The Little Black Book On Test Design

Rikard Edgren

Contents

Introduction

This little book describes ways to take advantage of the multiple information sources needed for ambitious system testing.

I started investigating it deeper after I for the Xth time felt that the existing test design techniques didn't capture the way I was working after 10 years with the same product suite. This resulted in an expanded generalization of ideas I primarily have learnt together with peers[1].

I will describe a holistic merge of test strategy, test analysis, test design and test execution; activities that can be split theoretically, but are intertwined in reality. It is not about details of individual test cases; test design can also be one-liners, charters, or a mental model in your head. It is about handling complexity, developing skill, learning as you go.

I want to emphasize the results of the testing; a broad sampling space, that enables serendipity[2] and aims for true saturation[3] for important areas. It is probably best suited for manual system testers that focus on any problems that are important, for people that want to find out more qualitative information than pass/fail on bare requirements.
It is about test design methods that I, and many others[4], have been using for a long time.

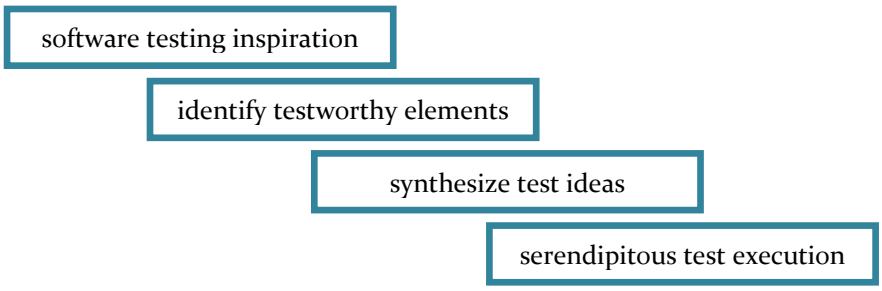
The Essence

Learn from multiple sources, generate relevant test elements,
synthesize testworthy test ideas, and execute with serendipity.

Software testers can follow the thoughts of social science Grounded Theory; we should use many sources of information regarding the subject (requirements, specifications, prototypes, code, bugs, error catalogs, support information, customer stories, user expectations, technology, tools, models, systems, quality objectives, quality attributes, testing techniques, test ideas, conversations with people, risks, possibilities); we should group things together, use our creativity, and create a theory consisting of a number of ideas capturing what seems important to test, together with the results.

I encourage testers to use a lot more than the requirements; to understand the need to combine things of different types, to look at the whole and the details at the same time. This piece of writing is for ambitious projects, probably spanning over several releases, and not worthwhile when resources are sparse, or when you are absolutely certain that the techniques you use are the best.

This might seem heavy, but people have phenomenal capacity.



[1] I have learnt from all colleagues, but especially Henrik Emilsson and Martin Jansson. Bibliography includes the major inspirations. Also thanks to reviewers who spent time and made valuable comments: Robert Bergqvist, Lisa Crispin and Matthew Heusser.

[2] Serendipity notion is gaining popularity in software testing. First occurrence I know of is Jonathan Bach, *Session-Based Test Management*, Software Testing and Quality Engineering magazine, 11/00, http://www.satisfice.com/articles/sbtm.pdf. Also see Rikard Edgren, *Testing is an Island, A Software Testing Dystopia*, EuroSTAR conference 2008 http://thetesteye.com/papers/redgren_testingisanisland.pdf

[3] More on saturation by Rikard Edgren, *Is your testing saturated?* - http://thetesteye.com/blog/2010/04/is-your-testing-saturated/

[4] There are not a lot of published material on testing efforts, but Fiona Charles has a grounded story in *Modeling Scenarios using Data*, http://www.quality-intelligence.net/articles/Modelling%20Scenarios%20Using%20Data_Paper_Fiona%20Charles_CAST%202009_Final.pdf

These activities depend on the skill of the tester, a skill that is developed over time, at best with a lot of knowledge of the product and project context, and speeded by generating and sharing your own rules of thumb, your specific test design heuristics.

Grounded Theory

I see a loose scientific, theoretical base in Grounded Theory[5] (Corbin/Strauss), used in social science as a qualitative, thorough analysis of a phenomenon.

The scientists examine the subject carefully, gather a lot of material, document codes and concepts in memos, combine them to categories, from which a theory is constructed.

There is no hypothesis to start with, as in the traditional natural science, and that's quite similar to my software testing; it would be dangerous if we thought we knew all important questions at once. They acknowledge that we are dealing with a sampling problem, and that serendipity is a factor to your advantage. Grounded theories are developed until no new, changing information is found; the theory is saturated.

I think this is a good match for software testing, especially since it emphasizes that the researcher must use her creativity.

When borrowing ideas from another area, you need to consider if there are special aspects about software testing that makes the ideas inappropriate. Grounded Theory is a research method, that tries to prove things, and others should be able to come to the same conclusions. It often uses people interviews in a greater sense than testing.

I have not tried to apply Grounded Theory to software testing; it is an inspiration for this description of my test design reality.

Test Design Theory

Unlike a traditional focus on test case design, this test design primarily encompasses test strategy, test analysis, test design, and test execution, but don't advocate a splitting of these activities. With a more holistic approach, you get interactions and interesting combinations by having multiple things on your mind at the same time; the activities inform each other.

It can fit with any test strategy, but the best match is to **not** have a defined strategy at start, and rather let the (multiple, diverse) strategies emerge together with test ideas and their result.

A matching high-level test strategy could be:

> *We are building off-the-shelf software that costs quite some money. Customers expect to be able to use the product with their data, in their environment and with their needs. We would like to avoid patches, and other problems that cause loss of image.*
> *We also want you to help developers go faster; create low-hanging fruit for them.*

The low-level test strategies (yes, they are many) will come naturally together with your learning, but you should probably check explicit requirements, and perform chartered or free-form exploratory testing.

Test analysis is a remarkably unelaborated area in testing literature, and my guess is that it is because traditional/ceremonial testing doesn't need this; what should be tested is already "completely" described in the requirements document. And also because *"Most of the published guidance on test techniques is based on unit/component test techniques"*[6]. This is unfortunate, since more important test decisions are made when you choose what test elements to consider, to which degree; when you try to find out good ways to investigate

[5] Juliet Corbin and Anselm Strauss, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory, Second Edition*, SAGE Publications, Inc., Thousand Oaks 1998

[6] Neil Thompson & Mike Smith, *The Keystone to Support a Generic Test Process: Separating the "What" from the "How"*, http://doi.ieeecomputersociety.org/10.1109/ICSTW.2008.46

areas, and design methods to find out important things about the software, from the users' perspective.
At the same time, the essence of Exploratory Testing[7] is a lot about this, and I am also more interested in testing than in checking[8].

Coming up with smart ways to test, based on multiple information sources, is the part of test design that fascinates me the most.
I am not interested in designing test cases, I think it is better to design test ideas, at least one level above test cases. The actual execution details are often trusted to the tester to decide upon. An expected result isn't necessary, often we can see what we find out, and report anything noteworthy.
Opposite to many existing test design techniques, my focus is on finding relevant sources, analyzing them, and choosing strategy/technique, rather than designing details of a test case.

What I'm writing is partly similar to *error-guessing* or *experienced-based*, but most to Exploratory Testing, which acknowledge the fact that the existing software, and results from tests, should be used to guide subsequent tests, if that helps. If you are seeking knowledge from many information sources, you focus on learning, which is a key part of exploratory testing.

Maybe this piece is just another take on a core area of exploratory testing, but at the same time, this test design can be used equally well for more scripted testing, that want to look further than to the requirement and specification documents.

What Is Important?

We know that complete testing isn't possible, since even if you execute all code and functions for all quality attributes, you won't get to all possible users' data, needs, environments and feelings. This means that we're dealing with a sampling problem, and that we want to sample mostly for important matters.

Test design should come up with test ideas that upon execution (with or without serendipity) will cover most of what is important (where requirements can be a good start.) This is more than difficult, because there are many things about software that are important, that's why "*Testing is an extremely creative and intellectually challenging task.*"[9]

I believe that in specific situations, either you know what is important, or you don't. So the basic strategy should be to learn a lot, because then you will **know** more often.

Your skill in understanding what is important will evolve over time, but is accelerated by curiosity and collaboration. If you work on a product for a long time, use this to your advantage by learning the different ways in which the product is useful; it's a long and fun learning journey. Domain knowledge helps, end user contacts give flesh to the concepts, imagination to conceive viable scenarios and new connections, critical thinking to spot holes, learn what can and should be done with the technology, get a tester's mindset[10].

We can look at the test space as an endless area, with a requirements box, and a software potato[11]:

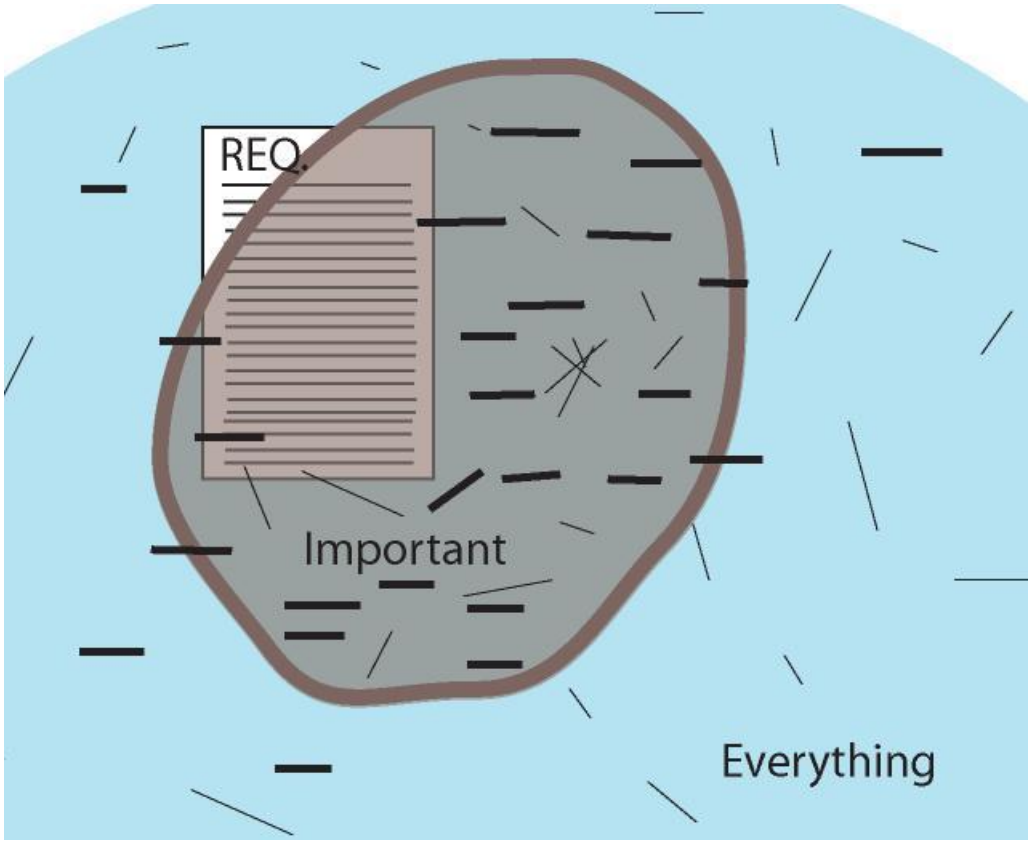
[7] The fundamentals of Exploratory Testing is described in Cem Kaner & James Bach, course *Black Box Software Testing, Exploratory Testing*, http://www.testingeducation.org/BBST/exploratory/BBSTExploring.pdf, Fall 2006.

