Software Testing Glossary

a damn simple definition dictionary

from testitquickly.com

a printing ready version

last build: July 11, 2019

Abstract

This is not an another '*Full glossary of terms used in Software Testing*', or '*Let's bring together every known term in our industry, because everyone needs it. . . '*.

I just had to notice my own *definition dictionary* of some terms, so I did it.

English is not my pet language, so any ping about ANY inaccuracy in this doc wil be appreciated. Thank you in advance.

Also you can:

1. download a new version of this pdf for free from `http://wp.me/pfyTc-W4`.
2. ask me, if something wrong or unclear.
3. understand, that some terms require a detailed explanation, which is a subject of a whole lesson, apart from of a glossary.
4. use and share this doc in any way with no commercial purposes.

Alexéi Lupán,
QA Trainer
Kyiv, Ukraine
astenix@testitquickly.com

Contents

1) Acceptance Criteria

> Understand what is a Criteria first [p.30].

An *Acceptance criteria* is a set of expectations.

Example: *'Yes, it works exactly as I expected! Shut up and take my money!'*.

Main idea: the *Acceptance Criteria* should be established directly by the user (or customer, or any other authorized entity), charged with authority to approve or reject, otherwise the whole idea has no sense.

> Expectations must be satisfied by development team in order to be accepted by their customer.
>
> So, who can/should establish them instead of Customer?

Fun is that a Customer has no needs of any requirements or even acceptance criterias — the development team required them. If nobody understand this simple issue — your project can slide to hell.

2) Acceptance Testing

A very ambiguous term. Depending of context:

1

A development project phase.

At the end of any development project phase a customer representatives can started a (sometimes — *very*) rigorous checking of every requirements, expectations and business capabilities implemented during the development process. Sometimes it is the unique solution to prevent a possible brilliant technical solution from failure.

Customer did not test, customer *use* the product. You *test* the product. You cannot test the product as a Customer does it.

> You can test till the death on the **Development** and **Testing** environments, but only the «**UAT**» (User Acceptance Testing) environment can be consid-

> ered as the closest to production environment. So, some serious bugs can be founded **only** on UAT environment.

That's why «Acceptance Testing» can be done only from the customer point of view — by the customer himself. We can only help him, by sharing our test-docs/ideas.

> And only by the Customer itself.
>
> Or only by his brave testers and accountants.

Only on UAT you can use for checkout not only 'Visa for testing' cards, but real payment issues.

Only on UAT you can use real payment filters, or a real database with customer's stuff, or a real connection with Warehouse.

And this is the reason to not invite development team to test on UAT and Production servers.

2

An approach to software development.

This approach is coming from TDD ('Testing Driven Development') family.

Implies that every requirement statement is published and handled by the customer as automated Test case (using with special software like 'FitNesse' wiki-system - see demo at https://youtu.be/wzmVJ3HYftA.

3) Actual Result

You always have some **Expectations** when you make an action for testing. Otherwise the whole testing has no sense.

> Testing is always to compare the *Expected* and the *Actual* results.
>
> When you have no expectations, then you can make an investigation, a research — anything but not testing.

Suppose, that I want to know if I can send an email notification from a Product page.

I have an expectation like *'I will call 'Send email' function, I will provide appropriate data in input fields, and I will receive a notification'*. This is the **Expected result** of a test case.

Suppose, that I am really open a Product Detail Page and I am looking for the 'Send email' function. It is available? If yes, than I really can send an email with it?

This will be the **Actual result**.

You can have more than one actual results.

A tester should always suppose the **Expected result** (a mandatory part of any Test Case [p.65]), but he will never be sure about the Actual result.

If any difference between Expected and Actual Result — this can be a Bug [p.22].

> Or the Expected result is wrong, and Test Case should be updated.
>
> Or something was unexpectedly changed during development.
>
> Or a third-party application has down.

4) Ad hoc testing

The term *'ad hoc'* has a Latin language origin, and can be translated like

> *'Emergency, jump on everybody, there is no time to think, we have no plan, just let's kick out some barbarians, just go-go-go!'*.

Figure 1: `Barbarians!`

For us this is a name of a testing approach, usually explained as testing, but without test cases or requirements[1]:

- without formal test preparation;

[1] See figure 'Simple explanation of Classic & Ad Hoc & Exploratory Testing' at p.11

- with no recognized test design technique is used;
- without expectations for results and arbitrariness guides the test execution activity.

But this is very, very stupid!

In fact, nobody said «DO NOT USE REQUIREMENTS in Ad hoc testing!»

> Use them!

Nobody said «DO NOT USE TEST CASES in Ad Hoc testing!»

> Use them!

The Ad Hoc is the second part of Classic Testing. Like

- White & Black,
- Guitar & Strings,
- Bonnie & Clyde.

The Ad hoc testing approach aims to *support* the formal testing approach (you call it Classic testing), where all testing activities are based on formal and logical execution.

The weakest side of formal testing is the lack of some scenarios, that can be omitted by development team, but they can be discovered by end-users. The Ad Hoc testing can help to identify possible bugs, that cannot be discovered using formal scenarios.

Some monkeys think that the Ad Hoc testing is the same as 'Monkey Testing' ([p.48]). They're wrong.

- In monkey testing tester even doesn't understand, what and for what he has doing with the application.
- In Ad Hoc testing tester understand his goals and application capabilities, he implies logic in his actions, but he can easily switch between his goals and scenarios, without any reason for doing it.

5) Agile Software Development

Means any software development approach, based and guided by agile principles (http://agilemanifesto.org/).

Warning, based on agile *Principles*, not Methodologies.

There is a huge difference between *Principles*:

> I will not tolerate then someone will have no answer to a question, I will try to explain it immediately. I don't know how I will do it, I can use Jira tickets, email, Skype, personal visits, this doesn't matter.

and a *Methodology*:

> Everybody shut up! From now every person, who will need my answer to any question, should:
>
> 1. formulate his question shortly and briefly (link to a guide),
> 2. raise an issue in Jira using a 'Question' template (link to a guide),
> 3. assign it to the appropriate person,
> 4. mention his QA Lead and Test Manager in watchers mandatory.

Feel free to feel the difference.

Yes, the 'Agile methodology' does not exist.

> The same with SCRUM [p.58] — this is not a methodology too, this is a software development *framework* (a common strategy) for managing product development, driven by agile principles.
>
> You can drive a development process by SCRUM framework, used only *over* a Methodology.

A lot of people wants to know about 'Agile' only '*You will always have a working product delivered*' (Lies!) or '*We can easily rework any issue, just tell us what to do*' (Deliberate lies!) , or '*True agile testers didn't use test cases!*' (Kill this bastard!), or '*We will have no requirements, because we are Agile, aren't we?*' (Not enough bullets. . .) and didn't care about how such goals will be achieved. This can drive the project to the hell, but we are brave enough to getting things done, aren't we?

6) Agile Testing

Even if it should be based on Agile Software Development principles[2] [p.11], the Testing activities still remain the same.

There are no such thing like *Agile Testing*.

> What is the difference between drinking water from a cup or from a bottle? Anyway, you just drink water.

The same with Testing — you should have to do all old things like

1. gather requirements,
2. understand expextations,
3. understand and preview situations for being tested,
4. the expected results
5. . . .

The most common problem started immediately from '*There are no Requirements*' issue.

Everything else in Agile Testing derive from this goddamned '*there are no Requirements*' mantra. You can have User stories & Use Cases, but for writing brilliant Test Cases you would scream for good brand old Requirements.

[2] http://agilemanifesto.org/

The Testing process in agile development requires a totally new logical approach, and you may not fit with it at all.

For example, for a tester in Agile is very important to be a master in Exploratory Testing or at least Ad Hoc testing, not a master in writing and executing Test Cases; and is important to handle very well the Automated Testing (from unit to complex functional testing), for increase speed of testing.

Some person can excellent fit, others will certainly fail — this is a matter of psychology.

And you can fail because you act with Agile like with just another *development process*, not like with an approach of do the development *closest to client needs*, and this is the second most common problem with Agile testing.

7) Alpha Testing

Alpha Testing is the name of an approach to testing, where the Development team ships his work to *internal* testing team only.

Get it? An *approach*, not a phase in testing process. But anything can be presented like a phase, so. . .

Alpha Testing approach is specific for development Internet Shops, games and almost any kind of software, where human beings opinion matters.

Surely, technically we can always publish an software with some bugs, and *'. . . if users will tell us about it, we will fix them quickly. It is just a question of 'cost of failure'*.

Well, today the cost of failure looks very low, because 'You can easily fix the software online'.

> Skype, Firefox, Twitter, Facebook — they always ship first, then fix.
>
> Why don't you do the same?

But 'Shops' always means 'Money', and money requires confidence. We cannot ship an Internet Shop with any bugs in functionality, because it can scarry future customers and this can burn out our asses. Or because they can cheat, and again, this is all about money.

And not every bug can be fixed quickly.

So we will frenzy test our software BEFORE it will be offered to our customers and will call this *Alpha Testing*.

The next big step will/can be **Beta Testing** [p.21]: this means that the Development team will select a small bunch of potential users of their product (usually outside from the development company), and will ship to them the Product 'as is', just for testing purposes.

> And, because you are asking, we are aware that there is no *Gamma* or *Delta* testing.
>
> But they was! An IBM PM Martin Belsky[3] has invented them just for name some logical steps.
>
> **The *A*-test** was a feasibility and manufacturability evaluation done before any commitment to design and development.
>
> **The *B*-test** was a demonstration that the engineering model functioned as specified.
>
> **The *C*-test** (corresponding to today's beta) was the *B*-test performed on early samples of the production design.
>
> **The *D*-test** was the *C*-test, repeated after the model had been in production a while, to verify final safety.
>
> We don't care about this today, so. . .

There is a **Main problem** with Alpha Testing — internal testing cannot reveal EACH issue, that can happens in production. It can assure only that all functionality works as expected in expected scenarios and conditions.

> Are you really sure, that all scenarios and conditions in your software was foreseen?
>
> Really?

Often this limitation is very clear to development, but not for the Customer.

[3] bit.ly/2kf9S9H

8) Automated Scripts

The scenario for the Automation Robot.

Shows how to interact with the application under testing.

Usually a simple human cannot read/understand such scripts, but there are a lot of exceptions. Scripts can be writted in a simulation of normal language.

You don't know, how the Robot will read and execute step by step this instruction, but you can understand at a global scale, what should be happen at each step.

9) Automated Testing

An approach to testing, where some functional testing in application under development can be done by special Robots instead of Human beings.

> Sometime it helps. Sometimes not.
>
> It can be fun. It can be stupid.

