was going nuts. Audrey, the SQA manager, couldn’t believe it. Build after build were exposing ugly bugs. The server crashed. The automated test suite could only run a few cases before choking. Memory leaks, viruses, and flakey OEM patches were the daily drama of “Communiqué”. The development team shifted the best developer onto a different project. Midway through, the VP of R&D assigned the project to me with a clear directive: “Converge the project and deliver a commercial release before our next board meeting”.

Why did we need the release? Survival! We needed continued financial support from private investors. They risked pulling the plug if we could not spin their capital into a commercial product. The next board meeting was our last chance to prove that we were not wasting their money.

My first step was to identify as many project context drivers as I could find. I looked for sources of information. I looked for sources of wisdom. I looked for business issues, technical issues, and even legal issues. Who were the investors? What were they looking for? What commitments did the company really make? To whom? Why? Were any requirements real needs? Which were wishes? Which were must haves? Why did the R&D department shift resources away from this critical product?

I arranged short daily stand-up meetings with all team members. I made the team baked goods every day. I listened to the team. What was blocking them? How were we progressing? I listened to their fears and frustrations. I discovered that the test team was trying to be diligent. I learned that developers tried to implement solid unit and integration testing but were block by weak OEM components. Developers had also provided us many testability hooks and a custom embedded test harness. I discovered that all the product manager wanted was a simple working solution that worked in many environments. I discovered that our board of directors cared a great deal about this project. They needed to raise additional financing. Investors had been promised a new product release based on the previous investment. If the board could demonstrate our ability to deliver a working solution the investors would very likely invest over $2.3 million dollars which would keep us afloat. I listened to the sales team and found an especially interesting sales lead at the French Stock Market in Paris.

The mid-project staff shift was caused by several VPs interested in showing investors our new web-based technology. These VPs did not realize that the new project put Communiqué at risk. They assumed that R&D could do both projects.

By actively listening to project context drivers, I devised a strategy that enabled the testing team to become project heroes. I proposed to refocus the project. Instead of targeting all of the environments, why not start with the French Stock Market environment. Get their Operating System, their network configuration, even their actual hardware. The testing team was able work with the customer to capture real test data. The test environment became the project focus. Direct communications channels enabled developers and testers direct access to our customer's technical staff. The customer loved the attention, leading to a great reference site generating plenty of solid sales leads.

We were able to stabilize this single configuration and we were able to confidently process customer transactions.

We delivered a working solution over a month in ahead of the critical board meeting. The company raised additional capital. The investors were delighted.

Decision Making

I've used the third principal, "Flexible decision making workflows", to help me deal with organizational politics and avoid escalation oriented deployment decisions.

How do we decide what to test? What not to test? What bugs to fix? What bugs to leave? What requirements to focus on? Which features should be deffered?

I set up the software engineering workflows at MVM, famous for virtual modeling at popular fashion sites. I involved product managers, architects, development leads, and testers in the prioritization of features, testing objectives, and bugs. I made sure that decision makers had access to the information they needed throughout the project. Delivering a quality product meant delivered value to all project stakeholders. Developers and testers knew specifically how each feature was valued and by whom.

One Friday afternoon, the decision making workflow broke down. An unexpected visitor arrived at our prioritization. It was an anxious gentleman with a big wad of paper. I did not know who he was or why he was barging into the meeting. He demanded our immediate attention. "Here is a list of 17 product issues. Drop everything you are doing. Resolve these 17 issues and ship the product. I don't care what you say or think." "Do as I say." He commanded.

This person was the company president whom I had never seen before. The president held weekly phone calls with each major customer. The last call was on Thursday in which he reviewed the customer's key issue list. The president had no idea how we made decisions. I had no idea that the president kept such great contact with all of the major accounts.

We did what the president demanded. We resolved the 17 issues. Some were already on our bug list. Some were completely new to us. We did nothing else and then we shipped the product. It took over three months to rework and fix the software development process to make up for all the problems introduced by the presidents list.