[8] See Michael Bolton's blog post series starting with *Testing vs. Checking*, http://www.developsense.com/blog/2009/08/testing-vs-checking/

[9] Page 19 in Glenford Myers, *The Art of Software Testing*, Second Edition, John Wiley & Sons, Inc., Hoboken 2004 (originally published 1979)

[10] Good points about the needed mindset in Michael Bolton blog post, *Why Do Some Testers Find The Critical Problems?*, http://www.developsense.com/blog/2011/02/why-do-some-testers-find-the-critical-problems/

[11] See Rikard Edgren, blog post *In search of the potato…*, http://thetesteye.com/blog/2009/12/in-search-of-the-potato/



With a grounded knowledge about the software and its context, you can perform variations that can capture what is important.

Understanding importance requires value judgments, and traditionally too much of a quantitative natural science[12] approach has been put on testing. Computers are not good at deciding what is important, humans are.

[12] I prefer Cem Kaner's notion *Software Testing as a Social Science*, Toronto Association of Systems and Software Quality, October, 2006., http://www.kaner.com/pdfs/KanerSocialScienceTASSQ.pdf

Software Testing Inspiration

It is often said, with right, that you need to consider a lot more than the explicit requirements in order to be able to test a product well. [13]

The most important rationale for this is that in testing, we know more about the product than before implementation, and testers look at different detail levels. Other reasons are that requirements always are incomplete (they weren't written by an almighty; implicit requirements are almost endless), some things must be tried out before you understand it, and you don't want to spend too much resources on requirements writing.

We should also ask ourselves what problem testing is trying to solve. I think there are more situations like this:

> *We know that there will be problems with the product we are building. We need your help to make sure that the released product can be used by customers without failing, and satisfactorily help them with their tasks.*

than like this:

> *We must verify that the actual product meets the statements in the requirements document.*

The insight of multiple information sources is often accompanied by a few examples of "the other information", e.g. something about the customer needs, or the necessity of reading the code, or knowing the technology the software operates in, or an understanding of quality characteristics in the specific context.

On page 10-11, there is a thorough list[14], which can be seen as a "checklist when building your own checklist". Everything is not always relevant, but I recommend spending a minute on each category. Think about what is important, and decide for yourself which information sources to dig deeper into. The information you get can give you good test ideas at once, can point to what is important, and eventually guide your test design regarding details. The holistic merge make us understand what is important, in many situations.

By using multiple information sources, including the actual product and customer needs, your tests will get a better connection with reality, they will be grounded.

Using Many Models

The requirements are an important model of what the software should accomplish, but it is more complex than what ended up in the requirements document. You should challenge the requirements, and through other sources try to find the implicit ones[15] that matter.

If there exists specifications of any type (conceptual, technical, functional, design) they are valuable input. You can also look at specifications for other functionality, and in that case test specifications might be the most interesting ones.

The actual code is also a model, and you might benefit from reading it, or let someone who has read it tell you about it. Take special considerations to old, new, shaky, unread, reviewed code.

A help models the functionality, hopefully from a user's perspective.

---

[13] One of the best examples is "A tester who treats project documentation (explicit specifications of the product) as the *sole* source of requirements is crippling his test process.", Lessons 32 and 33 in Cem Kaner, James Bach, Bret Pettichord, *Lessons Learned in Software Testing*, John Wiley & Sons, Inc., New York 2002

Another is "Testers should base their tests on the information they can get — and they can get plenty of information from sources other than specifications." Cem Kaner, *The Ongoing Revolution in Software Testing*, Software Test & Performance conference, 2004, http://www.kaner.com/pdfs/TheOngoingRevolution.pdf

[14] Rikard Edgren, Martin Jansson and Henrik Emilsson, *37 Sources of Test Ideas*, http://thetesteye.com/blog/2012/02/announcing-37-sources-for-test-ideas/

[15] A good start of implicit specifications on slides 8-9 in Cem Kaner & James Bach, course *Black Box Software Testing, Specification-Based Testing* http://www.testingeducation.org/BBST/specbased/BBSTspecBased.pdf

The very best model is the actual software, when you can see in reality what it does, you will understand how different parts are interacting. When using software, you always make a mental model of it, even though you aren't aware of it.

You have implicit mental models that aren't necessarily possible to communicate. This is OK; ask your friends how they visualize a year, because some don't do this, but still have a perfect understanding of what it means. Your combined tests and test results model your test space, and as for all models, you should look both inside and outside.

Be open to other people's models, testers are not perfect, and make mistakes as everybody else.

The Product & the Project

There is always a history, and there's a great chance you are working on a product for many releases, so you have a lot of data from previous runs to use.

Have you built an error catalogue for the type of bugs that are common at your place?
Can you tag your bugs and support tickets so they can be re-used as inspiration?
How did you succeed in finding the important bugs so early, and how did the patched ones escape?
How do customers try to solve the problem without your software?
Are you using your marketing material as a guideline for most important areas?
Do you search the web for information about real usage and problems?

You have, and can get, a lot of information about the functionality, and the associated risks. You are a part of the project, a part of the process, and you know about all the different deliverables.

You have a test execution context that should guide your test design:

When can you start with what, and how much time do you have?
What's the background and specialties of the testers?
What will be tested by others?
What new types of testing should you try this time?
Are you ready for the unexpected things that always happen?
You probably want a loose, open-ended plan that is easy to change.

If you are lucky, you have, or can create, quality objectives that guide towards what's important.
There might be information objectives that steer your activities in a good way.

People & Skills

There is a lot of knowledge among your team and stakeholders; ask for details when you get the chance. Earn respect from the developers, and you will get informal insider tips in your search for vulnerabilities.
What do you know about the user's needs, knowledge, feelings, impairments?
What is valued by users, the team and other interested parties?

To find out what should be tested, you should use your own, and your colleagues' skills:

- Knowledge, experience, intuition and subjectivity
- Investigation: explore and learn
- Analytical thinking: model details and the big picture, follow paths
- Critical thinking: see problems, risks and possibilities
- Creativity[16]: broaden test spectra, generate new ideas, lateral thinking
- The Test Eye: wants to see errors, sees many types, looks at many places, looks often, focus on what's important, look with others' eyes[17]

[16] Rikard Edgren, *Where Testing Creativity Grows*, EuroSTAR conference 2007, http://thetesteye.com/papers/where_testing_creativity_grows.pdf

More Inspiration

It is beneficial to know a lot about the actual and potential usage of the product. Take any chances of visiting or talking to customers, find out experiences that are channeled to support, learn about the "true" requirements: what problems are the customers trying to solve?

Can you use the product internally "for real"; eat your own dog food; sip your own champagne? Using the actual implemented software to accomplish something valuable is my favorite testing inspiration.

You can look at competitors in the form of software products, in-house tools, but also analog systems; how was similar functions performed before they were computerized?

In order to understand business needs, logic, information, knowledge, standards, laws; you could try pair testing with a business expert.

Knowing the technology means you know how the software can be operated, and what is most important. It is also vital to understand how to execute tests, and do it efficient. Using your primary platform for private work is a shortcut, knowing the relevant tools is necessary.

If you know a lot about existing test theory, you can use many different techniques/approaches and have a higher chance of finding important information. The classic test design techniques might not be suitable very often (except for the implicit equivalence partitioning that is used all the time), but in the right situation they can be very powerful.

Get inspired by generic test ideas like quicktests, tours, mnemonics, heuristics, quality characteristics; or tricks[18], attacks[19], checklists, and other knowledge from books, courses[20], blogs[21], forums, web sites, articles[22], conferences, conversations.

Don't use any idea without considering if it is important, or even relevant, in your product context. Over time you can create your own list of test idea triggers that are easy to transform to fruitful tests in your actual projects.

Double work?

It could be argued that using an extensive list of testing inspiration is doubling the work done by requirement analysts. This is partly true, and could be less costly if there were appendixes to the requirements document, that includes more about attributes and preferences, and some of the motivation information they collected.[23]

At the same time, testing goes into details in way requirements never do. The requirements will not state the response time for each and every action, in all circumstances, but manual testers will implicitly evaluate specific performances all the time, if they are allowed, or told to.

Testers (and developers) need to understand the true requirements (not the document) in order to do a great job. This will require some double work, but will also generate the richer, collective understanding that diversity gives.

Also, when doing this exercise, testers will get a whole bunch of sources to use as oracles[24] when evaluating interesting behavior on test execution.

---

[17] Rikard Edgren article *The Eye of a Skilled Software Tester*, The Testing Planet, March 2011 – Issue 4, http://wiki.softwaretestingclub.com/w/file/fetch/39449474/TheTestingPlanet-Issue4-March2011.pdf

[18] Many tips are available at http://www.quicktestingtips.com

[19] James Whittaker, *How to break Software: A Practical Guide to Testing*, Addison-Wesley, Boston 2002

[20] Free, great testing course material is available at http://www.testingeducation.org (mostly by Cem Kaner)

[21] A blog portal to start with is http://www.testingreferences.com

[22] Michael Bolton's Better Software columns are thoughtful and inspiring, http://www.developsense.com/publications.html

[23] A great book on requirements is Donald C. Gause/Gerald M. Weinberg, *Exploring Requirements: Quality Before Design*, Dorset House Publishing Co. 1989

[24] For a more manageable list of oracle heuristics, see Bach/Bolton's HICCUPPS (F) - http://www.satisfice.com/rst.pdf

37 Sources for Test Ideas

We recommend collecting test ideas continuously from a variety of information sources.
Consider the following, and think about values, risks, opportunities; find shortcuts to cover what is important.