Usually such testing activity can only demonstrate that your application works as expected — nothing was broken or damaged during development.

Automated test cases are very 'simple and stupid'.

> Suppose, that you have a link as an image on the page, and the image was lost for some reason.
>
> Automated tests will be 'blind and deaf' if no images will be available on page for a hyperlink — test script will click on link and the testing will go on without any error warnings.

Automated test cases are very unstable, they will fail immediately, if the locators of elements on the page will have ANY (even one symbol) changes. For this reason, each fail should be investigated apart, because nobody knows, if the fail happens because of a bug, or because the test script itself is obsolete or incomplete.

Automated test cases run very fast, they can interact with tested software faster than any human can see.

> For the first time this looks fun, but later you realize, that you cannot understand what happens on the page during testing. You lose the control.

Automated test cases can use an unlimited test data variety.

This approach involve a lot of special programming work before the testing begin and a lot of programming after.

This job can be done by developers or by skilled testers, but anyway — for being done, it requires the availability of well done developed and documented Test Cases.

The candidates for the test automation may be as follows by priorities:

1. test cases that are included in smoke testing;
2. test cases that are executed more frequently during regression testing;
3. test cases that are executed less often, such as low priority test cases that are not included either in smoke tests or in regression tests;
4. test cases that are difficult to execute manually because of large data sets, or because it takes too much time to setup and run).

10) Availability Testing

Availability is the probability that a system will work **as** required **when** required during the period of a mission.

You won't have to check it often in functional testing. Let's define it just for the sake of understanding.

Main idea: to reduce and eliminate the expected downtime, you should have a clear knowledge of how much time the app (or the machine) will be operational till the death, if nobody intervenes.

This is very important to know when we talk about web-servers (or aircrafts, or hard drives and so on).

Testing for availability means

- running an application for a planned period of time,
- collecting
  - failure events
  - and repair times,
- and comparing the availability percentage to the original service level agreement.

Availability testing is primarily concerned with measuring and minimizing the actual repair time. This is a simple mathematics.

But before of all you should know several things:

I. *Mean Time To Failure* (MTTF)

> Is the time, on average, that you would expect a clock-work to fail when it has been running. It is a simple indicator of clock-work reliability.
>
> Example: a clock-work runs for 500 hours. It breaks down 5 times during that period.
>
> *Mean Time To Failure* (MTTF) = 500 / 5 = **100 hours**

II. *Mean Time Between Failure* (MTBF)

> Is the time, on average, that you would expect a clock-work to fail including time lost for repairs are undertaken. It is an indicator of the combined reliability and maintenance effectiveness/efficiency.
>
> Example: a clock-work runs for 500 hours, and it has 200 hours downtime due to 5 failures.
>
> *Mean Time Between Failure* (MTBF) = (500 + 200) / 5 = **140 hours**

III. *Mean Time To Repair* (MTTR)

Is the time, on average, that you would expect a stoppage to last including time spent waiting for maintenance engineer, diagnosis, waiting for parts, actual repair and testing.

It is an indicator of maintenance effectiveness/efficiency.

Note that there is another indicator that can be used here, *Mean Corrective Repair Time* (MCRT).

> *Mean Corrective Repair Time* is only interested in the actual repair time assuming all tools, spares and required manpower are available (efficiency).
>
> *Mean Time To Repair* (MTTR) – *Mean Corrective Repair Time* (MCRT) = Waste and therefore gives you an indication of how ineffective your stores and maintenance resource strategy is.

Notice that as MTTR trends towards zero, the percentage availability trends towards 100%.

Still here? Here is an example.

As above, clock-work runs for 500 hours, has 200 hours downtime due to 5 failures.

Mean Time To Repair (MTTR) = 200 / 5 = **40 hours**

Therefore:

Mean Time To Failure (MTTF) + *Mean Time To Repair* (MTTR) = *Mean Time Between Failure* (MTBF)

Now, how to calculate the *Availability* — just use data from following variables:

1. *Mean Time To Failure* (MTTF)
2. *Mean Time Between Failure* (MTBF)
3. and *Mean Time To Repair* (MTTR)

Or more importantly, the unavailability of the clock-work due to maintenance causing failures.

Example, huh?

As above, clock-work runs 500 hours breaks down 5 times and 200 minutes are spent waiting repair / spares / repairing.

Availability:

$= ((500 + 200) - 200) / (500 + 200)$

$= (700 - 200) / 700$

$= 71\%$

Or

$$\frac{Mean\ Time\ To\ Failure\ (MTTF)}{Mean\ Time\ Between\ Failure\ (MTBF)} = Availability \quad (1)$$

or

$$\frac{Mean\ Time\ To\ Failure\ (MTTF)}{Mean\ Time\ To\ Failure\ (MTTF) + Mean\ Time\ To\ Repair\ (MTTR)} = Availability \quad (2)$$

$100 / 140 = 71\%$

Or

$100 / (100 + 40) = 71\%$

Ok. **What about *Unavailability*?**

$1 - Availability = 29\%$

Or

$200 / (500 + 200) = 29\%$

Or

MTTR / MTBF or MTTR / (MTTF + MTTR)

$40 / 140$ or $40 / (100 + 40) = 29\%$

Still here?

Here is an another example of the formula for calculating percentage availability:

$$\frac{\textit{Mean Time Between Failures}}{\textit{Mean Time Between Failures} + \textit{Mean Time To Repair}} \times 100 \tag{3}$$

"*Mean Time*" means, statistically, the average time.

"*Mean Time Between Failures*" is literally the average time elapsed from one failure to the next. Usually people think of it as the average time that something works until it fails and needs to be repaired (again).

"*Mean Time To Repair*" is the average time that it takes to repair something after a failure.

11) Best Practice

This is the worst thing, that you can get from something (or someone) and try to adapt it at your needs. There are no best practices, everything depends of the context. You cannot get the most valuable from the Capitalism and Communism, and mix it in a new religion for your own project.

> What is the best in Capitalism is immediately available with its worst things, and this things cannot be separated, like Mercedes cannot be separated from 'german approach of doing cars'. Just imagine the building of Mercedes in Russia.
>
> What is the best for one crew, can be crucial badly for any other.

Simple, honest alternatives to 'Best practices' are available:

- "Here's what I would recommend for this situation..."
- "Here is a practice I find interesting..."
- "Here is my favorite practice for dealing with x..."
- "Person X attributes practice Y for his success. Maybe you'd like to learn about it..."

Saying «best practice» is obviously irresponsible.

12) Beta Testing

Beta Testing is a testing approach, specific only for mass-market products (Word, Skype, Firefox so on).

Means that the Development team select a big bunch of already existed users of their product (usually this happens outside of the development company), and ship to them the Product in Testing 'as is'.

Main idea — developments will receive a lot of *'Something is going wrong when I tried to do following...'* messages. This can help in testing the Product on different testing configuration, without spending time trying to create it in our testing laboratory. And User Experience can reveal something really unexpected.

Main problem: not every message from beta testing users is a *Bug Report*. Will be a lot of garbage or already reported issues. Will be generated a lot of issues, which cannot be reproduced for some reasons (usually, because of a lack of information and because bug reporters didn't know that there are some common rules of writing bug reports).

And even if all messages will be Bug Reports, do you know how to deal with a thousand of reports about the same Bug?

For internet shops Beta testing is not applicable at all (users can be scared of bugs at Checkout, so only Alpha Testing [p.13] is appropriate), and is a part of standard testing process.

13) Boundary Value Testing

A test-design technique for explain the source where tester can search for ideas to create test cases.

Represent an input (or output) value which is on the edge of an equivalence partition or at the smallest incremental distance on either side of an edge (boundary).

Example: You can test a car at his minimum and maximum speed values (10 and 100 miles per hour).

If the test will be ok, you can suppose, that in the interior of this range the car will

work as expected.

So, instead of doing 90 test cases (10 m/h, 11 m/h, 12 m/h....) you can check only two test cases and suppose (just suppose) that car will work as expected.

Sometimes this approach can help to save time and materials.

Sometimes this approach totally sucks, if tester make a logical mistake or a false admit, that in the interior of the range everything will work as expected.

14) Bug

This is an 'umbrella' term.

Usually a *Bug* is a flaw in a component (or system) that can cause the component (or system) to fail to perform its required function, e.g. an incorrect statement or data definition.

But not every Bug is a Defect [p.31]. You cannot always be sure, what is the root cause of a flaw. An *Error* (a human mistake) in programming code can lead to a *Defect*. Sometimes a Bug can be nothing than a *Failure*[4], or a *Mistake*, or an *Error*...

It will be more convenient, if we will use a common term for all this artifacts — the *Bug*.

ANY deviation from the expected result during testing can be treated as a *Bug*.

But not every deviation is a bug.

Women wear dresses and earrings (this is an expectation).

When you will see a women wearing trousers, then '*Whoa, the actual result differs from the expected, this is a bug!*'?

Clearly not.

14.1) Why it is called 'A Bug'

A typical version of the story of the 'bug' term is:

[4]Deviation of the component or system from its expected delivery, service or result.

In September 9, 1945, Grace Hopper[5] work on the 'Mark II' computer at the Harvard Faculty.

Operators traced an error in the 'Mark II' to a moth trapped in a relay.

Dead bug was carefully removed and taped to the log book.

Stemming from the first bug, today we call errors or glitches in a program a bug.

Amen.

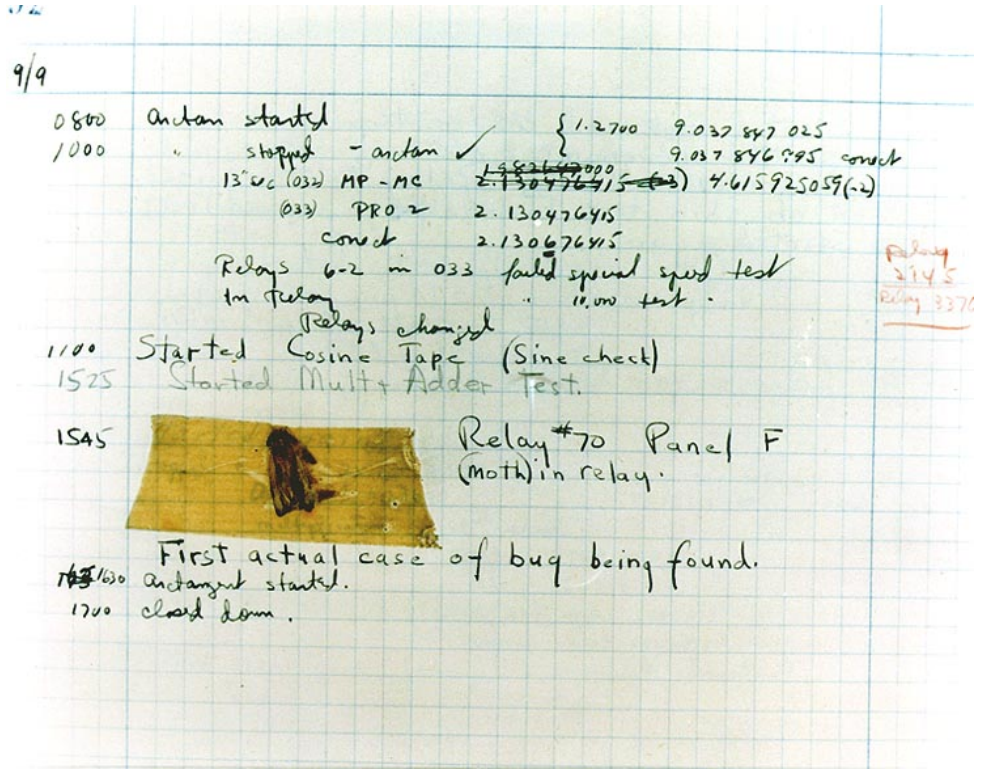

Figure 2: `'First actual case of bug being found'. September 9, 1947`

Well. . .

