STAREast 2000
May 1-5, 2000

High Speed Testing Cycles:
An Approach to Accelerated Delivery of Bug Free Software.

By Daniel Navarro-Ramírez
Banco Nacional de México S.A.

Introduction

Testing has never been easy, but when it comes to large institutions, things can get terribly complex. Take large banks for example. For years they developed their own solutions, building applications and bringing in all kinds of hardware and software available, in order to provide their customers with better products and services. Most of the times buying decisions were taken on the go, following no architectural model or plan, which , unfortunately, led them to rely on modern Frankesteins (a.k.a. automated beasts) to take care of their most valued asset: their customers.

Due to poor software engineering and testing, these Frankesteins experienced frequent disruptions, causing a poor service level for customers. Nevertheless, most of the times these service disruptions meant no major threat to banks, since competitors offered similar products and services with same poor quality. Customers had little or no option.

But suddenly, global competition started demanding better products (higher quality) at lower costs (less money) and at higher speed (getting there before the competitors did). While it was feasible for most mature IT organizations to face the challenge, those companies that were not mature enough faced extinction.

No better place to prove this statement than Latin America. For the last decade, Latin American banks experienced such fierce foreign competition in their own traditional markets, that almost half of them no longer exist. Those who survived had to evolve dramatically in a short period of time, changing and optimizing their current services and products while trying to modernize their IT processes.

Testing was no exception. In the following paragraphs you will see what a Mexican bank has done on testing to deal successfully with this new reality.

Assessing the beast.

1994 meant the beginning of new challenges for the largest bank in Mexico. As of January 1st of that year, it became effective the renowned NAFTA (North American Free Trade Agreement signed by Canada, the United States and Mexico) which set new rules of competition at the traditionally closed Mexican financial market.

The IT organization at Banco Nacional de Mexico (Banamex) realized that many things had to change in order to survive competition. We soon became aware that our software's robustness could determine the bank's ability to survive. Therefore, a lot of attention was put into the quality of the software we produced.

With our own mexican version of Frankenstein, we all realized that if we wanted to obtain high quality out of the current situation, we would have to struggle with millions of lines of code, dozens of mainframe boxes, hundreds of communications gadgets and, of course, hundreds of undisciplined developers and project leaders. Even worse, under this new "big leagues" competition, our systems and applications would need to change as fast as never before, letting little or no space for traditional software process improvement approaches.

After an in-depth assessment, we realized that, on the testing field, there were LOTS of problems to solve (areas of opportunity, as consultants nicely called them). Most of them related to testing methodology, organization and infrastructure. I will not discuss some of them since they exceed the scope of this document, but I must point out that they generated lots of actions that demanded a lot of support and hard work. Among them, we focused on testing cycles, which is the central topic of this discussion.

Leveraging our Test Facilities

We knew we required to improve our current testing facilities. According to testing gurus, we should establish some "testing cycles" in order to test properly all new changes released into production.

Sad as it may be, many people discuss about software quality but very few of them realize the large amount of investments it demands. At the beginning, a lack of testing infrastructure seemed to be the perfect excuse for developers to avoid testing. Even the user community seemed to favor the idea of bridging over the acceptance testing phase, so that they could release new software into production ASAP. But losing customers and revenue made them all think it over.

Setting a Model Office and a testing laboratory was no major obstacle. As a matter of fact, most of the System and Acceptance testing infrastructure already existed. It was simply scattered around “owned” by different groups or organizations who voluntarily donated them as soon as they received a couple of memos from our CIO asking them to support this idea. Out of the blue emerged a large amount of mainframe computers, PCs, servers, routers and many other components which we used as a basis for the testing lab. Even some corporate projects, such as Disaster Recovery and Y2K, provided us with additional funds to set a robust system and acceptance test environment.

Perhaps one of the most enthusiastic sponsors for our new testing facilities was the “Banamex University” group. An organization whose purpose is to train new tellers for our branches. They decided to connect their training centers (distributed around the country) to our centralized acceptance testing facilities in such way that trainees could learn to operate the on-line applications in “real life”. While being trained, they face similar situations to the ones they will confront once they begin working in a branch (which includes software failures). These teller trainees have ended up being some of the best inspirational testers for our applications, since their absence of paradigms lets them stress our applications in the most unexpected ways.

Establishing Test Cycles

Ok, ok, now we had all the necessary infrastructure in place, but, what would we do with it ? Things seemed to be pretty straight forward. All you had to do was to set “testing cycles”. “On line-Batch” sequences followed by an infinite repetition of them. Applications would be stressed by a big number of transactions generated from playback tools, testers, users, teller trainees, etc.

But in reality things seemed to be a little bit harder than that. First of all, we needed to establish some rules to install and uninstall new untested program versions. Our brand new testing factory realized that it should behave similar to a production data center. We learned the hard way that new software versions should be installed everyday before the on-line and batch services and not in the middle of the day.

By the end of that month, we convinced our first victim, ooops !, I mean, our first business project, to come test their application at our facilities. We would install their new software version at our shiny acceptance testing facilities and the user’s testing group would come down to our laboratory to test what they had requested.

Needless to say it was a success ! All our resources were dedicated to their project and to meet the dates they had previously set. Never had they experienced such a complete “real life” environment without impacting production environments. Success became public. It was so loud that most business groups decided to follow the path. We were radiant, but, inadvertently, it was the exact moment when the shadow of trouble appeared.