Product	**1. Capabilities.** The first and obvious test ideas deal with what the product is supposed to do. A good start is requirements, examples and other specifications, or a function list created from actual software. But also try to identify implicit requirements, things users expect that haven't been documented. Be alert to unwanted capabilities. **2. Failure Modes.** Software can fail in many ways, so ask "what-if" questions to generate test ideas that expose the handling of internal/external, anticipated/unexpected, (un)intentional, realistic/provoked failures. Challenge the fault tolerance of the system; any object or component can break. **3. Models.** A state model helps identify test ideas around states, transitions and paths. A system anatomy map shows what can be tested, and can highlight interactions. Create a custom model using structures like SFDPOT from Heuristic Test Strategy Model. A visual model is easier to communicate, and the modeling activity usually brings understanding and new ideas. Many and rich models give better test ideas. **4. Data.** By identifying intentional and unintentional data (there is always noise), you have a good start for a bunch of test ideas. Follow easy and tricky data through the application, be inside and outside boundaries, challenge data types and formats, use CRUD (Create, Read, Update, Delete), exploit dependencies, and look at the data in many places. **5. Surroundings.** No product is an island, so environment compatibility (hardware, OS, applications, configurations, languages) is one of many important testing problems, but also investigate nearby activities to your product. By understanding the big system you can get credible test ideas that are far-fetched when looking at functionality in isolation. **6. White Box.** By putting a tester's destructive mindset on developers' perspective of architecture, design and code, you can challenge assumptions, and also find mistakes that are cheap to fix. Pay special attention to decisions and paths you might not understand from a black-box perspective. Code coverage is not worthless, it can be used to find things not yet tested. **7. Product History.** Old problems are likely to appear in new shapes. Search your bug/support system or create an error catalogue, remember critical failures and root cause analyses. Use old versions of your software as inspiration and oracle. **8. Rumors.** There are usually lots of talk about quality and problems. Some hurt the product and the organization. Use the rumors themselves as test ideas. It is your mission to kill them or prove them right. **9. Actual Software.** By interacting with the software, you will get a lot of ideas about what is error-prone, connected, interesting. If you can eat your own dog food (euphemism: sip your own champagne), you are in much better position to understand what is important about the software. If "Quality is value to some person", it is pretty good if that person is "me". **10. Technologies.** By knowing the inner workings of the technology your software operates in, you can see problematic areas and things that tend to go wrong; understand possibilities and security aspects; which parameters to change, and when. You can do the right variations, and have technical discussions with developers. **11. Competitors.** By looking at different solutions to similar problems you can get straightforward test ideas, but also a feeling of which characteristics end users are interested in. There might be in-house solutions (e.g. Excel sheets) to be inspired by, and often there exists analogue solutions for similar purposes. Can you gain any insightful test ideas from your competitors support, FAQ or other material?
Business	**12. Purpose.** The overall purposes of the product will give you goals for your test ideas. Ask a couple of extra why? to find out the real purposes. These can give you the broadest benevolent start that can find very important problems, fast. **13. Business Objectives.** What are the top objectives for the company (and department break-downs) ? Are there any requirements that contradict those objectives? Do you know the big picture, the product vision and value drivers? **14. Product Image.** The intended behavior and characteristics of the product might be explicit or implicit, inside the walls and minds of the people producing or consuming the software. You will be able to write compelling problem reports if you know and can show threats to the product's image, e.g. by pointing to a violation of marketing material. **15. Business Knowledge.** If you know the purpose of the software, and the context it operates in, you can understand if it will provide vale to customers. If you can't acquire this knowledge, co-operate with someone who knows the needs, logic and environment. **16. Legal aspects.** Do you need to consider contracts, penalties or other legal obligations? What would cost the company the most in a legal issue? Do you have a lawyer that can give you hints on what must be avoided?
Team	**17. Creative Ideas.** All products are unique, and require some test ideas not seen before. Try lateral thinking techniques (e.g. Edward De Bono's Six Thinking Hats, provocative operators, the opposite, random stimulation, Google Goggles) to come up with creative test ideas. Metaphors and analogies is a good way to get you started in new directions. **18. Internal Collections.** Use or create lists of things that often are important in your context, some call these quality/test patterns, others have product-specific quicktests. **19. You.** You are a user. You might be a stakeholder. You matter. Take advantage of your strengths from experiences, skills, knowledge and familiarity with problems. Use your subjectivity and feelings to understand what's important. And don't forget to acknowledge your weaknesses and blind spots.

Project	**20. Project Background.** The reasons for the project are driving many decisions, and history from previous (similar) projects are well worth knowing about in order to make effective testing. **21. Information Objectives.** It is vital to understand the explicit and implicit purposes of the testing effort. If you can't get them, create your own quality objectives that steer test ideas for any feature. **22. Project Risks.** Some of the difficult things in a project can be addressed by testing. You want to know about which functionality developers are having trouble with, and you will adjust your schedule depending on risks that need mitigation first. **23. Test Artifacts.** Not only your own test ideas, logs and results can be used for sub-sequent tests, also try out test results from other projects, Beta testing reports, usability evaluations, 3rd party test results etc. What questions do you want to be able to answer in status reports? **24. Debt.** The shortcuts we take often give a constantly growing debt. This could be project debt, managerial debt, technical debt, software debt, testing debt or whatever you wish to call it. If the team keep track on what is on the debt list, you can map a set of test ideas against those items. **25. Conversations.** The informal information you get from people may contain things that are more important than what's included in specifications. Many people can help you with your test design, some are better judges of importance, what can you gain from MIP:s (Mention In Passing)? If developers know you can find interesting stuff, they will give you insider information about dubious parts of the software. A set of questions to a developer might be an innocent *"what do you think we should test?*" or "*what part of your code would you have liked to do better?*" **26. Context Analysis.** What else in the current situation should affect the things you test, and how? Do you know about the market forces and project drivers? Is there anything that has changed that should lead to new ways of testing? What is tested by others? Which strengths and weaknesses does the project and its people have? **27. Many Deliverables.** There are many things to test: the executable, the installation kit, programming interfaces, extensions, code & comments, file properties, Help, other documentation, Release Notes, readme:s, marketing, training material, demos etc. All of these also contain information you can use as inspiration. **28. Tools.** If something can be done very fast, it is a good idea to try it. Tools are not only the means to an end, they can also be used as the starting point for exploration.
Stakeholders	**29. Quality Characteristics.** Quality characteristics are always important for the project to be successful, although the OK zone can be easy to reach, or difficult and critical. Our definition includes capability, reliability, usability, charisma, security, performance, IT-bility, compatibility, supportability, testability, maintainability, portability, and a plethora of sub-categories. Many of these can be used as ongoing test ideas in the back of your head, executed for free, but ready to identify violations. **30. Product Fears.** Things that stakeholders are really worried about are much stronger than risks, they don't need prioritization, they need testing. Typical hard-to-verify, but useful-for-testing fears are: loss of image, wrong decisions, damage, people won't like the software. Different people have different fears; find out which matters most. **31. Usage Scenarios.** Users want to accomplish or experience something with software, so create tests that in a variety of ways simulate sequences of product behavior, rather than features in isolation. The more credible usage patterns you know of, the more realistic tests you can perform. Also try eccentric soap opera tests to broaden test coverage. **32. Field Information.** Besides knowledge about customer failures, their environments, needs and feelings, you can take the time to understand your customers both in error and success mode. Interview end users, pre-sales, sales, marketing, consultants, support people, or even better: work there for a while. **33. Users.** Think about different types of users (people you know, personas), different needs, different feelings, and different situations. Find out what they like and dislike, what they do next to your software. Setup a scene in the test lab where you assign the testers to role play different users, what test ideas are triggered from that? Best is of course unfiltered information directly from users, in their context. Remember that two similar users might think very differently about the same area.
External	**34. Public collections.** Take advantage of generic or specific lists of bugs, coding errors, or testing ideas. As you are building your own checklist suitable for your situation, try these: • Appendix A of Testing Computer Software (Kaner, Falk, and Nguyen) • Boris Beizer Taxonomy (Otto Vinter) • Shopping Cart Taxonomy (Giri Vijayaraghavan) • Testing Heuristics Cheat Sheet (Elisabeth Hendrickson) • You Are Not Done Yet (Michael Hunter) Learn some testing tricks or techniques from books, blogs, conferences; search for test design heuristics, or invent the best ones for you. **35. Standards.** Dig up relevant business standards, laws and regulations. Read and understand user interface standards, security compliance, policies. There are articles out there that describe how you can break something even if it adheres to the standards, can you include their test ideas? **36. References.** Reference material of various kinds is a good source for oracles and testing inspiration, e.g. an atlas for a geographic product. General knowledge of all types can be handy, and Wikipedia can be enough to get a quick understanding of a statistical method. **37. Searching.** Using Google and other means is a good way to find things you were looking for, and things you didn't know you needed (serendipity).

Analysis

The analysis[25] part of test design deals with the problem: *What should we test?* This involves the identification and elaboration of diverse information sources, but also the art of figuring out what is, or might be, important.

I want to elaborate a tester's natural instinct to break down product information to elements that can be used in testing. It can be details for a requirement, or insights from talking to a customer, or feature slogans from a web site, the list is long as seen in previous chapter.

Kaner calls this **learning**, Michael Bolton (and James Bach) in addition use **factoring** – "Factoring is the process of analyzing an object, event, or model to determine the elements that comprise it."[26]

Edward deBono has a general lateral thinking technique **fractionation**, which explicitly includes a creative aspect:

> *"One is not trying to find the true component parts of a situation, one is trying to create parts."*[27]

In software testing, this concept contains more than the mathematical factorization[28], which analyzes and finds the exact components that make up the whole.

In testing we are rather looking for elements that are useful, that contains something we want to test in order to find important information about the product. We might create unjustified artificial divisions, because they help us. If a use case includes Firefox 3.6, we can add all other browsers and platforms we think are relevant and worth to test; we will automatically think about settings in browsers.

We will add things we know or think we know; leveraging our understanding about what is important, to get elements to synthesize to useful test ideas. We will also look for many solutions, and the important thing is not that the factors make up the original information source, rather you want test elements that are useful, and it is perfectly fine if they contradict each other.

Focus not only on what is important, but also on what might be important, or might generate other information that is important. You want to create tests that are **testworthy**, and typical examples can be those tests where you don't know if it is important or not, but you want to make sure.

Analysis Heuristics

Analysis towards a broad and relevant test space is a skill that is difficult to learn and master; it is about understanding the whole, the details, and what is important. However, there are many heuristics[29], rules of thumb, you can use to make the intertwined process of analysis/design more powerful. Here are some examples in your search of your best ones:

1. ASAP – one small knowledge about the system can have big effect on the testing effort (and feedback can have big effect on the development effort.)
2. Look For More – Requirements, or Risks are good, but they are just the beginning.
3. Huh? Really? So? – you may not understand, what you understand may not be true, the truth may not matter, or may matter much more than you think.[30]
4. Mary Had a Little Lamb Heuristic - Take each word, elaborate and challenge it, use synonyms, antonyms, and combinations.[31]

---

[25] Analysis could also be called factoring++, because analysis in software testing = Factoring + Lateral thinking + Understand what's important + Many solutions.

[26] Michael Bolton, presentation *Confirmation Bias*, http://www.developsense.com/presentations/2010-04-ConfirmationBias.pdf

[27] Page 135 in Edward deBono, *Lateral Thinking - Creativity Step by Step*, Harper & Row, New York 1990 Perennial edition

[28] Wikipedia: Factorization, http://en.wikipedia.org/wiki/Factorization

[29] A broad definition of heuristics includes anything that can help you solve a problem, e.g. each item in the included posters is a heuristic. Here I focus on ways of thinking about the analysis part of your test design.

[30] Bach/Bolton heuristic for critical thinking in *Rapid Software Testing* course, http://www.satisfice.com/rst.pdf

5. Question everything, especially this statement – for important matters, challenge assumptions.
6. Ask people for details, clarifications and expectations.[32]
7. Ambiguity analysis[33] – look for contradictions, holes and things that are easy to misunderstand.
8. 'The "Unless…" Heuristic – Take whatever statement you care to make about your product, your process, or your model, and append "unless…" to it. Then see where the rest of the sentence takes you.' [34]
9. Apply Business Knowledge – Learn the business, and if you can't, co-operate with someone who has that knowledge.
10. Relevance Judgment – Do you think it is important? Does someone else think it is important? Might it become important? Will a user[35] be upset/annoyed about a problem?
11. What If? – brainstorming about what could happen, or be experienced.
12. Lateral thinking – artificial fractionation, analogies, generate alternatives, the opposite, random stimulation – anything that is useful will do.[36]
13. Test Idea Triggers – expose yourself to information/situations you think can generate good ideas.
14. Zoom in/out – go up or down in detail level, and examine the content.
15. New Connections – combine diverse knowledge in important ways, the details and the big picture; what is worth elaborations?
16. Instantiation of generalizations – contextualize ideas from generic tests, taxonomies, checklists (Hendrickson[37], Hunter[38], Kaner[39], Sabourin[40], Software Quality Characteristics.)
17. Make a picture, diagram or table – easier to communicate, and can trigger new ideas.
18. Mysterious silence – "When something interesting or important is not described or documented, it may have not been thought through, or the designer may be hiding its problems."[41]
19. Diversity – Think in different ways.[42]
20. Long Leash Heuristic – "Let yourself be distracted…'cause you'll never know what you'll find…but periodically take stock of your status against mission."[43]
21. Plunge In and Quit – start with the most difficult part, see what happens, quit when you want to.[44]

---

[31] Chapter 10 – Mind your meaning in Donald C. Gause/Gerald M. Weinberg, *Are Your Lights On? How to Figure Out What the Problem REALLY Is*, Dorset House Publishing Co. 1990 (originally published 1982)

[32] "Context-free questions" are listed in *Exploring Requirements* (Gause/ Weinberg) and Michael Bolton has *Context-Free Questions for Testing* at http://www.developsense.com/blog/2010/11/context-free-questions-for-testing/

[33] See slides 73-78 in Cem Kaner presentation *Developing Skills as an Exploratory Tester*, Quality Assurance Institute Worldwide Annual Software Testing Conference, Orlando, FL, November 2006, http://www.kaner.com/pdfs/ExploratorySkillsQAI2007.pdf

[34] The "Unless" Heuristic is described by Michael Bolton, http://www.developsense.com/blog/2007/03/white-glove-heuristic-and-unless/

[35] Use a broad definition of user if you want to catch issues like supportability, testability, maintainability.