[5]Grace Brews Murray Hopper (December 9, 1906 – January 1, 1992), the future United States Navy Rear Admiral; one of the first programmers of the Harvard Mark I computer in 1944; invented the first compiler for a computer programming language; one of those who popularized the idea of machine-independent programming languages which led to the development of COBOL, one of the first high-level programming languages.

You can call me a naïve idiot, but what if into the 'Mark II' has been climbed and deadly shocked not a moth, but a cat?

> Or a programmer?

In fact, the term "bug" to describe defects already has been a part of engineering jargon for many decades and predates computers and computer software. It may have originally been used in hardware engineering to describe mechanical malfunctions.

For instance, Thomas Edison wrote the following words in a letter to an associate in 1878[6]:

> It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise — this thing gives out and [it is] then that "Bugs" — as such little faults and difficulties are called — show themselves and months of intense watching, study and labor are requisite before commercial success or failure is certainly reached.

In '40 every computer guy was a radio mechanic or somehow similar to it (you cannot operate an old fashioned computer without being able to repair it several times by a day by your own hands and an soldering iron). So, computer operators were already familiar with the engineering term 'bug'. They amusedly kept the insect in the computer log, because it is really fun to discover a personalization of 'a bug'. That's why they noted it as the "*First actual case of bug being found*" :)

> Today this log book, complete with attached moth, is part of the collection of the Smithsonian National Museum of American History.
>
> Why a log: on those days every interaction with the computer had to be noted on a paper.

And Grace Hopper did not find the bug, as she readily acknowledged that. The notation "*First actual case of bug being found*" was made by a group of computer operators, including William "Bill" Burke — maybe he wrote the famous sentence.

[6]Edison to Puskas, 13 November 1878, Edison papers, Edison National Laboratory, U.S. National Park Service, West Orange, N.J., cited in Hughes, Thomas Parke (1989). American Genesis: A Century of Invention and Technological Enthusiasm, 1870-1970. Penguin Books. p. 75. ISBN 978-0-14-009741-2.

And the date in the log book was September 9, 1947, not 1945. The related term "debug" also appears to predate its usage in computing: the Oxford English Dictionary's etymology of the word contains an attestation from 1945, in the context of aircraft engines.

15) Bug Reporting

The process of explaining the steps, which lead to the bug happens. This notice with a proven screenshot should be stored in a bug-tracking system [p.25].

16) Bug Tracking System

Any software that facilitates the collaborative recording and status tracking of defects and changes.

Usually the https://www.atlassian.com/software/jira system is used for this.

> In fact, Jira is not a bug-tracking system, it's a Task-tracking system. but any defect recorded can be viewed as a 'task for fixing', so nobody cares.

I'll preffer https://www.mantisbt.org/.

Choose your own Bug Tracker.

17) Bug Verification

A process, where the tester try to reproduce on new build each bug, which was reported by developers as 'Fixed'.

18) Build

A compiled version [p.71] of a program.

Any program.

It comes from the https://en.wikipedia.org/wiki/Software_versioning.

19) Change Management

This is an approach to controlled way to effect a change, or a proposed change, to a product or service.

It is not only about software testing. But if you tend to be a manager in testing area. . . [7]

19.1) 'Change management' principles

1. At all times involve and agree support from people within system (system = environment, processes, culture, relationships, behaviors, etc., whether personal or organizational).

2. Understand where you/the organization is at the moment.

3. Understand where you want to be, when, why, and what the measures will be for having got there.

4. Plan development towards above No.3 in appropriate achievable measurable stages.

5. Communicate, involve, enable and facilitate involvement from people, as early and openly and as fully as is possible.

20) Change Request

A formal proposal for an alteration to some requirements or functions.

[7] See a nice 'How-to introductory guide' to Change management area at http://www.change-management.com/how-to-guide.htm

Often arises when the client want an addition (or alteration) to the agreed-upon deliverables for a project. Such a change may involve an additional feature or customization or an extension of service, among other things. So it had to be analyzed apart.

Because change requests are beyond the scope of the agreement, they generally mean that the client will have to pay for the extra resources required to satisfy them.

21) Checklist-based Testing

Stupid term, but it is very human. . .

If you need to dig a hole in the mother Earth, you will need a spade. And the process of creating a hole will be called 'to dig', not something like 'spade digging'. Because it doesn't matter, how you will dig a hole. You will use a spade, of course. Or your hands.

When you will use in testing a checklist, this is still the same testing, right?!

But if you want to feel yourself less dumb, you can say, that you will use the **Checklist-based Testing**. Something unusual! Something great! A new type of testing!

And if someone will interfere, just tell him following magic words: «*A checklist-based Testing is a test design technique whereby the experienced tester uses a high-level list of items to be noted, checked, or remembered, or a set of rules or criteria against which a product has to be verified. Step aside, whiffet!*»

All the chicks will starve to death admiring you.

22) Compatibility Testing

This is a part of non-functional testing. During this type of testing we try to understand, if the tested application is compatible with the computing environment or other specified application.

For example, an app designed to retrieve data from Internet should be compatible with all type of networks. Or an app, designed to work with Excel 97 format (.xls) should work with Excel 2010 format (.xlsx)

Often you will hear about Browser compatibility testing, which can be more appro-

priately referred to as user experience testing, but anyway. This requires that the web applications are tested on different web browsers, to ensure the following:

- Users have the same visual experience irrespective of the browsers through which they view the web application.
- In terms of functionality, the application must behave and respond the same way across different browsers and sometimes — operating systems.
- Hardware (different phones and applications).

23) Component Testing

A **Component** is an abstract common term. We suppose here any minimal software item that can be tested in isolation, without interaction with any other software items.

> E.g., user only can create new profile, but all other functions like setup user avatar or name are unavailable (or are not tested for some reason). The user profile is just a Component from the System point of view, and inside of this Component the ability to add an avatar is just a Component too.
>
> Your finger is a component of your hand. And your hand is a component of your body. And your body is a component of your country.
>
> Any complex System can be just a component of any other System.

Component Testing is aimed at verification of the correct work of **individual** software components, such as specific functional areas or/and features (or sub-features) without ensure that any other components works as expected.

It may include the functional testing, as well as testing of designs and content of the features or/and functional areas.

The **strong side** of this approach is obvious — the component can be tested from A to Z.

Common **weakness** of this approach: apart some components can work as expected, but then they are engaged in a system, they can interfere with other components and

some unexpected defects may appear. That's why only component testing will never be enough.

Usually internet shops are deployed as a bunch of components, which are already integrated. This means, that a clear isolation of some Components for testing purposes cannot be achieved. That's fine, because we always provide our shops packed with all amount of functionality.

24) Continuous Integration (CI)

In software engineering this is a practice of merging all developer working copies with a shared mainline several times a day.

It was first named and proposed as part of extreme programming (XP). Was originally intended to be used in combination with automated unit tests written through the practices of test-driven development.

Initially this was conceived of as running all unit tests and verifying they all passed before committing to the mainline. This helps avoid one developer's work in progress breaking another developer's copy. If necessary, partially complete features can be disabled before committing using feature toggles.

Continuous integration isn't universally accepted as an improvement over frequent integration.

25) Cost of Quality

This is a calculation of the total cost of quality-related efforts and deficiencies. The results of such calculations sometimes lead to sever managerial decisions at the whole Project scale.

It was first described by Armand Feigenbaum in a 1956 Harvard Business Review article.

Cost of Quality is the total costs incurred on quality activities and issues and often split into prevention costs, appraisal costs, internal failure costs and external failure costs.

If you don't know how much $$ costs the last founded Defect, then you cannot know

the Cost of Quality on your Project.

Advanced in the Project, this amount grows, because with the time each Project engage more and more software, time and materials. Sometimes managers try to reduce the cost of quality without reduce the quality. Well, sometimes this help a lot. Sometimes this is not the smartest decision. It depends of...

Check the 'Cost of poor quality' ideas https://en.wikipedia.org/wiki/Cost_of_poor_quality.

26) Coverage

The degree, expressed as a percentage, to which a specified coverage item has been exercised by a test suite.

27) Criteria

A standard of judgment or criticism.

Or a rule or principle for evaluating or testing something.

28) Debugging

A process, where developers start to prove their best skills.

The debugging is not the process of fixing bugs. Usually it looks like someone deliberately send to a program data with errors, trying to figure out where the calculation will be incorrect. When a weak point will be discovered, then the bug-fixing will begin.

In ancient times (1940-1970) the debugging was the primary testing approach. Those programmers had to understand clearly what (and how) the computers doing their job. And they act respectively.

29) Decision Table Testing

An excellent logical tool to capture certain kinds of system requirements and to document internal system design.

They are used to record complex business rules that a system must implement.

They can serve as a guide to creating test cases. Decision tables are a vital tool in the tester's personal toolbox.

Not every analyst, designer, programmer and/or tester are familiar with this technique[8].

Sad but true.

30) Defect

A flaw in a component or system that can cause the component or system to fail to perform its required function, e.g. an incorrect statement or data definition.

Usually we name it Bug [p.22].

30.1) Defect Management

This is a process of

- recognizing,
- investigating,
- taking action
- and disposing of defects.

It involves recording defects, classifying them and identifying the impact on the project or tested application.

[8]Quote from «A Practitioner's Guide to Software Test Design» by Lee Copeland, 2004, Artech House, Inc.

30.2) Defect Report

A document (an item in Jira) reporting on any flaw in a component or system that can cause the component or system to fail to perform its required function.

