Lightweight Characteristics Testing

This document contains inspiration for fast, lightweight testing using many different perspectives. But beware, in your situation it might be more appropriate test more thoroughly on just a few things.

Rikard Edgren

14-Feb-2014, V1.0

Lightweight Capability Testing

I think quality criteria/factors/attributes/characteristics are extremely powerful.
It helps you think in different ways, and makes it easy to get a broader coverage of your test ideas.
See Software Quality Models and Philosophies for McCall, Boehm, FURPS, Dromey, ISO 9126 models, or CRUSSPIC STMPL for a version without focus on measurability.
The granularity of this (and all other) categorization can be discussed, so here are suggested sub-categories for Capability, with some thoughts for inspiration:

Capability – The set of bigger and smaller things you can accomplish with the software. This is usually covered by requirements or similar, and with the addition of some help from developers telling about small or hidden features, you can cover this with thorough and hard work. I suspect some test efforts stop here, which might be a mistake.
The most lightweight testing approach is to ignore this totally, it has been considered by others, and you will cover parts in more interesting ways when performing other testing.

Completeness – Are the functionality really enough? Or are there small things in between, and on the edges that are needed to create a killer app? As a tester you won't decide the scope for a release, but you can identify small things that can be implemented almost for free, and since you have knowledge of the system you can identify bigger things that someone else can put up on the list.
Lightweight method: think about what is missing while performing system testing.

Accuracy – This could be seen as an obvious part of capability, but if you think about precision for real/double, or other corner cases, there are a lot to investigate, and probably a lot to ignore as well…
I don't know about any lightweight technique for this, except asking detailed questions about what's important.

Efficiency – Does the product do what it is supposed to do in an effective manner, without doing what it isn't supposed to do?
Lightweight: keep your eyes open and look at more places than the apparent ones.

Interoperability – In a requirements document you will find a lot of things the software should be able to do, but you will not get a list of all important combinations and interactions that surely exists. Pairwise testing is a theoretical solution to this dilemma (and I guess it is effective for some situations), but with knowledge about the product you will see that some combinations are more error-prone than others.
A lightweight testing solution consists of two test ideas: 1) turn on everything 2) turn off everything

Concurrency – Can the software or functions therein operate simultaneously? How many concurrent actions? What if they are dependent on each other?
Lightweight testing: start more operations now and then while system testing.

Data agnosticism – Use data as dirty as in reality. Use really tricky data throughout a full scenario.
Different cultures have different character sets, separators and date/time formats.
GUI controls that force input in one format is a way to avoid this, but these are not always easy to use.

Extendability – all features wanted by customers can't be implemented, so it can be nifty with an API that allows extensions of various types.
Lightweight testing: get hold of an API implementation, use it and change it.

Lightweight Reliability Testing

The big drawback and big advantage with reliability testing is that it is easiest and most effective to perform together with other testing. A separate automated reliability regression test suite could cost an awful lot to implement, but reliability in your spine when performing any type of manual test, together with deviations, is cheap, interesting, and powerful.
If you look at Reliability from a standards perspective, you will see a lot of measurement methods like Mean Time Between Failures. You don't need to use these. You can test and find important information anyway.
The most lightweight method is to ask these questions to heavy users of the product:

Reliability. *Does the product work well all the time?*

- Stability. Are you experiencing (un)reproducible crashes?
- Robustness. Are there any parts of the product that are fragile and have problems with mis-configurations or corner cases?
- Recoverability. Is it possible/easy to recover after (provoked) fatal errors?
- Resource Usage. What does the CPU, RAM, disk drive et.al. usage look like?
- Data Integrity. Are all sorts of data kept intact in the system?
- Safety. Is it possible to destroy something by (mis)usage of the system?
- Disaster Recovery. What if something really, really bad happens?
- Trustworthiness. Do you feel you can trust the system?

You may also want to perform some specific tests aiming at the different sub-categories.

