4. What You Forgot When You Planned Your Brand New Application

26. Transform and Extend Your Business with ESB

10. THE HARD WORLD of MICROBENCHMARKING

8 Principles of Successful Load Testing | Software Testing Biases
SHARE THE AI IDEAS

AICOMMUNITY

THE NEW PERSPECTIVE
TECHNICAL

What You Probably Forgot When You Planned Your Brand New Application
Mike Druzhinin / Luxoft USA

The Hard World of Microbenchmarking
Andrey Akinshin / JetBrains

Semi Automation Testing
Nguyen Tan Trung / Luxoft Vietnam

Software Testing Biases
Alona Sokolova / Luxoft Ukraine

Transform and Extend Your Business with ESB
Supriyo Debnath / Luxoft Vietnam

Your Bugfix Pull Request Should Have Multiple Commits
Michai Kordas / Luxoft Poland

LIGHT READING

8 Principles of Successful Load Testing
Andrey Dmitriev / JUG.RU

Choosing a Programming Language Strategically
Dimitar Kapitanov / Luxoft Bulgaria

CORPORATE NEWS

Luxoft Videos

Luxoft Events
Imagine you are starting a brand new application. You’ve already started or best-case scenario, you have already implemented it and are deploying into production right now for the first time.
HELLO WORLD

There are two things which you might have missed:

1. There is a huge gap between an application that is ready for first deployment and an application that is ready for operation in production mode.

2. When real users start working with your application, you no longer have time to close this gap and it will hurt you.

We will go through aspects which should be considered when you are actually starting the architecture of your new application for production. If you already in production and forgot something, then better late than never. Everything mentioned below seems to be obvious but for various reasons are regularly missed.

This is not a comprehensive list, just a few of the most painful. The result of missing one is usually very simple, requiring emergency releases, pulling all-nighters and overtime.

SO, LET’S GET STARTED…

All the topics bellow are called non-functional requirements, aka NFR (or quality attributes), or in short, how a system is supposed to be:

“NON-FUNCTIONAL REQUIREMENTS SPECIFY CRITERIA THAT CAN BE USED TO JUDGE THE OPERATION OF A SYSTEM, RATHER THAN SPECIFIC BEHAVIORS”.1

Today we will stick to NFRs related to system execution time and support operations around them. There are a lot of other non-functional requirements around the evolution of systems, testability, usability, security, etc. - we will cover them next time. Tools to support particular NFRs will also not be covered because tools selection depends on many different factors. Also, I assume that the system covers functional requirements in full.

Topics for today:

- Performance
- Availability and reliability
- Supportability
- Monitoring

PERFORMANCE

Every engineer learned about the importance of performance requirements in kindergarten but probably forgot all about them at the beginning of each project.

How many of you can tell me how many users your application planned for? How many operations they will perform? Are you sure? Do you know for sure that your application can handle it?

There are a couple of aspects of performance that can be easily measured:

- Response time – how fast the client will get the response for request
- Throughput – the rate of successfully processed requests within the time unit

For both you need to know what is expected from the system. Response time not only concerns UI interactions, but also background or asynchronous processing - meaning processes which return results to the client later. Here are things which significantly influence both these aspects.

1. The first thing typically overlooked is data volume. Someone thinks that they will have lot of data and brings a cargo ship instead of a cart. It will cost later in terms of ownership and support. Others do not think about it at all. As a result, when the application warms up to real volumes at some point you find out that the main dashboard loads from one to five minutes and to handle all asynchronous messages for a day system you need two and a half days of non-stop work with 100% CPU load.

For some lucky people, the workload will grow gently and there will be time to make changes to the application. For others, it will be something like a 10x spike in one day and Russian roulette.

1 https://en.wikipedia.org/wiki/Non-functional_requirement
Instead of this game, just follow a few simple steps:

1. **Calculate** the maximum data volumes of major entities in the system.

2. **Generate** required data in the system (please don’t say that a senior developer doesn’t know how to create a million random records in the database).

3. **Check** how the system works with this data for major scenarios.

The user count in the system can have a significant impact on data volume, so you also need to know how many users expect to work with it not just now, but also in the near future.

Next, you need to know where you will run everything: network, hardware, virtualization. Don’t be put in a position where your development desktop is more powerful than expected production environment.

And last but not least for advanced teams: **combine data**, a real-like environment, and expected workload and run performance tests to make sure that you get the expected response time. Automation here helps make sure that you keep timings right after changes.

I don’t think I need to mention that if a system isn’t able to handle the expected load, it’s critical to start finding a solution before you have real users.

### AVAILABILITY AND RELIABILITY

This part is about making sure the system performs well when needed and managing the risk of failure.

The major question to answer here is when the system should be available. There are two parts to consider: time and level. For complex systems, different parts/components will have different metrics.

The first part is time or “business hours,” that is, when users will operate the system. For example, for an online system this is 24/7. For a system providing functions for a company department, then the time frame is the working hours of that department.

The second part is how critical failure would be. I prefer to break this into three levels:

1. **Critical** – If a component is down it will significantly impact overall business (like purchases which the client won’t receive because of it). Usually such an impact can be directly measured in terms of money lost.

2. **Important** – If a component is down for X hours it will impact major business use cases, but without direct money loss (for example, a user won’t have access to full text search but will still be able to use major features).

3. **Support** - No direct impact on business (such as a configuration tool or internal user management system).

The grades are empirical and you may rate some component as critical because downtime will impact reputation or some other criteria.

The major goal here is to have the number of critical parts as low as possible. After that, you need to make sure that critical parts are protected and will be available during business hours. For important components, you need to have procedures in place which guarantee that component availability will be restored in x number of hours (before it starts affecting business).

Next, you need to understand what can go wrong and how it affects your components (this is especially critical). Some common things that can occur include hardware issues with the network server, unavailability of external systems, or an unexpected load on your system. Do not forget that everything that can fail, will fail. The database host will go down, the external service will provide a response in 30 seconds instead of 100ms, and somebody will create a client for your API that makes 1000 additional requests just because of a human mistake in code.

For critical parts you need to have redundancy (at least two nodes) with data replication to be protected from hardware failures. The failure of the whole datacenter is rare, so the usual practice here is just to have a procedure in place to restore the system in another datacenter in reasonable time.

Another thing that is usually missed is what to do when you receive more requests than you are able to process (assuming that the system is already able to han-
dle a normal operational load). Good patterns here include configured automatic scaling for the component and the throttling of requests from a particular client. Throttling strategy can be very different but usually it is something like, “If a client sent more than 1000 requests in the last 5 minutes, send an error instead of request processing.”

You also need to be ready for similar throttling behavior from external systems. The CircuitBreaker pattern can be used here to proactively manage connections with external systems, handle downtime instead of random failure on requests, and come back to normal operations when the external system comes back.

So try to be paranoid in term of possible issues, but only for components that are truly critical, instead of focusing on making things that are used twice per month more reliable.

**SUPPORTABILITY**

After the point when your system is able to handle the expected load and the failure of things outside of your code, it is time to think about how to investigate issues in your own production code and be able to fix them easily. I know that you create perfect code without any defects, but sometimes somebody from your team might make a mistake.

The first step here is to maintain good logs. You need to have log files with information about all significant operations, interactions with external systems, background operations from start to finish, and any unexpected behavior like cache miss or internal queue overflow - everything which shows what is going on in the system.

It’s a good idea for each business transaction to have a unique ID and login as well. It will help you quickly segregate one user’s interactions from another, and one click on UI from another. User ID and major data items IDs are also good candidates to be logged.

One feature which can save a lot of time is audit tracking. Many “defects” are just what a user executed or changed. The next level is to be able to reproduce unexpected behavior with real user data. This needs to be carefully combined with security requirements because you do not want to provoke a leak of user data, but it really helps in complicated scenarios. Also, you never know when you will need this functionality.

One option is to export the required dataset from production into some QA environment and play with the data in QA. You will need to support the export/import of required data between environments.
Another option is to be able to log into the system as a particular user. This option really helps the support team when a user is trying to explain what he thinks is wrong. It allows the reproduction of the issue without any copy of data and in a real environment. One needs to be careful here not to expose a person’s personal information to support personnel and also to log all activities performed via this feature properly (with user ID and support person ID). Otherwise you open a huge security breach that can be used by anybody with the appropriate access.

The next advanced level is to put in place specific support functions for the application which give you the ability to get into the running application (like a dump user session), look at which information was put into caches, and restart a background task or restart it with adjusted parameters.

**MONITORING**

Monitoring is a crosscutting topic for all the previous ones mentioned.

Do you know what is happening with your application right now? How many users you have? Operations per second? What is the root cause of failure?

First what you need to have is a set of metrics for your application. Each important aspect from a performance and operational perspective should be monitored. You need to have statistics on how this metric changes over time. We are not only talking about infrastructure metrics, which everybody is aware of, but also application specific metrics. Examples of such metrics include the count of users logged in to the system, the count of requests of a particular type, request response time, and job processing time. With a good metrics solution you can create business-oriented metrics like orders or purchases per day.

Next, you need to have an alert configured for your metrics so that responsible persons will be notified when something goes wrong. Also, here it’s good to track all exceptions in the system and be notified of them (as simple option, just configure the log to send an email to the development team distribution list for each exception).