30.3) Priority and Severity

Consider the impact of Defect at the whole system.

And here is two terms for this classification:

30.3.1) Severity

Critical — such severity is assigned when you lose the ability to work with the System because a Bug happens. You have lose information, or the Product cannot do his main task, or the Device is reset because of overloading... Almost every Bug at Checkout is Critical :)

Major – well, you can somehow continue to work, or you can switch to another task, but the system is seriously affected by this Bug.

Minor – bug is available, but it does not adversely affect the Product functionality.

Who is charged with authority to set the Severity?

First of all — the tester, who had discovered the Bug.

Second — the team lead.

Or the Manager.

Or the Client itself.

30.3.2) Priority

The level of business importance assigned to an item, e.g. defect.

For example, on our site user can buy some cool stuff, and on the checkout the field 'Provide your credit card info' is missed. This is a Critical bug with High priority.

Or on our site user can buy some cool stuff, and on the checkout the field 'Provide your credit card info' is available, but is named 'Provide your credit card info hzhzhzhzhzh'. This is a Major bug with High priority.

Or on our site user can buy some cool stuff, and on the checkout the field 'Provide your credit card info' is available, but is named 'Provide your credit card' instead of 'Provide your credit card info'. This is a Minor bug with High priority.

Or on our site user can buy some cool stuff, and they can download our worldwide stores list, but the file is missed on the file server. This is a Major bug with Minor priority. Yes, this is a bug, it is important, but we will fix it when we will have enough time. Forget about it. . .

31) Defect-based Test Design Technique

Well, this is not a technique (an approach, I suppose), but it's worthing your attentive attention.

If you will focus on searching defects, not on checking if requirements are implemented — what will you see?

When the target defects are determined based on taxonomies (a Taxonomy is a hierarchical list) that list root causes, defects, and failures, adequate coverage is achieved when sufficient tests are created to detect the target defects and no additional practical tests are suggested.

Example: list some potential defects for your app. Or make some assumptions, like 'User add several items to cart, and each entry will not disappear with each new item added' and — check it out. If this will happen for real — hooray, you just supposed a bug and you really made him discovered!

If not — suppose something else, and check it out.

This approach is far, far away from reading basic requirements, add an item to cart, as expected, and declare, that everything is ok.

32) Development Environment

A dev environment is any infrastructure, where the developers has the full authority to make any changes they needed.

Opposite is the Testing Environment. This can be a server with a full replication from Development server, but only QA makes decisions about the updating anything on their environment.

By the way, the software, aims to help in programming is a Development Environment too. But it is called 'Integrated Development Environment'.

> E.g., the 'Eclipse' is an IDE (Integrated Development Environment). And 'IntelliJ IDEA' is an IDE (Integrated Development Environment). And even MS Excel can be an IDE for VisualBasic developers.

33) Development of Test Cases

This is the best process in the testing work!

Development, development, development, development, development, development, development, development, development of Test Cases!

Can be done in a very clever way[9].

Or in a very dumb way.

34) Documentation Testing

A confusing term. There are no reason for test the documentation. Use it in testing, that's all!

Just grab requirements and think:

- what should be implemented,
- what can be tested.

Figure 3: 'Sorry, son...there's no app for that'

You will need only Notepad and your own imagination for this.

Note all your ideas and questions, search for answers, and repeat steps one by one (this is an incremental task) till the end of your career in testing.

35) Domain Analysis Testing

A very mature test design technique.

Used to identify efficient and effective test cases when multiple variables can or should be tested together.

Implies a lot of equivalence partitioning and boundary values analysis, so a junior tester will never be able to do it.

Not every Senior Tester is familiar with this advanced approach to design test cases either.

[9]The righteous way.

Sad but true.

36) End-to-end Testing

A convenient expression for establish the complexity of system being to test and the nature of test cases, designed for this.

Consider a platform, where individual component systems are integrated with each other, so a new shop (a new system) can be created without having to develop applications from new. Complexity of such software requires to test whether the flow of an application is performing as designed from start to finish.

Example — Promotions. Every step can be tested apart, but it is more important to be sure that the whole process (create and apply new Promotion) works as expected.

The purpose of carrying out end-to-end tests is to identify system dependencies and to ensure that the right information is passed between various system components and systems.

Someone can act such type of testing like very annoying. Well, complex thing requires complex testing.

37) Entry and Exit Criteria for Testing

> Understand what is a Criteria first [p.30].

Sometimes is very hard to agree about when we can start or stop testing.

For example, developers already had developed something, and Client call us to start testing. In his opinion, this is enough for starting testing — here are some functionality, just use your imagination and common sense and go on. But we still have no requirements, and we cannot be sure, that we are on the same page with developers about what and how should work. Can we just shut up and let's the mortal testing begin, or we will hang on our hands and refuse to start, because we had no time for preparation, no test cases, no requirements. . . ?

Yes, if we have common sense and we can discuss with our customer any issue, we can start without any requirements and fears.

But if the Customer is Al Capone itself, and in case of something will fail on our website you will wake up in the morning being silently buried in a black box under the heavy ground. . . We need to agree something before the process will start.

Entry/exit Criteria for Testing are a set of generic and specific Conditions discussed between the client and executor. This criteria doesn't exist as a law, and always can be/should be revisited.

37.1) Entry criteria for testing

The purpose of Entry criteria is to prevent a task from starting which would entail more (wasted) effort compared to the effort needed to remove the failed entry criteria.

Entry criteria for testing can be following:

- for each functionality we have a requirement available,
- we have enough human resources for the testing process,
- all third-party units are available for being used with our system under test,
- build is frozen and no new functionality will be added or changed...

37.2) Exit criteria for testing

The purpose of Exit criteria is to prevent a task from being considered completed when there are still outstanding parts of the task which have not been finished. Exit criteria are used to report against and to plan when to stop testing.

Exit criteria for testing can be following:

- all critical functionality was tested and here is the complete report,
- all discovered critical bugs was fixed and retested,
- the time for testing is out...

By the way, sometimes **Exit Criteria** for Testing is called as **Completion Criteria**.

38) Equivalence Class Testing

> This should be called **Equivalence partitioning of test cases**, but nobody cares.

Don't pretend that this technique can be used to reduce the number of test cases without lowering the test coverage, because reducing test cases always lower the test coverage level. Automobiles was created for consuming petrol or for carrying human beings from the maternity hospital to the cemetery as fast as possible?

38.1) Class

A class is a group of some items with some common attributes, characteristics, qualities, or traits.

Grouping is useful for science, it helps to discuss as single entities whole classes, not separate items.

Trick is that *one item* can inherit different characteristics and qualities *at the same time*. For example, an usual *man* can be added to several groups (classes) simultaneously:

- human being
- bus driver
- pig
- father

38.2) Equivalence

There are a lot of terms for showing that two items are equal: equal, comparable, identical, correspondent, duplicate, matched, uniform.

Any two things can be called equal, if they have the exactly same attributes.

Example: you have two brand new bottles of whiskey.

They exists apart in this Universe, but now they are identical, equal, the same, they duplicate each other.

If you will open one bottle, they immediately will become unequal (one is opened, the other is sealed).

But once both bottles are opened, they again become equal.

Ok, this bottles cannot be called 'equal' for real. But they can be declared as 'equal' only for our convenience.

There is an *meta* word for such statement: this two bottles are not equal (identical), but for the final result you can act with them like they ARE identical. This is called *Equivalence*.

Other example: you should use a passenger train for travel in metro. Will matter what kind of train you will have to use? His color, capacity, length, personal number and the manufacturing year, or something like that?

No! All those DIFFERENT trains from your route WILL BE equivalent one to others, while your journey.

If you will use one of thousand trains, you can assume, that all other trains can complete the same task.

Again, all those trains are not equal one to others. Each of them has his own number, different colors, different history and brake conditions, they was released in different ages. They are not equal, they are only equivalent.

Warning! Equivalence is not a substitution or something totally equal. 'Equivalence' is not 'Equality'.

38.3) Equivalence partitioning of test cases

An equivalence class *can* consists from a set of data that is treated the same by the module or that should produce the same result. Any data value within a class is equivalent, in terms of testing, to any other value. Specifically, we would expect that

if one test case in an equivalence class detects a defect, all other test cases in the same equivalence class are likely to detect the same defect.

A group of tests forms an equivalence class if you *believe* that:

- They all test the same thing.
- If one test catches a bug, the others probably will too.
- If one test doesn't catch a bug, the others probably won't either.

This approach assumes, of course, that a specification exists that defines the various equivalence classes to be tested.

And you should clearly understand, that your 'grouping' of test cases can be formally good, but totally wrong.

The **main gap**: for example, you get your car from the car service, and the mechanic says: «*We had to check all the wheels. Well, we checked only two wheels with brakes. You know, they're all four are equivalent to each other, so we have checked out those two, they are ok, so we* assume *that the remaining wheels are ok too. Have a nice day*». Will you have a nice day?

Divide into equivalence classes only Test cases, not the Functionality of a product.

39) Exhaustive Testing

A test approach in which the test suite comprises all combinations of input values and preconditions.

> All combinations!

As you know, nobody can test all input combinations into a modern software, so every moron knows that exhaustive testing is totally impossible.

In fact, exhaustive testing it is totally possible, but not all the time necessary. You cannot test everything. But you can predict everything that can happen with the tested software. Predictability lead to control.

40) Exploratory Testing

A very cool, but informal test design technique.

It is hard to explain this approach in a few words, because it is completely unusual for almost all testers. But it can be demonstrated and explained step-by-step (it's like riding a bicycle).

