First published in Software Testing and Quality Engineering Magazine, 11/99

Heuristic Risk-Based Testing

By James Bach

This is risk-based testing:

1. Make a prioritized list of risks.
2. Perform testing that explores each risk.
3. As risks evaporate and new ones emerge, adjust your test effort to stay focused on the current crop.

Any questions? Well, now that you know what risk-based testing is, I can devote the rest of the article to explaining *why* you might want to do it, and how to do it *well*.

Why Do Risk-Based Testing?

As a tester, there are certain things you must do. Those things vary depending on the kind of project you're on, your industry niche, and so on. But no matter what else you do, your job includes finding important problems in the product. Risk is a problem that might happen. The more likely the problem is to happen, and the more impact it will have if it happens, the higher the risk associated with that problem. Thus, testing is motivated by risk. If you accept this premise, you might well wonder how the term “risk-based testing” is not merely redundant. Isn't all testing risk-based?

To answer that, look at food. We all have to eat to live. But it would seem odd to say that we do “food-based living”. Under normal circumstances, we don't think of ourselves as living from meal to meal. Many of us don't keep records of the food we eat, or carefully associate our food with our daily activities. However, when we are prone to eat too much, or we suffer food allergies, or when we are in danger of running out of food, then we may well plan our lives explicitly around our next meal. Same with risk and testing.

Just because testing is motivated by risk does not mean that explicit accounting of risks is required in order to organize a test process. Standard approaches to testing are implicitly designed to address risks. You may manage those risks just fine by organizing the tests around functions, requirements, structural components, or even a set of pre-defined tests that never change. This is especially true if the risks you face are already well-understood or the total risk of failure is not too high.

If you want higher confidence that you are testing the right things at the right time, risk-based testing can help. It focuses and justifies test effort in terms of the mission of testing itself. Use it when other methods of organizing your effort demand more time or resources than you can afford.

If you are responsible for testing a product where the cost of failure is extremely high, you might want to use a rigorous form of risk analysis. Such methods apply statistical models, or comprehensively analyze hazards and failure modes. I've never been on a project where we felt the cost of rigorous analysis was justified, so all I know about it is what I've read. One well-written and accessible book on this subject is *Safety-Critical Computer Systems*, by Neil Storey. There is also a technique of statistically justified testing taught by John Musa in his book *Software Reliability Engineering*.

There is another sort of risk analysis about which relatively little has been written. This kind of analysis is always available to you, no calculator required. I call it *heuristic risk analysis*.

Heuristic Analysis

A heuristic method for finding a solution is *a useful method that doesn't always work.* This term goes back to Greek philosophers, but George Polya introduced it into modern usage in his classic work *How to Solve It*. Polya writes, "Heuristic reasoning is reasoning not regarded as final and strict but as provisional and plausible only, whose purpose is to discover the solution of the present problem." (p.113).

Heuristics are often presented as a checklist of open-ended questions, suggestions, or guidewords. A heuristic checklist is not the same as a checklist of actions such as you might include as "steps to reproduce" in a bug report. It's purpose is not to control your actions, but help you consider more possibilities and interesting aspects of the problem. For a wonderful set of heuristics for developing software requirements, see *Exploring Requirements: Quality Before Design*, by Don Gause and Gerald M. Weinberg.

Two Approaches to Analysis

Let's look at some heuristics for exploring software risk. I think of risk analysis as either inside-out or outside-in. These are complementary approaches, each with its own strengths.

Inside-Out

Begin with details about the situation and identify risks associated with them. With this approach, you study a product and repeatedly ask yourself "What can go wrong here?" More specifically, for each part of the product, ask these three questions:

- *Vulnerabilities:* What weaknesses or possible failures are there in this component?
- *Threats:* What inputs or situations could there be that might exploit a vulnerability and trigger a failure in this component?
- *Victims:* Who or what would be impacted by potential failures and how bad would that be?

This approach requires substantial technical insight, but not necessarily *your* insight. The times I've been most successful with inside-out risk analysis was when making “stone soup” with a developer. I brought the stones (the heuristics); he brought the soup.