After you have it configured, you need to test the configuration to make sure that the alert really generates when something goes wrong and that people receive it and know how to react. Also, you need to make sure that people do not receive unnecessary alerts so that they don’t start to ignore them, because at some point they will ignore something that’s important. One of the worst-case scenarios is one where you have configured monitoring, but nobody reacts to alerts from it.

**SUMMARY**

In summary, just a few simple takeaways:

- Understand what is really expected from the application and have a plan to deliver on it.
- Do not overcomplicate things.
- Keep as close as possible to the real environment.
- Do not trust your experience and intuition - run tests.
- Keep monitoring and altering each important metric so that you are aware of failure before the user tells you.

Do not forget one last thing about IT: there are people doing backups and those that aren’t. If you aren’t, and you haven’t been affected by any of the above issues, it means you are either really smart or really lucky. Do backups and sleep well at night.

Mikhail Druzhinin has over 10 years of experience in software development. He has participated in full-cycle application development leading development groups and providing hands-on experience in a variety of programming languages and technologies, including solution design and architecture. He works as a trainer at Luxoft’s Training Center, where he conducts trainings in the area of architecture and Java development. His major areas of interest are cloud-based and distributed applications and DevOps.
DIVE INTO TECHNOLOGIES

ARCHCOMM
Architecture Community

THE NEW PERSPECTIVE
Performance is an important function of software in the modern world. Everyone wants to write fast programs. If you want to improve the speed of your code, the first thing you have to learn is performance measurement. Usually, it is enough to take a good profiler, take a snapshot of your program, find hot methods, and try to optimize it. However, what if you want to do a complete performance analysis and work with methods which take milliseconds, microseconds, or even nanoseconds? A profiler is not good enough in such cases – you have to know how to write a correct benchmark which provides a high level of precision. Most people write very simple Stopwatch-based measurements which often produce wrong results and lead to bad decisions.
WHY IS BENCHMARKING SO HARD?

There are many common benchmarking pitfalls. There are many situations in which you can deceive yourself. Let’s take a look briefly at some of them.

1. Many people still use DateTime.Now for performance measurements. This is wrong because it provides a pure precision level and has some timing-specific problems. Even if you are using DateTime.UtcNow, there are still many surprises which could arise on the level of the operating system. The correct API for high-precision measurements is Stopwatch.

2. Many people still use the Debug build for benchmarking. Never do it! In the Debug build, there are few important optimizations. It could differ from the release build (which you hopefully used in production) by ten to a hundred times when we are talking about micro-benchmarking. Moreover, of course, you have to run the benchmarks without an attached debugger.

3. Turn off any CPU-consuming apps. If you are using a laptop, keep it plugged in. If you are running benchmarking in the background from VisualStudio and go to look at funny pictures in a browser or watch a movie in your favorite multimedia player, such application use could significantly affect the results.

4. If an operation takes nanoseconds, it is impossible to measure it with a single invocation. You have to write a loop with many invocations and calculate an average.

5. Warm-up vs. cold start: Keep in mind that the first few iterations (or a couple dozen in some cases) can run a lot longer than the subsequent iteration. If you want to know the cold start performance, you have to measure only the first iteration (or the few first iterations). Otherwise, you have to do a warm-up first until a steady state is achieved.

6. If you run a benchmark several times, you probably expect to get the same results for each run. Unfortunately, it is impossible: you will get a new result each time. Thus, you have to do many repetitions of the benchmark and consider all the measurements.

7. Sometimes it’s impossible to express the performance of an application with a single number because you have to deal with distribution. If you’re lucky, it is a normal distribution with a small standard deviation. However, in practice, you could have strange distributions with huge variance and several maximums. Carefully look at the form of your distribution and use it in your conclusions.

8. If you’re running several benchmarks in the same process, they could affect each other. The order of your benchmarks (even if they are independent at first glance) could impact the results. There are many possible reasons: tricky interface method dispatching, autotuned GC, conditional jitting, and so on. Thus, it is better to run each benchmark method in its own process.

9. Beware of JIT optimizations. When you are trying to make your software fast they are your friends, but when you are benchmarking, they are your worst enemies. Such things as dead code elimination, inlining, and constant folding could totally spoil results of a synthetic microbenchmark.

10. Be careful with CPU effects. Things like the branch predictor, CPU cache, instruction level parallelism, false sharing (and many others; actually, it is a pretty long list) could play a major role in the speed of your code. Little changes in the input data (which shouldn’t affect performance from the point of view of asymptotic analysis) could easily modify the results by 10 times.

11. Your program could be run in many different environments. Just because it works fast on your computer with a specific version of an operating system and .NET Framework, it does not mean that your user will get the same performance.
BenchmarkDotNet TO THE RESCUE!

It looks hard to keep all the details in mind and write benchmarks without any mistakes. Fortunately, there is a good library for .NET benchmarking called BenchmarkDotNet (supported by the .NET Foundation). Of course, if you are using this library, it does not mean that all of your benchmarks will be correct; there is no magic here. However, it will help you to easily design a benchmark without stupid mistakes. Just mark your methods with the [Benchmark] attribute and BenchmarkDotNet will do all the following: automatically generate a separate project per method with important boilerplate measurement code; run it several times (the number of iterations will be automatically selected based on the first results) against all the requested environments; carefully calculate all the statistics; check some additional program characteristics like allocated memory per invocation; and produce a nice summary, reports in various formats, and beautiful plots.

This is a cross-platform library (you can use it on Windows, Linux, MacOS). It supports several runtimes (Full .NET Framework, CoreCLR, Mono) and languages (C#, F#, VB.NET). The powerful API will help you to easily compare different run configurations, enumerate sets of input parameters, and export results. There are also many built-in analyzers which look at the results and automatically warn you if you are doing something wrong.
However, you have to understand that it is just a tool. You still should know a lot about the ins and outs of modern runtimes and hardware. The final analysis of the results, which is the most important part of any performance investigation, is still your duty. Treat it responsibly.

Let's discuss a few more things which you should consider during benchmarking.

**KNOW YOUR ENVIRONMENT**

If you write some code and choose the input data, you still can't talk about performance in general without a chosen environment. Here is a small list of things which also could affect the performance:

- **Compiler**: Different versions of C# compilers produce different code with various sets of optimizations.

- **Runtime**: The Full .NET Framework, CoreCLR, and Mono: They have entirely different JIT logic and sometimes it is impossible to predict which runtime produces the fastest assembly code.

- **OS**: An implementation of the runtime can be different on different operating systems. If you are working with the OS level API, the final performance depends only on the current operating system.

- **JIT**: Even if you are working with the Full .NET Framework on Windows, there are still LegacyJIT-x86, LegacyJIT-x64, and RyuJIT-x64 (RyuJIT-x86 is coming). Each of the JIT compilers works differently, and it often makes sense to compare them.

- **GC**: The garbage collector also has some parameters. Real applications often produce a lot of memory traffic and spend time resources on GC. Thus, if you are benchmarking with Client GC and using Server GC on the production system, some performance surprises are waiting for you.

- **Toolchain**: You can use a classic approach with the JIT-compiler. Alternatively, you can use NGen and reduce cold start time. Or you can use .NET Native or coret which can significantly improve the performance of a .NET application.

- **Hardware**: Most developers probably understand that there is performance difference between Intel Core i3 and Core i7 because of the basic CPU characteristics. However, not all developers are thinking about how low-level hardware features can affect performance, SSE/AVX support, the size of out-of-order execution window, the internal logic of branch predictor, the size of CPU cache, and so on. It’s great if you can keep in mind where the bottleneck of your algorithms is and which hardware properties are important in the current context.

```csharp
[ClinJob, MonoJob, CoreJob]
public class Md5VsSha256
{
    private readonly byte[] data = new byte[10000];
    private readonly SHA256 sha256 = SHA256.Create();
    private readonly MD5 md5 = MD5.Create();

    public Md5VsSha256()
    {
        new Random(42).NextBytes(data);
    }

    [Benchmark]
    public byte[] Sha256() => sha256.ComputeHash(data);

    [Benchmark]
    public byte[] Md5() => md5.ComputeHash(data);
}
```
The first three attributes here mean that we want to run it against all the popular .NET runtimes: the Full .NET Framework, CoreCLR, and Mono. We should not run it three times: BenchmarkDotNet will do all the dirty work for us and print a summary like this:

As you can see, we have different performance pictures for each runtime. Here we are only talking about a few simple methods on a single laptop and single OS. In practice, when we are benchmarking a tricky logic that will run on different hardware and operating systems, the performance analysis could be much harder.

**STATISTICS IS YOUR FRIEND**

Imagine that you run your benchmark 100 times. Now you have 100 different performance measurements. What’s next? How do you deal with a set of time intervals instead of a single number? Here it would be nice to have basic knowledge of statistics. Here are some main things which you should consider:

- **Outliers**: You could have expected or unexpected outliers. For example, if you are doing a network benchmark, some of the invocations can take much longer than others. You should also consider expected outliers. Otherwise, you could have outliers that you don’t expect (for example, because of 3rd apps CPU usage). These should be detected and probably removed from the final results.

- **Median vs. mean**: Usually, these two metrics are very close to each other. However, if you have outliers, a big difference can arise between them. Carefully look at both values and don’t reduce the final summary table to only one characteristic.