Here is the simplest explanation of this approach:

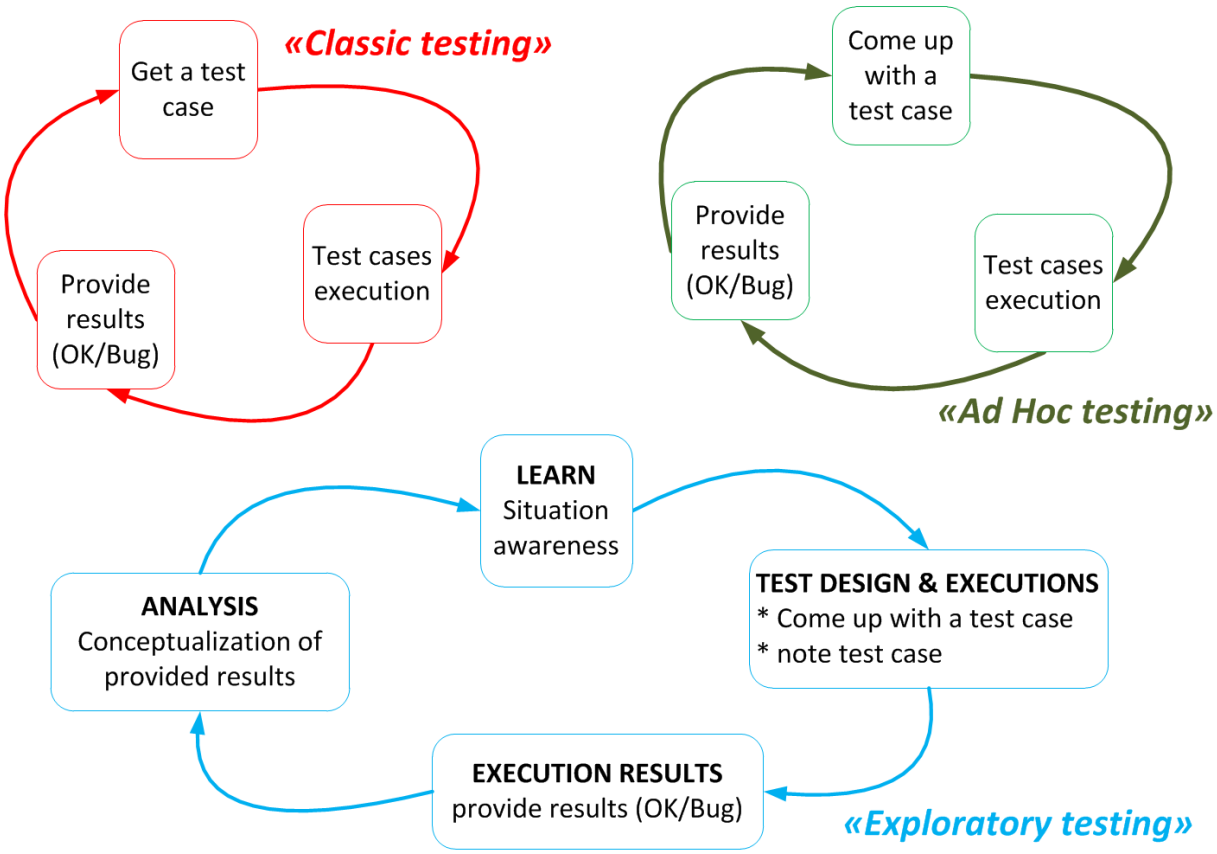

Figure 4: `Simple explanation of Classic & Ad Hoc & Exploratory testing`

41) Feature

An attribute of a component or system specified or implied by requirements documentation (for example reliability, usability or design constraints).

42) Function

> Cat's functions are 'to make purrrr' and to hunt mouses.

The **Function** is what a given entity does in being what it is.

So, the **Functionality** is an ability of software to perform a task.

Word is taken from the Latin "*functio*" - to perform.

43) Functional Requirement

In programming a Function is a named section of a program that performs a specific task.

In the same way that a File is a named section of some data on your hard-disk drive.

> The term function is also used synonymously with operation and command.
> For example, you execute the 'delete' function to erase a word.

A **Function**, in its most general use, is what a given entity does in being what it is ([p.42]).

The **Functionality** is an ability of software to perform a task.

A **Functional Requirement** is an abstract, which describes the functionality task that a software system should **do** for the user needs.

> And a Non-functional requirements usually place constraints on **how** the system will do so. See [p.50] for details.

An **example** of a functional requirement would be that a system must send a an email whenever a certain condition is met (e.g. an order is placed, a customer signs up, etc).

> A related non-functional requirement for the system can be that emails should be sent with a latency of no greater than 12 hours from such an activity.

Then **Functional Requirements** are done, it's time to write **Functional Specifications**.

You understand the difference between Requirements and Specifications?

44) Functional Specification

This is an extention of a Functional Requirement [p.42].

Functional Requirement is an abstract, which describes *what* a software system should do from the user point of view.

> I want to have a red car for driving on the streets.

Functional Specification describes *how exactly* the functional requirement will be implemented.

> I want to have a car painted in #ff0000 with engine 1.6 and McPherson suspension.

45) Functional Testing

Any testing based on an analysis of the specification of the functionality of a component or system.

The functional testing will be executed to evaluate the compliance of a system or component or third-party with specified functional requirements and corresponding predicted results.

Functional testing is performed for each planned feature and is guided by approved by client requirements.

46) Impact Analysis

In fact, this should be named **Change impact analysis**, but anyway — this is a method to identify the potential consequences of a change in a complex system.

Consider 'Notepad' as a simple software under test. We can add new functions (or modify existed). We would like to set assure, that this change will not alter existed functions. How?

We can proceed to regression testing (consider the level of boring testers and costs). Or we can setup a table, where we will trace all Notepad functionality and their junction points.

Now, we can see, which functionality will be affected by changes, and can retest only areas, where changes will have the impact.

Problem with this impact tables is the same as for any other project documentation — they should be updated on a regular basis.

And yes, if the project is short or 'small' — we can omit such documents.

47) In Scope / Out of Scope

A 'scope' is an abstract term for setting the area or subject relevant for the discussion.

> For example, two men meet for a coffee. For sure, they will start to discuss business issues, and only business issues will be in scope of their conversation. Any other stuff (cars, pets, women) will be 'out of scope'.
>
> But when this two will have to exterminate the second bottle of 'Johnnie Walker', business issues will become 'out of scope', and chat will be oriented to cars, women, pets and other important for business issues.

Same thing relate to testing: testers have to establish from the start which functionality will be tested during next testing session; they should list all functions 'in scope' and all the functions, which will not be tested during this session, and agree with Development and Client about why this will be and those will not be tested.

By the way, such documents are called Testing Scope and they are an indispensable part of any Test Plan.

Why testers must define the Testing Scope? This should be done especially to cover testers heads in case of strange questions appears, like *'But why this and those things wasn't tested?'*.

Sometimes testers can save their lives with this 'in scope' and 'out of scope' lists.

Sometimes not.

48) Integration Testing

Integration testing is a process, where different parts of system under testing are engaged one by another.

Such testing is done to ensure that all engaged components (may be included third-parties) maintain data integrity and can operate in coordination with other system artifacts in the same environment.

This type of testing is performed to expose defects in the interfaces and in the interactions between integrated components or systems. There are a huge possibility, that each software component can be bugless apart, but on the integration phase can appear some bugs, that cannot be imagined, supposed and identified. They can be discovered only at the Integration testing phase.

49) Item Pass/Fail Criteria

Understand what is a Criteria first [p.30].

Pass/Fail Criteria will be used to ensure the completeness of the test process.

It sound very obvious, but try to explain your judgement about *'How do you know, that the testing of Change Avatar in User Profile is done?'* Sometimes is very easy to understand this (I can change user Avatar, all satisfied citizens are going home). But sometimes not.

Where is the previous user Avatar picture? It is still available, but was marked as 'Deleted'? Or it was completely deleted from file server?

50) Iterative Development Model

Check http://en.wikipedia.org/wiki/Iterative_and_incremental_development, please. There are more nice pictures too.

Iterative Development ModelIterative and Incremental development is any combination of both iterative design or iterative method and incremental build model for software development. This process may be described as an 'evolutionary acquisition' or 'incremental build' approach.

The relationship between iterations and increments is determined by the overall software development methodology and software development process. The exact number and nature of the particular incremental builds and what is iterated will be specific to each individual development effort.

Iterative and incremental development are essential parts of the Modified waterfall models, Rational Unified Process, Extreme Programming and generally the various agile software development frameworks.

It follows a similar process to the 'plan-do-check-act' cycle of business process improvement.

51) Maintenance of Test Cases

Maybe, the most annoying process for every smart tester in the world.

Requirements (and Expectations) in a Product can happen many times during a product development lifecycle, so test cases can become totally obsolete. Rewrite them all, or write new test cases.

What would you do with this issue?

Real challenge, huh?

52) Metaphor

A *Metaphor* is a figure of speech that refers, for rhetorical effect, the attributes of one thing by mentioning another thing.

It may provide clarity or identify hidden similarities between two ideas. A *Metaphor* directly equates this two items, and does not use "like" or "as" as does a *Simile*.

Example[10]:

> The moon has looming over the horizon like a big orange.

From the other side, nobody will understand the same metaphor, used vice versa:

> An orange has lying on a plate, like a moon over the horizon.

Sound stupid, isn't it?

Some methapors works perfect in both sides, some not.

53) Migration Testing

Consider that a company switch from one DBMS to another (from MySQL to Oracle). Or the architecture of DB requires a massive innovation. Then will be required a database Migration testing.

Database migration may be done manually, but it is more common to use an automated ETL (Extract-Transform-Load) process to move the data.

Database migration testing may encounter problems when:

1. The data in the source database(s) changes during the test;
2. Some of the source data is corrupt;
3. The mappings between the tables/ fields of the source databases(s) and target database are changed by the database development/ migration team;

[10]Yeah, the word "like" was used, but anyway

4. A part of the data from the source database is rejected by the target database;

5. Data migration takes too long because the database migration process is too slow or the source data file is too large.

The test approach for database migration testing consists of the following activities:

1. Design the validation tests. In order to test database migration, SQL queries are created either by hand or using a tool, such as a query creator. The test queries should contain logging statements for the purpose of effective analysis and bug reporting after the tests are complete.

2. Set up the test environment. The test environment should contain a copy of the source database, the ETL tool (if applicable) and a clean copy of the target database. The test environment should be isolated so that it is not changed externally during the tests.

3. Run your validation tests Depending on the test design, the database migration process does not need to completely finish before starting tests.

4. Report the bugs.

54) Monkey Testing

A metaphor for naming the producer of any input for a program.

The name 'monkey' comes from the the 'Infinite monkey'[11] theorem. There is an adage that '*thousand monkeys at a thousand typewriters will eventually type out the entire works of Shakespeare*'.

For example, a monkey test can enter random strings into text boxes to ensure handling of all possible user input or provide garbage files to check for loading routines that have blind faith in their data. The test monkey is technically known to conduct Random testing.