I learned a valuable lesson. When you are developing software in a highly turbulent context, it is very important to get buy-in and support from executive sponsors and project stakeholders about the mechanism and workflow used to make decisions. Now, I am always on the lookout for stakeholders missed. I need their support in how we make discussions, especially decisions made on the fly about focusing requirements, tests, and product bugs.

Ruthless Triage

The "Ruthless triage of requirements, testing, and bugs" is my forth principal. Prioritization is like emergency medical triage. For each issue I consider business criticality, technical risks, and the available testing skills. Each and every decision is triaged.

Edsger Dijkstra taught that software testing can show the presence of bugs but can never demonstrate their absence. For even the simplest application, there are more possible tests which can be run than there are particles of matter in the universe.

In turbulent projects, I face daily changes in business, technological, and organizational context drivers. In order to deal with turbulence and focus testing, I have always set up a mechanism for test triage. In test triage, a small group of people review and evaluate test findings, development progress, bugs, and testing priorities.

I set up the mechanism to enable a short meeting to triage testing issues. We reprioritize our testing based on business knowledge, technical risk, and our findings to date. We decide what to test and what not to test. I assess the benefit and consequence of implementing test objectives. I assess the benefit and consequence of skipping test objectives.

I manage test objectives as a prioritized heap. Bubbling to the top are test objectives which offer more value to our stakeholders. Bubbling down are test objectives which we might skip entirely. Low priority

objectives are assigned less testing effort, while high priority items are tested earlier and given more effort.

While testing, I continuously collect and triage new test ideas. Often, the most valuable test objectives are identified during testing by testers. Sometimes, objectives which appeared high priority at the start of the project become low priority as the project context is revealed.

The most reliable product I ever worked on was the AVT-710, an embedded system driving a video display terminal. It was a fixed delivery date project. We were to replace a weak, failing product in the Middle Eastern market. Unexpected project turbulence was largely due to freight rates which increased dramatically during the First Gulf War. A few months before release, we refocused our development to the Eastern and Central European markets. Ruthless triage daily was the key to project success. All team members were aware of the project purpose and we reacted quickly to major changes. The AVT-710 was on the market for over 10 years. There were no field reported software defects over the life of the product. The AVT-710 won the BYTE MIDDLE EAST product of the year award for its reliability. But, the AVT-710 had over 300 known bugs in it. The success of the product was due to the effectiveness of triage. Deciding what to test. Deciding what not to test. Deciding what to fix. Deciding what not to fix. Triage was based on a solid understanding of how the product would be used and excellent relationships between developers and testers. The message is to ruthlessly triage, purposefully triage. Triage always. Never give up. Never surrender. Use triage to decide what bugs to fix and which bugs to keep.

Always Know the Last Best Build

The fifth principal, “Always know the last best build”, guides me when I have a fixed delivery date. When the release date comes up, I want to ship the best build I can. Typically, I know the least about the most recent build, but one of the most recent builds may strike a reasonable balance between working features and known weaknesses that best meets our project goals.

My first commercial release was on June 4, 1982. I completed and delivered the 2-d graphics package Quarto. In its day, Quarto was pretty slick technology. Quarto allowed applications to drive bus mounted graphics cards on PDP-11 computers and included support for digitizing tablets and frame grabbers. Quarto supported windowing and menus years before Xerox, Apple or Microsoft developed their Windowing systems.

Every year I celebrate the release of Quarto. In 2007, I took my wife Anne on a month long romantic tour of New Zealand to commemorate the 25th anniversary of the Quarto launch. Of course, Anne does not believe we celebrated a software release. Anne thinks we celebrated something completely different, our 25th wedding anniversary. We were married on June 5th 1982. Quarto was release on June 4th 1982. I was not waiting for the Quarto release before marrying Anne, providing I was able to provide for my family. The Quarto release was a fixed delivery date release. I was the main developer, the main tester, and the technical writer responsible for Quarto. Without me, nothing was going to get done.