Stability. Run the product for a long time, without restarts.
Automate a simplistic scenario and run it thousands of times in a sequence, maybe AutoHotkey can help you?
Count the number of non-reproducible crashes per day that happens to your team.
Try really hard to reproduce the non-reproducible issues.

Robustness. Provoking error messages is fun, and don’t forget to check spelling and if error message helps.
When Robustness is important: hit hard, hit many times.

Recoverability. Turn off the power for machines performing important things, restart and look at behavior.
Whenever an error occurs, try to recover, and consider if it is easy and intuitive.

Resource Usage. Look at system resources now and then.
Stress the system in various ways (but only spend time on this if the project is interested in results.)

Data Integrity. Use all types of data (numeric, strings, out-of-range, invalid, empty, Unicode), in different sizes (small, medium, large) on different systems (localized OS, regional settings, different fonts) through all parts of the system.

Safety. Do thorough brainstorming around scenarios where people can get hurt. Be aware that ambiguous or missing information can be very dangerous if they affect important decisions.

Disaster Recovery. You probably don’t want to test this for real. But you can ask developers or others if there are possibilities of continuing using the software after a crucial machine has disappeared. You can test if the backup can be used to restore the system. This is one of those characteristics that either is irrelevant, or very important.

Trustworthiness. Note down all inconsistencies in behavior, or moments when you are unsure what the product is up to.
Tell the project how you feel about the reliability.

The best way to get reliable software is to have really solid code, and properly customized code checker tools can help you with this.
There are also many situation where implementation of a tool designed for your specific purposes is what you need.

Lightweight Usability Testing

When reading about usability testing, it often involves an outside person trying to use a feature for the first time. Now this is a good thing, but there are a lot more to Usability than Learnability.
There also exist systematic inspection methods, e.g. Heuristic Evaluation by Jacob Nielsen (http://www.useit.com/papers/heuristic/heuristic_list.html), that should be performed by super-experts on usability.
He recommends alternating between user testing and evaluation, but I think a manual software tester can do both at the same time, if she has (some) knowledge about the domain, (some) knowledge about the product, and (some) knowledge about usability problems.

Here is a very cheap way to do usability testing; ask the following questions to someone with many hours of using the product (probably a real user or a manual system tester)

Usability. *What's your most important usability issues?*

- **Affordance**. Would you say that the products invites to discovering possibilities, or are there features you learned too late?
- **Intuitiveness**. Is it easy to understand and explain what the product can do?
- **Minimalism**. Have you seen redundant information anywhere? Any important information missing?
- **Learnability**. Are there any areas you feel are difficult to understand and learn?
- **Memorability**. Do you forget how to perform some actions?
- **Discoverability**. Can you discover all available operations by systematic exploration of the user interface?
- **Operability**. Are any common operations cumbersome?
- **Interactivity**. Are there situations where you aren't sure of which kinds of interactivity is supported?
- **Control**. Do you know what is happening, and how to control the behaviors of the product?
- **Clarity**. Are you ever confused?
- **Errors**. Have you encountered strange error messages, or situations that were difficult to recover from?
- **Consistency**. Is the user interface annoyingly different anywhere, or hard to understand?
- **Tailorability**. Are there any (default) settings you would like to change?
- **Accessibility**. Did you notice any problems while using High DPI and mostly the keyboard? Color-blindness?
- **Localization**. Any experiences of running translated product or data?
- **Documentation**. Are there any information you would like to see in the Help?

You might already know about many of the issues in the answers, but now you know more about their importance.
You won't get any quantitative measurements, but hopefully a lot of valuable information.

Even better might be to have questions like these in the back of your head when executing manual tests, you'll probably get valuable information for free. There is no need to memorize the categories, the important ones will emerge.

Usability Quicktest

Look at the screen and note down anything annoying.
Look at default keyboard position.
Press F1, Enter, Escape.
Check tab order and access keys (on Mobile device, zoom in, change something, zoom out)

Lightweight Charisma Testing

One heavyweight way of testing charisma is to use dozens of potential users on dozens of alternative product solutions/prototypes.
For lightweight charisma testing, it is often fast and fruitful with an awareness of charisma violations.
This method requires an understanding of the unique charisma for your product.