[11] http://en.wikipedia.org/wiki/Infinite_monkey_theorem

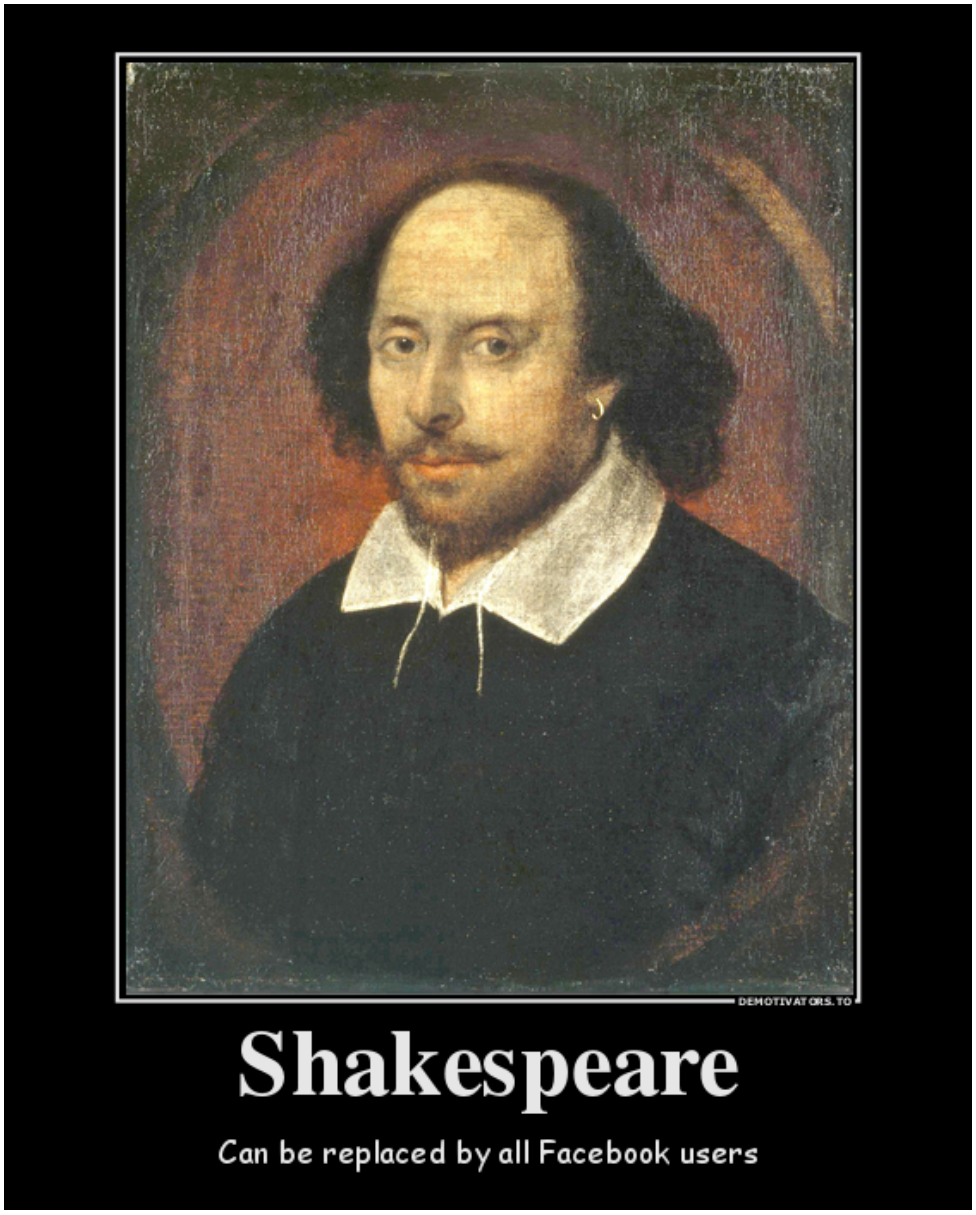

Figure 5: `This is Shakespeare. He looks at you as at an infinite monkey`

55) Negative Testing

A test designed to determine the response of the system outside of normal parameters. It is designed to determine if the system performs error handling with an other than

expected input.

Understand, that when you check something like 'Enter following abracadabra and system should explain that there is an unaccepted symbol, please enter data without errors' — this is Positive Testing ([p.55]), because you have the expectations about systems response.

56) Non-functional Requirement

A requirement that does not relate to functionality, but to attributes such as reliability, efficiency, usability, maintainability and portability.

Typically non-functional requirements fall into areas such as:

- Accessibility
- Capacity, current and forecast
- Compliance
- Documentation
- Disaster recovery
- Efficiency
- Effectiveness
- Extensibility
- Fault tolerance
- Interoperability
- Maintainability
- Privacy
- Portability
- Quality

- Reliability
- Resilience
- Response time
- Robustness
- Scalability
- Security
- Stability
- Supportability
- Testability

  How you will test the Effectiveness of MS Word processor? :)

Non-functional requirements are sometimes defined in terms of metrics (something that can be measured about the system) to make them more tangible. For example, the Effectiveness of MS Word processor can be described binary — it is effective or not. Or it can be described, for example, from 1 to 10 points.

Suppose the level of user excitement when he drive an Cadillac Eldorado 1959 — this is a non-functional requirement.

Suppose the level of user annoying when he use an app, which crashes each time when wi-fi connection is down for a second — this is a non-functional requirement.

56.1) Non-functional Testing

Testing the attributes of a component or system that do not relate to functionality, e.g. reliability, efficiency, usability, maintainability and portability.

57) Pair Testing

Sometimes more than one human beings work together on a single task. If the task is about testing something — welcome to Pair testing.

One does the testing and the other analyzes or reviews the testing. They can always switch their roles.

Pair Testing can significantly speed up the testing. One person is a **Driver**, second is a **Navigator**.

Figure 6: `Beyond the limits`

Both minds can generate testing ideas, and immediately execute them. While Navigator stay for make a note about what was doing and the results, Driver can run and generate next idea, and so on. This can be done between one tester and developer or business analyst or between two testers with both participants taking turns at driving the keyboard. No limits, really!

This approach come from Exploratory Testing area [p.41], and it's very controversial. From the manager's point of view, Pair Testing looks like TWO working units do ONE job, so one testing task costs double. While this, will be better if those TWO working units were doing at the same time TWO testing tasks.

For sure, this if very OK, when the task is to dig a hole at the land. And sometimes

testing really looks like digging.

> You remember the '*You see, in this world there are two kinds of people, my friend: those with a loaded gun and those who dig. You dig*' © from 'The Good The Bad and the Ugly' movie?
>
> We must understand and admit this truest statement.

But there are a lot of tasks in testing, when we need more brains for work, than hands. For such tasks Pair Testing can be a wonderful approach, where TWO smart people can work as ONE unit best and quicker than two working units apart.

At the end of Pair Testing this two persons survive so many challenges, that as a nice persons, they have to marry each other, so be careful with this software development technique.

58) Pairwise

A powerful test design technique. Test cases are designed to execute all possible discrete combinations of each pair of input parameters.

It can significantly reduce the number of all combinations of variables to test all pair combinations.

Usually this powerful technique is used as a waste of time with respective attitude. Please, do not try it at home. You can fake that you totally understand what is talking about.

59) Performance Testing

Initially, *Performance* is an activity that a person does to entertain an audience, such as singing a song or acting in a play on a Broadway.

In technical language, *Performance* is the ability of a mechanism to do an required action (or activity). It can be evaluated only as a numerical value.

Example: the old Cadillac Eldorado 1953 engine, we have reached a level of 100 km/h in 42 seconds, and this was possible because of 210 horsepower expected and available.

Performance can be evaluated, varied and measured in follow mode: 'Let's find out, if will be possible to reach a level of 100 km/h in 32 seconds at the old Cadillac Eldorado 1953 engine, if we will setup 240 horsepower instead of 230'.

Performance can be evaluated, varied and measured in follow mode 'Let's find out, how much time the old Cadillac Eldorado 1953 engine will work, if we will put a brick on the accelerator pedal and will have to wait'.

59.1) Load & Stress testing

Performance testing simply implies **Load** and **Stress** testing. Difference between this testing types is logical and lies in the fact that different types of performance testing answer to different business questions.

Load testing help us to understand the behavior of the system under a specific **expected** load.

Stress testing is normally used to understand the **upper limits** of capacity within the system.

Example: there are an application with DB based on the MySQL. MySQL is a popular choice of database for use in web applications, an open-source relational database management system.

MySQL can easily support 200 hits per second (12 000 per minute, and 720 000 per hour, and 17 280 000 per day). How to test the performance of this web-site?

We can use JMeter for generate 100 hits per second. Is anything ok?

Let's generate 190 hits. Is anything ok?

Let's generate 210 hits. Is anything ok?

Let's generate 300 hits. Is anything ok?

Let's generate 500 hits. Is anything ok?

Let's generate 210 hits during 6 hours. Is anything ok?

We have the same utils and we do the same things. But first it was simple Performance testing, then it became Load testing, then it became Stress testing, then it again became Load testing.

59.2) Reliability Testing

'Reliability' is a term for testing a software's ability to function, given environmental conditions, for a particular amount of time.

For example, turn on the coffee machine and use it 100 hours without any stop. Will be some problems in the software and functionality? Will be coffee available after this 100 hours? And if Yes, then what about 200 non-stop hours? Or 1000?

59.3) Maintenance Testing

Is that testing which is performed to either identify equipment problems, diagnose equipment problems or to confirm that repair measures have been effective.

It can be performed at either the system level, the equipment level, or the component level.

Maintenance testing uses system performance requirements as the basis for identifying the appropriate components for further inspection or repair.

A good testing program will maintain a record of test results and maintenance actions taken. These data will be evaluated for trends and serve as the basis for decisions on appropriate testing frequency, need to replace or upgrade equipment and performance improvement opportunities.

60) Positive Testing

This is nothing else than the main approach to testing.

Our main intention is to prove that an application will work 'as expected' on giving valid input data.

In Negative testing ([p.49]) our intention is to prove that positive testing did not cover some situations.

Suppose following situation: I will type an email address without '@', system should alert me about my error. What kind of testing type is that? Is positive, or negative?

61) Precondition

Environmental and state conditions that must be fulfilled before the component or system can be executed with a particular test or test procedure.

61.1) Postcondition

Environmental and state conditions that must be fulfilled after the execution of a test or test procedure.

Warning, this is not the Expected result [p.8].

62) Quality

The degree to which a component, system or process meets specified requirements and/or user/customer needs and expectations.

A *value* to some *person* at some time.

63) Regression Testing

This process is very annoying and costs are high, but we can drive it in a smart way, trough Impact Analysis ([p.44]).

So, every system (social, political, military or software system) should continuously expand his functional possibilities to survive. This is a **Progress**.

But with more opportunities, the more relationships appears, and the more chances to find a defect. Or not to find it. This is a **Regress**.

Regression testing is performed for getting know, if regression appears.

> Testing brings information, remember this simple idea?

This activity only looks like we should test some modification in software to ensure that defects have not been introduced or uncovered in unchanged areas of the software, as a result of the changes made. In fact, 'Regression testing' can be done periodically, without any changes in software.

> Now deal with it.