- **Percentiles**: In the case of expected outliers, it makes sense to look at percentiles. For example, it can be a good marker if 99% of the target runs take fewer milliseconds than a threshold value.

- **Standard deviation**: Many real benchmarks have a lot of variance, which should also be considered. A random question: Is there any significant difference between two methods which take 100ms and 116ms (e.g. these are mean values)? You can’t give the correct answer without additional numbers. If the standard deviation is about 1ms, you can probably say that the first method is faster. If the standard deviation is about 80ms, it’s hard to compare them. You have to do some additional analysis.

- **Standard error and confidence intervals**: In a case where the variance is too big, confidence intervals can help you perform the analysis. You can increase the number of iterations until the standard error is too small, then you can work with the confidence intervals. By the way, if you have requirements for the standard error, it can be a good criterion for the amount of iteration needed. Often developers say “We will do 5 (10, 100, 100) iterations. That will be enough,” but it is the wrong approach in a general case without additional research.
• Distribution: When the distribution has only a single local maximum, it is easy. In real life, it is ok to have several maximum values. In this case, you can’t only work with mean values; you have to compare distribution.

Statistics is tricky. Just remember that it’s often not enough to peek at only one random metric; instead, you should make a conclusion based on all obtained measurements.

CONCLUSION

IT IS NOT EASY TO DESIGN A CORRECT BENCHMARK. IT IS ALSO NOT EASY TO ANALYZE IT AND MAKE CONCLUSIONS THAT WILL HOLD TRUE GENERALLY. HOWEVER, IT DOESN’T MEAN THAT YOU SHOULDN’T TRY! WRITE YOUR OWN BENCHMARKS, MEASURE THE PERFORMANCE OF YOUR CODE, AND TRY TO WORK WITH THE MEASUREMENTS AND ANALYZE THEM. ANY PERFORMANCE INVESTIGATION HELPS YOU TO BETTER UNDERSTAND WHAT IS GOING ON UNDER THE HOOD OF YOUR SOFTWARE AND IMPROVE YOUR PROGRAMMING SKILLS. JUST BE CAREFUL, REMEMBER THAT PITFALLS ARE EVERYWHERE, AND DON’T MAKE ANY HASTY CONCLUSIONS.

USEFUL LINKS

• github.com/dotnet/BenchmarkDotNet
• www.nuget.org/packages/BenchmarkDotNet/
• aakinshin.net/en/blog/dotnet/datetime/
• aakinshin.net/en/blog/dotnet/stopwatch/

ANDREY AKINSHIN

Andrey Akinshin is a .NET MVP, a Ph.D., a silver medalist of ACM ICPC, the maintainer of BenchmarkDotNet, a fan of micro-optimizations and one of the developers of Rider at JetBrains.
Nowadays manual and automation testing are done separately but there is a testing methodology that can bring them together to improve coverage, reduce cost and effort, and still maintain reliability.
WHAT IS SEMI-AUTOMATION TESTING?

Semi-automation testing is the combination of manual and automation testing. In manual testing, test cases are executed manually by tester without any tool support by interacting with the System Under Test (SUT) and checking the response from SUT to verify the test result.

In automation testing, the interaction with SUT is performed by the Automation Tool (AT). The tester requests the AT to execute testing, the AT then interacts with SUT, receives a response from SUT, and does a predefined check to verify then sends the test results back to tester. In this model, the tester has no interaction with SUT.

Semi-automation testing defines a new topology by which a tester can interact with both SUT and AT to perform more actions and verifications during test execution. In this model, the AT still communicates with SUT to execute testing, but for specific steps that need the tester's action the AT pauses and prompts the tester to execute and confirm the result. Once the tester has finished, he or she requests the AT to continue execution. Test results and logs are stored by the AT and can be tracked in cases of failure.

WHY SEMI-AUTOMATION TESTING?

When automation is mentioned, the assumption is that the AT can communicate to all components of SUT. In fact, many testing environments are a mix of multiple components (software and hardware) from different platforms and the AT can’t communicate to all of them. In addition, some verification (usability, animation, etc.) can’t be done automatically. Manual testing is considered time-consuming and prone to human error. Repetitive and accurate timing execution are also limitations of this testing type.

With semi-automation, testing can maximize coverage, reduce cost and effort, and improve testing quality because the missing coverage in automation is fulfilled by manual coverage. Both effort and the risk of human error are reduced and timing is granted by automation. Let’s analyze an example to gain a better understanding.

Suppose there is an Inventory Management page that needs to be tested. With different user privileges, the rights to delete different types of products are permitted accordingly. A sample test case for testing user privilege is as follows:
STEP 4
- **Manual**: After getting a passing result from the tester in Step 3, the AT prompts the tester to enter captcha and clicks “Confirm.” The tester then checks and confirms the login to the AT.

STEP 5 TO 6
- **Automation**: The AT proceeds to select Product A, click “Delete,” check deletion, log out and verify results, then add the result to the log.

The test case is now fully covered with minimal effort and maximum quality.

### ADVANTAGES/DISADVANTAGES

Let’s look at the disadvantages of manual and automation testing:

<table>
<thead>
<tr>
<th>MANUAL</th>
<th>AUTOMATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>HUMAN ERROR DURING EXECUTION</td>
<td>LIMITED USABILITY TESTING</td>
</tr>
<tr>
<td>HUGE EFFORT TO EXECUTE TESTING</td>
<td>NOT FLEXIBLE</td>
</tr>
<tr>
<td>NOT REUSABLE</td>
<td></td>
</tr>
</tbody>
</table>
Semi-automation testing provides solutions to improve both. Its advantages are listed below:

- It saves effort and avoids the risk of human error during execution because repeated steps are executed quickly and correctly by automation.
- Usability check is no longer limited, since it is done manually.
- There are flexible actions/verifications during execution.
- Automation script can be re-used in test cases.

The disadvantages of semi-automation testing include:

- The risk of human error still exists because manual testing is still performed.
- The tester still needs to monitor the test during execution.
- It can’t be applied with an AT that can’t handle interruptions while running.

**PRACTICAL APPLICATION**

In Automotive Instrument Cluster functional testing, functionality and hardware components in the instrument cluster need to be tested, including evaluating HMI display, sound, electrical and physical matter. The evaluation is based on cluster behavior upon receiving signals or under specific conditions.

The diagram below shows the environment setup:

- **Connection box** – the hub where all interactions with the cluster happen.
- **Signal** – simulated by Simulation software.
- **Power supply** – supplies power to cluster and other devices.
- **Switch controller** – used to automate the switch toggle.
- **External devices** – include external lamps and resistance input and multimedia devices from head unit, like radio, CD, MP3 player, navigation device, etc.
- **Measurement devices** – measure voltage, current, frequency and resistance, such as oscilloscope, multimeter, etc.

Semi-automation testing is the most suitable methodology for this environment:

- Components of this testing environment come from different manufacturers and platforms. Some devices don’t even have a communication port so it isn’t possible for the AT to control all components. That means full automation can’t be achieved but still automation can be used for components sharing the same communication protocol.
• Some information from the instrument cluster and external devices can’t be observed automatically. For example, the CD player is connected to the system, so when songs shuffle through the CD, the instrument cluster must sync to display the changes. The tool cannot observe the CD and cluster display – that must be done by the tester.

• The tester isn’t able to perform test steps that need accurate timing, e.g. activate two signals at the same time or within 200ms.

• The effort to manually control the simulation SW, power supply or switch controller can be reduced by automation.

For this environment, the AT is developed based on a framework that supports communication protocols from simulation software, switch control and power supply; hence, these can be controlled by the AT. Tester manually controls the remaining components.

When applying semi-automation testing, the testing execution sequence is as follows:

1. The tester selects test cases to be run and requests the AT to execute.

2. The AT controls power supply to provide power.

3. The AT controls simulation to send signal.

4. AT informs tester to interact with external devices, verify measurement devices data and cluster behavior, and confirms test result.

5. The test result and logs are stored in AT.

REFERENCES


TRUNG TAN NGUYEN

Trung is a software tester working on the Continental Project. Returning to Luxoft Vietnam from Luxoft Singapore, he is taking on the role of leading the offshore functional testing team. Trung has expertise in Automotive, Telecom, Network, Customer Relationship Management domains, and experience in working in multiple locales (Europe, US, Asia).
A WORLD FULL OF DATA

BACOM

LUXOFT Analysts Community

THE NEW PERSPECTIVE
Human perception is directly related to a person’s way of thinking and to the judgments an individual makes about a particular situation. Unfortunately, the thinking and judgments of the individual do not always reflect objective reality; systematic errors and deviations may be observed in them. In such cases we speak about cognitive biases.
“YOU SHOULD WORK TO REDUCE YOUR BIASES, BUT TO SAY YOU HAVE NONE IS A SIGN THAT YOU HAVE MANY.”

NATE SILVER

Cognitive biases are systematic errors in human thinking and patterned deviations that occur in certain situations that are based on distorted beliefs. Each person tends to create his or her own subjective reality, which depends on perception. This reality determines the person’s behavior in society, at home, and at work. Cognitive biases are often the cause of false or erroneous judgments, distorted interpretations, or irrational behavior.

Software is created by people, tested by people and, in most cases, used by people. Every person in this chain is subject to his or her own cognitive biases. The worst nightmare of any responsible tester is missing a serious bug: the tester works hard, thoroughly analyzing requirements, writing test cases, and carefully checking the product – and then a week after the release a user reports a critical problem.