Here’s what that looks like: In a typical analysis session we find an empty conference room that has a big whiteboard. I ask "How does this feature work?" The developer then draws a lot scrunched boxes, wavy arrows, crooked cylinders and other semi-legible symbology on the board. As he draws, he narrates the internal workings of product. Meanwhile, I try to simulate the mechanism in my head as fast as the developer describes it. When I think I understand the process or understand how to test it, I explain it back to him. The whiteboard is an important prop because I get confused easily as I assimilate all the information. When I lose the thread of the explanation, I can scowl mysteriously, point to any random part of the diagram, and say something like “I'm still not clear on how this part works.”

As I come to understand the mechanism, I look for potential vulnerabilities, threats and victims. More precisely, I make the *developer* look for them with questions such as:

- [pointing at a box] *What if the function in this box fails?*
- *Can this function ever be invoked at the wrong time?*
- [pointing at any part of the diagram] *What error checking do you do here?*
- [pointing at an arrow] *What exactly does this arrow mean? What would happen if it was broken?*
- [pointing at a data flow] *If the data going from here to there was somehow corrupted, how would you know? What would happen?*
- *What's the biggest load this process can handle?*
- *What external components, services, states, or configurations does this process depend upon?*
- *Can any of the resources or components diagrammed here be tampered with or influenced by any other process?*
- *Is this a complete picture? What have you left out?*
- *How do you test this as you're putting it together?*
- *What are you most worried about? What do you think I should test?*

This is not a complete list of questions, but it’s a good start. Meanwhile, as the developer talks, I listen for whether he is operating on faith or on facts. I listen for any uncertainty or concern in his voice, hesitations, or a choice of words that may indicate that he has not thought through the whole problem of requirements, design or implementation. Confusion or ambiguity suggests potential risk. When we identify a risk, we also talk about how I might test so as to evaluate and manage that risk.

A session like this lasts about an hour, usually, and I leave with an understanding of the feature, and a list of specific risks and associated test strategies. The tests I perform as a result of that conversation serve not only to focus on the risks, but also to refute or corroborate the developer’s story about the product.

There are wonderful advantages to this approach, but it requires effective communication skills on the part of the developer and tester, and a willingness to cooperate with each

other. You can perform this analysis without the developer, but then you have the whole burden of studying, modeling, and analyzing the system by yourself.

Inside-out is a direct form of risk analysis. It asks “What risks are associated with this thing?” Inside-out is the opposite of the outside-in approach, which asks “What things are associated with this kind of risk?”

Outside-In

Begin with a set of potential risks and match them to the details of the situation. This is a more general approach than inside-out, and somewhat easier. With this approach, you consult a predefined list of risks and determine whether they apply here and now. The predefined list may be written down or it may be something burned into your head by the flames of past experience. I use three kinds of lists:

- *Quality Criteria Categories*

These categories are designed to evoke different kinds of requirements. What would happen if the requirements associated with any of these categories were not met? How much effort is justified in testing to assure they are met to a “good enough” standard?

- *Capability.* Can it perform the required functions?
- *Reliability.* Will it work well and resist failure in all required situations?
- *Usability.* How easy is it for a real user to use the product?
- *Performance.* How speedy and responsive is it?
- *Installability.* How easily can it be installed onto its target platform?
- *Compatibility.* How well does it work with external components & configurations?
- *Supportability.* How economical will it be to provide support to users of the product?
- *Testability.* How effectively can the product be tested?
- *Maintainability.* How economical will it be to build, fix or enhance the product?
- *Portability.* How economical will it be to port or reuse the technology elsewhere?
- *Localizability.* How economical will it be to publish the product in another language?

I cobbled together this list from various sources including the ISO 9126 standard, Hewlett Packard’s FURPS list, and a few other sources. There is nothing authoritative about it except that it includes all the areas I’ve found useful in desktop application testing. I remember this list using the acronym CRUPIC STeMPL. To memorize it, say the acronym out loud and imagine that it’s the name of a Romanian hockey player. With a little practice you’ll be able to recall the list to mind any time you need it.