One of our interest is to discover negative behavior, that's why we are doomed to perform a 'Regression testing' each time when new functionality is added to a software.

And 'Regression testing' it is sometimes performed when the environment is changed.

> Environment, not software. You can call this 'Integration testing', but what kind of Integration it is, if the software still the same?

You will be a great donkey, if you will advance in 'Regression testing' having only old test cases, '*Cause this is just a re-test, right, ma?*'. You will need new test cases because of ANY significant change in software structure.

64) Requirement

A condition or capability needed by a user to solve a problem or achieve an objective that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.

Specification is the next step after Requirement. Requirements tells what to do, and Specification tells how to do it — this can be two different docs.

65) Root Cause Analysis

A root cause is an initiating cause of a causal chain which leads to an outcome or effect of interest. In short: *find why someone was killed, and you can find the killer*.

Root cause analysis is a method of problem solving that tries to identify the root causes of faults or problems. A lot of developers can expect such analyse from testers, but this is insane. Root cause analysis practice solve problems by attempting to identify and correct the root causes of events, as opposed to simply addressing their symptoms. Focusing correction on root causes has the goal of *preventing problem recurrence*, and here testers fails.

There are many different tools, processes, and philosophies for performing Root Cause Analysis. However, several very-broadly defined approaches or "schools" can be identified by their basic approach or field of origin: safety-based, production-based, process-based, failure-based, and systems-based.

66) SCRUM

The term itself comes from Rugby game, and describes a way of starting play again, in which players from each team come together and try to get control of the ball by pushing against each other and using their feet, when the ball is thrown in between their feet.

You can say *'Jump everybody and let's kill them all!'* and millions will respond — yes! Well, this is Rugby.

But you will suppose the SCRUM as a development approach, huh?!

You will even say 'A software development methodology!', do you?!

This is not a methodology.

This is a software development framework (a strategy) for managing product development, driven by agile principles. For being applied, the dev team already should have a development methodology.

Again, framework OVER a methodology.

Figure 7: `This is the real Scrum`

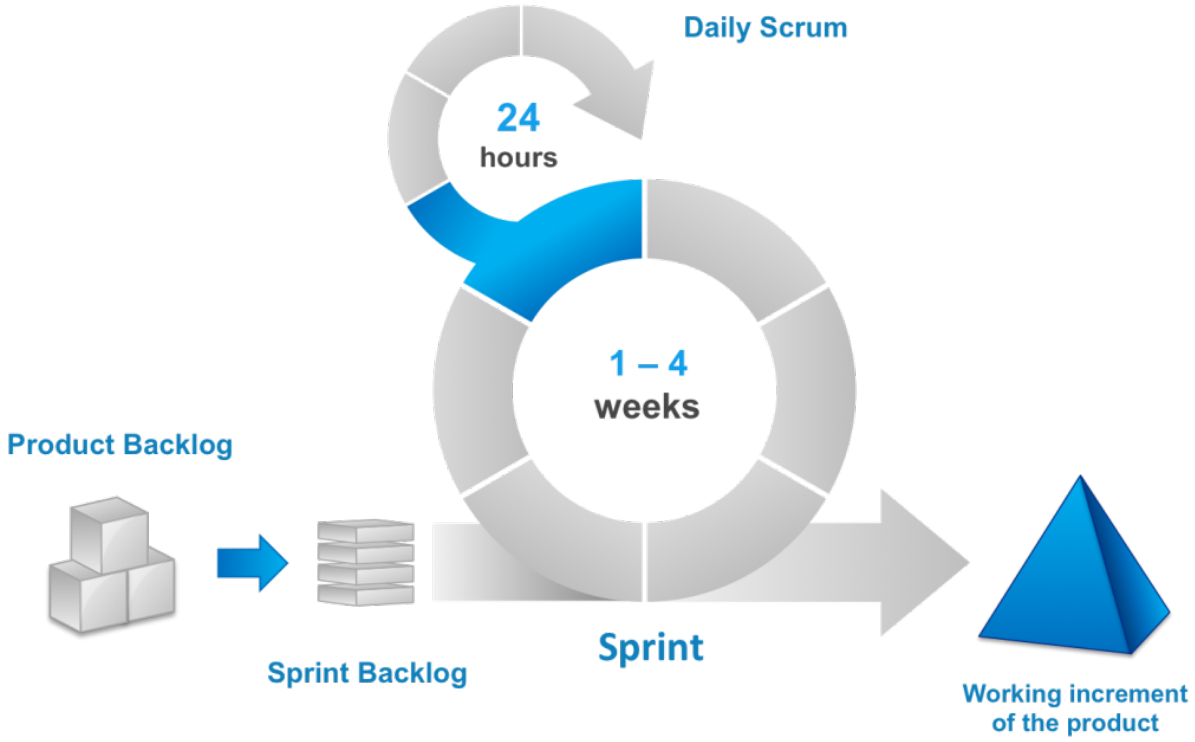

Figure 8: `SCRUM Framework`

```
ELSE
        Fail();
```

See Agile Software Development explanation ([p.11]) & agile principles at (http://agilemanifesto.org/) or GTFO.

67) Security Testing

Usually — a process intended to reveal flaws in the security mechanisms of an information system that protect data and maintain functionality as intended.

It looks like a simple 'Let's find some possibility to stole credit card numbers from our shop!' approach, but in fact Security Testing represent a totally different approach than testing some functionality, involving a lot of programming skills. Usually it looks like someone explore/decompile a program and write his own program for explore and decompile others.

> If you are bored for a little by Functional Testing, and you are looking for something else, then Security is the latest area for you. This is a whole profession, whis involve a lot of programming and administration servers skills and knowledge.

Well, winter is coming... Let prepare for cold times. Are we enough strong for say 'Hello!' to winter? Are our bodies prepared for that?

Imagine a simple Doctor. At university he studied the human body, starting with cutting the frogs in the school laboratory, and then progressively started the autopsy of human bodies in the dissecting room.

Now suppose yourself. Do you know about medicine something more than 'If I'll get a chill, I will drink a hot tea with lemon and Aspirin!?

Feel the difference?

Who can effectively suppose/assume/find some weaknesses in human body? You, with tea and Aspirin, or a Doctor, who has killed during his university ages more frogs than you saw movies with Bruce Lee?

Only just because it is called 'testing' (exactly like 'Functional Testing'!), each year a new generation of junior-jedi testers tries to study 'something new in testing'.

To succeed in security testing, it is necessary not only to understand how something is built, but why someone made certain technical decisions.

68) Server

A program, that can simultaneously accept a lot of references to it, put them in a queue and execute them successively, one by one.

Depending on context, the term can refer to the hardware, where the Server lives.

69) Smoke Testing

This is just a convenient metaphor, defining a process of superficial testing, required to being done before the real, deeper advanced testing started.

The idea consist in following: before starting to explore new build, set assure that all at least main functionality (differ from project to project) is on:

- product catalog is available,
- you can add a product to cart,
- you can run trough checkout process,
- user profile are available,
- and so on.

If smoke testing will fail, then the whole testing process will be worthless.

If smoke testing is ok, then we can advance in testing, as expected.

Usually developers call this checking 'sanity testing', others (factory engineers) 'integrity testing'. As manual functional testers, we called it 'Smoke testing'.

> A lot of years ago the testing of electronic devices started with real 'smoke testing' — plug the device to an outlet and set power 'on'.
>
> If you see light and smoke — forget about testing, the device is broken, you will burn.

Smoke testing is always based on a subset of defined/planned test cases, that cover the main functionality of a component or system, to ascertaining that the most crucial functions of a program work, but not bothering with finer details.

A daily build and smoke test is among industry best practices.

70) Sprint

The term *Sprint* is coming from sport. It is the act of running over a short distance at top speed.

> Opposite is 'stayer' — a long-distance runner.

Depending of physical statement, runners specialize in running on long or short distances, because this are two different strategies of running.

> Sprinters runs very fast, but only at short distances (no more than 100 m). They should make an extra effort from start, immediately, without any compromises, without trying to save some power or even air breathing. Human physiology dictates that a runner's near-top speed cannot be maintained for more than 30–35 seconds due to the accumulation of lactic acid in muscles. Just get up and run!
>
> Stayers starts slowly, at first they should try to hold on the group of all runners, at maximum saving energy. They will need all their power and speed in the latter stages of running, closer to the finish band. For example, marathon runners can run 42 km (plus 195 m).

In Agile Software Development ([p.11]) the term 'sprint' means almost the same — a short-time activity (typically one week long), where all the people involved in a

project, being focused on development of some task for the project, give maximum of their power and attention.

It is obvious, that to be a real sprinter 'week by week' without long rest it is impossible. In fact, everyone involved in project use a Stayer strategy. But nobody cares, because not everyone even understand the meaning of 'Sprint' term. You want me to say 'a sprint' instead of 'a week'? No problem. Scrum, sprint, scope — whatever. You are the boss.

71) Staging

In theater a Staging is the platform on which actors perform their 'To be, or not to be. . . ' So, this is a place, where something is shown to an appreciative audience.

A 'staging server' is any server that stores the software with testing purposes, but the environment is almost production-similar.

Usually only client-side testers have access to Staging server, where they can, for example, use real Credit Card for testing.

And customer usually check the software on the staging server, not on testing server.

Sometimes this thing is called UAT — User Acceptance Testing.

72) Strategy

> **Source**: Merriam-Webster's Learner's Dictionary & https://en.wikipedia.org/wiki/Strategy

It is a very ancient word, means "`art of military troops leader`".

It comes from the science (and art) of military command exercised to meet the enemy in combat under advantageous conditions for our glorious troops and disadvantageous conditions for the disgraceful enemy.

> And the enemy tries to do the same, like in chess game.

Strategy is important because the resources available to commandment are usually limited.

Strategic planning and strategic thinking involves

- setting goals,
- determining a general pattern of actions to achieve the goals,
- and mobilizing resources to execute the actions.

You don't really know how this goal will be achieved, because it is a matter of the *Tactics*.

73) Suspension and Resumption Criteria

> Understand what is a Criteria first [p.30].

Entry and Exit criteria are the conditions which when satisfies, the test team starts (enters) the testing process and stops (exits) it.

Suspension and Resumption criteria are the conditions which when satisfies, the test team temporarily suspends (suspension) the testing process and resumes (resumption) it.

A test process may have to be suspended temporarily until the show stopper bugs are resolved.

Examples:

> The project scope was agreed and approved by the client at the time indicated in the project schedule, but something goes wrong with the testing environment
>
> The project scope is agreed, the requirements are defined and approved by the client, but requirements was unexpectedly changed
>
> The credentials to the third-parties or/and external tools was available at the time indicated in the project schedule, but SUDDENLY they are not working.