How many bugs do we miss that are obvious to others? Our biases impact all tests, both manual and automated. Knowledge of our own biases and their consequences is the key to creating effective test cases, implementing them and finding bugs.

I would like to share with you some of cognitive biases that affect the work of a software tester and different methods that can reduce their negative effects.

1. CONFIRMATION BIAS

One of the most dangerous and yet common cognitive distortions is confirmation bias. This is the tendency of aperson to seek, interpret or give preference to information that is consistent with his or her own beliefs or hypothesis and refuse to see facts that contradict it. Confirmation bias is a filter that shows only the reality that meets the expectations of the person. This bias makes us think selectively and reduces the desire to look for additional facts. This bias is responsible for the existence of conspiracy theories.

How dangerous is the bias for testers? For example, let’s say that to test some functionality you decide to check with the developer to find out which areas, in his or her opinion, should be paid more attention to. Based on the information you receive, you prioritize testing.

The approach works well; you find serious problems in the high-risk areas and no problems in the low-risk areas, as expected. However, after the release users find a critical bug in a non-critical area. It is highly probable that you saw the bug or its manifestations but due to the fact that you considered the area bug-free, your brain simply filtered it.

A quick and easy way to reduce the risk of confirmation bias is to involve another person in testing the same functionality. It is highly probable that this person will have his or her own confirmation trap, but it is likely to be a different one than yours.

2. COGNITIVE DISSONANCE

Another equally widespread bias is cognitive dissonance, which is the state of mental discomfort caused by the confrontation of contradictory notions: ideas, beliefs, values, or emotional responses in the consciousness of an individual. The danger of this trap is that humans don’t like internal conflicts and humans are lazy. It is easier to adjust our opinion than to understand the causes of the conflict.

It is important for testers to recognize when they are in a state of cognitive dissonance and learn to ask questions before they make a decision. Once an individual makes a decision, he or she will find all kinds of confirmation proving the decision he or she made is correct. If during a test the tester needs additional information about a product and receives contradicting responses from two sources, it causes cognitive dissonance. It’s easy to choose the response from the most reputable source or the most expected response, but it is harder to spend time and effort to explore the matter in detail. If there is contradiction in some area then very likely there will be bugs in that area as well.

3. CONJUNCTION FALLACY

The simultaneous occurrence of two independent events cannot be more probable than the probability of either one occurring alone. People often forget about that and attribute a higher probability to a combination of events, mistakenly associating the number of events with an increase in probability.
This can impact the interpretation of documents in testing. When preparing test cases, the tester should understand how often a particular case will be used and how critical the case is.

To avoid the risk of conjunction fallacy, discuss details in advance and check statistics related to the use of a specific functionality.

4. ILLUSORY CORRELATION

Illusory correlation is the perception of a close relationship between variables when no such relationship exists in reality, or it is significantly lower than expected.

People make illusory correlations in many areas of their life. Everyone has heard the success stories of Bill Gates or Mark Zuckerberg who dropped out of college to start their businesses where they earned billions. We attach high importance to these stories and discuss them with our friends and acquaintances. Meanwhile, you never hear about the indolent students who did not succeed and did not create world-renowned companies. In the flow of information we catch only the most extraordinary cases; we scratch the surface, while ignoring hundreds or even thousands of stories of people who dropped out of college but did not meet the success paradigm.

To reduce the likelihood of this trap, you should use paired testing and involve other participants in test planning.

5. COGNITIVE ILLUSION

The simplest example of cognitive illusion is visual illusion. An individual looks at one thing, and the thing seems to be something completely different than in reality. This is the most widespread trap, and often explains how intelligent people can say silly things, because their brains don’t actually see what is happening.

To avoid this kind of trap the tester can try looking at the situation from a different perspective; however, this is very hard and requires experience and time.

6. SURVIVORSHIP BIAS

Survivorship bias is the tendency to underestimate data from “non-survivors” because of their lack of visibility. This logical trap was discovered during the Second World War, but it can occur in peacetime as well. During the war, US military authorities decided to reduce bomber losses to enemy fire and issued an order to determine which areas of their aircraft required additional armour. They started to analyze aircraft returning from missions and found holes in the wings and tail, so they decided to add armour to these areas. At first glance it seems quite logical, but it wasn’t the reality. Abraham Wald came to the aid of the military when he explained that they might have made a fatal error. In fact, the holes in returning aircraft signified their strong areas, not their weak areas. The bombers “wounded” in other areas – for example, the engine or the fuel tank – simply did not return from the mission.

Today we still need to think about the wounded-survivor principle when we jump to conclusions on the basis of asymmetric information from any two groups.

7. PATTERNS FROM NOTHING

People tend to find non-existent relationships. It is human nature to look at a set of data and find a pattern or images when looking at clouds. An experiment was conducted with humans and rats where a green or red light lit up in random order and where the green light lit up randomly in 75% of cases. The rats received a reward for guessing which color would light up. They quickly understood that the green light illuminates most often and always chose it, winning in 75% of cases. People searched for relationships and won in 65% of cases.

In testing, the invented relationship prevents a bug from being repeated. This is especially important during exploratory testing where attempts to confirm a pattern lead away from the main task.

8. FOCUSING ILLUSION

The more an individual thinks about something, the more important it seems. The individual does not pay attention to anything other than a particular area.

For example, the Mona Lisa gained popularity thanks to the manifestation of the focusing illusion in hu-
About 100 years ago the painting was completely unknown, but in 1911 it was stolen. Tabloids publicized the portrait to mock police. After a few months of media exposure, everyone knew the painting.

The focusing illusion can occur in testing when an uncritical bug requires extensive joint work between the tester and the developer. As a result, the tester begins to believe that the uncorrected product cannot be released, though the bug priority and importance are low.

9. NEGATIVITY BIAS

The human brain is designed to pay more attention in the short-term to the negative rather than the positive. Focusing on the negative leads to the expectation of its manifestation.

Negativity bias can occur in testing in two cases:

1) The fear of change, which leads to saying “no” to any changes. On the one hand, this is good because it forces supporters of innovation to think over their arguments and rethink the need for the change. On the other hand, it can inhibit development as a whole.

2) Lack of trust of developers. On the one hand, the work of the tester is to “trust but verify.” At the same time, excessive checks require time, which the tester may not have. For example, it is very difficult for many testers to end task testing if they did not find a bug, even a small one.

10. EMOTIONS

Emotions, though they are not a cognitive bias, affect human perception similarly. Therefore an individual can see the same thing in a positive or negative light depending on his or her emotional state. Most importantly, emotions affect focus and concentration. When a person is very happy, he or she is joyful and perhaps really doesn’t want to delve deep into the maze of a complex and boring algorithm.

As a rule, developers test their code while writing it or immediately after. In this case, coding and testing occur in one emotional state, and therefore the same blind spots exist. Thus, it’s a good idea to have the code tested someone who didn’t write it.

CONCLUSION

People will always lean towards taking into account conclusions and judgments that support their beliefs without spending the time and effort it takes to deliberate on concepts to see what they actually agree with. Beware of beliefs, assumptions and categorical assertions - both those of your own and those of your team members. Communication, discussion and critical thinking skills can help reduce the effect of cognitive biases.

There are many more fallacies and distortions not mentioned in this article, but you can find them here: www.fallacyfiles.org/taxonomy.html.

I would like to end the article on a positive note:

AS LONG AS DEVELOPERS ARE SUBJECT TO VARIOUS FALLACIES AND DISTORTIONS, THERE WILL ALWAYS BE A NEED FOR TESTERS.

USEFUL LINKS

• changingminds.org/explanations/theories/theories.htm
• www.fallacyfiles.org/taxonomy.html
• leanpub.com/thepsychologyofsoftwaretesting

ALONA SOKOLOVA

Alyona Sokolova started working in the IT sphere in 2007 as BI Developer. Since 2009 she has been promoted from Junior Manual QA to Automation QA to QA Lead/Test Manager. She is interested in process optimization development and product quality improvement.
The other day I met an old friend who invited me to do a technology review of her new e-commerce business idea. After thoroughly assessing the architecture, I suggested that she implement an Enterprise Integration layer in the near future. For the targets she had set for business it was really necessary. But my friend comes from a business administration background, so she simply asked, “Why should I invest millions of dollars in implementing an ESB layer?” I answered: “If you dream of a billion customers for your business, it’s worth investing a million dollars.”
OVERVIEW

Today businesses across all industries need to respond to the threat of becoming obsolete. The rise of mobile channels and cloud services has completely changed the dynamics of customer interaction and asset growth. Since the last decade, we have experienced a transition from discrete hubs to the Internet of Things. If a business wants to stand tall among the competition, it needs to strategize with Hybrid Enterprise Integration. By adapting to Enterprise Integration, you can be absolutely sure of being provided three things:

1) A reliable engine which is designed to connect your system of interaction with your system of records, your system of records to your system of sales, and so on

2) A scalable, lightweight, highly available, maintainable application programming interface at the core of your business data center

3) A secure connection to extend your business to the Cloud and to external business partners precisely for B2B communication.

