*The best way to learn complex activities is to try to do them*

2014-10-17

# Trying to Teach Testing Skills and Judgment

*An experience report from a higher vocational studies teacher*

Rikard Edgren

## Summary

Since 2011, I have spent 1500 hours of actively teaching software testing to higher vocational students in Sweden. It is educations spanning 1 to 2 years aimed at providing professionals that the industry needs. My colleague Henrik Emilsson and I set out to enable the students to master the activity of testing. We tried to create courses that let the students capture both explicit knowledge like testing techniques, but also tacit skills and judgments like these:

- Asking good questions
- Critical thinking
- Understanding what is important
- Rapid learning
- Seeing many perspectives and test ideas
- Selecting effective test strategies
- Capturing serendipity
- Realizing when testing is good enough

With a lot of calendar time available, it was possible to create many learning experiences; including lectures, stories, live examples, small exercises and complex exercises. We used real software most of the time, so the students would gain diverse experiences, and be forced to learn a lot, about the applications, and about testing. The most important part of the learning process is probably individual feedback on the work they do.

The different learning methods led to the possibility of giving students authentic problems; complex software with a real mission, so they had to use skills and judgment together with the testing lessons they hopefully had learnt.

I don’t have any proof that these methods are better than others, but I hope sharing our stories can help others advance the craft.

## Background

In 2011 I moved to Karlstad and switched company. Henrik Emilsson was involved in a soon-to-start 2-year higher vocational studies that had received funding, and I was excited to be a core part of this course, trying to make qualified testers out of non-testers with a variety of background (some people straight from school and others with years of experience from IT or elsewhere.)

We had for years discussed what constitutes a “good tester”, and now we had the chance of creating learning opportunities so students could become that. We did not know of any existing educational material that accomplishes this, even though the BBST[1] has great theoretical material, and the Rapid Software Testing course[2] inspires many, and sets a spark that help testers become great.

We did this work as consultants, but earning money was not our motivation, to say the least. Countless hours of unpaid preparations and examinations were put in, because we felt this was worthwhile, challenging, and a learning experience also for us.

The 2-year course mentioned has been run twice, with engagement from us for 30 weeks. We have also been involved 15 weeks in Gothenburg, 19 weeks at a distance course in Stockholm (third year is currently running), plus minor engagements for other programs, all being higher vocational studies. This means that our material has evolved, so this paper describes the current “findings”, not exactly what we did from start.

The programs cover a broad range of areas: testing theory, testing techniques, practical testing, test management, test tools, requirements analysis etc. This paper describes the tacit skills and judgment that is embedded within all of these areas. Within the programs there can also be courses for programming, databases, communication and more.

Within higher vocational studies in Sweden, an important part is LIA, Lärande i Arbete (Learning at Work), where the education takes place as internship at a company. A supervisor at the company helps the student, and this gives practical training and also networking opportunities. Students from other higher vocational studies can come to LIA with a theoretical ISTQB-based experience, and we wanted “our” students to have both theoretical and practical experience, so they can solve problems they will face in the industry, and make LIA an even better experience (for both the student and the company.)

## Skills and Judgment in Testing

The skills and judgment parts I find most interesting and valuable in software testing are:

- Asking good questions
- Critical thinking
- Understanding what is important
- Rapid learning
- Seeing many perspectives and test ideas
- Selecting effective test strategies
- Capturing serendipity
- Realizing when testing is good enough[3]

[1] BBST has three courses: Foundations, Bug Advocacy and Test Design. Available at http://testingeducation.org
[2] See www.satisfice.com for more information. Rapid Testing Intensive is the closest I have seen to what we do, but it spans over only 3 days. Also, the Per Scholas STEP program does similar things.
[3] Pretty much summing up core abilities of skilled exploratory testing?

…and more things I can't even express…

It is intentionally vague and intangible, it is deeper skills and judgments, which mechanical repetition cannot help improve significantly. Many of these skills are natural, and the students already have them. So one part of the teaching is to lure these skills out, that's where open and challenging exercises are crucial. It could also have been called tacit knowledge, which is defined by Harry Collins[4] as "*knowledge that is not explicated*", knowledge that isn't transferred as strings.

A common example of tacit knowledge is our ability to recognize faces, almost all of us can do it, but we can't really explain how we do it.

## Teaching Philosophy

These are constructions made afterwards, but Henrik has confirmed we operated under these values:

- **Motivation is key** – motivated students that have fun will learn more
- **Not for the money** – we create material we believe in, not what is fastest to do
- **It's not about us** – we are there for the students' sake, not to show off (even though that also happens…)
- **Encourage new ideas** – students' need to make mistakes, and aim for new perspectives
- **Don't be afraid** – with authentic, unique problems, we don't know what will happen; be confident that we will sort things out

## Testing Landscape

There are lots written about testing, and our teaching is rooted in the works of Cem Kaner and James Bach. We see testing as both a technical and humanistic activity, and we rely heavily on the Heuristic Test Strategy Model[5]. We add our own interpretations and stories from our combined 30 years of testing experience.

We put more emphasis on testing as an activity, and less on artifacts; the planning is more interesting than the plan, what is communicated is more important than a documented test report.[6]

## Developing courseware

The main reason there doesn't exist a lot of good educational material around testing's tacit skills and judgment is probably that it is difficult and time-consuming; it can't be explained, but it can be experienced. We can't say we know we have succeeded with this, but this is the basics of what we did.

Each sub-course has learning objectives that the class aims at and the teacher grades against. This brings a lot of examination responsibility that will not be dwelled upon here. Depending on the objectives, the courses will be different, but embedded in many of them are the skills and judgment this paper deals with. So when testing techniques are covered, we try to also get the students to be able to understand **when** a test technique is more

[4] Harry Collins, Tacit and Explicit Knowledge, 2010
[5] James Bach, Heuristic Test Strategy Model, http://www.satisfice.com/tools/htsm.pdf
[6] It is a bit ironic that grading mostly is done against documents students produce, their description of what they tested rather than the actual performance.

or less suitable. When automation tools are used, we stress the judgment of what to test, and why. To be able to do this, you need to know the tools and techniques, so some of the material is quite similar to other testing courses.

Book-wise, we prefer to use the following:

- Lessons Learned in Software Testing – Cem Kaner, James Bach, Bret Pettichord
- Essential Test Design – Torbjörn Ryber[7]
- Exploring Requirements – Daniel Gause, Jerry Weinberg
- Explore It! – Elisabeth Hendrickson[8]

We also developed slides material with a combination of BBST material (creative commons) and our own learnings we wanted to share. As an example, lectures on *scenario testing* are a lot of BBST plus our own exercises, but lectures for *Thinking like a tester* are mostly our own, with some Rapid Software Testing extracts.

Here is an example of areas covered in a first 7 week course with testing fundamentals:

1. Introduction to testing
2. Think like a tester
3. Test Strategy[9]
4. Exploratory and scripted testing
5. Test design techniques I (function testing, domain testing, specification-based testing)
6. Test design techniques II (risk-based testing, scenario testing)
7. Bug reporting

The most interesting parts are the exercises and assignments. This is where hands-on experiences generate a deeper understanding and enables acquiring of tacit knowledge (you will never learn to ride a bike without actual cycling.) A few exercises[10] were borrowed, but we have created about 100 of our own[11]; some simple, and some more advanced. Many exercises are based on open source applications, so there has been a lot of try-outs at sourceforge.net over the years. A hypothesis is that too many testing exercises are based upon finding one answer, teachers like them because they are in control. Our experiences indicate that all software has interesting things within them, and by using real applications, you build experiences you learn more from. There is no need to inject faults when the real world is full of them.

## Test Objects

The education is aimed towards the business, it is being held because there is an explicit market need for software testers. Hands-on skills are important, and a lot of practical testing should take place in the education. For this, virtually any software could work, but it is more interesting, and memorable if there are problems the students can find for themselves.

[7] Lee Copeland's A Practitioner's Guide to Software Test Design is equally good, but Ryber's book is also available in Swedish.
[8] Published in 2013, so only used recently.
[9] It might seem too early with test strategy, but this will be revisited later.
[10] Surprisingly, many publically available exercises did not feel good enough. I am not sure it's because they aim differently, or if it is because your own exercises feel a lot better. Two examples I use are a couple of black box machines from James Lyndsay, http://www.workroom-productions.com/black_box_machines.html, and BBST Testing Mission assignment, http://www.testingeducation.org/BBST/foundations/MissionOfTestingSpring2011a.pdf (from BBST Foundations.)
[11] Not counting the exercises I didn't think became good enough to use.