[36] Edward deBono, *Lateral Thinking - Creativity Step by Step*, 1990 Perennial edition

[37] Elisabeth Hendrickson, *Test Heuristics Cheat Sheet*, http://testobsessed.com/wp-content/uploads/2011/04/testheuristicscheatsheetv1.pdf

[38] Michael Hunter, *You Are Not Done Yet*, http://www.thebraidytester.com/downloads/YouAreNotDoneYet.pdf

[39] Appendix A, Common Types of Errors Cem Kaner, Jack Falk, and Hung Q. Nguyen, *Testing Computer Software, Second Edition*, John Wiley & Sons, Inc., New York 1999. Available at http://www.testingeducation.org/BBST/testdesign/Kaner_Common_Software_Errors.pdf

[40] Robert Sabourin, *10 Sources of Testing Ideas*, http://www.amibugshare.com/articles/Article_10_Sources_of_Testing_Ideas.pdf

[41] Cem Kaner & James Bach workshop *Risk-Based Testing: Some Basic Concepts*, QAI Managers Workshop, QUEST Conference, Chicago 2008, http://www.kaner.com/pdfs/QAIRiskBasics.pdf

[42] Many good examples of this in James Bach, Jon Bach, Michael Bolton, *Exploratory Testing Dynamics* v2.2, http://www.satisfice.com/blog/wp-content/uploads/2009/10/et-dynamics22.pdf

[43] Bach/Bolton heuristic for critical thinking in *Rapid Software Testing* course, http://www.satisfice.com/rst.pdf

22. Rumsfeld's Heuristic[45] – investigate the known unknowns, consider unknown unknowns.
23. Don't Stop – when you find what you were looking for, think some more to see if there are better things waiting around the corner.
24. Always Pause – Analysis (and design) is a continuous activity, pause all the time and never.
25. Done by Others – Find out which types of testing that are performed by others; you can skip or cover them lightly.[46]
26. Context Analysis – Find out which factors in the current situation that can guide your testing effort.[47]
27. Test Framing[48] – Link your testing activity to your testing mission, and learn more.
28. What Assumptions I Took – explain your testing activities, and challenge them.[49]

These activities are surrounded by **understanding what's important**. This is essential in order to get good test elements, and by performing this activity, you will generate the same understanding. It's a positive spiral, and well spent time.

Invisible Analysis

Most of the time, the analysis will take place rapidly in your mind; you will notice something important, and react with a test idea (written or executed) immediately. This is perfectly fine, documented analysis is only needed if the process should be reviewed.

If you train your skill in this, you can do it faster, and share your thinking in a better way.

Creativity is important, the more you learn and think about the product, the more elements and ideas will appear. Put them on the list, also the crazy ones; you can always dismiss them later. As Robert Sabourin puts it: *Collect all testing ideas you can find!, My experience is that it is better to omit a test on purpose than to skip it because you ran out of time or forgot about it!*[50]

Memos

For your own sake, but primarily for others, you can write short memos as you discover important things from your variety of information sources. This will make the process reviewable, possibly re-creatable, and powerful as knowledge transfer.

---

[44] Page 100-103 in James Bach, *Secrets of a Buccaneer-Scholar*, Scribner, New York 2009

[45] Original Rumsfeld quote is "There are known knowns. There are things we know we know. We also know there are known unknowns. That is to say, we know there are some things we do not know. But there are also unknown unknowns, the ones we don't know we don't know."

[46] Good example: "the specific technique - checking single variables and combinations at their edge values - is often handled well in unit and low-level integration tests. These are much more efficient than system tests. If the programmers are actually testing this way, then system testers should focus on other risks and other techniques.", Cem Kaner, blog post *Updating some core concepts in software testing*, http://www.satisfice.com/kaner/?p=11

[47] A good guide is Project Environment section in James Bach, *Heuristic Test Strategy Model*, http://www.satisfice.com/tools/satisfice-tsm-4p.pdf

[48] Michael Bolton, *Test Framing*, http://www.developsense.com/resources/TestFraming.pdf

[49] , Alan Richardson, blog post *Challenge your assumptions and presuppositions to identify useful variation*, http://www.eviltester.com/index.php/2008/04/18/challenge-your-assumptions-and-presuppositions-to-identify-useful-variation/

[50] Robert Sabourin, *Just-in-time testing*, EuroSTAR 2010 tutorial, http://www.amibugshare.com/courses/Course_Just_In_Time_Testing.zip (typo *expecrience* corrected in quote)

You can use coding, e.g. describing tags, on any of your relevant documents, to be able to return and re-arrange the information.[51]

Typical areas where a periodically updated memo can be suitable (and useful to many) are:

- Our Sources for Test Ideas
- Customer Configurations
- (User) Work-arounds
- Root causes for support incidents
- Finding Bugs at XX Software
- Our Most Important Quality Characteristics
- Tests we should run first
- Typical bugs to catch in unit tests

Maybe you can persuade requirement analysts to share memos, to give some details, some desires not expressible in requirement format. Some flesh to the users; when requirements are formulated in words, so much information is lost…

Quality Characteristics

Quality characteristics[52] describe attributes that most software benefit from[53]. They can be used on the whole product or for details. *The whole consists of the details. The quality of a detail is defined by the whole.*

These characteristics are generic, and fruitful as test idea triggers for any software. Some of them are irrelevant for you, some are easy to satisfy, and some are very important and difficult. See poster on next pages for a thorough list of inspiring concepts in the areas of Capability, Reliability, Usability, Charisma, Security, Performance, IT-bility, Compatibility, Supportability, Testability, Maintainability, Portability. There are no numbers attached to these descriptions; metrics are dangerous[54], because they hide what's really important.

Go through the list with your peers/stakeholders, and they can help you focus, and at the same time get an understanding of the complexity of software (testing). Discuss with the project for each case when characteristics are conflicting.

This categorization can also be a starting point for a more deliberate non-functional[55] testing, with a fruitful mix of specialized testing and lightweight[56] methods.

[51] If you want to go heavy into this, try free trial of qualitative analysis tool Atlas.ti. You can also use any text tool with search abilities, a wiki, or a blog.

[52] An updated version of this poster is included in this paper (1.1 release is upcoming), original announcement at http://thetesteye.com/posters/TheTestEye_Software-quality-characteristics-1-0/ (written by Rikard Edgren, Henrik Emilsson, Martin Jansson)

[53] A too long definition of software quality could be phrased:
Software quality is **more** than the **perceived** sum of **relevant** quality characteristics like capability, reliability, usability, charisma, security, performance, IT-bility, compatibility, supportability, testability, maintainability, portability for many **different** persons

[54] A more forgiving view is "Metrics that are not valid are dangerous.", http://www.context-driven-testing.com/

[55] "Non-functional" is a strange word, but Kaner's "parafunctional" is not wide-spread, and "testing by characteristics" won't be either.

[56] Check the tag "lightweight" on http://thetesteye.com/blog for simple non-functional test ideas for everyone.

Software Quality Characteristics

Go through the list and think about your product/features. Add specifics for your context, and transform the list to your own.

Capability. *Can the product perform valuable functions?*

- *Completeness:* all important functions wanted by end users are available.
- *Accuracy:* any output or calculation in the product is correct and presented with significant digits.
- *Efficiency:* performs its actions in an efficient manner (without doing what it's not supposed to do.)
- *Interoperability:* different features interact with each other in the best way.
- *Concurrency:* ability to perform multiple parallel tasks, and run at the same time as other processes.
- *Data agnosticism:* supports all possible data formats, and handles noise.
- *Extensibility:* ability for customers or 3rd parties to add features or change behavior.

Reliability. *Can you trust the product in many and difficult situations?*

- *Stability:* the product shouldn't cause crashes, unhandled exceptions or script errors.
- *Robustness:* the product handles foreseen and unforeseen errors gracefully.
- *Stress handling:* how does the system cope when exceeding various limits?
- *Recoverability:* it is possible to recover and continue using the product after a fatal error.
- *Data Integrity:* all types of data remain intact throughout the product.
- *Safety:* the product will not be part of damaging people or possessions.
- *Disaster Recovery:* what if something really, really bad happens?
- *Trustworthiness:* is the product's behavior consistent, predictable, and trustworthy?

Usability. *Is the product easy to use?*

- *Affordance:* product invites to discover possibilities of the product.
- *Intuitiveness:* it is easy to understand and explain what the product can do.
- *Minimalism:* there is nothing redundant about the product's content or appearance.
- *Learnability:* it is fast and easy to learn how to use the product.
- *Memorability:* once you have learnt how to do something you don't forget it.
- *Discoverability:* the product's information and capabilities can be discovered by exploration of the user interface.
- *Operability:* an experienced user can perform common actions very fast.
- *Interactivity:* the product has easy-to-understand states and possibilities of interacting with the application (via GUI or API).
- *Control:* the user should feel in control over the proceedings of the software.
- *Clarity:* is everything stated explicitly and in detail, with a language that can be understood, leaving no room for doubt?
- *Errors:* there are informative error messages, difficult to make mistakes and easy to repair after making them.
- *Consistency:* behavior is the same throughout the product, and there is one look & feel.
- *Tailorability:* default settings and behavior can be specified for flexibility.
- *Accessibility:* the product is possible to use for as many people as possible, and meets applicable accessibility standards.
- *Documentation:* there is a Help that helps, and matches the functionality.

Charisma. *Does the product have "it"?*

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

Security. *Does the product protect against unwanted usage?*

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

Performance. *Is the product fast enough?*

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

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

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

Compatibility. *How well does the product interact with software and environments?*

- *Hardware Compatibility*: the product can be used with applicable configurations of hardware components.
- *Operating System Compatibility*: the product can run on intended operating system versions, and follows typical behavior.
- *Application Compatibility:* the product, and its data, works with other applications customers are likely to use.
- *Configuration Compatibility*: product's ability to blend in with configurations of the environment.
- *Backward Compatibility:* can the product do everything the last version could?
- *Forward Compatibility:* will the product be able to use artifacts or interfaces of future versions?
- *Sustainability:* effects on the environment, e.g. energy efficiency, switch-offs, power-saving modes, telecommuting.
- *Standards Conformance:* the product conforms to applicable standards, regulations, laws or ethics.