Now everybody wanted to try it. The whole user community wanted to test all the changes they had requested BEFORE they went into real life production. And, of course, we were eager to serve them. At the time, we hadn’t realized we had incurred into an elementary mistake. Our resources were scarce and we could only serve a limited amount of projects at the same time. Concurrence and a lack of rules became our cancer.

As soon as two projects concurred in our testing environment, we were unable to serve them both as well as we had done the first time. Not to mention when five different project coincided (Or should I say collided ?). They all had entered our facilities at different times; they all required different database dates and different applications; they all wanted to have the whole laboratory for themselves; some of those projects demanded to have three “daily” cycles per natural day, while others expected to have two or three day long “single day” cycles. Catastrophe was at sight.

An airborne solution for testing concurrence problems

Funny, but even though I am a frequent flyer, I had not noticed the great similarities between airline operations and testing environments management. Airlines, same as testers, have a limited infrastructure . They have a great demand for services (on certain routes at least). They have tight schedules to deal with and their customers expect to get a high quality standard service as well as a safe and pleasant journey. Not to mention that they expect to arrive right on time to their destinations (same as IT projects).

After a century of flying experience, airlines have learned how to deal with these demands. They have a fixed number of flights that serve a fixed number of destinations (they do not commit to more than they can handle). They have a booking system to avoid collisions among passengers. They take off on time (at least most of the times) regardless of the presence/absence of confirmed passengers.

They fly as much as they can (time is money) and, above all, they follow a strict methodology. Personnel’s rolls are well differentiated. Maintenance periods on engines, fuel repositories and airplane parts is strictly followed.

Well, we discovered that most of these centennial routines could apply to our testing environment, because we have similar constrains that determine our capacity to provide concurrent software testing environments. If we think of each testing environment (a laboratory for instance) as an “airplane”, and of a project under test as a “passenger”, we will need to set “departure” and “arrival” dates for all passengers. “Passengers” must reserve a seat in advance. If they require a non standard menu or service on board then they ask for it. Those “passengers” who don’t show up before take off, can not get on board in the middle of the flight (unless we’re talking about Harrison Ford).

While on board, they can enjoy a standard service. If passenger’s behavior “hurts” other passengers or he is unable to coexist with other passengers or with the crew (operating systems, unchanged applications, etc.) he will be “invited” to jump down. Some minor changes may apply as long as they do not jeopardize the purpose of the flight and the arrival date.

Setting a Testing Train

What we did was to set a yearly schedule and a few rules for successful system and acceptance testing. The assumptions behind this methodology are 1) that maintenance testing and most large scale projects usually take a standard period of time to be performed and 2) the idea that concurrent projects may coexist while sharing resources (specially expensive resource such as mainframe cycles).

Through an extensive use of intranet we published the important dates for a testing train such as: Booking deadline, Software arrival and installation, Test beginning, Test end, Release to production and Field installation. We also published all standard services available as well as a way for users to demand non-standard services. A “reservations system” was put in place so that “passengers” could book in advance.

Gaining Train Acceleration

Once in place, trains started to deliver better software, BUT we must admit that they slowed down development. At the time we made no difference between big “passengers” and “small” ones, so they all had to wait for the train ending regardless of the date they had actually finished. Our original trains considered a long time for manual regression testing.

Since the testing discipline started to be a normal part of every project, we knew it was the right time to begin test automation. Previous efforts to automate testing through capture and playback tools had failed, mainly because the testing phase was not even considered in projects, so there was no time to test nor automate. But now the whole organization realized how important testing was and they were ready to invest time and money to design test cases and automate them.

In some cases, we started to use data slicers to extract data from production databases in order to reduce processing time. In some other cases we used data comparison tools to automate result analysis.

Once you integrate the use of this combination of tools (capture & playback, data slicers and data comparison) you can reduce dramatically your acceptance testing phase. It is then when trains can be reduced in time. Migrating from a slow motion train into a faster train can be difficult when both of them share same infrastructure. Client/Server type of applications can move more easily than legacy systems since many times they can be tested in isolation. Web based applications apply too. Most of the times you will require to keep both types of trains running in parallel, specially when “passengers” from both trains need to be in touch.

Lessons Learned

Evolution in testing takes time, upper management support and discipline. In our case, establishing testing trains helped us get the time and initial resources required for testing. Once people heard the first successful stories, more and more “passengers” started to “buy tickets”. Some of them even became our sponsors and funded the acquisition of new infrastructure. CIO commitment helped us bring in the last few groups that resisted.

Tools were not used until we were mature enough. Trying to use capture and playback tools without a proper and stable testing environment wouldn’t have given us permanent results. It was until we had the testing trains that we could aspire to use testing tools to help us accelerate the process.

Setting a yearly schedule for train departures helped the whole company to plan in advance. In fact, testing trains are used as a reference for diverse matters including the launching of marketing campaigns. You can hear business directors say: “We will launch the new credit card at train number six”. In this way, the whole company plan their yearly job on these testing/release trains.

Surprisingly, setting a robust testing environment and a yearly schedule helped other people and projects do a better job. As mentioned above, the Banamex University group now relies on our testing environment to train new personnel on most of the on-line applications they will use in real life. Needless to say that training is schedule according to our testing trains.

References

“Structured Testing of Information Systems”
Pol, Martin & Veenendal, Erik van
Ed. Kluwer

“Test Process Improvement”
Koomen, Tim & Pol, Martin
Ed. Addison-Wesley