In order to find understandable software that isn't too good (is there any?) we decided to mainly use open-source applications. At least 100 applications have been tried, and many of them are used in different exercises[12].

Many of them are client applications for easier distribution and control, but we also have dedicated servers with web applications where they are allowed to use any kinds of tools.

As one example, I will tell you more about a core application we have relied heavily on: RedNotebook.

> "RedNotebook is a modern journal
>
> It includes a calendar navigation, customizable templates, export functionality and word clouds. You can also format, tag and search your entries. RedNotebook is Free Software under the GPL."[13]

This description is from the project web site, and also an exercise in itself:

> "What does this really mean? How should we test it"?

RedNotebook was perfect for our means, since it is

- Easy to understand
- Full of bugs
- Appropriately complex

We have used it in many exercises, for instance in bug reporting assignment: "In groups of two, find four bugs each (of different kinds) that you think the development team wants to fix, and report them in the best way you can. Review each other's bug reports, and turn in your assignment for feedback."

This assignment involves explicit knowledge: bug reporting a la RIMGEN[14]. And also tacit knowledge: to see problems, to understand which ones are important.

These exercises and other experiences helped me create "The Game" exercise, where 14 tasks are given.

1. Make RedNotebook impossible to start
2. Do a journal entry with 100.000.000 characters
3. Find a language problem
4. Provoke saving failure
5. Find an Undo + Redo problem
6. Fool word cloud with something that is interpreted as something else
7. Make export differ too much from original content
8. Identify an Operability problem
9. Find an incorrect calculation
10. Identify something that really should be mentioned in Help Contents
11. Trick RedNotebook regarding what date it should show
12. Find an insert image problem
13. Isolate combinations of formatting to a fixworthy bug
14. Find a spell-checker problem not related to actual spelling

---

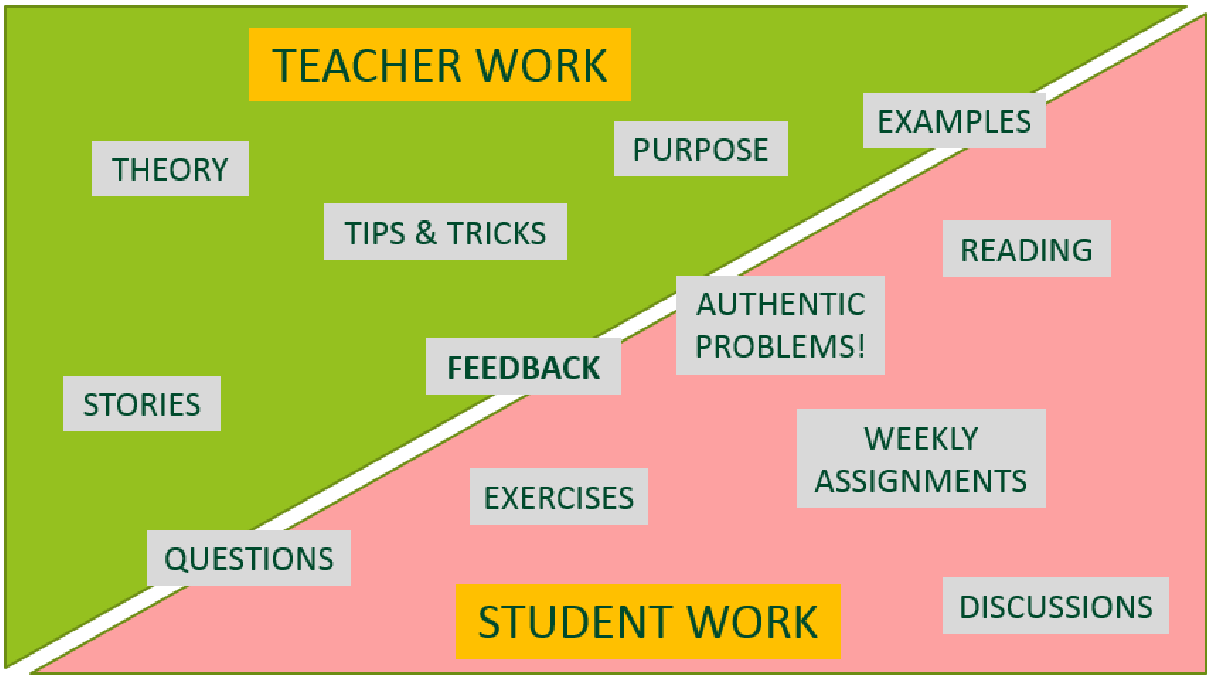
[12] Teacher tip: It is difficult to have an exercise in mind, and then find a suitable application. It is easy to find an application, and make a good exercise of it.
[13] Slogan at http://rednotebook.sourceforge.net/
[14] RIMGEN is Cem Kaner's mnemonic for better bug reports: Replicate, Isolate, Maximize, Generalize, Externalize, Neutral Tone, see BBST Bug Advocacy - http://www.testingeducation.org/BBST/bugadvocacy/

This exercise encourages bug hunting and learning skills; the game aspect forces more interaction with the teacher.

Publishing this paper might mean the end of RedNotebook as a core application, but that's about time.

## Diverse Learning Experiences

People learn in a variety of ways, and for difficult, complex things, several different kinds of learning opportunities are needed to enable deeper learning. The exact methods vary, but here is a categorization of different types of teaching we have used:



## Lectures

Lecturing is easy when you are well-prepared. It is useful to teach concepts and theoretical aspects of testing. It is a great way to inspire and help students find their intrinsic motivation. It can also be boring, un-engaging, and it doesn't help much for the tacit parts, you can't (in my experience) explain how to learn rapidly.

For testing skills and judgment, lectures should be mixed up with exercises. This makes it easier to keep focus, and the exercises also give experiences students can relate to, and examples that the teacher can use when explaining different ideas. The typical workflow is

1. explain the basics
2. simple exercise
3. the "full" theory
4. tips & tricks from the real world
5. more complex assignment
6. individual and group feedback

As an example, I believe testers should be inquisitive, which is easy to lecture about and tell stories. To teach it, it can help with thought-worthy slogans like

*“Question everything, especially this statement”*

This can be a starting point for discussions about What if-questions, but also about understanding that time is limited, and not everything should be questioned. (To reach the practical knowledge, you also need to encourage this during hands-on testing work.)

## Stories

A new area would typically start with a real world story, for two purposes:

- Build motivation with an engaging example
- Provide second-hand experiences for students

The first is most important for testers with experience, the second most important for newcomers to the field.

Here is an example I use to explain the importance of understanding your testing mission.

> **Our Biggest Failure**
>
> At one occasion we got the opportunity to test add-on products that were made for a few customers only. We were happy to help, and did fast and effective testing identifying dozens of bugs that were mailed to the developers. But we didn’t get an answer. They seemed occupied with other things, and after a while we understood that we had completely misunderstood the testing mission. We believed we should try to find as many bugs as possible, but what they needed was confirmation that the main functionality was OK, and an overall judgment about quality.
>
> A short conversation about expectations would have prevented this, in that case we would have reported the two important bugs we found. Worst in this failure is that we ruined a collaboration that took long time to repair.

## Live examples

Besides stories from our experiences, we perform actual testing during lectures. This is especially important for newcomers to the field that don’t know at all what testing actually looks like. I prefer to use something I never have tested before, because then it inspires my thinking and I can explain what I am doing in an authentic way. In this way they can see that it isn’t strange or unnatural, and I explain that the difficult part lies in the thinking, the figuring out of what is important, not so much in the actual physical execution[15]. My experience is that simplifying the testing activities and not running too fast, will set the students’ challenge at an appropriate level.

It is also comfortable to perform the same testing for different educations, where my two favorites are

- RedNotebook 1.6 introduced a feature where image width can be specified. How should that be tested?
- Investigate Spotify error handling when switching between online/offline[16]

[15] Depending on the situation I might downplay or emphasize how difficult testing is. In the beginning I want them to feel motivated and think “I can do this”, because they seem to learn more in that case.
[16] Video recording made in English available at https://www.youtube.com/watch?v=o0MXE8Onkh4

For distance students, I create videos where I practically show examples of testing. A few examples have been made in English, and are available at YouTube[17].

## Test artefact examples

We also share our own examples of test strategies, test analysis, test ideas, test reports etc. These are in Swedish, but two examples have been translated for your convenience and are available in Appendix A and B.

Some are real world products, and some are exercises we have done for ourselves, where we can explain in more detail the thinking process behind the artifacts.