Internal Software Quality Characteristics

These characteristics are not directly experienced by end users, but can be equally important for successful products.

Supportability. *Can customers' usage and problems be supported?*

- *Identifiers:* is it easy to identify parts of the product and their versions, or specific errors?
- *Diagnostics:* is it possible to find out details regarding customer situations?
- *Troubleshootable:* is it easy to pinpoint errors (e.g. log files) and get help?
- *Debugging:* can you observe the internal states of the software when needed?
- *Versatility:* ability to use the product in more ways than it was originally designed for.

Testability. *Is it easy to check and test the product?*

- *Traceability:* the product logs actions at appropriate levels and in usable format.
- *Controllability:* ability to independently set states, objects or variables.
- *Observability:* ability to observe things that should be tested.
- *Monitorability:* can the product give hints on what/how it is doing?
- *Isolateability:* ability to test a part by itself.
- *Stability:* changes to the software are controlled, and not too frequent.
- *Automation:* are there public or hidden programmatic interface that can be used?
- *Information:* ability for testers to learn what needs to be learned...
- *Auditability:* can the product and its creation be validated?

Maintainability. *Can the product be maintained and extended at low cost?*

- *Flexibility:* the ability to change the product as required by customers.
- *Extensibility:* will it be easy to add features in the future?
- *Simplicity:* the code is not more complex than needed, and does not obscure test design, execution and evaluation.
- *Readability:* the code is adequately documented and easy to read and understand.
- *Transparency:* Is it easy to understand the underlying structures?
- *Modularity:* the code is split into manageable pieces.
- *Refactorability:* are you satisfied with the unit tests?
- *Analyzability:* ability to find causes for defects or other code of interest.

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

- *Reusability:* can parts of the product be re-used elsewhere?
- *Adaptability:* is it easy to change the product to support a different environment?
- *Compatibility:* does the product comply with common interfaces or official standards?
- *Internationalization*: it is easy to translate the product.
- *Localization:* are all parts of the product adjusted to meet the needs of the targeted culture/country?
- *User Interface-robustness*: will the product look equally good when translated?

Rikard Edgren, Henrik Emilsson and Martin Jansson - thetesteye.com v1.1

Synthesizing Test Ideas

It is difficult to describe the process of synthesizing test ideas[57]. It involves a multitude of information sources, a sense of what's important, and a dose of creativity to come up with ingenious test ideas, and effective ways to execute them.

The easiest way is to take the requirements document, re-phrase each item, and optionally add some details using equivalence partitioning. The best way is to use a variety of information sources and generate testworthy test ideas that have a good chance of being effective. An impossible way is to combine all important information in all possible ways.

Rather you should use each element that is important, and each combination you think is important. Reviews are helpful, the actual product invites to some tests, and also the speed of the tests will guide you to the best ideas in your situation. A team collaboration on a whiteboard mind map doesn't have to take too much time.

I recommend writing down the test ideas, at least with a high-level granularity, so they are easy to use for discussion and planning. If reviewing isn't done, or the software already is available, it is faster to write the noteworthy ones after execution, together with the result.

Don't try to cover everything, because you can't. Rather make sure there is a breadth, and count on serendipity to find the information needed for a successful product. Some of the test ideas will be generated on-the-fly, during your test execution when you see what you actually can do with the software.

And don't stop when you reach an adequate test idea. Think some more, and you might find better ways, or new solutions to old problems.

The techniques to use are the classics: equivalence partitioning, boundary value analysis, state models, classification trees, data flow[58], but also a straightforward statement of what you want to test[59] or anything else you find useful.

For training purposes it might be good with categorizations like risk-based, specification-based, domain testing et.al, but in my reality it is more powerful to switch between these rapidly, and make up your own methods for your diverse material.

Nonetheless, here is an alternative classification of test ideas:

Ongoing Test Ideas

Ongoing test ideas can't be completed at once; they keep going as long as more relevant information is revealed. A good example[60] is quality characteristics like stability; the more you test, the more you know about the stability of the product. You probably don't do only ongoing tests (in the background) for important characteristics, but as a complement, it is very powerful, and very resource efficient.

Another example is ongoing usability testing for free, where the tester keeps the relevant attributes in the back of their head during any testing, and whenever a violation occurs; it is noticed, and communicated.

---

[57] Terminology "test idea" originates from Brian Marick. Martin Jansson's (http://thetesteye.com/blog/2010/10/discussion-around-the-content-of-a-test-proposal/) definition is almost perfect: "the essence of a test in the minimal amount of words bringing out the most important aspects of it" (it excludes visual test ideas though.)
For benefits of one-liner test ideas, see Rikard Edgren, More and Better Test Ideas, EuroSTAR 2009 conference, http://www.thetesteye.com/papers/redgren_moreandbettertestideas.pdf

[58] Two good books on this are Lee Copeland, *A Practitioner's Guide to Software Test Design*, Artech House Publishers, Boston, 2003 and Torbjörn Ryber, *Essential Software Test Design*, Fearless Consulting Kb, 2007

[59] There is no established name for the test design technique that doesn't use any elaborating technique, see discussion at http://thetesteye.com/blog/2010/12/test-design-technique-name-competition/ although I believe Human Test Design Technique might be the right name.

[60] Not sure about this, but ongoing activities might **only** be for quality characteristics.

Classic Test Ideas

Classic test ideas deal with something specific, and can be written in "verify that…" format. They can be a mirror of requirements, or other things the testers know about.

Don't get me wrong, these are important. For central parts of the functionality, the classic test design techniques, with clear expected results, might be necessary to cover everything needed, so you should know how to use them. For functional testing, you can look at isolated capabilities, or use a systems perspective[61].

For review and reporting's sake, beware of granularity; rather use one test "verify appropriate handling of invalid input (empty, space, string, accented, Unicode, special ASCII, long, wrong delimiters)" than a dozen of tests. In some situations you can even use "Verify explicit requirements are met", and put focus on the more interesting test ideas.

Also take the opportunity to recognize which tests are better suited as automated, and which are safest to test automatically, and tool-aided and manually.

Combinatorial Test Ideas

Many problems don't appear in isolation (that's why unit tests are very good, but far from a complete solution), therefore testing wants to use features, settings and data together, in different fashions and sequence.

Scenario testing[62] is one way to go, and pairwise might suffice[63] if it's not worth finding out what is most common, error-prone, important, or best representatives (but a tester with product knowledge can tell you at once which interoperability can't be neglected.)

Combinatorial test ideas use many test elements together, because we think they can have effect on each other, or we don't know.

Unorthodox Test Ideas

Requirements are often written so they can be "tested" (but what usually is meant is "verified".) This easily results in requirements being quantified and often in a contractual style, instead of describing the real needs and desires.

The risk of missing what is really important can be mitigated with testing that isn't necessarily aimed towards falsifying hypothesis. "*You are one of few who will examine the full product in detail before it is shipped*"[64]

Unorthodox test ideas are open, based on something that is interesting, but without a way to put a Pass/Fail status, and more aimed towards investigation and exploration for useful information.

> With creativity you can come up with original ways of solving a testing problem.
> Usability testing can be addressed by asking any heavy user what they think.
> Charisma might be best evaluated by having a quick look at a dozen of alternative designs.
> If there are a hundred options, you might want to take a chance and only look at five random.
> An unknown area can be vaguely expressed, focusing on enabling serendipity.

These are especially vital to get feedback on, since one unorthodox test idea can generate two others, that are better. It is also a chance to dismiss them as irrelevant.

---

[61] An inspiring list of 41 system aspects can be found in Joris Meerts, *Functional Testing Heuristics - A Systems Perspective*, http://www.testingreferences.com/docs/Functional_Testing_Heuristics.pdf

[62] A very good overview in Cem Kaner, *An Introduction to Scenario Testing*, 2003. http://www.kaner.com/pdfs/ScenarioIntroVer4.pdf

[63] For a good and nuanced analysis of pairwise, see James Bach, Patrick Shroeder, *Pairwise Testing: A best Practice that Isn't,* http://www.testingeducation.org/wtst5/PairwisePNSQC2004.pdf

[64] Page viii in Cem Kaner, Jack Falk, Hung Quoc Nguyen, *Testing Computer Software, Second Edition*, John Wiley & Sons, Inc., New York 1999

Visual Test Ideas

Some things can't be expressed with words. And many things are much more effective to communicate with images. Try to use visual representations of your tests, also because they can generate new thoughts and understandings.

Examples: state models, architectural images, technology representation, test elements relations, inspiring images of any kind.

The visual test ideas, as all the others, can be transformed to other types, morphed to many, or what is best suited.

What and How?

Most test ideas will probably be focused on what to test. Some will also state how to test it. This can have many benefits, e.g. if you write you're going to use Xenu's Link Checker, and someone points out that there's another tool that will fit the purpose a lot better. Sometimes it might be far and difficult to go from a test idea to an actual test, and this enables trust and opportunity grabbing for the test executor (which might be yourself.)

Be aware of the "what" and the "how", but treat them as one whole, because that will generate more effective and innovative tests, and a lot of synergy between different areas. Use a holistic mindset: at the same time as you dwell in details, you should keep an eye on the whole picture, the big system.

These categories are artificial, and only used to better explain the processes. When you know enough, forget about them, and re-build your own models of the test design that suit your, your colleagues' and your company's thinking. Make sure you change and add to your test ideas as you learn more. Get your test ideas reviewed. Your questions, hypothesis and answers will emerge.

Test Design Heuristics

All means are allowed when designing your tests. Use any information source, with any deviation, and rely on your instinct that this might be important to test.

Also take advantage of these rules of thumb, if they seem applicable:

1. All Analysis Heuristics can be used as Design Heuristics.
2. ALAP – Specify details for tests as late as possible. Things will change, and you will know more.[65]
3. Diverse Half-Measures – "it's better to do more different kinds of testing to a pretty good level, than to do one or two kinds of testing perfectly."[66]
4. Faster is better – so you can get more information from the product.
5. "automate 100% of the tests that should be automated."[67] – you know this better after you have done the test at least once.
6. No Flourishes Heuristic[68] – 1) You can get far with simple tests. 2) You don't have to use fancy test techniques, especially when it isn't needed.