74) System

A collection of components organized to accomplish a specific function or set of functions.

Your body is a system.

75) Test

Initially a Test is an assessment intended to measure the respondents' knowledge or other abilities. Remember your class years?

In our industry a Test is the process of verifying that a software program works as expected from the point of Verification or/and Validation view ([p.70]).

To perform a Test, brave Quality Assurance engineers should previously create their brilliant Test Cases. For doing this they need in advance the Requirements ready for review ([p.57]).

76) Testability

The capability of the software product to enable modified software to be tested.

Sounds weird, but in fact, this is a good topic for a scientific research in the name of almighty Allah.

77) Test Case

An instruction for create a test situation.

$$\text{Case} = \text{Situation}.$$

Usually test cases looks like a set of

1. input values,

2. execution preconditions,

3. expected results

4. and execution post-conditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement.

Note, that it's all about a the test situation, that can be created artificially.

78) Test Design

The process of analysing everything in software and/or software requirements for testing purposes.

It is somehow similar with inventing poetry without inspiration.

It is not 'How to write test cases. . . ', but after analysing test cases will be created.

79) Test Idea

A conception or a statement about any software expectations or requirements, that potentially can/should be verified with boolean logic (YES/NO, TRUE/FALSE).

Usually each title of a test case is a Test Idea.

80) Test Scenario

The simplest explanation about what steps need to be taken in order to create expected test situation to verify a software requirement without getting into specific detailed test steps, test data, and/or expected results.

Example: "*How to put an elephant into a refrigerator?*"

1. open the fridge door,

2. put the elephant in a refrigerator,
3. close the refrigerator door.

Sometimes waste, complex test cases are preceded by test scenarios.

And it is obvious, that every test case aggregate a test scenario (step one, step two, step three. . .)

81) Test Suite

A set of several test cases for a component or system under test, where the post condition of one test is often used as the precondition for the next one.

Sometime a suite it's just a logical 'folder' for a list of test cases, focused on testing something specific.

One test case can apper in several test suites.

82) Traceability

Traceability is the ability to identify related items in documentation and software, such as requirements with associated tests.

A traceability matrix is a document, usually in the form of a table, that correlates any two baselined documents that require a many-to-many relationship to determine the completeness of the relationship.

It is often used with high-level requirements (these often consist of marketing requirements) and detailed requirements of the product to the matching parts of high-level design, detailed design, test plan, and test cases.

A requirements traceability matrix may be used to check to see if the current project requirements are being met, and to help in the creation of a request for proposal, software requirements specification, various deliverable documents, and project plan tasks.

Common usage is to take the identifier for each of the items of one document and place them in the left column. The identifiers for the other document are placed across the top

Requirement Identifiers	Reqs Tested	REQ1 UC 1.1	REQ1 UC 1.2	REQ1 UC 1.3	REQ1 UC 2.1	REQ1 UC 2.2	REQ1 UC 2.3.1	REQ1 UC 2.3.2	REQ1 UC 2.3.3	REQ1 UC 2.4	REQ1 UC 3.1	REQ1 UC 3.2	REQ1 TECH 1.1	REQ1 TECH 1.2	REQ1 TECH 1.3
Test Cases	321	3	2	3	1	1	1	1	1	1	2	3	1	1	1
Tested Implicitly	77														
1.1.1	1	x													
1.1.2	2		x	x											
1.1.3	2	x											x		
1.1.4	1			x											
1.1.5	2	x												x	
1.1.6	1		x												
1.1.7	1			x											
1.2.1	2				x		x								
1.2.2	2					x		x							
1.2.3	2								x	x					
1.3.1	1										x				
1.3.2	1										x				
1.3.3	1											x			
1.3.4	1											x			
1.3.5	1											x			
etc.…															
5.6.2	1														x

Figure 9: `Traceability Matrix sample`

row. When an item in the left column is related to an item across the top, a mark is placed in the intersecting cell. The number of relationships are added up for each row and each column. This value indicates the mapping of the two items. Zero values indicate that no relationship exists. It must be determined if a relationship must be made. Large values imply that the relationship is too complex and should be simplified.

To ease the creation of traceability matrices, it is advisable to add the relationships to the source documents for both backward traceability and forward traceability. That way, when an item is changed in one baselined document, it's easy to see what needs to be changed in the other.

83) Usability Testing

Usability testing is a technique used in user-centered interaction design to evaluate a product by testing it on users. This can be seen as an irreplaceable usability practice, since it gives direct input on how real users use the system. This is in contrast with usability inspection methods where experts use different methods to evaluate a user interface without involving users.

Even if it is called 'testing', it doesn't look like usual functional testing, done by QA engineers.

> But this is a testing, because it involved some research and it brings information.

Usability testing focuses on measuring a human-made product's capacity to meet its intended purpose. Examples of products that commonly benefit from usability testing are foods, consumer products, web sites or web applications, computer interfaces, documents, and devices. Usability testing measures the usability, or ease of use, of a specific object or set of objects, whereas general human-computer interaction studies attempt to formulate universal principles.

84) Use Case

In software and systems engineering, a *Use Case* is a list of steps, typically defining interactions between a role (known in Unified Modeling Language as an "actor") and a system, to achieve a goal. The actor can be a human, an external system, or time.

In our industry Use Cases become a special form of a Requirement.

> We may say that Requirements evolve in Use Cases.
>
> Or — this is another form of Requirements, nothing to worry about it.
>
> Anyway, Use Cases derive from User Stories ([p.70]).

Use Cases have been widely used in modern software engineering over the last two decades. With its iterative and evolutionary nature, use case is also a good fit for agile development.

When you have good Use Cases, you will be able to create a lot of good Test Cases.

85) User Story

A high-level user or business **requirement** commonly used in agile software development, typically consisting of one or more sentences in the everyday or business language capturing what functionality a user needs, any non-functional criteria, and also includes acceptance criteria.

Like Use Cases ([p.69]), the 'User Story' is a special form of a Requirement ([p.57]).

86) Verification

This is the basic level of any testing, it comes from the verb `to verify`.

Verification is the simplest confirmation of any *expectation* (or assumption) by examination and through provision of objective evidence that specified requirements have been fulfilled.

Example: have a pie.

> Or a *borscht*.
>
> Or a car.

Verification is the basic checking that this pie can be eaten, and if so, then the quality of the pie is ok.

Sometimes it is more than enough just to be sure, that software does what was expected.

But sometime not.

I am sure, that you will expect, that the pie should be tasty, right?!

86.1) and Validation

Confirmation by examination and through provision of objective evidence that the informal requirements for a specific intended use or application have been fulfilled.

Any pie can be cooked as expected by recipe, and this can be tested (verified). But even if recipe is only one, the taste of pies will vary from one cook to other. And because it is very hard to set the requirements of 'how to do a tasty pie', this start to lead to informal requirements. And such reqs can be tested only at the *Validation* level.

This two terms are always presented in testing, but they are not declared.

Any test case can be declared as '*This is a validation test*' or '*This test is about verification only*'. But obviously, nobody will ask '*Let's write some validational test cases, please*'.

87) Version

The versioning is the process of assigning either unique version names or unique version numbers to unique states of computer software.

Usually the version of a program is presented like this:

> ver. `3.12.242`

Let's read it:

Major release — #3.

> The software was developed to the third release.

Minor release — #12.

> In the third major release was added some functionality. Maybe was added 12 new functions in one day.
>
> Maybe not, maybe to 11 already existed functions today was the 12.
>
> Maybe in major release #3 was added only one new function, with 11 important updates.

Build — #242

> I can suppose, that someone had a bad day and made 242 bugfixes. Each bugfix increase build numbering with +1.

At a fine-grained level, revision control is often used for keeping track of incrementally different versions of electronic information, whether or not this information is computer software.

We can change numbers at any time. Now we have ver. `3.12.242`. Add one new feature, and this can be ver. `3.13.0` Apply some small modifications, and we have ver. `3.13.1` Add one new feature, and here is the ver. `3.14.0`

88) White Box / Black Box Testing

It is completely not about colors (Black/White), but nobody cares (and clearly will not).

Please, do not share the stupid idea that *'White Box can be done when you look into the source code, and the Black Box can be done when you don't have an acces to source code'*. There are thousand of situations, when you'll look directly into the source code, and you will perform classic Black Box testing.

> Later you will realize, that developers understand this 'boxes' better than testers...

'White & Black' boxes are just a very convenient *metaphors* (see p.47) in testing terminology. They perfectly explain the source where tester has searched for ideas to create test cases[12].

Keep it simple. As a tester you have to

- realize what kind of situation may happen when user will start to interact with the application (and which situations can happen, but should not),
- create these situations one by one and see what's happen.

This is the whole *Testing*.

You can have all those situations listed in Requirements (p.57). If it so, then you are a lucky bastard (or you are in army). But for a mass-market application (any modern

[12] Situations to be tested

website) you will be out of this luxury. So, you have to invent or discover them by yourself. How to do it?

Sometimes you can know exactly how your app should working. You will imagine situations (so test cases will appear), and you will swtich to the illuminated zone — the 'White box' metaphor.

> Doesn't matter, where you will bring this information. You can read requirements? Now you know. You can read source code? Now you know. You can ask someone who knows about it? Now you know too.

Sometimes you know nothing, and you slide to a darker zone (the 'Black box' metaphor), where you have to often suppose what should happen than knowing exactly. You should explore, like 'Let's provide to this input field several characters and see how the software will react to it. . . '.

As you can see, the *researching* (or, sometimes, exploration) is not a testing. They are always close, but they are not the same.

> In testing you ALWAYS know the expected result. And you can compare it with actual result.
>
> In researching you may be blind about the expected result, but when you will know — you can test. That's why texting and researching are always close.

88.1) Why 'boxes'

The software, in the eyes of a tester, is like a box with some magic inside.

You don't know exactly how and why the coffe-machine works, but you can interact with it. The 'box' is 'black'.

You may not understand how and why an engine works (or how it looks like), but you can drive a car. The 'box' is 'black'.

You don't know what to do with your miserable life, but you still live it. The 'box' is 'black'.

And if you know How and Why it works — the box became 'transparent' for you.

That's why someone can say 'A glass box', or even 'The transparent box', opposite to 'The black box'.

88.2) What about strategies

Bearded old school testers (covered by bearded old school developers) claims that the 'White/Black Box Testing' is a *strategy*[13] (see p.63). And if this is a strategy, then we can combine the 'black' and 'white' to the 'gray box'.

Well, it really looks alike. But no.

And there are no 'gray' boxes in testing.

[13] «Art of Software Testing» by Glenford J. Myers, 1979, John Wiley & Sons, Inc.

THE END