Especially for plans and reports, I stress that it is examples and not templates; good testing requires a lot of understanding, which probably will give a unique documentation for the specific situation.

## Readings

Besides mentioned books, we reference blogs and articles that articulate testing thoughts with other words than our own. A few examples:

- James Bach. Heuristic Test Strategy Model: http://www.satisfice.com/tools/htsm.pdf
- Kaner/Fiedler, BBST Test Design: http://www.testingeducation.org/BBST/testdesign/BBSTTestDesign2011pfinal.pdf
- Michael Bolton, A Map By Any Other Name: http://www.developsense.com/articles/2008-11-AMapByAnyOtherName.pdf
- Jonathan Kohl, How do I create value with my testing? http://www.testingeducation.org/BBST/foundations/Kohl_Blog_CreateValue.pdf

These were sometimes accompanied by discussions with the class; what did you like/dislike; what didn't you understand?[18]

For students that are eager to learn more about an area, we also had extra discussions outside lecture time[19], and could recommend further material to study.

## "Simple" hands-on exercises

To start with, it is handy to have very simple applications to use as test objects. This means the practice can be more focused, with less distractions of different kinds (those distractions are part of reality, but they will be included in more complex exercises.)

A good "first practical testing" software is JShown[20], a software that shuts down your machine at a specific time. Simple UI, simple function, but still quite a lot to explore and test. No specific instructions are needed, but

[17] Spotify Space, Image Galumphing http://www.youtube.com/channel/UCjH3G-Gdpk72WhZx5Q3pzcw
[18] Signs of not understanding can be uncomfortable for a student, so student groups typically handle some questions better themselves.
[19] This is not only for the students' sake, new discussions give new ideas also for the teacher.
[20] Available at SourceForge: http://sourceforge.net/projects/jshown/

17 generic test ideas are provided if they get stuck[21]. This program is interesting because it doesn't work very well. It has state-dependent problems, it has interesting error handling (especially for students that tamper the configuration file) and it works differently on different platforms. And from experience I can say that just having a real software to investigate is motivating to most, if not all, students.

We have also created some small applications of our own that are simple, but still hands-on. The "best" one is called GuessTheNumber and details are given in Appendix C. This one is more of a puzzle, but there are plenty of things to think about and discuss in a debrief.

Another home-made application is FizzBuzz[22], resulting in the simplest test strategy exercise I could come up with. Since testers are the target group in this application, they can understand the purpose and what is important, and potentially come up with a multi-faceted test strategy. The exercise can be just strategy, or also with testing and reporting (parts can be automated in different ways). I am not happy with the functionality challenge in it (too simple), but it works well as a test strategy exercise. In Appendix D more details are given, and this time in an Exercise Instructor Notes format I am experimenting with.[23]

The exercises alone are not enough, they should be accompanied by:

- Introduction
- Guidance
- Debrief

It is not the results inside the exercise that matters, it is what happens in the students' minds. This can't be measured, but seeing the students evolve is a main benefit as a teacher.

## Thinking exercises

Abstract thinking exercises are useful to train your thinking in different ways. But it's not only the actual exercise, it is also how you use it, how you sum up together with students, and point them in further directions.

One interesting thinking exercise comes from http://wtquestions.blogspot.se/2009/04/first-question.html :

> There is an empty house. One person goes in and a bit later two persons come out.
>
> Explanations:
> By Biologist: Procreation – amitosis, fission, (asexual) reproduction etc
> By Mathematician: if one more person entered the building it would be empty again.
>
> How would a tester think?[24]

This can be discussed in groups, or as a whole class. Many ideas are encouraged, and there will probably be ideas about:

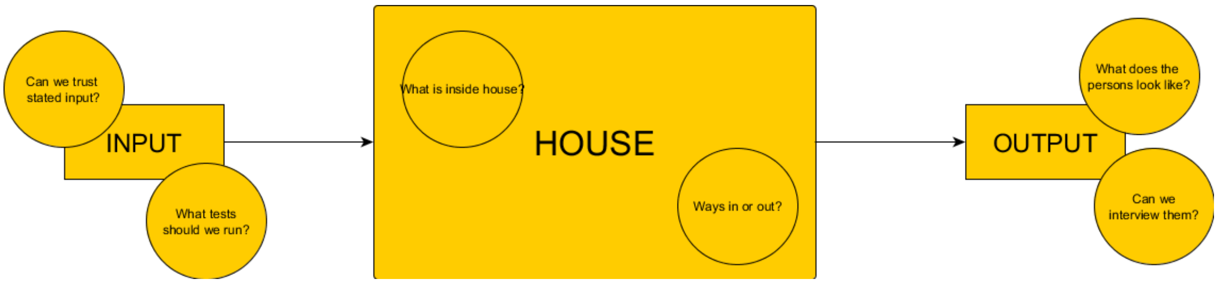
- Questioning – are the instructions (the specification) right? What exactly are the results?
- Reproduce – does it always happen?

[21] Rikard Edgren, Seventeen Test Ideas, http://thetesteye.com/blog/2011/10/seventeen-test-ideas/
[22] A children's game, replace numbers divisible by 3 with "Fizz", and divisible by 5 with "Fuzz", see http://en.wikipedia.org/wiki/Fizz_buzz
[23] The actual exercises used in the education program are in Swedish, and not very well documented.
[24] Original exercise stated "How would a tester explain the situation?", but I think that frames the question too much.

- Variations – for any person, for two persons
- Side-effects – does anything else happen? Is it an identical clone?
- Purpose – what's the house supposed to do? Without knowing, can we say something is wrong?

These are all relevant to testing, and can be further explained by real life testing examples.

It is also an opportunity to introduce models and systems thinking, so if the moment is right, the following model can be drawn:

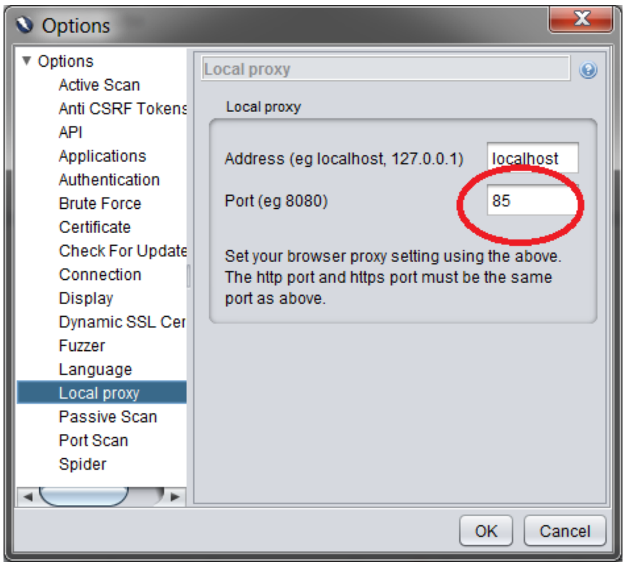

Can we trust stated input?
INPUT
What tests should we run?
What is inside house?
HOUSE
Ways in or out?
What does the persons look like?
OUTPUT
Can we interview them?

Not revolutionary, but mental models are important to testing, and this is one way to introduce them lightly.

## Intertwined areas

The courses are divided in different areas, but in reality testing activities belong together. My best experiences come when there are opportunities arising within the lectures and exercises.

One example happened the first time I taught equivalence partitioning (the technique all testers use all the time.) The test object was OWASP Zed Attack Proxy, a light-weight security testing tool I wanted them to use in later courses.



Address (eg localhost, 127.0.0.1) localhost
Port (eg 8080) 85

Set your browser proxy setting using the above. The http port and https port must be the same port as above.

We performed the testing together, looking at valid and invalid values, exploring the three different boundaries this one had: 65535, 2147483647, and about 10.000.000 characters.

We did string partitioning for the address field and discovered the interesting problem that an extended character, e.g. a misspelt "localhöst" would make ZAP impossible to start.

I didn't know about these problems, so it was a perfect opportunity to teach some bug reporting. So we searched the bug database, isolated the problems together, wrote a compelling reason and reported four bugs publicly, http://code.google.com/p/zaproxy/issues/detail?id=225 (and id's 224, 226 and 227, all of which were fixed.[25])

The issue with allowing millions of characters got this comment (bug selling):

> "This might not be important from a functionality point of view, but since the application is for security testers, it would be appropriate with robustness everywhere."

I believe this made the example more memorable, and also gave hands-on transferring of judgment skills like "maximizing consequences" and "bug advocacy".

Serendipitous teaching makes even more sense in testing, since it is a messy business with a lot of inter-twined activities.