[65] Robert Sabourin, EuroSTAR 2010 tutorial *Just-in-time testing*, http://www.amibugshare.com/courses/Course_Just_In_Time_Testing.zip

[66] Lesson 283 in Cem Kaner, James Bach, Bret Pettichord, *Lessons Learned in Software Testing*, John Wiley & Sons, Inc., New York 2002

[67] This is often fewer than you might think, because of the (maintenance) cost and implicit regression you already get. Quote comes from Alan Page, page 104 in *Beautiful Testing*, O'Reilly Media Inc., Sebastopol 2010

7. Complicate – 1) Complex test can make you really effective. 2) You cannot be too careful when designing complex tests.[69]
8. Coverage For Free – With one test, you can cover many different things. Utilize ongoing test ideas for quality characteristics. Keep the ideas in the back of your head and notice violations when they occur.
9. Appropriate Granularity – Consider the detail level of your tests, in order to improve reviewability.
10. Best Representative[70] – Sample by using the most common, error-prone, all-inclusive, important.
11. Importance Principle – Focus most tests on the main purpose, but look for other purposes, and probable mistakes.
12. Serendipity – Tests are better if they enable finding things we didn't know were important.
13. OK to Throw Away – Don't be scared to throw away tests you no longer believe are relevant.
14. I Know It When I Feel It – try the software and you will get a feeling about what's important, what is error-prone, and how to test for it.
15. Intuition – "Trust your intuition when thinking about things that are difficult to predict and when there is little information."[71]
16. The five examples from Lessons Learned in Software Testing – "Test at the boundaries … Test every error message … Test configurations that are different from the programmer's … Run tests that are annoying to set up … Avoid redundant tests."[72]
17. Align With Information Objectives – Can tests be tweaked towards various information objectives?
18. Good Tests - power, valid, value, credible, likely, non-redundant, motivating, performable, maintainable, repeatable, important, easy to evaluate, supports troubleshooting, appropriately complex, accountable, cost-effective[73], easy to understand, enables serendipity.
19. Change Strategy – If your testing doesn't reveal new, interesting information; it is time to change strategy.
20. Use Elsewhere? – Can this great test idea be applied to other areas?
21. Who can do it? – You may include test ideas that you aren't capable of performing.
22. Diverse Review – If reviewers don't have additions, use someone who thinks more differently.
23. Unread Test Design – If developers don't read your high-level test design, ask them why.
24. Favorite Technique – Your best test design technique[74] might not be a perfect match, but it might be the one that will give you the best result.
25. Any skill associated with exploratory testing, experience-based testing, or error-guessing can be useful.
26. Meta Questions – For tricky design problems, you can make use of questions like CIA Phoenix Checklist[75].

---

[68] Rikard Edgren, blog post *No Flourishes and New Connections Heuristics*, http://thetesteye.com/blog/2011/06/no-flourishes-and-new-connections-heuristics/

[69] From mail conversation with Henrik Emilsson. It should be noted that none of these heuristics are brand new; they have been used many times by many testers.

[70] E.g. Lesson 50 in Cem Kaner, James Bach, Bret Pettichord, *Lessons Learned in Software Testing*, John Wiley & Sons, Inc., New York 2002

[71] Gerd Gigerenzer, *Gut Feelings: The Intelligence of the Unconscious*, Penguin Group, New York 2007

[72] Lesson 38 in Cem Kaner, James Bach, Bret Pettichord, *Lessons Learned in Software Testing*, John Wiley & Sons, Inc., New York 2002

[73] Cem Kaner, *What is a Good Test Case?*, STAR East, May 2003, http://www.kaner.com/pdfs/GoodTest.pdf

[74] A good overview of test design techniques is available in chapter 3: Testing Techniques, Cem Kaner, James Bach, Bret Pettichord, *Lessons Learned in Software Testing*, John Wiley & Sons, Inc., New York 2002

[75] Pages 139-141 in Michael Michalko, *Thinkertoys: a handbook of creative-thinking techniques*, Ten Speed Press, Berkeley 2006

Testworthy

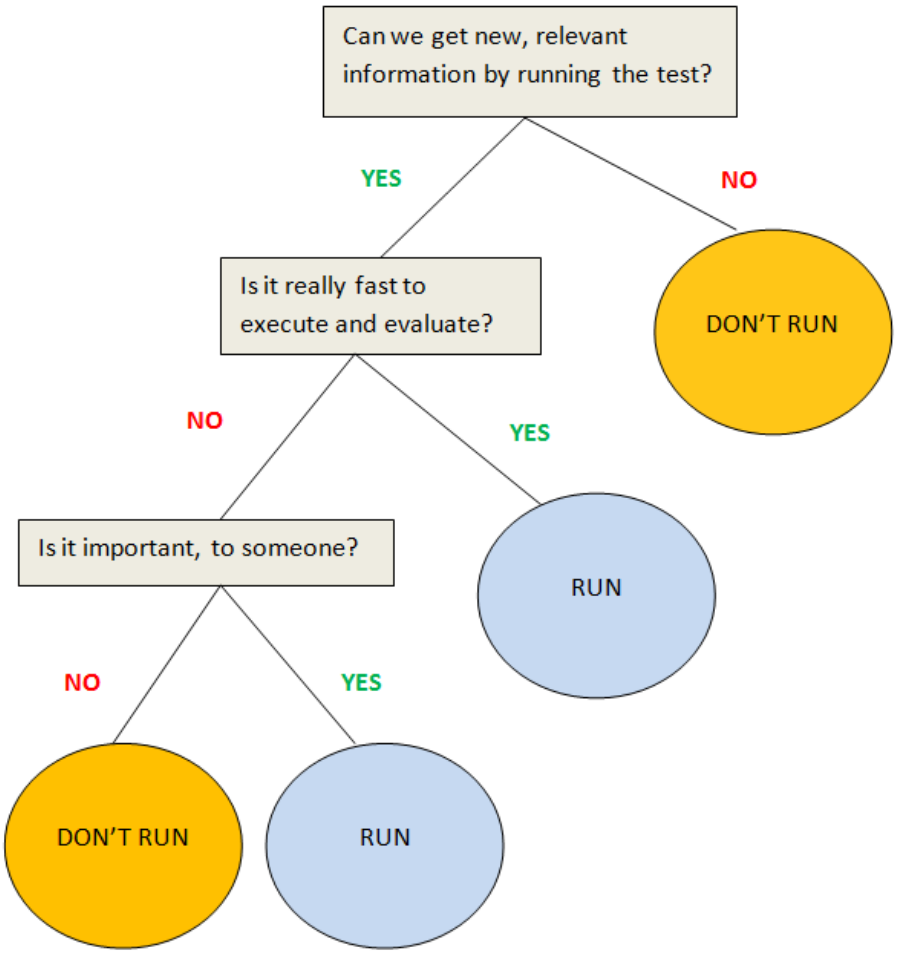
At some point you need to decide which tests to run and evaluate. It can be done, or thought about during analysis, or when synthesizing test ideas, or when executing tests. Probably you will do it all the time, and I think you will decide on the tests that you find most important. This is not magical, but includes many aspects: risk, speed, opportunity, promises, history, serendipity-enabling, goodness.

You don't want to spend a lot of time prioritizing tests, rather it feels natural to have a boundary called testworthy:

> *A test is testworthy if we think the information we can get is worth the time spending on it, regardless of risks, requirements, test approaches et.al.*

Gerd Gigerenzer describes in Adaptive Thinking[76] his theory (that originates from Herbert Simon) that intuition is based on heuristics, that we are using fast and frugal decision trees in order to make good **judgments** in a **complex** world, with **limited** information.

In order to explain and share the reasoning, give transparency to your intuition; you can try using a decision tree[77] for test triage[78]:



[76] Gerd Gigerenzer, *Adaptive Thinking: Rationality in the Real World*, Oxford University Press, New York 2000

[77] This is not a water-proof method, but might yield the very best results according to Gerd Gigerenzer, *Gut Feelings: The Intelligence of the Unconscious*, Penguin Group, New York 2007

[78] More on test triage by Robert Sabourin, *What Not to Test*, Better Software magazine November/December 2004, http://www.amibugshare.com/articles/Article_What_Not_To_Test.zip

The order and content of your questions might vary. In this instance, I check speed before importance, because I want to have chances of finding things we didn't know were important. At the other end, the "relevant" question might take away tests that would have been perfect, but this is OK. We can't include everything.
There is also a balance if it is worth the effort of running the test.[79]

The purpose of this paper is to help testers finding the most testworthy among the endless possibilities of software usage.

[79] Erik Jacobson, blog post *Eight Things You May Not Need To Test*, http://www.testthisblog.com/2012/01/eight-things-you-may-not-need-to-test.html

Test Execution

Some of the test design is better left until test execution. With the software at hand, you can make better decisions regarding details, and you can judge which variations and deviations that are fast and fruitful. You will find some of the information you were looking for, and some information you didn't know you needed. You will learn which areas are error-prone, which things can be tested fast, which things we want to know more about, what you actually can do with the software, how things are connected, and perhaps most importantly: what can get better.

Feed this information back to the test design, re-create and re-arrange your tests, go back to the information sources that were more important than you initially believed.

There are also a lot of detailed variations needed when trying to pinpoint possible problems, additional tests to design and execute in order to make a really good problem report.

Degree of Freedom

This is covered by the modern definition of Exploratory Testing:

> *"Exploratory software testing is a style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the value of her work by treating test-related learning, test design, test execution, and test result interpretation as mutually supportive activities that run in parallel throughout the project."*[80]

If testing is an endless sampling problem, you don't want any limiting factors. Diversity demands freedom, and ability to grab opportunities in the spur of the moment.

With a high degree of freedom you can change your test environment as needed; you can do variations and deviations to maximize the sort of diversity that is needed in your situation.
You can use uncontrolled environments and testers, because they can provide information you didn't know you were looking for.

Test Execution Heuristics

There exists many test execution heuristics; here are the ones I deem connected to the design aspect[81].

1. All Analysis Heuristics and Design Heuristics can be used as Execution Heuristics.
2. Basic Heuristic – if it exists, I want to test it.[82]
3. Important Problems Fast – Try to find the most important problems first.[83]
4. Benevolent Start – First see if easy, big chunks works OK.
5. Broader Partitioning – If "this" was OK or Not OK, we can skip "that".
6. Background Complexity – use "documents" that are more complex than necessary.[84]
7. Do One More Thing – additionally do something error-prone, popular or what a user might do. Don't think too much; just do something, and see what happens.

---

[80] Cem Kaner, blog post *Defining Exploratory Testing*, http://www.satisfice.com/kaner/?p=42

[81] E.g. many heuristics for when to stop testing can be found at Michael Bolton's blog, http://www.developsense.com/blog/2009/09/when-do-we-stop-test/

[82] continuing with: "(The only exception is if I have something more important to do.) " James Bach, blog post *Should Developers Test the Product First*, http://www.satisfice.com/blog/archives/54

[83] Lesson 5 in Cem Kaner, James Bach, Bret Pettichord, *Lessons Learned in Software Testing*, John Wiley & Sons, Inc., New York 2002

[84] Rikard Edgren, blog post *Background Complexity and Do One More Thing Heuristics*, http://thetesteye.com/blog/2011/03/background-complexity-and-do-one-more-thing-heuristics/

8. Dead Bee Heuristic – If I change a data file and see that it no longer crashes the application I'm testing, the next thing I do is change it back again so I can see the crash one more time.[85]
9. Rumble Strip – When product does strange things, a big disaster might be about to happen.[86]
10. Galumphing – Doing something in a deliberately over-elaborate way.[87]
11. User Mistakes – investigate unintentional (typical) mistakes.
12. Method acting – **be** a "real" user.[88]
13. Bisect[89] – when something strange happens, remove "half" until you find the necessary ingredients.
14. Flipability Heuristic[90] – Try to get value from unintended usage.
15. Basic Configuration Matrix – identify few platforms that spans over "boundaries".[91]
16. Concurrent Test Execution – Run several test ideas at the same time to get new interactions and ideas.
17. Déjà vu Heuristic – When you encounter an error message, repeat it once more (if it's not exactly the same, code is suspect.)[92]
18. Look at Many Places – The results of tests can be interpreted via several locations.
19. Problem Elsewhere – Can problems found exist at other places?
20. Hunches – "Trust your instincts. Try any test that feels promising"[93].
21. Pair Testing – Collaborative test execution stimulates new and faster thinking.
22. Test Aid – Could a tool help me right now?
23. Polarities Dynamics[94] – Switch between careful vs. quick, playful vs. serious, objective vs. subjective et. al.
24. Right Now – "what's the best test I can perform, right now?"[95]
25. Emotions – Use your senses.[96]
26. Fresh Eyes Find Failure – look at new things; let others look at your things[97].
27. Improvise![98]
28. Problem (Un)Certainty – There are always problems, but they might not be important.
29. I Might Be Wrong Heuristic - if I find many problems, it might be my model that is faulty.