Enterprise Service Bus (acronym: ESB) is the term used for the architectural layer of Enterprise Integration. ESB provides connectivity between enterprise applications and facilitates universal data transformation in any system architecture. Now businesses can eliminate tightly coupled, less secure, crash-prone, point-to-point connections and processes regardless of IT infrastructure, server platform, connection protocol and data format. There are multiple ESB solutions available on the market by leading IT organizations like IBM, Oracle, TIBCO, Microsoft, Red Hat, Mule Soft, etc.

I will narrate the recommendation of ESB through two different business scenarios.

THE PROBLEM

Company X decides to expand its business from a core pharmacy chain to a retail mart. Although the present infrastructure and line of business should not be affected, in the initial stage of this new expansion there should be fair visibility of existing customers of the company. The company already has a stable IT infrastructure for its Healthcare business, so the company decides to build an entirely new IT infrastructure for its expansion to the Retail and Consumer Goods business, thereby ensuring zero impact to the existing setup.

Now the challenge comes of achieving visibility of existing customer data from the existing IT setup to the new IT setup. Customers should get information about the new services offered by this retail business; the privileged customers of pharmacy business should get some benefits from the retail services; the sales agents of the retail business should have enough details about customers from the pharmacy business; new customers from the retail business should be encouraged to register with the pharmacy business and vice-versa, and so on. Moreover, the company needs to maintain a unified visibility of customers from both the pharmacy business and the retail business sides.

THE SOLUTION

To build a bridge between the pharmacy system and the newly proposed retail system, Company X decides to implement ESB as an IT solution. The pharmacy system maintains its customer information in a centrally-managed database and all interaction is processed through Unix server. The newly proposed retail system also develops its own database for customer information. As the retail system needs information about existing customers from the pharmacy system to begin its initial level of operation, the first ESB solution is built to provide visibility of existing customers to the new system.

#1. EXPANSION FROM PHARMACY CHAIN TO RETAIL MART

BACKGROUND

Company X is a renowned establishment in the pharmacy business with close to 5000 stores. It is expanding nationwide and has more than 70 years of market background. It specializes in healthcare and wellness products, consulting medical treatment through in-store clinics, filling prescriptions, selling proprietary drugs as prescribed and other pharmacy services.
This process is operated in batch mode to transport the data of existing customer information from the pharmacy system to the retail system.

Next, there is a parallel requirement where new customers of the retail business should get a notice about the pharmacy channel and similarly, new pharmacy customers should be encouraged to register with the retail channel. Now the company wants to make sure that each customer is registered or informed about both lines of business. To achieve that end, two more ESB solutions are built.

This process captures customer activity information from the pharmacy system and sends it to the retail system in real time.

Similarly, this process captures the newly registered customer information from the retail system and sends it to the pharmacy system in real time.

Now it’s time to discuss the third and probably the most important solution built on ESB for this new proposal - the operational master data hub. Company X must have unified visibility of all customers from both the business systems, pharmacy and retail. The capacity to transfer data and at the same time avoid data loss are of prime concern. To achieve this highly complex functionality in a simple way, the third ESB solution is developed:

This process is implemented in batch mode so that once a day, preferably at night, it propagates all the operational information from both business systems to the independently managed operational master data hub. It is also capable of ignoring duplicate or already pushed data while retrieving data from both business systems. Upon completion, the business owner’s group receives an email with basic audit information.

So logically, two different systems with completely disparate IT infrastructures can communicate with each other and provide visibility to each other without compromising the security of their own architecture with the help of Enterprise Service Bus. All of these solutions are highly scalable, so that if the company decides to expand further into new sectors, the system can easily accommodate new IT systems and provide a strong foundation for business.
first, the data mediation has to be maintained as it is among the applications, and second, a strong security firewall has to be built between the cloud-space and the on-premise center.

THE SOLUTION

Company Y implements the ESB solution to maintain enterprise-level security between cloud and on-premise services and to mediate data as it used to before migrating to the cloud to ensure no damage in current workflow.

Prior to moving data from the on-premise system to the cloud-space, the foremost requirement is to enable an enterprise standard security gateway between the existing infrastructures to the new SaaS infrastructure. So the first ESB solution is built to achieve this functionality.

#2. MIGRATION FROM ON-PREMISE PLATFORM TO CLOUD-SPACE

BACKGROUND

Company Y is a global provider of insurance, annuities and employee health benefit programs. It has more than 100 years of market experience and has expended to over 40 countries.

THE PROBLEM

Company Y decides to restructure some of its IT system components, like its Customer Relationship Management system, Alerts & Notification services, and Application Processing system by implementing SaaS. The company faced challenges earlier like the diminished availability of services worldwide, additional maintenance costs for discrete infrastructure, extra manpower in IT support and so on. In order to overhaul the current infrastructure and make it more efficient with fewer company resources, a decision was made to migrate some components to a cloud service.

For obvious reasons, the company has to think about data integrity before migration. The present architecture is supported by an on-premise data center and is quite heterogeneous in nature. Detaching some components from the present setup and migrating to the cloud presents some strong challenges to its engineers. Two things become very important at this stage:
This process works as a DMZ (demilitarized zone) between the on-premise center and the cloud-space to ensure maximum security. It runs in an independent server and works in a gateway mode. In order to transfer data to the cloud service or vice-versa, this layer has to be crossed. It takes care of authentication, authorization and policy-based security enforcement to handshake with different cloud services. It also reports on whether or not the secured connection has been established or any security violation is noticed.

The next target is to achieve data integrity between the cloud and on-premise services. The on-premise service includes a variety of service architectures, from REST Service to Big Web Service and from windows service to legacy application. Once detached, the heterogeneity of interfaces needs to be conjugated to ensure zero impact in functionality. ESB is able to smoothly take care of those challenges. Based on communication protocols, end interface category and message formats, solutions are built which directly invoke the security gateway and thereby the cloud service.

This process works in publish-subscribe mode and sends publications with events from different cloud services to on-premise storage. The data is always published via DMZ to maintain security between the cloud and the on-premise server. Therefore, highly complex service components are smoothly migrated to the cloud space with the help of ESB solutions. The key part of any cloud service is providing enterprise standard security using ESB that can be easily achieved without any compromise in performance. In the future, should the company decide to add new components in cloud, these can also be integrated with the existing system both in the cloud and on-premise with the help of present ESB solutions.

SUMMARY

As an ESB developer in IBM stack, I have experienced the large scale business transformation achieved by ESB implementation. In this article, I did not talk about the features that IBM Integration Bus provides in ESB scope. In the next edition, I would like to highlight some key features of IBM offerings for enterprise integration. Until then, this article sums up my recommendation for all business owners and enterprise architects to take advantage of ESB.
When a code that you wrote is not committed to a repository, it effectively doesn’t exist, since it isn’t visible to anyone except for you. Moreover, Git doesn’t take care of such files and they may end up lost with no hope of recovery. If we commit often it’s easy to go back to the previous state when something fails. We can share changes regularly and receive timely feedback about ideas and solution architecture. Other people know what is going on. There are less merge conflicts and less duplicate work. Exact statuses are shared even before stand-up meetings. All of these are basic continuous integration principles and we cannot say we have proper CI when people haven’t committed changes for days.
COMMIT EARLY AND OFTEN

If it looks like the rule is to commit early and often, we end up quickly with dozens of commits for relatively easy changes in our GitHub pull requests. This enables us to track progress, see the chosen development path and review only the new stuff that gets added instead of reviewing the entire change again and again. Perhaps contrary to the above, many projects have adopted an only one commit per pull request rule and they always require the use of squash commits. The main reason is to have a short, clear and readable history of all significant changes that serves as project documentation. The idea is that such history better explains all the WHYs, which are much more important than HOW exactly the change was made, and which steps were taken. Simple history is easier to browse and comprehend from user interfaces (like GitHub) without resorting to advanced techniques involving command-line. When a project is large, having a leaner and smaller repository will also be more performant for all the contributors. Lastly, we can apply the single responsibility principle in various contexts, even for a pull request. This means that we should be focusing only on one concern and doing it great.

ALWAYS CHOOSE THE SWEET SPOT

Generally the above convinces me and I use squash commits. I always used to do it, even during development, with git commit --amend and git push --force. Now I understand that this isn’t ideal and there are many exceptions. For me, a couple of well-made commits may simplify review significantly and make the history even clearer in certain specific cases, particularly non-trivial bugfixes or features that should be verified at the system/integration level.

Squashed commits:

1. Test, bugfix and refactoring mixed together

Sample solution with 3 commits:

1. Test that reproduces a bug at the system/integration level. For this commit, the build should fail for the proper reason, and should have a red tick in GitHub.

2. The simplest possible bugfix, so that no one has any doubts where the bug was, should already have a green tick on GitHub.

3. Minimal refactoring is needed to pass code review; additional refactoring can be done in other PRs before/after.

Ideally, a responsible code reviewer should make sure that the test really reproduced the bug and failed for the proper reason. It is too easy to create a test that always passes, even if it failed initially. Having a commit with just a test can be evidence that throughout changes to PR the test is still failing, and enables the reviewer to easily verify assertion/error message (which is also very important!).

Moreover, a commit with just a bugfix makes the real functional problem immediately obvious. It doesn’t get lost between refactoring or other unrelated changes and can be reviewed separately with greater care.

Of course, you can still squash commits just before merge if reasonable. GitHub even recently introduced the “Confirm squash and merge” button that preserves development commits, but only one squashed commit with an adjusted title and description goes to the master branch.