## Weekly assignment

Typically there would be one bigger compulsory assignment per week, so the students are forced to work themselves, and to get something tangible to give feedback on. Some assignments were in groups, and some were individual.

We wanted them to put in practice what they had learnt, and here is an example from the week covering risk-based testing and scenario testing.

> **Risk-based scenario testing**
>
> 3-4 persons per group.
> The group selects the application you want to use.
>
> Develop product risks for the application.
> Create scenario tests towards appropriate risk(s)
>
> Perform the testing and describe problems you encounter.
>
> Hand in a report with risks, scenarios and bugs.
> Adjust your ambition so it requires about 8 hours per person.
>
> We will read, give feedback and the grade Pass or Not Pass.

This is a favorite assignment, since it produces unexpected results for a wide range of software, and often with very high quality.

## Group discussions

We let students do a lot of work in pairs or groups. One reason is that students tend to work better; but testing-wise it is especially important, since so much of the skills and judgment are tacit, they have to be experienced and learned by the students themselves. And this learning is accelerated by group work, by discussions where they see others' perspectives, and have to explain their own reasoning. When they explain testing to each other, they learn more about it.

[25] It doesn't always turn out this good, a getting started issue in JMeter 2.10 was quite dismissed: https://issues.apache.org/bugzilla/show_bug.cgi?id=55959

For on-site students we changed groups often, so they would also learn to co-operate with different people. For distance students, where communication is more difficult, there were pretty fixed groups, so they had similar study time, and easier communication.

Students in good groups get better than students in groups that don't work well together.

## Authentic problems

This is where it really happens. Maybe all the other activities are made in order to be able to learn for real in authentic problems.

We have used 2 different types of authentic problems:

1. Simulated situation. We build up a fake environment, with names and objectives, needs, obstacles and opportunities. No information is given up-front, in groups they have to discuss and ask the teacher questions. The software is real though, we have for instance used CiviCRM[26] with a made-up "Chrome support" scenario, where the project manager Coleman sends this letter after initial investigations:

   Hi!

   It has been exciting to read your test plans, this looks promising.
   We don't expect complete testing of CiviCRM, but it is important that Chrome (configured in many ways) is at least as good as IE and Firefox with regards to functionality, perceived performance, usability and charisma (thanks for that terminology!)
   You don't have to write any test cases in advance; but we want your test coverage summarized afterwards.
   We have never had test experts before, so please look around for important issues previously undiscovered.
   Based on your reports, we will make a decision to have dedicated testers in upcoming releases.
   Good luck!

   [illegible]

2. Real software. A group of 3 or 4 students choose an appropriate complex open source product, contacts the development team, and performs testing that has real value. Almost all development teams are glad to get testing help, and the groups also have the possibility to choose software that interest them (getting even more motivated to do a good job.)
   Depending on how much time is available, this could be done in different steps, with ongoing feedback:
   Preparations (analysis and strategy, getting to know the software to understand what is important)
   Execution (we emphasize testing in many different ways)
   Reporting ("*testing is never better than the communication of the results*")

   This is where we want them to use the things we have covered earlier: test strategy a la HTSM; choosing appropriate test methods in the given situation; reporting that is clear and understandable.

[26] CiviCRM is an open source software for constituency relationship management, http://civicrm.org

> The results of course vary, but often better than one could expect.
> One example is Sweet Home 3D, a mature interior design application, where a group of students used their own method "scenario based exploratory testing", including a Pippi Longstocking building for many kinds of angles. They entered at the start of Sweet home 3D 3.4 Beta, and in the end the developer asked them if they felt it was ready for release.

Especially the second kind of authentic problem have numerous possibilities and no fixed, correct answer. It is situations where teacher has little control, but can help with appropriate feedback.

You want the students to apply what they have learned in a new situation. So this is typically not done when the area is covered in the course, but rather something you hope will happen in other assignments given. Students won't apply everything they learn, they will get their favorite test techniques, just as any other tester.

## Pair testing

Tacit skills and judgment can also be transferred by performing testing together with students. This is time-consuming for larger groups of students, but can also show real-time examples and gives plenty of opportunities for direct feedback in the testing situation. This is something I would have liked to do more of, and it is extra difficult for distance students.

## Examination

Also the examination should be a learning process. We have no intention of letting students memorize different things, therefore written exams uses open questions that requires active thinking in order to be answered.

A typical 12-point question could be:

> A new health record system is about to be introduced at a health center. The product should store information about patients, treatments, and medication doses. All computers will run the same operating system and software. It has been decided that doctors are the most important users.
>
> Base your answer on these quality characteristics:
> capabilities, reliability, usability, charisma, security, performance, it-bility, compatibility.
>
> Choose the three quality characteristics you think are most relevant to test, and explain how they should be tested (a small test strategy.)

For this very question, one student[27] wanted to use everything he learned in a stressed exam situation. So he chose three quality characteristics, and for each of them he applied SFDIPOT, generating many different and interesting ideas. To combine CRUCSPIC STMP and SFDIPOT is nothing new, but to do it explicitly, for *very important characteristics* is nothing less than brilliant. That's why we now call this method the David Maneuver.

[27] David Jonasson, talented tester.

Also being examiner can give a negative power situation. It is important that students can trust me; that they know I won't trick them, and that I'll do my best to help them learn.

## Feedback

That the students get feedback on the work they do is essential for learning, but also for motivation. This felt difficult in the beginning, but after a while it was remarkably easy to see which areas students needed help with. I want them to struggle, and I know they will struggle, but at different places. Exercises typically don't need planted traps, because there are so many things that can "go wrong". Almost regardless of type of assignment, you could see areas where they needed strengthening.

That the feedback is individual and specific is obvious, but of course it helps to use similar exercises, so you know different things that might help each student to reach next level.

To some degree it is hard to decide if it is **teaching** or **coaching**, and I believe I do "wisdom-based coaching. That's where the coach knows the subject matter."[28] Examples of things to talk about:

- Someone might not be asking questions?
- Some might not act on answers.
- Students might test too narrow.
- Or look to broad without focusing deep on the most important.
- Or misunderstand the software.
- Learners might only use their own feelings as oracles,
- and some disregard subjective matters totally.
- They can observe too shallowly, being serendipity-free.
- We all have problems understanding when we have tested too little, or too much.

All of these can come from a misunderstanding of what the exercise really is, so letting them explain the mission with their own words might be the best way forward (or rather to start all over again.)

I use at least four styles of giving feedback:

- encouraging, "it is only difficult if you believe it is difficult"
- inquisitive: "why do you want to do that?"
- critical: "I see no variety in test data here"
- enlightening: "you could also try...”

There are things I want them to do as testers, but main focus is not on right or wrong, it is about seeing the thinking, encouraging new thoughts and explaining why.

All skills mentioned are intangible, tacit stuff that requires judgment for the specific situation. I can't "teach" this, but I can help them learn it.

After exercises and assignments I might give generic feedback to the whole class with typical mistakes made, and pointing to important things to remember for next time.

When I have the opportunity, I also like to have an individual conversation with each student towards the end of a course or the whole education. This gives them opportunity to ask their very own question, but is foremost

[28] James Bach on Twitter, 10-mar-2014

an opportunity to tell the student which strengths I believe they have, and what they should try to practice more.

The feedback from the teacher might be less important than the student's honest reflection it enables.

## Results

With such small samples and great diversity in courses and student background, there can of course be no evidence that these methods are better than others. However, we have observed many assignment hand-ins from students that are a lot better than what we generally see in the field. The students perform very well at the jobs they got after completing the education. (We are probably biased and put more focus on good examples (supporting our great courses…) than bad or non-existing examples.)