---

[85] James Bach, blog post *Dead Bee Heuristic*, http://www.satisfice.com/blog/archives/39

[86] James Bach, blog post *Testing Heuristic: Rumble Strip*, http://www.satisfice.com/blog/archives/8

[87] James Bach, blog post *Three New Testing Heuristics*, http://www.satisfice.com/blog/archives/467

[88] Rob Lambert, blog post *There's method in the madness*, http://thesocialtester.posterous.com/theres-method-in-the-madness

[89] Included as a feature in James Bach's Perlclip, http://www.satisfice.com/tools/perlclip.zip

[90] Rikard Edgren, blog post *Flipability Heuristic*, http://thetesteye.com/blog/2011/05/flipability-heuristic/

[91] Rikard Edgren and Henrik Emilsson trick, see blog post *BCM – Basic Configuration Matrix*, http://thetesteye.com/blog/2008/05/bcm-basic-configuration-matrix/

[92] Bj Rollison, blog post *Boundary Testing – Hidden Loops and the Deja Vu Heuristic*, http://www.testingmentor.com/imtesty/2009/11/13/boundary-testing-hidden-loops-and-the-deja-vu-heuristic/

[93] Page 6 in Cem Kaner, Jack Falk, and Hung Q. Nguyen, *Testing Computer Software, Second Edition*, John Wiley & Sons, Inc., New York 1999

[94] Many examples of fruitful dynamics in James Bach, Jon Bach, Michael Bolton, *Exploratory Testing Dynamics* v2.2, http://www.satisfice.com/blog/wp-content/uploads/2009/10/et-dynamics22.pdf

[95] James Bach, *Exploratory Testing Explained*, http://www.satisfice.com/articles/et-article.pdf

[96] Michael Bolton, *Lightning Talk on Emotions and Oracles*, http://www.developsense.com/2007/05/lightning-talk-on-emotions-and-oracles.html

[97] Lesson 43 in Cem Kaner, James Bach, Bret Pettichord, *Lessons Learned in Software Testing*, John Wiley & Sons, Inc., New York 2002

[98] Jonathan Kohl, blog post *Getting Started with Exploratory Testing - Part 2*, http://www.kohl.ca/blog/archives/000186.html

Quicktests

Spice your execution with some fast, not always nutritious tests (these come from Kaner/Bach et.al.)[99]

30. Shoe Test – Find an input field, move the cursor to it, put your shoe on the keyboard, and go to lunch.
31. Boundary Testing – test at boundaries because miscoding of boundaries is a common error.
32. Whittaker Attacks[100] – Inputs: force error messages, default values, explore data, overflow buffers, find interactions, repeat many times. Outputs: Force different, invalid, property changes, screen refresh. Explore stored data and computations. Fill or damage file system; invalid/inaccessible files.
33. Interference Testing - Cancel, Pause, Swap, Abort, Back/Next, Compete, Remove, Logoff.
34. Follow up recent changes – unintended side effects.
35. Explore Data Relationships – exploit dependencies, trace data, interfere, delete.
36. Variability Tour – Vary as far as possible, in every dimension possible.
37. Complexity Tour – Look for the most complex features and data, create complex files.
38. Sample Data Tour – Employ any sample data you can, and all that you can. The more complex the better.
39. Continuous Use – While testing, do not reset the system. Leave windows and files open. Let disk and memory usage mount. You're hoping the system ties itself in knots over time.
40. Adjustments – Set some parameter to a certain value, then, at any later time, reset that value to something else without resetting or recreating the containing document or data structure.
41. Dog Piling – Get more processes going at once; more states existing concurrently.
42. Undermining – Start using a function when the system is in an appropriate state, then change the state part way through to an inappropriate state.
43. Error Message Hangover – Make error messages happen. Test hard after they are dismissed.
44. Click Frenzy – Testing is more than "banging on the keyboard", but that phrase wasn't coined for nothing. Try banging on the keyboard[101]. Try clicking everywhere.
45. Multiple Instances – Run a lot of instances of the application at the same time. Open the same files.
46. Feature Interactions – Discover where individual functions interact or share data. Look for any interdependencies. Tour them. Stress them.
47. Cheap Tools! – Learn how to use InCtrl5, Filemon, Regmon, AppVerifier, Perfmon, Task Manager, Threadhijacker, Zed Attack Proxy, Color Oracle (all of which are free.)
48. Resource Starvation – Progressively lower memory and other resources until the product gracefully degrades or ungracefully collapses.
49. Play "Writer Sez" – Look in the online help or user manual and find instructions about how to perform some interesting activity. Do those actions. Then improvise from them.
50. Crazy Configs – Modify O/S configuration in non-standard or non-default ways either before or after installing the product.[102]

---

[99] The included quicktests are quotes or interpretations of http://www.testingeducation.org/BBST/riskbased/BBSTRisk2005.pdf Many were collected at LAWST workshops, also see James Bach, blog post *Rapid vs. Hyper-Rapid Testing*, http://www.satisfice.com/blog/archives/9 and *Rapid Software Testing* course, http://www.satisfice.com/rst.pdf

[100] James Whittaker, *How to break Software: A Practical Guide to Testing*, Addison-Wesley, Boston 2002

[101] Use keys that often mean other things, like F1, Ctrl+P, Alt+03.

[102] See also Rikard Edgren, blog post *An Error Prone Windows Machine* http://thetesteye.com/blog/2008/04/an-error-prone-windows-machine/ which includes an important quicktest like installing and/or running as restricted user.

51. Grokking – Find some aspect of the product that produces huge amounts of data or does some operation very quickly.

Interpretation

Everything is interpretation.The purpose of testing is not to make it easy to set Pass or Fail, which doesn't mean much.[103] You should rather be open and broad in your search of important information. Sometimes it is necessary to start a test without an expected result, or an oracle at hand, and rather use your judgment and curiosity to pinpoint problem details or noteworthy information, and what is important. Often you need to do subsequent tests to find out what the last result meant. You can look at many places to avoid interpretation errors.

You can communicate anything you think is valuable to recipients, testers find stuff like bugs, risks, issues, problems, artifacts, curios, tests, dependencies, questions, values, approaches, ideas, improvements, workarounds, frames, connections.[104]

If you have a perfect key without doubt, of course use it as your oracle.

Serendipity

Even with the very best test design, you will not find everything that is important about the product. But with an open-minded and curious test execution, you can take advantage of the serendipity that is oozing in software testing.

If you don't experience serendipity, use this super-good practice[105]:

52. *be open to serendipity* –the most important things are often found when looking for something else. If we knew exactly where all bugs were located, it would suffice with automation or detailed test scripts (or maybe we wouldn't have to bother with tests at all?) So make sure that you look at the whole screen, and many other places as well. Odd behavior might be irrelevant, or a clue to very important information. Testing always involves sampling, and serendipity can be your friend and rescue.

Serendipity is not luck, but some good fortune is involved.

[103] Rikard Edgren, article *Addicted to Pass/Fail?*, Tea-time with Testers, August 2011 – Issue 5, http://issuu.com/teatimewithtesters/docs/tea-time_with_testers_august_2011__year_1__issue_v

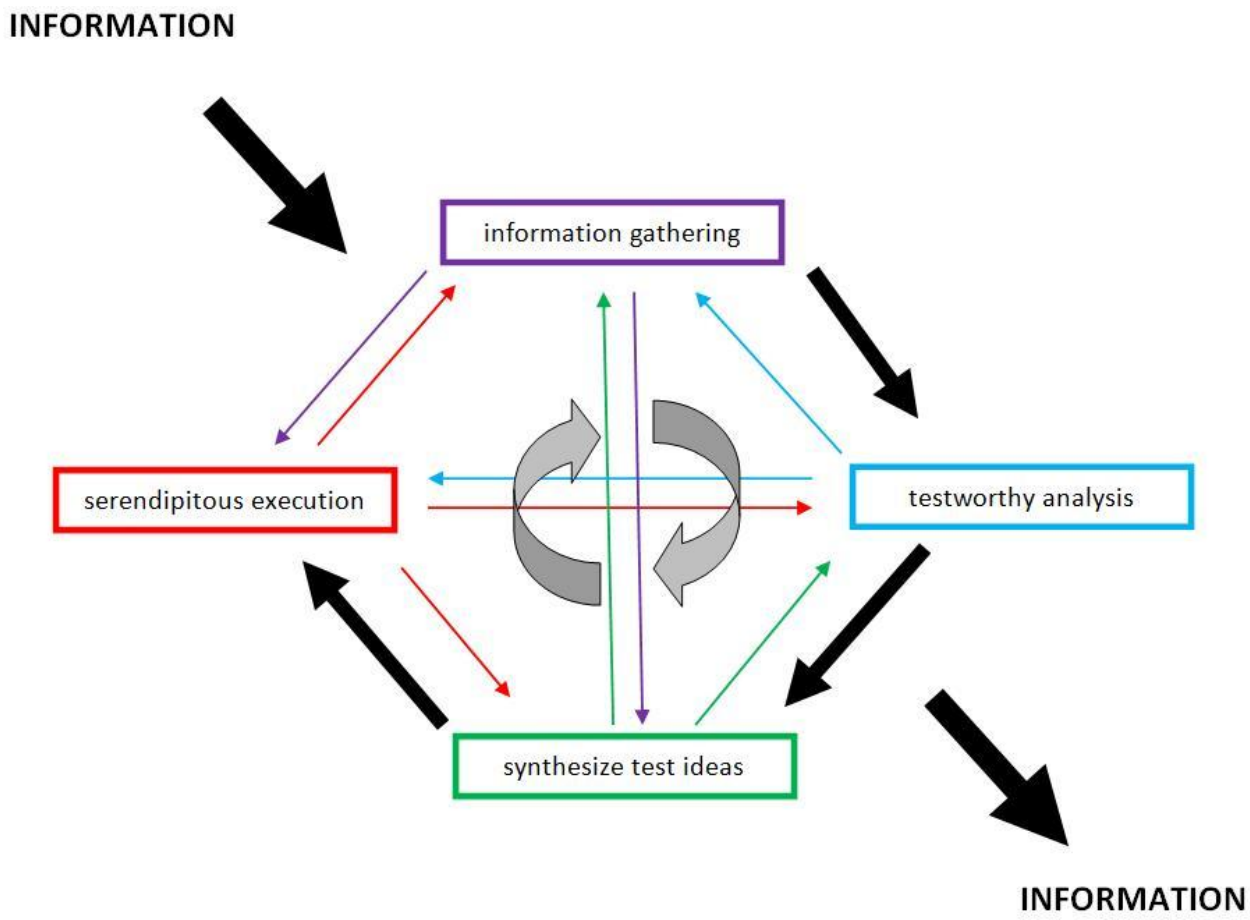
[104] James Bach blog post *What Testers Find* (and comments), http://www.satisfice.com/blog/archives/572
Elaborated by Michael Bolton at http://www.developsense.com/blog/2011/03/more-of-what-testers-find/ and http://www.developsense.com/blog/2011/04/more-of-what-testers-find-part-ii/

[105] See Rikard Edgren, blog post *Exploratory Testing Best Practices**, http://thetesteye.com/blog/2010/09/exploratory-testing-best-practices/

Finale

The described phases of test design are in reality not done in a linear fashion. Probably they go in all directions, span over several releases, and with a lot of changes and new ideas as test results are interpreted.