Testers probably won't be in charge of developing an explicit charisma model, but you can take part, e.g. by recommending the following process:

1. Write down what you believe are the special things about the software (gamers might use "playability", but please be more specific)
2. Get inspired by thetesteye's generic list of Charisma characteristics (provided below)
3. Document the important characteristics with as much details as possible (to make'em truly mean things in your context.)
4. Share, review, refine and gain approval for your Charisma Guidelines across the teams

When you have this list, it is fun and attention-inspiring to keep these things in the back of your head whatever type of testing you are performing.
Just remember to report appropriately, probably verbal.

Charisma: Does the product have “it”?

- **Uniqueness:** the product is distinguishable and has something no one else has.
- **Satisfaction:** how do you feel after using the product?
- **Professionalism:** does the product have the appropriate flair of professionalism and feel fit for purpose?
- **Attractiveness:** are all types of aspects of the product appealing to eyes and other senses?
- **Curiosity:** will users get interested and try out what they can do with the product?
- **Entrancement:** do users get hooked, have fun, in a flow, and fully engaged when using the product?
- **Hype:** should the product use the latest and greatest technologies/ideas?
- **Expectancy:** the product exceeds expectations and meets the needs you didn't know you had.
- **Attitude:** do the product and its information have the right attitude and speak to you with the right language and style?
- **Directness:** are (first) impressions impressive?
- **Story:** are there compelling stories about the product’s inception, construction or usage?

Lightweight Security Testing

Many testers (including myself a couple of years ago) believe security testing is something only for the experts.
The truth is that you might already be performing security testing, at least when going by TheTestEye's classification:

Security. Does the product protect against unwanted usage?

- **Authentication:** the product's identifications of the users.
- **Authorization:** the product's handling of what an authenticated user can see and do.
- **Privacy:** ability to not disclose data that is protected to unauthorized users.
- **Security holes:** product should not invite to social engineering vulnerabilities.
- **Secrecy:** the product should under no circumstances disclose information about the underlying systems.
- **Invulnerability:** ability to withstand penetration attempts.
- **Virus-free:** product will not transport virus, or appear as one.
- **Piracy Resistance:** no possibility to illegally copy and distribute the software or code.
- **Compliance:** security standards the product adheres to.

Personally, I cannot do a thorough security pen-testing report, but I can look at some areas, and at least see if there are really big problems.

Web Tools

I believe in OWASP's Zed Attack Proxy, they are building it to be easy to use, in order to widen the security testing of web applications. It doesn't show if the application is secure, but it can show if it is (seemingly) insecure.

Things That Mean Other Things

A good addition to any equivalence partitioning is to try to find strings that might mean something else. Hackers want to exploit the interpretations, you might also want to try SQL injections and URL-tampering.

Known Issues

The most common security problem is old versions with known security issues. If everyone upgraded server applications to the latest versions, there would be less security problems (but perhaps more functionality problems...)
Search the web for "vulnerability taxonomy" and see if the information helps.

Crashes

Many crashes you report are potential security problems. If someone says, “no one would do that on purpose”, you can respond "maybe that's their exact purpose..."

Moving Forward

There are a lot of places to get more information on this, here is one:
http://blog.applabs.com/index.php/2010/11/a-heuristics-based-approach-to-security-testing-of-web-applications/

Lightweight Performance Testing

If performance is crucial for product success, you probably need pretty advanced tools to measure various aspects of your product, to find all bottlenecks and time thieves.
For all other software, performance is just very important, and you might get by with lightweight test methods.
You may, or may not have quantified performance requirements, but you should test performance to some degree anyway; for the whole, but also for each detail (when appropriate.)

In TheTestEye's classification, performance consists of:

Performance: Is the product fast enough?