An open question sent on Skype to a hundred students only resulted in 4 answers (hopefully they are busy doing better things….) The answers are over-positive, but it is interesting to see students' own words[29]:

*Rikard has an educational style where he challenges students to think for themselves regarding the task. Rikard encourages questioning his views and the assignments, which creates a stimulating and creative learning environment for those that are curious and want to learn more. If a student wants a crystal clear education with easy assignments, you will probably not like it, but the setup creates the right conditions for becoming testers in that one after the education feels at home in the environment a test project is.*

*In short, Rikard's courses give both the needed knowledge, and creates the challenges needed to become a good tester.*
[Magnus Pettersson]

*Rikard's courses were among the best I received my whole time at the education provider. He has a good way of creating a relationship with his student which bring about comfort to ask questions that are not only related to our education but to future careers and personal development. I got introduced to Ruby during Rikard courses which has helped me develop not just within Ruby but with Python as well. Some of the skills I acquired during Rikard's courses includes: Ruby, JMeter, Python, Selenium, Sikuli, creative thinking etc.*

*I`m a better Automated Tester today, thanks to Rikard and the education provider.*
[Kelvin Yoreme Agadagba]

*The courses you and Henrik held gave something I haven't previously experienced, it was a combination of passion, interest and appreciation of the profession that I haven't seen elsewhere, including the 4 months I have worked as a tester after the education.*

*The course opened the door to the huge world of testing, and you introduced it in a way one could understand your passion and interest, and even experience it as a student.*

[29] I sent a general question in students' forum after completed education, which very few responded to: "Could you tell a few sentences about Rikard's and Henrik's teaching, and what you learned from it?" This of course will give a biased view from those who liked the course a lot. I would have loved to see more balanced and negative opinions, but did not force anyone to respond.

*Furthermore, your strong practical knowledge and brilliant theory was the icing on the cake that made me feel I really want to work with this as long as I can! Thanks!*
[Krzysztof Prieditis]

*The courses I took with Rikard was very well-prepared in terms of balance between theory and practice. The content and assignments are, according to me, very relevant for us future testers since they have dealt with current methods to write, review and execute tests. I have learnt to handle different test tools, and think more like a tester when it comes to creating a test plan and write test cases.*
[Rafael Silva]

The educations also have quantitative surveys, where the results are good, but they don't add value to this experience report.

Apart from students results, it is worth mentioning that the funding stakeholder (Swedish state) seem satisfied based on that more courses for testers are being held. The “industry” people I have talked to seem happy to get motivated employees with a good basic testing knowledge. The education providers hire us again, and I and Henrik Emilsson have learned a lot about testing and teaching.

## Finale

These methods were used in a context were there was a lot of time available. Students studied full-time, and courses spanned over many weeks. Hopefully this paper can inspire also for shorter courses, but especially the "authentic problems" probably require more calendar time. I would also recommend creating your own exercises, they will be much more meaningful for your purposes.

Of course there are a lot of tacit knowledge involved for the teacher of this. So this is not a handbook, but hopefully it can inspire some of you to get to the heart of the matter in your exercises and training. And this is not the end of our teaching; we already train at companies, and there are additional higher vocational studies and other upcoming initiatives.

I also want to thank reviewers who gave valuable feedback that improved this paper: Henrik Emilsson, Tina Lungström, David Jonasson, Per Amdahl, Ruud Cox, James Lyndsay and Michael Bolton.

# Appendix A: Libre Scenario Testing Example

This document is an example of a scenario test, and reporting of results. It is written for education purposes, but is designed to find relevant problems in LibreOffice.

Background: If LibreOffice is to take market shares from Microsoft Office, it is important that Microsoft documents can be converted easily. I have a complex document with a lot of formatting, and decided to do a scenario test, since I saw that the result wasn't exactly the same.

Scenario testing is suitable since it is not only functionality issues that matter, it is also important how they are experienced and how they can be worked around.

## LIBREOFFICE WORD COMPATIBILITY SCENARIO TEST

Purpose: Investigate if complex Office documents can be converted to LibreOffice, and look equally good.

Setting: Train ride from Stockholm to Karlstad. 2,5 hours available, including interruptions. New Ultrabook, sound turned off on machine. Office 2013, LibreOffice 4.1.3.

Actor: Rikard, software tester, advanced Office user, novice LibreOffice user, technically savvy, attention to detail, allergic to usability problems, author of The Little Black Book on Test Design.

Objective: To convert Word version of The Little Black Book of Test Design into a LibreOffice document that can produce good-looking PDF-booklet.
We know the document won't become exactly the same, so how long it will take to "fix" it, is important.

Motivation and emotions: Rikard might switch to LibreOffice for his software testing material, and is curious about the capabilities for conversion of his Microsoft Office documents.
He has a lot of others things on his mind, so there is a risk of losing focus, and temper.

Plot:
1. Open PDF The Little Black Book on Test Design in Adobe Reader, to use as oracle.
The document is 32 pages long, includes images, many footnotes and formatting to be printer-friendly.
2. Open .docx file in Word and re-save also as .doc and .rtf (to have three documents to compare, maybe one of them is better suited for LibreOffice.)
3. Open .docx file in LibreOffice writer, and look at all pages quickly
4. Produce a PDF and compare visually with original PDF. Write down differences.
5. Do the same with .doc and .rtf document, continue to work with the "best" one.
6. Edit formatting to produce a document that still is 32 pages. It doesn't have to be the exact same font, size, alignments, distances; but it has to "look good".
Write down any actions that feel cumbersome.
7. Now and then, produce a PDF to see that the end result gets better.
8. When the document is deemed good enough, put up the PDF on a web site.
9. Download and inspect the document at one other computer, and two mobile phones.

Evaluation: How long does it take to produce an equally good PDF?
Does Rikard still want to try using LibreOffice?

Events: Switch to Exploratory Test Design tutorial in PowerPoint format, with the same purpose.

## Test execution

Performed the test at 12 November 2013.

## Preparations

Installed LibreOffice 4.1.3.2 on Windows 7, 64-bitars Samsung Series 9 Ultrabook. Typical installation that caused no problem.

Loads the PDF in Acrobat Reader to have as oracle.

Creates three input files (.docx, .doc, .rtf)

Details: Font: Constantia 10 (8 in footnotes), Posters are using Cambria 9
Custom margins, e.g. 37 Sources have 0.59" in top and bottom, and 0.79" to the left and right. Quality Characteristics have 0.5" on all sides.

There are no warnings in Word when saving as.rtf and .doc. PDF exported from .rtf looks good.

Files are available in ”input” folder.

## LibreOffice

Starts LibreOffice Writer and language is Swedish, good.

Opens .docx-file (which is most modern, and probably most important)) which is 41 pages. Some problems are apparent when comparing with oracle:

Problem 1: Pages doesn’t fit. Title ”On Test Design” does not fit on one line. The margins are not the same as in Word (0.79, 0.79. 0.79, 0.49 – the last one should be 0.59)

Problem 2: Same page margin for all pages, in Word there are different margins for different sections.

Problem 2b: Usability: Difficult to find where to change page margins, finally found right-click, Page, tab Page.

Problem 3: For each footnote there are four extra newlines.

Problem 4: Footnote 54 and above have a newline after the footnote number.

Problem 4b: It is not fast to fix the footnotes reliably, so I delete the newlines manually. It took about 10 minutes, and I feel a bit annoyed.

Problem 4c: When I reload the saved document, the first space in footnote text has disappeared (text too close to footnote number.) A couple of more minutes of work …

Problem 5: Footnotes have more space between rows than oracle. I find no way to change this (I want them to close to each other.)

Problem 6: Some page breaks have disappeared (especially for the Posters.)

Problem 7: Strange page break between Test Execution Heuristic 14 and 15 (page area is wasted) Note: This was automatically fixed when opening the document later.

Problem 8: All text is not visible in table cells for 37 Sources. Fixed manually in two minutes, good operability on that one.

Problem 9: The vertical test in 37 Sources is not aligned in the same way. This was tricky to fix. Had to use Cell, On Top, plus remove Paragraph, Indentation, and add two spaces for it to look good. On the other hand, the result became better than in Word.

Problem 10: Table of Contents refers to wrong page numbers (fixed when updating table.)

Problem 11: The borders around Posters have disappeared.

Problem 11b: Can’t figure out if it should be possible to be able to set page layout (margin and border) for specific pages (bigger “break” than page break?) I have to live without borders, and customized margins.

Problem 12: Red triangle in text boxes (”software testing inspiration” et.al.) on page 3. Looks like an extra newline. Easy to remove manually, but hard to understand what it meant.

Problem 13: Can’t remove the red triangle for QR Code on page 10.

Problem 14: ”Software Quality Characteristics” can’t fit the same amount of text on one page, due to page footer. Disabled that one, and it looked good (but it felt dangerous as Libre Office said that information would be deleted.)
This solution is not optimal, since page numbering disappears.

Problem 14b: Put page numbers back, but found no easy way to get book layout (different adjustment on left and right pages.) With page numbers I get too many pages again, so I remove them …

The .rtf file has even more problems that I won’t delve into.

.doc file is better for some things (footnotes, borders) and worse in others (table of contents only at top level.) This one would probably be faster to fix.

Continuing with the.docx, and try to fix all problem (interesting observations added above.)