So don’t blindly follow the rules about making just one or many commits per PR and always choose the sweet spot.

MICHAŁ KORDAS

Michał is a fully cross-functional member of a strongly agile team. A software developer, “quality maniac” and agile enthusiast, his particular focus is on well-written code, good requirements and efficient process. Unlike the majority of QAs, he believes that being a truly good QA is not only about having testing-related skills. It’s also about being great software developer with all the knowledge necessary to deliver production code of the highest possible quality.
What does the customer want? Do you believe your customer is really interested in the number of tests that you have successfully completed? Is the number of tests in the database really of significant value? In fact, the number of successful (or unsuccessful) tests is not something the customer wants to know. They want to know that substantial work has been done to ensure that their application satisfies certain requirements. If we refer to load and performance requirements, then today we will talk about testing applications under load. We will consider all tests on the basis of several principles.
**PRINCIPLE #1: SEE THE FULL PICTURE**

It isn’t enough to build plans only on the basis of data provided by the customer. Oftentimes additional research needs to be conducted to collect all the puzzle pieces. As a rule, component diagrams help identify the main areas of research. For example, based on the diagram below we can assume that some data migration or reconciliation will be used and the search and interaction with third-party systems (Active8) will be implemented in some way.

**THE ARTIFACTS THAT WE PROVIDE**

Before we start testing we need to determine which goals we want to achieve.

**PRINCIPLE #2: REGISTER ALL PARAMETERS EXPECTED FROM THE SYSTEM**

Let us assume that a system just being developed must withstand a load of say, 3,000 queries per day. Moreover, the queries can be of one type only and will barely differ from one another.

Great, this data is enough to build a workload model. The first artifacts (something that has value in the software development process, for example code, documentation, configuration files, etc.) are non-functional requirements, or NFRs.

**NFRs**

When we develop a system, we write the code on the basis of certain requirements called Functional Requirements (FR). When we implement a system and the requirements do not directly affect the functions performed by the system, these are called Non-Functional Requirements (NFRs).

In other words, it is a list of all possible system requirements not related to the functionality of the system. It may include:

- security requirements (which rights will be granted to which user roles);
- reliability requirements (how the system will behave in case of failure or the failure of any of its parts);
- system speed requirements (within what timeframe queries must be processed).

NFRs can be quite bulky. It may look like this:

<table>
<thead>
<tr>
<th>PHASES</th>
<th>DESCRIPTION</th>
<th>DATA VOLUME</th>
<th>DATES</th>
</tr>
</thead>
</table>
| Phase 1 Preparation | Set up data sources on onsite test environment:  
1. BORIS (ORACLE)  
2. ONPS (ORACLE)  
3. Cisco ISC (SYBASE)  
4. ProJEN (ORACLE) | 14.500 Engineering Orders | XYZ               |
| Phase 1 Execution | Executing data migration on onsite test environment | 14.500 Engineering Orders | XYZ               |
| Phase 2 Preparation | Set up all data sources on onsite test environment:  
1. BORIS (ORACLE)  
2. eDesigner Evolve VPN (MYSQL)  
3. eDesigner Evolve EWAN (MYSQL)  
4. ONPS (ORACLE) | 34.500 Non-Engineering Orders | XYZ               |

This table shows the number of different types of objects to be migrated from other systems.
This table shows the number of user scenarios to be executed per hour and the maximum response time.

<table>
<thead>
<tr>
<th>#</th>
<th>AREA</th>
<th>PHASE</th>
<th>DESCRIPTION</th>
<th>REPORT CRITERIA</th>
<th>TX/H (AVG)</th>
<th>RESPONSE TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FUF</td>
<td>PHASE 2</td>
<td>ORDER PERFORMANCE WITH RESPECT TO THE ORDER TARGET DATE</td>
<td>MONTHLY OR QUARTERLY PERFORMANCE</td>
<td>16</td>
<td>1 MIN</td>
</tr>
<tr>
<td>2</td>
<td>FUF</td>
<td>PHASE 2</td>
<td>ORDER PERFORMANCE WITH RESPECT TO THE ESTIMATED DELIVERY DATE - COMMITMENT DATE</td>
<td>MONTHLY OR QUARTERLY PERFORMANCE</td>
<td>16</td>
<td>30 MIN</td>
</tr>
<tr>
<td>3</td>
<td>FUF</td>
<td>PHASE 2</td>
<td>ORDER VOLUMES - CREATED</td>
<td>MONTHLY OR QUARTERLY PERFORMANCE</td>
<td>16</td>
<td>5 MIN</td>
</tr>
</tbody>
</table>

This table shows the number of user scenarios to be executed per hour and the maximum response time.

STRATEGY

While NFRs are something we should test, the strategy is the document that answers the question of how to test them. On the one hand, an optimal strategy shouldn’t miss any important tests. On the other hand, it should ensure solution quality with minimal effort on the part of the project team. It is easier to do this if we know the entire scope (a set of all tasks to be performed) and set priorities for each item.

Unfortunately, tests are often performed in an environment that differs from the environment of the production system.

PRINCIPLE #3: ALWAYS SPECIFY LIMITING VALUES AND ACTUAL RESULTS

To do this you need to know the characteristics of the two environments.

PLAN

This is a typical work plan prepared in Excel, MS Project, JIRA, etc. We will not discuss it in detail here.

REPORT

This is the most important part of the testing process. The report presents the results with detailed information about how the tests were run, in what environment they were run and the results that were obtained.

<table>
<thead>
<tr>
<th>APP CPU MODEL</th>
<th>8-CORE INTEL XEON E5-2650 V2 PROCESSOR, 2.60GHZ</th>
<th>8-CORE INTEL XEON E5-2640 V3, 2.60GHZ</th>
<th>ACCORDING TO SPEC BENCHMARK CPUS ARE ALMOST EQUAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>APP QUANTITY</td>
<td>2</td>
<td>1</td>
<td>NOT EQUAL (PRODUCTION ENVIRONMENT WILL BETTER PROCESS THROUGHOUT WORKLOAD THEN OFFSITE SVT ENVIRONMENT)</td>
</tr>
<tr>
<td>APP MEMORY</td>
<td>64</td>
<td>48</td>
<td>EQUAL (FROM JVM PERSPECTIVE)</td>
</tr>
<tr>
<td>APP LOCAL STORAGE</td>
<td>300</td>
<td>160</td>
<td>EQUAL</td>
</tr>
<tr>
<td>ODB CPU MODEL</td>
<td>12-CORE INTEL XEON E5-2697 V2 PROCESSOR, 2.70GHZ</td>
<td>10-CORE INTEL XEON E5-2660 V3, 2.60GHZ</td>
<td>ACCORDING TO SPEC BENCHMARK CPUS ARE ALMOST EQUAL</td>
</tr>
<tr>
<td>ODB QUANTITY</td>
<td>2</td>
<td>1</td>
<td>EQUAL (DUE TO ACTIVE-PASSIVE CONFIGURATION)</td>
</tr>
<tr>
<td>ODB MEMORY</td>
<td>128</td>
<td>256</td>
<td>EQUAL (CONFIGURED AS ON PRODUCTION)</td>
</tr>
<tr>
<td>ODB LOCAL STORAGE</td>
<td>300</td>
<td>300</td>
<td>EQUAL</td>
</tr>
<tr>
<td>ODB ORACLE RAC</td>
<td>Y</td>
<td>N</td>
<td>NOT IMPORTANT (DUE TO ACTIVE-PASSIVE CONFIGURATION)</td>
</tr>
<tr>
<td>DB CACHE SIZE</td>
<td>75G</td>
<td>75G</td>
<td>EQUAL</td>
</tr>
<tr>
<td>SGA MAX_SIZE</td>
<td>85G</td>
<td>85G</td>
<td>EQUAL</td>
</tr>
</tbody>
</table>
FOCUS ON DETAILS

When planning a test, pay attention not only to business scenarios but also to data amount and structure. The number of scenarios to be executed per unit of time and the amount of data to be put in our database before testing can be estimated by making a selection in the existing database for the previous period. If such a database does not exist or if a query on it is impossible, then you can use projected estimates.

Most important business scenarios used in load testing can be determined by looking at the frequency of execution and assessing the complexity of business logic or database queries. Why do we need to know the amount of data in the database?

1. in order to estimate how long the data migration from the old to the new systems may last (if applicable).

2. We need to know to be able to test queries at appropriate volumes, because queries may be executed differently in a dataset of 100 records versus a dataset of 100 million records.

3. We need to know how many users will be using the system concurrently and what reports they will be building. We even need to know what data they will be searching for.

FROM THE NUMBERS TO THE TEST STRATEGY

You need to balance the conditional “utility” of the scenario when selecting it for load testing. The price of the test is made up of the costs of developing a test and the costs to support it in the future. The application will change, the scenario will change, and the corresponding test will need to be changed as well. This is why this price should be included in the plan from the very beginning. If we talk about selecting scenarios to implement as tests, we need to understand which scenarios best demonstrate the system’s behavior.

Of the entire set of scenarios performed on the system we need to select only those that 1) load the system most of all, 2) are performed frequently and 3) have an acceptable price range for development and support. For example, if 40,000 queries of a particular type are expected during system operation but they don’t load the system heavily, this means other scenarios need to be considered. Similarly, if the test is difficult to implement and potentially highly sensitive to changes in functionality, then it is better to search for other scenarios comparable to the complexity of the execution but easier to implement and support.