- **Capacity:** the many limits of the product, for different circumstances (e.g. slow network.)
- **Resource Utilization:** appropriate usage of memory, storage and other resources.
- **Responsiveness:** the speed of which an action is (perceived as) performed.
- **Availability:** the system is available for use when it should be.
- **Throughput:** the products ability to process many, many things.
- **Endurance:** can the product handle load for a long time?
- **Feedback:** is the feedback from the system on user actions appropriate?
- **Scalability:** how well does the product scale up, out or down?

Be aware of different definitions of performance testing, e.g. some include reliability, stress handling, robustness, and what stakeholders believe is most important might differ (even when using the same words...)

Ongoing Violation Awareness

The number one lightweight method starts by finding out which of these characteristics that are relevant for your product.
Then keep them in the back of your head, and whenever you see something fishy, investigate further and communicate.
Often the OK zone is easy to reach, but testers should notice when violations occur.
When appropriate, apply the destructive principle: Increase the amount of everything that can be increased.

No Tools

Perceived performance is what matters for end users (but maybe not for a product comparison check list) so think about how it feels, and try using a stop watch.
You might get pretty far by load testing with colleagues with several instances each.

Tools

There exists limiters for CPU, RAM, bandwidth etc. and many of them are free (and some of them become obsolete.)
A task manager/resource utilization tool can give you hints on memory, CPU, disk, network et.al.
Scripting your product to run over weekend is good for endurance and stability testing.
JMeter is free and often quick to get running.

Summarizing

Summarizing performance test results is difficult. Aggregations of measurements don't tell the full story, and the whole story takes a long time to tell.
Communicate what is important, which is easier if you have asked stakeholders beforehand.

Warning: For many products, users aren't as interested in Performance as the developers...

Lightweight IT-bility Testing

In TheTestEye's classification, IT-bility consists of:

IT-bility: Is the product easy to install, maintain and support?

- System requirements: ability to run on supported configurations, and handle different environments or missing components.
- Installability: product can be installed on intended platforms with appropriate footprint.
- Upgrades: ease of upgrading to a newer version without loss of configuration and settings.
- Uninstallation: are all files (except user's or system files) and other resources removed when uninstalling?
- Configuration: can the installation be configured in various ways or places to support customer's usage?
- Deployability: product can be rolled-out by IT department to different types of (restricted) users and environments.
- Maintainability: are the product and its artifacts easy to maintain and support for customers?
- Testability: how effectively can the deployed product be tested by the customer?

The starting point is to understand which platforms that are most important to test: popular, error-prone, best representatives.
Installation testing can be quite easy, or very cumbersome, but you will often try these broad strokes:

File Analysis

Compare snapshots of files and registry entries before and after installation, and after use and uninstallation.

"Restricted User"

Many end users are restricted in what actions they are allowed to perform on their computer. Many developers have administrative rights when they are testing the functionality they have built.
That's why you want to test with similar (or same) restrictions as actual user will have.

Ongoing Violation Awareness

The most lightweight method is, as always, to know about IT-bility characteristics, and have a look at them while performing other testing.
Installation

Dog-fooding

If you spend some setup time, you can have deeper testing, e.g. by using customer-like, upgraded clients and/or servers.
Being able to use dogfooded machines for several upgrades give a breadth that is hard to match with clinical, testlab upgrades.

Avoid

If you don't want to spend too much time on your IT-bility testing you don't want to

- install in uncommon folders
- test on many unsupported platforms

Lightweight Compatibility Testing

In testing text books you can read that compatibility testing should be performed after the functionality testing and bug fixing is completed. I guess the reason is that you don't want to mix these categories, but hey, what a waste of resources.
My suggestion is to try to perform the compatibility testing at the same time as you are doing your other testing; when problems arise, trust that you will deal with them.
In thetesteye's classification, compatibility testing involves hardware, operating system, application, configuration, backward/forward compatibility, sustainability and standards conformance.
Here follows some lightweight methods to tackle these areas.