Created a PDF without changes, one PDF half way through, one when I felt finished, and notes more problems:

Can’t see any difference between 100% .jpeg-quality instead of 90 which is default (but picture quality isn’t very good in Word either...)

Problem 15: text in text boxes is cut off at bottom of the produced PDF. Maybe need more margin for borders in LibreOffice? Fixed by using bigger text boxes.

Problem 16: Page number in table of contents is not shown in exported PDF. Possible side effect of changing page margin. Fixed by updating.

Problem 17: Get an unexpected error message when choosing ”Show PDF after Export”:
”Can’t find file C:\work\software_testing_projects\writing\scenariotesting\output\tmp.pdf. Make sure you entered the correct name and try again.”

Problem 18: If you look carefully at the created PDF, the text isn’t equally good looking. It is slightly, slightly blurred, and doesn’t give a light and elegant impression. Might be because the PDF is much smaller, 425 KB instead of 1323 KB (but smaller size is good!)

Problem 19: Page 23 has a lonely word – ”everything.” Fixed by doing hard line break (not Shift+Enter.) At least partly author’s mistake.

I’ll stop here and publish a PDF on the web to have a look at it in a smart phone (same appearance as on computer.)

## Overall judgment

The whole process took longer than I expected. It is too many things that have changed; some things took too long time to fix; and some things I didn’t manage to fix. To use .doc file looks a bit more promising, but should that really be needed?

I am not happy with the end result, and will stick with Word.

Rikard Edgren, 13-november-2013, translated 7-march-2014

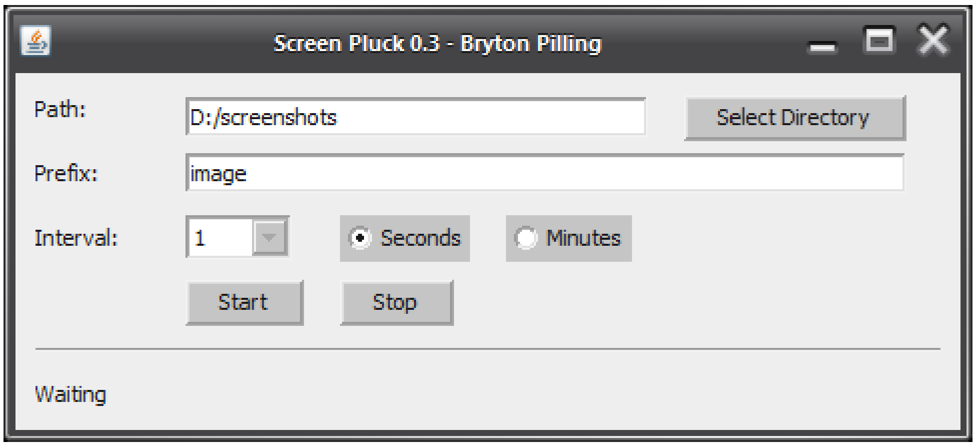
# Appendix B: Screen Pluck Test Analysis

This is a test analysis written by Rikard Edgren, version 0.4.
It is done for educational purposes, the idea is that it should provide an example of how to think in order to get the basis for testing a product well.

It is a very small application, that's why it is possible to go into detail also with the documentation.
The thought processes are roughly the same as for more complex software, but also totally different.

## Introduction

Screen Pluck is an application that takes screen captures at time intervals specified by user.


Screen Pluck 0.3 - Bryton Pilling

Path: D:/screenshots

Prefix: image

Interval: 1 [✓ Seconds] Minutes

Waiting

The user specifies file names and folder location, the software will add an increasing number.

## Strategies for test ideas

I start with using the software benevolently, trying to understand the purpose, and see its capabilities.
I will get a lot of ideas about things to test, and I prefer to write them down in one-liner format.

To be sure I test the software well I make a model, because it helps me think about each ”object” and ”action”.

Next I will have a look at "Software Quality Characteristics" to see I didn't miss something important.

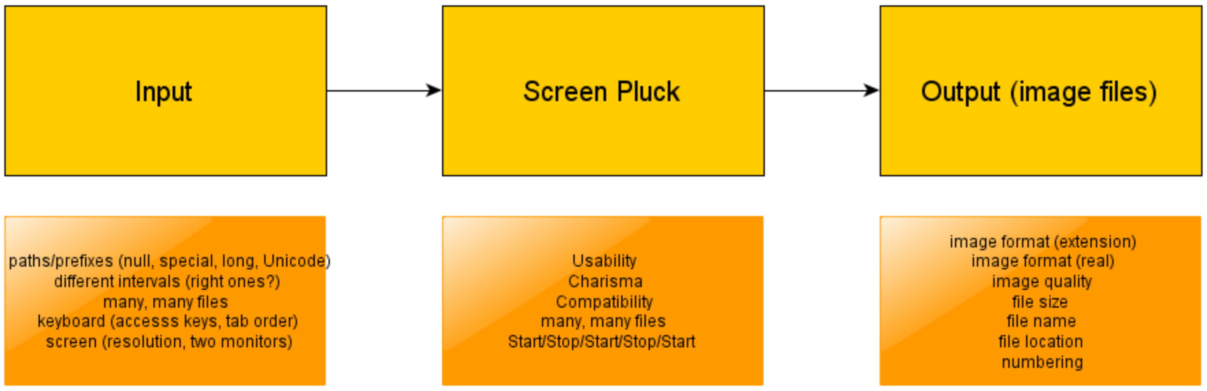
I do an SFDIPOT-analysis, where I can omit parts I already have considered (I usually start with SFDIPOT, but this time I did it later. You use what you want, in your preferred order, the important thing is that the result is an identification of things important to test, plus a good understanding.)

I conclude by considering risks that can be important.

As a complement I had a brief look at the source code.

## A model

A simple model where I look at each "object" and their interactions.



For each thing in a model, you can ask yourself *"What if this one doesn't exist?"*
In this case it gave test ideas: run Screen Pluck without monitor, shut down Screen Pluck when running, Save to non-existing location.

## Quality Characteristics

I look at the generic quality model http://thetesteye.com/posters/TheTestEye_SoftwareQualityCharacteristics.pdf, pick the relevant ones, and try to make them as specific as possible for Screen Pluck. This is more difficult than it seems, and the core ingredients are an understanding of the software, plus experience from other software.

These quality characteristics seem most important for Screen Pluck:

**Capability.** *Test that the functions perform what is promised*
- *Completeness:* available time intervals should be the "right" ones.
- *Accuracy:* screen pluck each 10 seconds should create 100 images after 1000 seconds.
- *Interoperability:* could there be problems with "path vs. prefix" or "numbers vs. second/minute"?
- *Concurrency:* what happens if four Screen Plucks are run at the same time?
- *Data agnosticism:* test valid and invalid values for path and prefix

**Reliability.** *Can you trust the product in many and difficult situations?*
- *Stability:* run Screen Pluck over the weekend
- *Robustness:* test error handling for (very) invalid folders and file names
- *Stress handling:* test what happens if disk is full; test with very little available memory or CPU

**Usability.** *Heuristic evaluation of usability sub-categories.*
- *Intuitive:* is it easy to understand how to get started?
- *Learnability:* appropriately small program that is easy to learn and remember?
- *Operability:* correct tab order and access keys?
- *Interactivity:* is it easy to see if product is running or paused?
- *Control:* does the user feel in charge?
- *Clarity:* is terminology easy to understand for "everyone"?
- *Error handling:* look at error message for invalid entries.
- *Tailorability:* will Screen Pluck persist path and interval when restarted?
- *Accessibility:* can Screen Pluck be used by blind? (e.g. Microsoft Narrator)
- *Documentation:* can I get relevant Help about Screen Pluck?

**Charisma.** *Free software need charisma to reach an audience*
- *Professionalism:* is GUI trustworthy, is it appropriately good looking?
- *Directness:* let ten persons look at Screen Pluck and say how they like it

**Security.** *Can be ignored for Screen Pluck?*

**Performance.** *Is Screen Pluck fast enough?*
- *Capacity:* use an incredibly large monitor, and take one screen capture per second
- *Resource usage:* is reasonable amounts of memory and processor used?
- *Response time:* is perceived performance good?
- *Endurance:* can it run for weeks?

**IT-bility.** *Screen Pluck doesn't have an installer, but should be easy to handle*
- *System requirements:* test on computers without Java, or with too old Java version.