The process rather looks like this:



Testing adds quality to the information about the product

The most informal test design might happen extremely fast in the expert tester's head when thinking of a completely new idea while performing exploratory testing; or it could be the base for test scripts/cases/scenarios, that are carefully investigated and designed in advance. Good software testing uses many different methods.

Follow the scent, look at many places, with different perspectives, vary many things, few things, listen to your feelings, use data you know, don't restart the system, make it simple, complex, improvise!

Coverage

It is dubious to write about test design without mentioning coverage. But a representative coverage model is at least extremely difficult, and impossible to get in quantitative format without removing information about what is important (which is the coverage we should pursue.) For an example like code coverage, we can ask if it is more important to test the code than if the product is useful.

The proposed test design rather generates qualitative information, that is subjective, and sensitive to what is important in the same way as the customers' relation with the product. You can report what is tested and not, and your important findings, without mentioning coverage numbers.

There are two purposes of coverage models from a testing perspective:

1) Identify things to test
2) Decide when testing is completed

As we have seen in the previous chapters, there are many models and approaches to use in order to identify what to test. I doubt that molding all of these, and others, into multi-dimensional coverage models, or a model of models, is helpful.
As for the "when to stop" question, we can rather look at it from the opposite direction:

> **The test execution can continue until no new, relevant information is found. Change strategy, or stop; testing is saturated.**

If I have to choose a test coverage model, I would go for trying to cover everything deemed important.

Drawbacks

Although I believe this little book has a lot of useful information, it is far from perfect.
These are the most important problems I know of:

- The first ambitious collection of test design heuristics, and it can certainly be improved
- Not for everyone – too theoretical, too much/condense, irrelevant for many
- Pass/Fail test style is expected/demanded in many contexts, so alternative approaches won't work
- Lot of double work, especially with fragmented responsibilities
- A full-blown example would have given flesh to the ideas

Ending

Testers should break free from limitations of requirements-only, verification-focused test cases, and design many kinds of tests; you need to look at the whole picture to see the important details.

I am not claiming that the tests will be a true reflection of the reality, but I think they will have a good chance of finding things about the forthcoming reality of the relation between the user and the software.

By giving transparency to your test design you have an embryo for status reporting.

You should also strive for diversity in the people involved. Besides having a mixed test team, it also means that other roles should get themselves involved in different ways; not because they don't trust the testers, but because they all care about making a great product, many eyes see more. In the same fashion; you can be involved early in the project, and help with requirements gathering/analysis/understanding.

I think I have covered the most important things I know about test design. It's not the complete story, and only some parts are relevant to you. I do hope that everyone reaching this far have found at least a couple of new ideas. If you haven't, I hope you write your own paper, and send it to me.

Bibliography

Most of the content is experience-based. Some have been explored with colleagues, and things have been discussed with other testers. When available, I have tried to make references to books, articles, courses or blogs.

Books

James Bach, *Secrets of a Buccaneer-Scholar*, Scribner, New York 2009

Edward deBono, *Lateral Thinking - Creativity Step by Step*, 1990 Perennial edition

Lee Copeland, *A Practitioner's Guide to Software Test Design*, Artech House Publishers, Boston, 2003

Juliet Corbin and Anselm Strauss, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*, Second Edition, SAGE Publications, Inc., Thousand Oaks 1998

Donald C. Gause/Gerald M. Weinberg, *Exploring Requirements: Quality Before Design*, Dorset House Publishing Co. 1989

Donald C. Gause/Gerald M. Weinberg, *Are Your Lights On? How to Figure Out What the Problem REALLY Is*, Dorset House Publishing Co. 1990

Gerd Gigerenzer, *Adaptive Thinking: Rationality in the Real World*, Oxford University Press, New York 2000

Gerd Gigerenzer, *Gut Feelings: The Intelligence of the Unconscious*, Penguin Group, New York 2007

Adam Goucher and Tim Riley, *Beautiful Testing*, O'Reilly Media Inc., Sebastopol 2010

Cem Kaner, Jack Falk, and Hung Q. Nguyen, *Testing Computer Software*, Second Edition, John Wiley & Sons, Inc., New York 1999

Cem Kaner, James Bach, Bret Pettichord, *Lessons Learned in Software Testing*, John Wiley & Sons, Inc., New York 2002

Michael Michalko, *Thinkertoys: a handbook of creative-thinking techniques*, Ten Speed Press, Berkeley 2006

Glenford Myers, *The Art of Software Testing*, Second Edition, John Wiley & Sons, Inc., Hoboken 2004 (originally published 1979)

Torbjörn Ryber, *Essential Software Test Design*, Fearless Consulting Kb, 2007

James Whittaker, *How to break Software: A Practical Guide to Testing*, Addison-Wesley, Boston 2002

Articles

James Bach, *Exploratory Testing Explained*, http://www.satisfice.com/articles/et-article.pdf

James Bach, *Heuristic Test Strategy Model*, http://www.satisfice.com/tools/satisfice-tsm-4p.pdf

James Bach, Jon Bach, Michael Bolton, *Exploratory Testing Dynamics*, http://www.satisfice.com/blog/wp-content/uploads/2009/10/et-dynamics22.pdf

James Bach, Patrick J. Schroeder, *Pairwise Testing: a Best Practice That Isn't*, http://www.testingeducation.org/wtst5/PairwisePNSQC2004.pdf

Jonathan Bach, *Session-Based Test Management*, http://www.satisfice.com/articles/sbtm.pdf

Michael Bolton, *Test Framing*, http://www.developsense.com/resources/TestFraming.pdf

Michael Bolton, Better Software columns, http://www.developsense.com/publications.html

Fiona Charles, *Modeling Scenarios using Data*, http://www.quality-intelligence.net/articles/Modelling%20Scenarios%20Using%20Data_Paper_Fiona%20Charles_CAST%202009_Final.pdf

Rikard Edgren, Henrik Emilsson and Martin Jansson, *Software Quality Characteristics*, http://thetesteye.com/posters/TheTestEye_SoftwareQualityCharacteristics.pdf

Rikard Edgren, Henrik Emilsson and Martin Jansson, *37 Sources for Test Ideas* , http://thetesteye.com/posters/TheTestEye_SourcesforTestIdeas.pdf

Rikard Edgren, *Addicted to Pass/Fail?, http://issuu.com/teatimewithtesters/docs/tea-time_with_testers_august_2011__year_1__issue_v*

Rikard Edgren, *More and Better Test Ideas*, http://www.thetesteye.com/papers/redgren_moreandbettertestideas.pdf

Rikard Edgren, *Testing is an Island, A Software Testing Dystopia*, http://thetesteye.com/papers/redgren_testingisanisland.pdf

Rikard Edgren, *The Eye of a Skilled Software Tester*, http://wiki.softwaretestingclub.com/w/file/fetch/39449474/TheTestingPlanet-Issue4-March2011.pdf

Rikard Edgren, *Where Testing Creativity Grows*, http://thetesteye.com/papers/where_testing_creativity_grows.pdf

Elisabeth Hendrickson, *Test Heuristics Cheat Sheet*, http://testobsessed.com/wp-content/uploads/2011/04/testheuristicscheatsheetv1.pdf

Michael Hunter, *You Are Not Done Yet*, http://www.thebraidytester.com/downloads/YouAreNotDoneYet.pdf

Cem Kaner, *The Ongoing Revolution in Software Testing*, http://www.kaner.com/pdfs/TheOngoingRevolution.pdf

Cem Kaner, *What is a Good Test Case?*, http://www.kaner.com/pdfs/GoodTest.pdf

Cem Kaner, *An Introduction to Scenario Testing*, http://www.kaner.com/pdfs/ScenarioIntroVer4.pdf

Joris Meerts, *Functional Testing Heuristics - A Systems Perspective*, http://www.testingreferences.com/docs/Functional_Testing_Heuristics.pdf

Robert Sabourin, *10 Sources of Testing Ideas*, http://www.amibugshare.com/articles/Article_10_Sources_of_Testing_Ideas.pdf

Robert Sabourin, *What Not to Test*, http://www.amibugshare.com/articles/Article_What_Not_To_Test.zip

Neil Thompson & Mike Smith, *The Keystone to Support a Generic Test Process: Separating the "What" from the "How"*, http://doi.ieeecomputersociety.org/10.1109/ICSTW.2008.46

Courses

James Bach, Michael Bolton, *Rapid Software Testing*, http://www.satisfice.com/rst.pdf

Michael Bolton, *Lightning Talk on Emotions and Oracles*, http://www.developsense.com/2007/05/lightning-talk-on-emotions-and-oracles.html

Michael Bolton, *Confirmation Bias*, http://www.developsense.com/presentations/2010-04-ConfirmationBias.pdf

Cem Kaner, *Developing Skills as an Exploratory Tester*, http://www.kaner.com/pdfs/ExploratorySkillsQAI2007.pdf

Cem Kaner & James Bach, *Risk-Based Testing: Some Basic Concepts*, http://www.kaner.com/pdfs/QAIRiskBasics.pdf

Cem Kaner & James Bach, *Black Box Software Testing, Exploratory Testing*, http://www.testingeducation.org/BBST/exploratory/BBSTExploring.pdf

Cem Kaner, *Software Testing as a Social Science*, http://www.kaner.com/pdfs/KanerSocialScienceTASSQ.pdf

Cem Kaner & James Bach, *Black Box Software Testing, Specification-Based Testing* http://www.testingeducation.org/BBST/specbased/BBSTspecBased.pdf

Cem Kaner & James Bach, *Black Box Software Testing, Risk-Based Testing* http://www.testingeducation.org/BBST/riskbased/BBSTRisk2005.pdf

Cem Kaner & Rebecca Fiedler, *Black Box Software Testing, Test Design* http://www.testingeducation.org/BBST/testdesign/BBSTTestDesign2011pfinal.pdf

Robert Sabourin, *Just-in-time testing*, http://www.amibugshare.com/courses/Course_Just_In_Time_Testing.zip

Blogs

Blog portal http://www.testingreferences.com

Michael Bolton, http://www.developsense.com/blog/

Context-Driven Testing, http://www.context-driven-testing.com/

Rikard Edgren, Henrik Emilsson, Martin Jansson, http://thetesteye.com/blog/

Cem Kaner, http://www.satisfice.com/kaner/

Jonathan Kohl, http://www.kohl.ca/blog/

Rob Lambert, http://thesocialtester.posterous.com/

Quick Testing Tips, http://www.quicktestingtips.com

Software testing is a sampling problem, and we don't know exactly what we are looking for.

You can get a broader and better test space by

* investigating many, diverse information sources

* understanding what is important, and break down to elements

* synthesizing testworthy ideas by a range of heuristics

* explorative, serendipitous execution

* adapting to learnings

These people-centered, versatile test design methods are free from common limitations like requirements-only, verification-focused test cases, and rather emphasizes questions like *How can we learn the most about important aspects of the software?*

Rikard Edgren, started with software testing 1998, after a background with music and philosophy.
10 years with same product suite gave a depth that has been generalized in this paper.
More material can be found at thetesteye.com.
rikard.edgren@thetesteye.com

Including

Software Quality Characteristics 1.1
37 Sources for Test Ideas 1.0
104 test design heuristics

v1.1.1, August 2012