- Generic Risk List

Generic risks are risks that are universal to any system. These are my favorite generic risks:

- *Complex:* anything disproportionately large, intricate, or convoluted.
- *New:* anything that has no history in the product.
- *Changed:* anything that has been tampered with or "improved".
- *Upstream Dependency:* anything whose failure will cause cascading failure in the rest of the system.
- *Downstream Dependency:* anything that is especially sensitive to failures in the rest of the system.
- *Critical:* anything whose failure could cause substantial damage.
- *Precise:* anything that must meet its requirements exactly.
- *Popular:* anything that will be used a lot.
- *Strategic:* anything that has special importance to your business, such as a feature that sets you apart from the competition.
- *Third-party:* anything used in the product, but developed outside the project.
- *Distributed:* anything spread out in time or space, yet who elements must work together.
- *Buggy:* anything known to have a lot of problems.
- *Recent failure:* anything with a recent history of failure.

▪ Risk catalogs

A risk catalog is an outline of risks that belong to a particular domain. Each line item in a risk catalog is the end of a sentence that beings with “We may experience the problem that...” Risk catalogs are motivated by testing the same technology pattern over and over again. You can put together a risk catalog just by categorizing the kinds of problems you have observed during testing. Here's an example of part of an installation risk catalog:

- Wrong files installed
    - temporary files not cleaned up
    - old files not cleaned up after upgrade
    - unneeded file installed
    - needed file not installed
    - correct file installed in the wrong place
- Files clobbered
    - older file replaces newer file
    - user data file clobbered during upgrade
- Other apps clobbered
    - file shared with another product is modified
    - file belonging to another product is deleted
- HW not properly configured
    - HW clobbered for other apps
    - HW not set for installed app
- Screen saver disrupts install
- No detection of incompatible apps
    - apps currently executing
    - apps currently installed
- Installer silently replaces or modifies critical files or parameters
- Install process is too slow

- Install process requires constant user monitoring
- Install process is confusing
    - UI is unorthodox
    - UI is easily misused
    - Messages and instructions are confusing

For an example of a very broad risk catalog, see Appendix A of *Testing Computer Software*, by Cem Kaner, Jack Falk, and Hung Nguyen.

You can use these risk lists in a number of ways. Here's one that works for me:

1. *Decide what component or function you want to analyze.* Are you looking at the whole product, a single component, or a list of components?

2. *Determine your scale of concern.* I like to use normal, higher, and lower. Everything is presumed to be a normal risk unless I have reason to believe it's a higher or a lower risk. Use a scale that's meaningful to you, but beware of ambiguous scales, or scales that appear more objective than they really are.

3. *Gather information (or people with information) about the thing you want to analyze.* Obviously, you need to know something about the situation in order to analyze it. When I'm doing outside-in analysis on a product, I gather whatever information is convenient, make a stab at the analysis, then go to the people who are more expert than I and have them critique the analysis. Another way to do this is to get all those people in the same room at the same time and do the analysis in that meeting.

4. *Visit each risk area on each list and determine its importance in the situation at hand.* For each area ask "Could we have problems in this area? If so, how big is that risk?" Record your impression. Think of specific reasons that support your impression. If you're doing this in a meeting ask "How do we know that this is or is not a risk? What would we have to know in order to make a better risk estimate?"

5. *If any other risks occur to you that aren't on the lists, record them.* Special risks are bound to occur to you during this process.

6. *Record any unknowns which impact your ability to analyze the risk.* During the process, you will often feel stumped. For example, you might wonder whether a particular component is especially complex. Maybe it's not complex at all. What do you need to know in order to determine that? As you go through the analysis, it helps to make a list of information gathering todo items. At some point, go get that information and update your analysis.

7. *Double-check the risk distribution.* It's common to end up with a list of risks where everything is considered to be equally risky. That may indeed be the case. On the other hand, it may be that your distribution of concerns is skewed because you're not willing to make tough choices about what to test and what not to test. Whatever distribution of risks you end up with, double-check it by taking a few examples of equal risks and asking whether those risks really are equal. Take some examples of

risks and differ in magnitude and ask if it really does make sense to spend more time testing the higher risk and less time testing the lower risk. Confirm that the distribution of risk magnitudes feels right.

I recommend including a variety of people from a variety of roles in this analysis. Use people from technical support, development, and marketing, for instance.

Three Ways to Organize Risk-Based Testing

Whether you employ outside-in, inside-out, or some hybrid approach to doing the analysis, I can suggest three different ways to communicate the risks and organize the testing around those risks: risk check list, risk/task matrix, or component risk matrix.