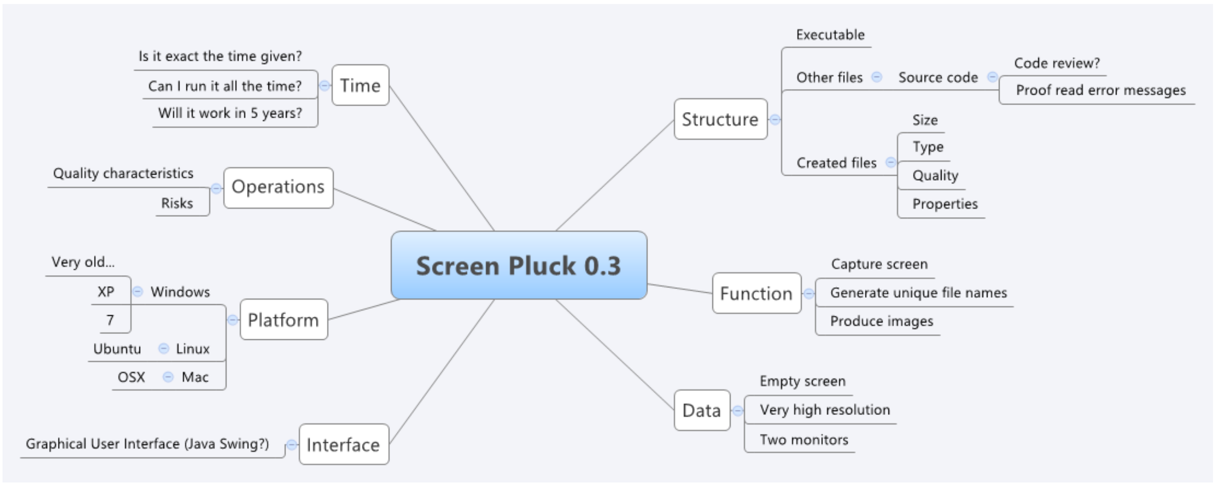
**Compatibility.** *How well does the product interact with software and environments?*
- *Hardware Compatibility*: find out how graphics card might matter.
- *Operating System Compatibility*: test on at least Windows, Linux and Mac (latest, and very old version)
- *Application Compatibility*: test with different kinds of screen "painting", e.g. full-screen games.
- *Configuration Compatibility*: test with Large DPI on Windows, and other special settings.

**Supportability.** *Can customers' usage and problems be supported?*
- *Identifiers:* does image properties state they were created by Screen Pluck?
- *Diagnostics:* is it possible to get details from a customer situation?
- *Troubleshootable:* is it easy to pinpoint an error (e.g. log file) and help?
- *Debugging:* can you observe internal states and variables if necessary?
- *Versatility:* is it possible to use Screen Pluck in other ways than intended?

**Testability.** *Will be easier to evaluate after testing has started.*
- *Traceability:* what should a log file contain?
- *Controllability:* if free-form intervals were available (10 captures per second), it would be easier to evaluate performance.

## SFDIPOT model

A lot has already been covered, but there were some new ideas when thinking through Structure, Function, Data, Platform, Interfaces, Operations, Time:



What about computer hibernation mode?

## Risks

I start by reading public information about the product at http://sourceforge.net/projects/screenpluck/

> Description:
> A simple screenshot utility that takes screenshots at user selected intervals and saves them as png files to a directory selected by the user. Written in Java using the Netbeans IDE. Works on any modern operating system.
>
> Features:
> * Small and Light Weight
> * Works on any operating System
> * Takes Screenshots at timed intervals
> * Images created are in .png format

I investigate keywords which gives me a few risks I think a product owner would agree with.
It is my understanding of this and other software that help me do this (an SFDIPOT model can also help.)

**simple** - risk that software isn't easy to use
**screenshots** - risk that complex "screens" aren't supported
**user intervals** - risk that frequency can't be adjusted as user wish
**png** - risk that images aren't saved in best way (quality, size, properties)

**NetBeans** – low risk that this component has serious problems in itself
**any OS** - risk that Screen Pluck won't function on some operating systems
**lightweight** - risk that installation and running isn't as smooth as one believes

At project page there are no reviews, but three "thumbs up".
There is no help or other product information accompanying the software (this could be seen as a problem in itself.)
There is a blog where the developer informs a little about new versions (0.4 is the latest, adding an "Overwrite" check box.)

I google "Screen Pluck" and mostly find it on download sites.
I find other blogs from developer where he explains that the software was written when he wanted to document working with a building in Second Life.

Lateral thinking helped me identify this risk:
Risk that developer don't have the rights to distribute the code (if it is borrowed from other places.)

## Test methods for important risks

Next step is to choose which risks to test, and how.
One can start with some kind of prioritization, and think about ways to test.

| Prio | Risk | Test method |
|---|---|---|
| 1 | Screen capture doesn't match reality | Test basic functionality on may platform, with double monitors, and very high resolution. |
| 2 | Software is not easy to use for all | Do a heuristic evaluation of the identified usability characteristics.<br><br>Start a reference group and interview them. |
| 3 | Not compatible on all platforms supporting Java, version 6 | Test supported platforms (executed at the same time as risk 1) |
| 4 | Charisma | Let the reference group tell what they think after 5 minutes (no instructions) |
| 5 | Software is not stable and robust | Perform stress testing over a weekend. |
| 6 | General Public License violations | Read the code to notice interesting things.<br>Run software that checks if code is ”borrowed”. |
| 7 | Other risks | Do exploratory risk-based testing on things not covered, and from other inspiration (competitors, SFDIPOT model, etc.) |

The last one is very open, which also enables adding new ideas we come up with as we test.

## Test ideas

Not everything can be tested, so the judgment about what to do is difficult and important.

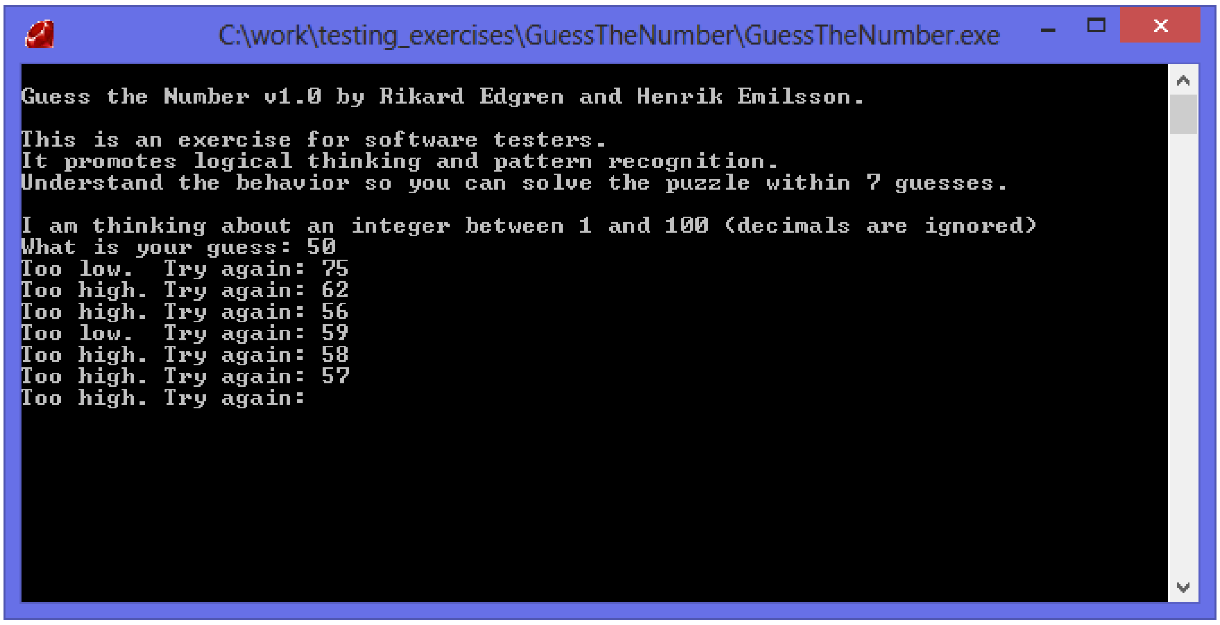
These test ideas seem most important, and should be included, whatever method you have been using. In some kind of priority order:

1. Test that the created images reflects what is shown on screen
2. Test that vaild paths and prefix can be used
3. Test on supported operating systems
4. Test that interval time is at least reasonably correct
5. Evaluate usability
6. Investigate if image size and quality is appropriate.
7. Run Screen Pluck for a long time to evaluate reliability, endurance and resource usage.
8. Investigate if software has any charisma
9. Test error handling for various incorrect input.