REPORT FOR THE CUSTOMER

Let’s assume that we have successfully implemented a strategy, carried out the necessary measurements and are ready to present our results. What should be included in the final report?

<table>
<thead>
<tr>
<th>#</th>
<th>NFR</th>
<th>SCENARIO</th>
<th>DESCRIPTION</th>
<th>STATUS</th>
<th>ACTUAL DURATION</th>
<th>NFR DURATION</th>
<th>DATA VOLUME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>#NFR-DATA MIGRATION</td>
<td>DATA MIGRATION PHASE 22 (ITERATION 22.2)</td>
<td>PHASE 22 PATCH2</td>
<td>PASSED</td>
<td>3H 15 MIN</td>
<td>24H</td>
<td>86G</td>
</tr>
<tr>
<td>2</td>
<td>#NFR-DATA MIGRATION</td>
<td>DATA MIGRATION PHASE 22 (ITERATION 22.3)</td>
<td>PHASE 22 PATCH2</td>
<td>PASSED</td>
<td>4H 28M</td>
<td>24H</td>
<td>86G</td>
</tr>
<tr>
<td>3</td>
<td>#NFR-DATA MIGRATION</td>
<td>DATA MIGRATION PHASE 22 (ITERATION 22.4)</td>
<td>PHASE 22 PATCH2</td>
<td>PASSED</td>
<td>4H 20M</td>
<td>24H</td>
<td>86G</td>
</tr>
</tbody>
</table>
The level of detail depends on who is getting the report, but the general rule is as follows: At the beginning, the report should contain high-level information about the tests, including a description of system configurations, tests performed, assumptions, and summary tables on the number of tests run, passed, and not passed. Then more detail should be provided on each measurement with numbers and diagrams.

The report should end with recommendations on scalability and potential risk areas.

**PRINCIPLE #6: PROVIDE A FORECAST FOR SYSTEM SCALABILITY**

<table>
<thead>
<tr>
<th>SCENARIO</th>
<th>100%, 2 HOURS</th>
<th>120%, 2 HOURS</th>
<th>150%, 2 HOURS</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TARGET</td>
<td>ACTUAL</td>
<td>TARGET</td>
<td>ACTUAL</td>
</tr>
<tr>
<td>XXXXXXXXXX</td>
<td>20</td>
<td>20</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>XXXXXXXXXX</td>
<td>20</td>
<td>20</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>XXXXXXXXXX</td>
<td>2516</td>
<td>2512</td>
<td>3019</td>
<td>3019</td>
</tr>
<tr>
<td>XXXXXXXXXX</td>
<td>1132</td>
<td>1109</td>
<td>1358</td>
<td>1332</td>
</tr>
<tr>
<td>XXXXXXXXXX</td>
<td>12</td>
<td>12</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>XXXXXXXXXX</td>
<td>1400</td>
<td>1400</td>
<td>1680</td>
<td>1677</td>
</tr>
</tbody>
</table>

This table shows three measurements at different loads (100, 120 and 150 percent of full load).

<table>
<thead>
<tr>
<th>SCENARIO</th>
<th>APPLICATION SERVER NODE 1</th>
<th>APPLICATION SERVER NODE 2</th>
<th>DATABASE SERVER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100%</td>
<td>120%</td>
<td>140%</td>
</tr>
<tr>
<td>CPU UTILIZATION</td>
<td>10.7%</td>
<td>14.1%</td>
<td>14.4%</td>
</tr>
<tr>
<td>MEMORY UTILIZATION</td>
<td>44%</td>
<td>54%</td>
<td>75%</td>
</tr>
<tr>
<td>MEMORY USED</td>
<td>43.5GB</td>
<td>48GB</td>
<td>55GB</td>
</tr>
</tbody>
</table>

This table shows the increase of CPU utilization on different servers depending on the load level.
PRINCIPLE #7: PROVIDE DETAILED INFORMATION ON SYSTEM STATUS IN THE CONTEXT OF TIME

KSar can build such graphs based on the data collected by sar utility.

JVM GC log activity graph.
PRINCIPLE #8: SEND A CHECKLIST OR SURVEY TO THE CUSTOMER BEFORE YOU START LOAD TESTING

Here is a checklist of questions that you can ask a customer to identify the major system components requiring load testing. This is not a complete list but rather a set of guidelines to help you find the right questions to ask your customer. Good luck!

1. How big is the database? How will it change by the date of release?
2. Which entities does the database contain? How many?
3. How many users will be using the system, and how many will be using it at the same time?
4. What data migrations will take place? What types of data and amounts will migrate?
5. What user scenarios will be used? Which are executed most often? Which are the most difficult to execute?
6. What batch jobs will be implemented? How often will they run?
7. What reports will users build? For which periods? What additional parameters will be used?
8. What search criteria will be set by users? How often will they search in this way?
9. What types of queries to the database will be used most often?
10. What are other third-party systems that need to be integrated?

FOR FURTHER READING

Atul Gawande, The Checklist Manifesto: How to Get Things Right.

ANDREY DMITRIEV

Andrey is a Java Programmer with experience developing JDK in Sun and Oracle companies. Andrey led an Android development team at QuickOffice and later worked to develop an app for Google for a year. Andrey has participated in several load testing projects of various sizes and in different roles, from Line Engineer and Team Lead to Test Manager; hence, he has experience in both building test plans and implementing specific steps, such as writing tests and collecting and analyzing solution metrics. In addition, he can assess the entire process of load testing from the perspective of various roles in the project. In his spare time he teaches on the Mathematics and Mechanics Faculty of St. Petersburg State University and organizes conferences for developers.
WE ARE YOUR SUPPORT

xtf: champions
Trusted Delivery Managers Community
If we ask programmers which programming language we should learn, we get many different answers. Everybody has his or her own preference. Everyone protects and promotes his or her language of specialization. This is natural human behavior – if we invest years in something, we tend to protect that investment, even if it was made by accident. We all have witnessed emotional discussions about which programming language is superior. Opponents quickly raise argument after argument with religious fervor sparkling in their eyes. (“Language X is the very height of software evolution – everything else is a step back.”) In the end, no one manages to convince anyone else. Such discussions are a waste of time, so let’s remove the emotional side of the equation and look at the question from a practical perspective.
IT IS AN INVESTMENT!

Learning a programming language is actually an investment - it requires significant time and effort. If we are going to make investment, we want to have a return on it, right?! Let’s assume we are (or want to be) professional programmers – to write code for a living. Which programming language should we invest in? The answer is, probably one of the most used and popular languages. Let’s check out the collective wisdom on popular languages. Here is the language popularity indices PYPL (http://pypl.github.io/PYPL.html):

This is the trend of popularity of programming language for the recent years, that gives us our first selection strategy:

TARGETING POPULAR LANGUAGES

These languages have high job demand and a large online community presence. At the moment, Java is the most popular language. In general, the most popular programming languages are also the most well-paid. Looking the chart above, we notice the stable rise of Python. JavaScript also has a strong position as the dominant language for web browsers. It can also run on the server side, but there is a trap that comes with it that we will discuss later. Keep reading.

Delphi and Perl (added for comparison) have long been decreasing in popularity, so it’s best not to invest in them now. The pay is higher for the compiled languages used by big corporations (Java), simply because big corporations have a lot of funds. On the other side, startup companies prefer scripting languages like Python and Ruby.

But what if we are not mainstream and we don’t really care about what language the majority uses? Here is another strategy:

TARGETING NICHE LANGUAGES

We have all heard the stories about the programmer that knows a rare language (Cobol, Fortran, Abap, etc.) who gets piles of money because almost no one knows it anymore. Is it a good strategy to target a rare language? In most cases, no. The competition is low, but you would be trapped supporting decades-old software with little to no online help.

Moreover, there isn’t any margin for creativity (your job is to just keep the thing working). Old systems are gradually being rewritten to modern languages, causing the general demand for niche languages to decrease.
There is one exception. If we are old generation programmers no longer interested in learning the next fashionable thing (which appears every year and does the same thing), then the niche language is an option. Also, it’s an option if we are just looking for a calm job to last until retirement. But let’s say we are far away from retirement. We already have one primary language in our arsenal. Why not experiment with few other languages and pick up another one? This is the next possible strategy.

LANGUAGE JUMPING

We know some of programmers who jump on every new language bandwagon they read about: C, C++, Delphi, Assembler, Java, Ruby, JavaScript, Scala and so on. They invest tremendous time in experimenting. Their resumes contain a long list of language names. Does this behavior pay back? The common argument for learning another language is that this will make us better programmers and will enlarge our vision. Every modern programmer has to be polyglot and to know multiple languages, right? Hmm. Do you remember that thing about investment? Time is our most precious resource. That’s why language jumping is unwise.

Learning an additional language makes sense only if the new language is applied immediately (or soon). If we don’t apply our new Ruby skills after we acquire them, they get rusty. This is the same principle as with learning human languages – if we don’t practice it, we forget it (use it or lose it). Learning Ruby in our spare time will not help on our current Java project. Without applying it, one year later our knowledge about Ruby will be mediocre at best, and if we start a project with Ruby at all, we would have to relearn much of it (investing time in it again).