- Risk watch list

This is probably the simplest way to organize risk-based testing. A risk watch list is just a list of risks that you periodically review to ask yourself what your testing has revealed about those issues. If feel you don't have enough recent information about problems in the product that are associated with a risk, then do some more testing to gather that information.

- Risk/task matrix

The risk/task matrix consists of a table with two columns. On the left is a list of risks; on the right is a list of risk mitigation tasks associated with each risk. Sort the risks by importance, with the most important risks at the top. Think of each row in the matrix as a statement of the form "If we're worried about risk X, then we should invest in tasks Y."

The risk/task matrix is useful mainly as a tool for negotiating for testing resources. I like using this technique in situations where management would not accept poor testing, yet also would not provide enough testing staff to do that job. The matrix helps bring management expectations in line with available resources. It's a lot easier to get testing resources when you can explain the impact of not having enough.

A disadvantage of this approach is that some tasks mitigate more than one risk. Also, some mitigation tasks cost so much or take so much time that they actually add more problems to the project than they're worth in terms of the problems they help detect. Still, it's a simple way to show the gross relationships between risk and test effort on a project-wide basis.

- Component risk matrix

The component risk matrix consists of a table with three columns. Break the product into 30 or 40 areas or components. These components can be physical code, such as "install program", functions, such as "print", or data, such as "clipart library". In other words, a component is anything that is subject to testing. In the leftmost column of each row of the matrix, list a component. In the rightmost column, list all of the known risk heuristics that indicate significant risk in that component (if a risk heuristic applies equally to all

components, don’t bother listing it). In the middle column write a risk judgment of "higher", "lower", or "normal".

Component	Risk	Risk Heuristics
Printing	Normal	distributed, popular
Report Generation	Higher	new, strategic, third-party, complex, critical
Installation	Lower	popular, usability, changed
Clipart Library	Lower	complex

What this matrix helps you do is help communicate and negotiate which components will get more effort. I use a general rule that higher risk items get twice the effort as normal items, which in turn get twice the effort as the components that are lower risk. This is just an approximation, of course.

The risk heuristics are included in the table because they help provoke questions about your risk judgments, but remember— there is no hard relationship between the heuristics and any particular judgment. You may find yourself in a situation where you will argue that one component is more risky than another, even though the first component has more heuristics driving its risk than the second. Risk analysis is a matter of *evaluating* factors that influence risk, not merely counting them.

As the project proceeds, you pay testing attention to different components in rough accord with their associated levels of risk. A disadvantage of this approach is that it focuses only on highlighting risks that increase the need to test, and not on those factors that decrease the need to test. You could add those risk lowering factors into the matrix, of course, but I find that it makes the matrix too complicated.

Making it All Work

Always keep this in mind: your risk analysis is going to be incomplete and inaccurate to some degree, and may be very wrong. All you really have at the beginning of a project are rumors of risks. As the project progresses, and you gain information about the product, you should adjust your test effort to match your best estimation of risk. Also, to deal with the risk of poor risk analysis, don’t let risk-based testing be the only kind of testing you do. Spend at least a quarter of your effort on approaches that are not risk focuses, such as field testing, code coverage testing, or functional coverage testing. This is called the *principle of diverse half-measures*: use a diversity of methods, because no single heuristic always works.

Finally, if I were to choose two vital factors needed to make risk-based testing work, I would name *experience* and *teamwork*. Over a period of time, any product line or technology will reveal its pattern of characteristic problems (assuming that you pay attention to problems found in the field). Learn from that. And do whatever you can to invite different people with different points of view into the risk analysis process.

If there's a magic to risk-based testing, it's the magic of noticing the signs and clues, all around you, about where the problems lie. Some people do this without consciously thinking about it, and maybe that's good enough. But when a problem slips by you because you couldn't do perfectly exhaustive testing, you may be called upon to explain why you did what you did. Management may assume that you did a sloppy job, and they may not be impressed with the standard argument that all testing is incomplete. That's when it's nice to have that risk list or matrix. With risk-based testing, you can show management that you strive to make the best use of the resources they invest. They'll respect you for that.

I thank Cem Kaner and Brian Lawrence for their advice and criticism in the development of this article.