Basic Configuration Matrix

Basic Configuration Matrix is a short list of platform configurations that will spot most of the platform bugs that could exist in your currently supported configuration matrix.
The simplest example is to use one configuration with the oldest supported operating system, oldest browser etc; and one configuration with the newest of all related software. A more advanced example could use several configurations that use different languages, Application Servers, authentication methods et.al.
Often it will take quite some time to run most tests on BCM; so alternate between the configurations while testing your product. Do variations on configurations when appropriate.

Error-prone Machine

Another trick is to setup machines so they have a high chance of stumbling on compatibility issues. You can vary this on your BCM, your personal machine or whatever is suitable. The idea is to get some compatibility testing almost for free.
Examples on Windows include: run as Restricted User, use Large DPI setting, use German Regional Settings, install support for all of the worlds characters, non-English language in Internet Browsers, move system and user Temp folder, activate IE 'display a notification about every script error', move Task Bar to the left side of the screen, Windows Classic Theme & Color Scheme, use User Access Control, use an HTTP proxy, use Data Execution Prevention, install new versions as they come, e.g. latest hotfixes, MDAC etc, never install software on default location, run with 2 screens, on a 64-bit system, use different browsers, turn off virtual memory swapping, , install Google/Yahoo toolbar, run Energy Save Program, pull out the network cord every time you leave the computer; and put it back in when you return, turn on the sound!

Technology Knowledge

If you know a lot about the environment the software operates in, you know which things will happen in reality, which settings that usually are altered, and how it is commonly operated.
The lightweight method is to use this knowledge and make sure you test the most important things.

Letting Others Do the Testing

Many compatibility problems happen on basic usage, which indicates that you can let others do a big part of the compatibility testing: developers can use different machines and graphics cards, Beta testing can be done in customers' production-like environment. If the product is free of charge, you might even get away with addressing problems after your users encounter them (but make sure you have an easy and compelling way for them to do this reporting.)
Crowd-testing could be a way, but so far the payment models from the testers' perspective are not ethically defensible, to me.

Reference Environments

To quickly investigate if you are experiencing a compatibility issue, it is handy to have reference environments available. It could be someone else's, a virtual machine, a quickly cloned image, your own machine etc.
Personally I prefer having a physical machine that is running similar things, but on a different OS, different language and/or earlier version. The last years I have had three machines and three monitors, and by switching, I get a lot of compatibility testing done at the same time as testing new features. When I check things on an older version, I can save documents and use them for next tests.

Backward/Forward Compatibility

Backward compatibility is easiest done if you can use real customers most complicated files/data/documents. Use these as you test any functionality (Background Complexity Heuristic).
Forward compatibility should be designed in the architecture, as a tester you can point this out.

Sustainability

Have conversations around the question: *Is the product compatible with the environment?* Have we considered energy efficiency, switch-offs, power-saving modes, support work from home and the likes?

Standards

A lightweight method for standards conformance is to identify which ones are applicable, and ask the experts if they understand it, and successfully have managed to adapt it to the new context.
Let's finish with a non-lightweight method: you can become the standards expert.

Lightweight Internal Characteristics Testing

Supportability

Can customers' usage and problems be supported?

The best lightweight method is probably to have conversations with support people, the best heavyweight method is to work there for a while.
You can also think about customer incidents when encountering problems, what would developers need in order to pinpoint this; should logging be improved?

Testability

Is it easy to check and test the product?

Testability improvements can be huge time-savers. So whenever you feel testing is difficult and takes a lot of time, consider additional capabilities in the product that would make it faster to test, e.g. internal validation, or better observability of internal states and variables. Then ask developers if they want to help out.

Mainainability

Can the product be maintained and extended at low cost?

This is often the developers' area, but as a tester you can help by describing your experiences of the addition of new features in various areas.

Portability

Is transferring of the product to different environments and languages enabled?

If the product is translated, use another language from time to time when doing your normal testing. Localized products doesn't only have language issues, in my experience, ANYTHING can break.
It is also possible to test languages you don't know. I remember how I looked at a whole Help system (Windows style with navigation tree to the left) in Japanese, by rapidly pressing right arrow and Enter, it was easy to spot places where non-Japanese characters were used.