Experimenting with a new programming language can show us different ways of doing the same things, but from the investment point of view it doesn’t pay. To become better programmers, we need to write more code in the language we know best. Let’s not forget that some people don’t strive to be so pragmatic. They just want to have fun and play with a new language. The next strategy is for them.

CHOOSING BY SYNTAX

What about concise syntax? Here is one line of Perl code:

```perl
print(\n"\n\".($line=join("\n","\n")) =~ s/\n/\n\$/g;i++?";\$/ge,$line))
```

It is really short, isn’t it? The C++ colleague writes 30 lines, whereas in Perl it only takes 5 to do the same. We should also consider wording differences between languages. Let’s look at the following examples:

<table>
<thead>
<tr>
<th>JAVA</th>
<th>KOTLIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>public boolean isFree(Person forWhom)</td>
<td>fun isFree(forWhom: Person) : Boolean</td>
</tr>
<tr>
<td>Public (method?) which returns boolean and is called isFree with parameter of type Person named forWhom</td>
<td>Function isFree with parameter forWhom of type Person which returns Boolean</td>
</tr>
</tbody>
</table>

In the first case, the type is before the name, in the second – the opposite. Some languages read more naturally than others. But for the current strategy, personal taste decides.

GENERAL HINTS

Looking at newly created languages, we can see the general direction they are taking. The trend in programming language evolution is toward multi-platform, compiled languages. The compiler helps to write correct code. The code base is more supportable (refactor-able) than the code bases of scripting languages.

On the browser side JavaScript still dominates but this is slowly going to change. Many companies are trying to escape from JavaScript because of its language weaknesses. So they invest in replacements: Dart (by Google), TypeScript (by Microsoft), ActionScript (by Apple), CoffeeScript, Haxe, Elm, GorillaScript, Spider, etc. The upcoming WebAssembly project will push these efforts even further. The message is that big players are moving away from JavaScript and they have the power to reshape browser programming.

When we choose a programming language, we should think of long-term investment. Achieving mastery in this area requires effort and (roughly) 10,000 hours of practice. So... be wise!
At Luxoft, solving your business challenge is our mission and technology is our passion. We know how to connect the dots to create just the solution or service you need. And because we live and breathe technology, it’s perfectly natural for us to see what others have missed.
See the link: goo.gl/APzswg

**THE NEW PERSPECTIVE**

Beautiful historic city. Untouched nature. Beckoning sea. Luxoft invites you to Poland’s most desirable region to live in: Become a part of our awesome team and discover great opportunities for growth in a rapidly developing company. Tricity waits for you. Set your sails!
See the link: goo.gl/JeX9qi

**LUXOFT TRICITY**

Ukraine is the #1 software development center in Central and Eastern Europe and the fourth largest exporter of IT products and services in the world. We owe our success to the talent of over 100,000 IT professionals, a strong education system and the support of the Ukrainian government. We are the best in what we do in the global arena – 10 IT companies operating in Ukraine have been selected among the one hundred best.
See the link: goo.gl/iShdnf

**UKRAINE OPEN FOR IT**

Running is living. Luxoft actively supports multiple sporting activities around the world, and the International Moscow Marathon 2016 is one of the biggest yet: 30,000 runners joined this massive event – including Luxoft’s running team! Created in March 2016 with five enthusiasts, our team of more than 50 Luxofters took part in the marathon, in both the 10 km and 42.2 km runs. We would like to thank the Luxoft Running Club for its support, inspiration and care. See us run next year!
See the link: goo.gl/NERKr3

**LUXOFT MOSCOW MARATHON**

Check out our Youtube channel - www.luxoft.com/youtube
ACROSS

1) it represents the right to perform an action
3) an open-source operating system with a flightless logo
7) 1024 bytes
10) the capital of the country with the first fully electronic voting system
11) some clever indicator
12) the opposite of false (a Boolean value)
14) an experimental, multi-paradigm (and a little bit spicy) programming language
16) one of the general-purpose programming languages or the Biblical tower
17) short for eXtensible Open XHTML Outlines (full of hugs and kisses)
21) a programming language or the alphabet song
24) a mobile app with self-destructing selfies
27) … ipsum
28) ancient command-line operating system
30) the process of replacing a product with its newer version
32) “It’s not a bug, it’s a …”
33) a complete set of coded instructions that performs a particular operation
35) short for toolkit
36) Human Resources
37) the act of solving a problem
38) Job Control Language

Answers from previous issue:

DOWN

1) a tile-matching puzzle game from 80s
2) a web-based Git repository hosting service
4) a numeric value (between 0 and 999999999)
5) a virtual address
6) a numbering scheme in which only two values are possible for each digit
8) a cryptocurrency
13) Enterprise Application Integration
15) a giant network of connected “things” (including humans)
18) the best way to keep your manager up-to-date (Carbon Copy)
19) a value used to initialize a pseudorandom number generator
20) a source code that has a complex and difficult to read structure, unpopular even in Italy
22) a firmware used during the computer booting process
25) the biggest rival to the Apple iOS
26) F5
29) a clean shortcut for Simple Object Access Protocol
30) well-known Linux distribution for personal computers, tablets and smartphones
31) production, storage and communication of information using computers and microelectronics
32) a keyword to introduce a for-loop
34) a set of best practices for IT service management
MEET US LUXOFT@ EVENTS

Joker, Russia

Heisenberg, Russia

LoGeek Omsk

HackerX, Krakow

Back to School, Bulgaria

Scrum Workshop, Bulgaria

Join the team!
<table>
<thead>
<tr>
<th>Location</th>
<th>Event</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIETNAM</td>
<td>3rd LoGeek Night</td>
<td>February 15</td>
</tr>
<tr>
<td></td>
<td>Information Technology – Car Race Event</td>
<td>February 29</td>
</tr>
<tr>
<td></td>
<td>HCMC University of Technology – BKIT Quiz 2017 Contest</td>
<td>March</td>
</tr>
<tr>
<td>BULGARIA</td>
<td>QA Challenge</td>
<td>February 18</td>
</tr>
<tr>
<td></td>
<td>Automotive Open Day</td>
<td>March</td>
</tr>
<tr>
<td>UKRAINE</td>
<td>LoGeek Night, Dnipro</td>
<td>February 9</td>
</tr>
<tr>
<td></td>
<td>LoGeek Night, Odessa</td>
<td>February 23</td>
</tr>
<tr>
<td></td>
<td>Kids Robofest, Dnipro</td>
<td>March 5</td>
</tr>
<tr>
<td></td>
<td>LoGeek Night, Kyiv</td>
<td>March 16</td>
</tr>
<tr>
<td></td>
<td>JUG meetup, Odessa</td>
<td>March</td>
</tr>
<tr>
<td>POLAND</td>
<td>LoGeek Night, Warsaw</td>
<td>February 25</td>
</tr>
<tr>
<td></td>
<td>LoGeek Night, Gdansk</td>
<td>March</td>
</tr>
<tr>
<td></td>
<td>Test Fest Conferences</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Exclusive Trainings</td>
<td></td>
</tr>
<tr>
<td>RUSSIA</td>
<td>LoGeek Night, Saint Petersburg</td>
<td>February 16</td>
</tr>
<tr>
<td></td>
<td>Dev2Dev Krasnoyarsk</td>
<td>February 18-19</td>
</tr>
<tr>
<td></td>
<td>C++ Conference Moscow</td>
<td>February 24</td>
</tr>
<tr>
<td></td>
<td>Omsk Open Days</td>
<td>March 16</td>
</tr>
<tr>
<td>MALAYSIA</td>
<td>Grand Office Opening Ceremony</td>
<td>February 21</td>
</tr>
<tr>
<td></td>
<td>KDU Job Fair</td>
<td>March 7</td>
</tr>
<tr>
<td>ROMANIA</td>
<td>Luxoft Technology Series Event, Automotive</td>
<td>March</td>
</tr>
<tr>
<td></td>
<td>Kids in Tech</td>
<td>March</td>
</tr>
<tr>
<td></td>
<td>Best Hackathon</td>
<td>March</td>
</tr>
<tr>
<td>MEXICO</td>
<td>LoGeek Night in Colima</td>
<td>February</td>
</tr>
<tr>
<td></td>
<td>2nd Bootcamp for Data Structure and Algorithms</td>
<td>February</td>
</tr>
<tr>
<td></td>
<td>Scrum Bootcamp</td>
<td>February</td>
</tr>
<tr>
<td></td>
<td>Hackathon in Pueblo</td>
<td>March</td>
</tr>
</tbody>
</table>
CareerCon, Wroclaw

LoGeek Night, Bulgaria

Code Europe, Krakow

Java Day, Kiev

HolyJS, Russia

Back to School, Bulgaria

LoGeek Night, Bulgaria
TOP ENGINEERING TALENT
Work with top professionals

EXCEPTIONAL QUALITY & EFFICIENCY
A development process built on agility and innovation

GROWTH & NEW OPPORTUNITIES
You grow together with the company

TECHNOLOGY EXCELLENCE
We live and breathe technology

EPIC ATMOSPHERE
Enjoying what we do

Find opportunities at:

CAREER.LUXOFT.COM
Editor: Dilya Salakhetdinova

For any comments, suggestions or requests for becoming a part of the next issue please write to LGmagazine@luxoft.com