The Quarto project used daily builds and included automated test harnesses to help regress the application. Every day we would take a build into the testing environment and exercise it based on our knowledge of what had changed. We kept a simple log of the product features. We had a lot of interesting regression bugs due in part to the harsh environment with very limited amount of memory available and high speed responses required. We implemented such risky programming practises as self-modifying code to optimize line drawing and area filling primitives. When a modification was made to one part of the software, it could risk breaking some other part of the software.

Every build was assessed and scored on each functional area or testable object. We used four grades, confident, partial knowledge, no knowledge, or blocked. Every day, I was able to review the stability and status of all the builds ever made. When hitting the fixed delivery date, I did not ship the last build which was the build I knew the least about, but rather I decided to ship the last best build.

Shipping the last best build has let me meet many fixed delivery date releases in many development contexts, especially those with regulated or contractually binding release targets.

Putting the five Pressure Principals into action

On time, on quality, and on budget are meaningless unless you are on purpose. I have achieved great purposeful testing when everyone involved in testing understood why we are developing the software. When you are well informed, you are able to make better microscopic decisions when testing and better macroscopic decisions when planning your test strategy. When you observe an unexpected emergent behaviour, your understanding of the project "why" will guide you in determining how much effort you should invest exploring it. Knowing the project purpose guides bug reporting, helping you describe how the consequences of not fixing the problem might work against the project purpose. Coming up with new testing ideas varies dramatically when the testers are in sync with "why" we are testing the project. I urge testers to understand the "why" before getting caught up in the tyranny of the urgent. Testing without purpose is often wasteful. Testers should ask "why" every time they are assigned a test objective.

Active context listing is all about being proactive. There is always a reason for project turbulence. If the testing team can anticipate changes, then they can adapt. If the testing team is the last group to know about a change, they may have wasted a lot of time focusing on aspects of the project which are no longer important. Build relationships with your peers in other departments. Development peers can give you a heads up on technological changes. Sales peers can help you understand who will be buying the product and what they plan to do with it. Customer support peers can help you understand potential areas of weakness and trends in system usage. Accounting and finance department peers can help you understand the potential financial impact of the product under development. Living in a restricted testing silo leads to being locked out of important information.

The last thing I want to do when a critical decision is required is to start asking how we might make such a decision. Decisions related to prioritizing requirements, tests, and bugs should be made long before you start development. The decision making process should vary depending on the level of testing. Unit

testing issues might be left up to the developer. Integration testing bugs might be prioritized without the input of product management because there could be many phantom bugs due to the use of stubs and drivers in place of missing code. System test issues might be decided with the combined input of product managers, developers, and testers. Customer acceptance testing bugs may be exclusively prioritized by the customer and account manager. Set up the decision making workflows in advance and then adapt them as required if context changes.

Triage can be implemented by extending the traditional bug prioritization or change control board sessions. Ruthless triage means you must come to a decision. To quote Yoda from Star Wars, "Do or do not. There is no try". As early as you can, insist that triage meetings come to a conclusion. Encourage people to commit. Fix it now. Fix it later. Do not fix. Avoid and eliminate priority levels such as: Fix (optional) or Fix (if we have time). The priority of bugs can be reviewed after important project milestones and should also be reviewed whenever the decision making workflows change.

As soon as development starts issuing builds to be tested, I start reporting information about those builds. I answer the question, "What choices do we have if we were to ship today?" Some builds are stable but missing features. Some bugs work on some platforms but fail on others. Some builds are feature complete but unstable. I report information showing the alternatives we have available if we were forced to ship the product today. In my experience, stakeholders quickly grasp the significance of these reports. I limit my report to the best five builds. My assessment is based on features or capabilities identified by product management.

Testing with these principals helps me focus on what matters and successfully react to change. Applying these principals leads to a dynamic testing experience tied closely to the important business drivers of the project. When operating under extreme time pressure, I am able to consistently focus precious testing resources on what matters the most.