All the other test ideas in this document can be treated as “run if they are fast and we have nothing better to do.”

## Appendix C: Guess the Number excerpt

One of our custom made applications is such a one-hit wonder that we only want to show a small excerpt of it. It is a console application that typically render the following output the first time the students run it.



```
Guess the Number v1.0 by Rikard Edgren and Henrik Emilsson.

This is an exercise for software testers.
It promotes logical thinking and pattern recognition.
Understand the behavior so you can solve the puzzle within 7 guesses.

I am thinking about an integer between 1 and 100 (decimals are ignored)
What is your guess: 50
Too low.  Try again: 75
Too high. Try again: 62
Too high. Try again: 56
Too low.  Try again: 59
Too high. Try again: 58
Too high. Try again: 57
Too high. Try again:
```

When 56 is too low and 57 is too high, the interesting thinking parts start to happen.

Some students say they are done, the application is broken. I respond that they need to dig deeper, and that the instructions are correct.
Some use decimals: 56.5 (they are ignoring the instruction “decimals are ignored”), and get a warning from the application: “It seems you accidently used the same integer as last time.” To try decimals anyway is encouraged, as testers we take nothing for granted!
Some create hypothesis like the number being changed all the time. I respond (accurately) that the number is randomized at start, but not changed after that.
Some can get frustrated, and some laugh on remarks from the software like “I think you are trying to test the error handling. The exercise is about solving the game. I will count that as 2 guesses, for not following instructions."”

I will not reveal the solution here, but they struggle, they question, they perform different test, and learn about the application, and hopefully also about their thinking.

When the time is right (after between 20 to 60 minutes) the testing stops and we debrief in big group what happened, what they were thinking, and which tests they designed. I generally talk about:

- Critical Thinking. What can be wrong? Instructions, calculation, evaluation, fantastic theories, tester’s mistake?
- You need to do many experiments, in different ways, so you learn stuff. More tests, of different kinds, are the solution to the ideas your critical thinking produces, and more tests give more clues.
- Logical thinking. 2 raised to 7 equals 128. Bisecting as a testing technique.
- Observation about what happens, ability to analyze many observations and draw conclusions.

The exercise is done in pairs, otherwise it is even more difficult (can be solved in 5 minutes, but can also be a complete mystery after an hour.)

# Appendix D: FizzBuzz Exercise Instructor Notes

Blog post at http://thetesteye.com/blog/2014/01/lateral-tester-exercise-v-fizzbuzz/

## Purpose:

Test strategy exercises usually takes quite some time to perform properly. This software is so simple, so there is not a lot of functionality to get bogged down with. The learner is the target group, so they can grasp what is important. It should be possible to take a "whole product" approach, and design a test strategy that would find important issues, not only around functionality.

## Scene:

This exercise can be given online, without teacher interaction, but as with most exercises, a teacher who can give instant feedback is better.

Provide learners with the executable and the ruby file (so they can read the code, even if they don't have Ruby installed.)

Give them the instructions below, or let them read them when running the software.

If I want to speed things up, I will ask if anyone has any question, and if silence I'll say: "Are you sure you don't want to know about the purpose of the software or your testing mission?"

## Impromptu feedback

Some students might run it from a temp folder and not see the log file. Good time to give them feedback that they generally should make sure they have full control of the software they are running.

## Learner instructions:

This program is an exercise for software testers. As input it takes an integer between 1 and 1000, and repeats it as output. But if the number is a multiple of three, it should print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five it should give "FizzBuzz" as output.

The functionality is so straightforward so you will have time to investigate other things as well. Your testing mission is to find any threats to this being a useful exercise for testers around the world.

Test as much as you want, and answer these questions:

1. What would be a good test strategy?

2. What is your test coverage?

3. What are the results from your testing?

4. If you would use this exercise as a teacher, what would you talk about in the debrief?

Ruby file: FizzBuzz.rb, http://www.thetesteye.com/code/FizzBuzz.rb

Executable (Windows): FizzBuzz.exe, http://www.thetesteye.com/code/FizzBuzz.exe.zip

## Inspiration

http://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/

Write a program that prints the numbers from 1 to 100.

But for multiples of three print “Fizz” instead of the number and for the multiples of five print “Buzz”.

For numbers which are multiples of both three and five print “FizzBuzz”.

## FAQ

### What is the point of the exercise?

The point of the exercise is to practice testing skills like asking questions, acting on answers, test strategy, modeling, test design, test execution, result interpretation, test reporting.

### What is the purpose of the software?

The software is the testing exercise it is. I want it to be challenging and engaging, and a good learning experience for any tester that want to improve their skills.

### What is the testing mission?

Find things that would be threats to the value of it as a testing exercise. Also suggest improvements to it.

### What would be threats to the exercise?

Too many bugs, stupid bugs, no bugs, boring exercise, not possible to test in many ways…

### Should it be possible to translate?

Yes, that would be very good.

### What are your own results from testing it?

Can’t reveal that…

### Do you have instructor’s notes? Can we have them?

Yes, there are instructor notes with examples of solutions, and an FAQ (including this very question.)

No, you cannot have the instructor notes. You’ll get them after completing the exercise.

### Are you interested in Performance, usability, reliability, etc?

Yes, probably, can you be more specific?

### This was easy, I’ll just automate all values.

What is “all” values?

How will you do it?

Is there nothing more than functionality to this?

### Are there unit tests?

Yes, they are available at http://www.thetesteye.com/code/FizzBuzz_tests.rb

## My solutions:

See folder AllValuesAutoHotkey for external all values solution with AutoHotkey (took 20 minutes to produce)

See folder AllValuesRubyUnitTests for internal solution with Ruby unit tests (took 15minutes)

See folder FizzBuzzFuzz for many invalid strings, testing endurance, robustness (took 10 minutes)

### 1. What would be a good test strategy?

a) I would start by executing and getting a feel of it. Usability aspects will be evaluated, as well as noting interesting behavior.

b) I would do manual samples of fizz, buzz, fizzbuzz, number, too high, negative, way too high, too much input, strings, special words (fizz, ruby, null)

c) I would proof-read all text, including log file

d) pay a lot of attention to testability, especially test the content of log file

e) I would review the code

f) I would get a handful of testers to do the exercise to see how useful, and inspiring it is

g) Hopefully these testers have diverse platforms, but some additional operating systems and Ruby versions should also be tested.

h) I would write my own program that produces the same output, to check that all 1000 values are correct. Tests correctness, stability, endurance, and is a bit of fun as well. Feed these values into unit tests. (I have two examples of this; one with AutoHotkey, and one with Ruby unit tests.)

i) I would run many inputs with AutoHotkey, both valid and invalid, to see endurance and robustness.

j) I would try to talk to someone knowledgeable to make sure the requirements are good, and correctly understood by me.

### 2. What is your test coverage?

Answered in question 1?

### 3. What are the results from your testing?

a) Functionality is good, even though decimal interpretation could be discussed.

b) Code should be better structured, with classes, to enable better unit test solutions than copying fizzbuzz evaluation code.

d) Strings could be even better organized to enable easier translation (good idea with #Encoding: UTF-8 to the start of ruby file, but it won’t work for many MACs that have older than Ruby 1.9 pre-installed.)

e) The executable only runs on Windows 8, and is quite large (2,4 MB) I know this is what you get with Ruby, so another language, e.g. Java, should be considered.

f) AS DESIGNED. The functionality part is probably too easy (might be OK for a 20 minute exercise?), it should be more of a challenge in order for the exercise to have more charisma, and be more motivating. Maybe there should also be some bugs, people often remember more from those.

g) Software crashes when .exe is launched from C:\ on Windows.

## 4. If you would use this exercise as a teacher, what would you talk about in the debrief?

a) Sometimes the “Brute Force Heuristic” is suitable, and this is one example. Test all values, not just for functionality, but also for stability, and endurance. Do this by writing your own code and compare the results with a massaged log output form FizzBuzz (I would use AutoHotkey to create this) or by using the source code and add unit tests.

b) You have to know the context to get a good test strategy, and to do that, you need to ask questions. “How will FizzBuzz be used?” is answered by: “It is the testing exercise you are given, should be reusable by self-educating testers or teachers.”

So it’s about more than functionality. The answer indicates that understandability is very important for this software. That the text is clear, friendly, and not too difficult to translate. Compatibility with platforms is important, so Ruby might be a dubious choice.

Testability is a major concern, so you need to consider different ways to test (including unit testing) to see if it meets testing possibilities.

c) It is important how you communicate the strategy and the results. Strategy communication is a good way to get better answers about what should be tested.