FIFTY QUICK IDEAS TO IMPROVE YOUR TESTS

by
Gojko Adzic,
David Evans and Tom Roden

Fifty Quick Ideas to Improve Your Tests

Gojko Adzic, David Evans and Tom Roden

This book is for sale at http://leanpub.com/50quickideas-tests

This version was published on 2016-01-08

Tweet This Book!

Please help Gojko Adzic, David Evans and Tom Roden by spreading the word about this book on Twitter!

The suggested hashtag for this book is #50quickideas.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#50quickideas

Contents

Introduction

This book will help you test your software better, easier and faster. It's a collection of ideas we've used with various clients in many different contexts, from small web start-ups to the world's largest banks, to help team members collaborate better on defining and executing tests. Many of these ideas also help teams engage their business stakeholders better in defining key expectations and improve the quality of their software products.

Who is this book for?

This book is primarily aimed at cross-functional teams working in an iterative delivery environment, planning with user stories and testing frequently changing software under the tough time pressure of short iterations. The intended audience are people with a solid understanding of the basics of software testing, who are looking for ideas on how to improve their tests and testing-related activities. The ideas in this book will be useful to many different roles, including testers, analysts and developers. You will find plenty of tips on how to organise your work better so that it fits into short iterative cycles or flow-based processes, and how to help your team define and organise testing activities better.

Who is this book not for?

This book doesn't cover the basics of software testing, nor does it try to present a complete taxonomy of all the activities a team needs to perform to inspect and improve the quality of their software. It's a book about improving testing activities, not setting up the basics. We assume that readers know about exploratory testing and test automation, the difference between unit tests and integration tests, and the key approaches to defining tests. In short, this isn't the first book about testing you should read. There are plenty of good basic books out there, so read them first and then come back. Please don't hate us because we skipped the basics, but there is only so much space in the book and other people cover the basics well enough already.

What's inside?

Unsurprisingly, the book contains exactly fifty ideas. They are grouped into four major parts:

- *Generating testing ideas*: This part deals with activities for teams to engage stakeholders in more productive discussions around needs and expectations. The ideas in this part are equally applicable to manual and automated testing, and should be particularly useful to people looking for inspiration on improving exploratory testing activities.
- *Designing good checks*: This part deals with defining good deterministic checks that can be easily automated. The ideas in this part will help you select better examples for your tests and specifications, and in particular help with the given-when-then style of acceptance criteria.
- *Improving testability*: This part contains useful architectural and modelling tricks for making software easier to observe

and control, improve the reliability of testing systems and make test automation code easier to manage. It should be particularly useful for teams that suffer from unreliable automated tests due to complex architectural constraints.
- *Managing large test suites*: This part provides tips and suggestions on dealing with the long-term consequences of iterative delivery. In it, you'll find ideas on how to organise large groups of test cases so that they are easy to manage and update, and how to improve the structure of individual tests to simplify maintenance and reduce the costs associated with keeping your tests in sync with the frequently changing underlying software.

Each part contains ideas that we've used with teams over the last five or six years to help them manage testing activities better and get more value out of iterative delivery. Software delivery is incredibly contextual, so some stories will apply to your situation, and some won't. Treat all the proposals in this book as experiments.

Where to find more ideas?

There is only so much space in a book, and some of the ideas described deserve entire books of their own. We provide plenty of references for further study and pointers for more detailed research in the bibliography at the end of this book. If you're reading this book in electronic form, all the related books and articles are clickable links. If you're reading the book on paper, tapping the text won't help. To save you from having to type in long hyperlinks, we provide all the references online at 50quickideas.com.

If you'd like to get more information on any of the ideas, get additional tips or discuss your experiences with peers, join the discussion group 50quickideas.

This book is part of a series of books on improving various aspects of iterative delivery. If you like it, check out the other books from the series at 50quickideas.com.

Explore capabilities, not features

As software features are implemented, and user stories become ready for exploratory testing, it's only logical to base exploratory testing sessions on new stories or changed features. Although it might sound counter-intuitive, story-oriented exploratory testing sessions lead to tunnel vision and prevent teams from getting the most out of their effort.

Stories and features are a solid starting point for coming up with good deterministic checks. However, they aren't so good for exploratory testing. When exploratory testing is focused on a feature, or a set of changes delivered by a user story, people end up evaluating whether the feature works, and rarely stray off the path. In a sense, teams end up proving what they expect to see. However, exploratory testing is most powerful when it deals with the unexpected and the unknown. For this, we need to allow tangential observations and insights, and design new tests around unexpected discoveries. To achieve this, exploratory testing can't be focused purely on features.

Good exploratory testing deals with unexpected risks, and for this we need to look beyond the current piece of work. On the other hand, we can't cast the net too widely, because testing would lack focus. A good perspective for investigations that balances wider scope with focus is around user capabilities. Features provide capabilities to users to do something useful, or take away user capabilities to do something dangerous or damaging. A good way to look for unexpected risks is not to explore features, but related capabilities instead.

Key benefits

Focusing exploratory testing on capabilities instead of features leads to deeper insights and prevents tunnel vision.

A good example is the contact form we built for MindMup. The related software feature was that a support request is sent when a user fills in the form. We could have explored the feature using multiple vectors, such as field content length, email formats, international character sets in the name or the message, but ultimately this would only focus on proving that the form worked. Casting the net a bit wider, we identified two capabilities related to the contact form:

- A user should be able to contact us for support easily in case of trouble. We should be able to support them easily, and solve their problems.
- Nobody should be able to block or break the contact channels for other users through intentional or unintentional misuse.

We set those capabilities as the focus of our exploratory testing session, and this led us to look at the accessibility of the contact form in case of trouble, and the ease of reporting typical problem scenarios. We discovered two critically important insights.

The first was that a major cause of trouble would not be covered by the initial solution. Flaky and unreliable network access was responsible for many incoming support requests. But when the Internet connection for users went down randomly, even though the form was filled in correctly, the browser might fail to connect to our servers. If someone suddenly went completely offline, the contact form wouldn't actually help at all. None of those situations should happen in an ideal world, but when they did, that's when users actually needed support. So the feature was implemented correctly, but there was still a big capability risk. This led us to offer an alternative contact channel for when the network was not accessible. We displayed the alternative contact email address prominently on the form, and also repeated it in the error message if the form submission failed.

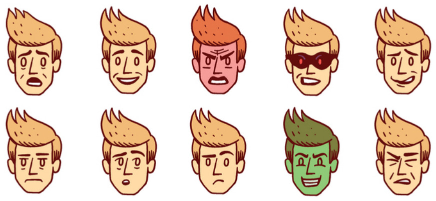
The second big insight was that people might be able to contact us, but without knowing the internals of the application, they wouldn't be able to provide information for troubleshooting in case of data corruption or software bugs. That would pretty much leave us in the dark, and disrupt our ability to provide support. As a result, we decided not to even ask for common troubleshooting information, but instead obtain and send it automatically in the background. We also pulled out the last 1000 events that happened in the user interface, and sent them automatically with the support request, so that we could replay and investigate what exactly happened.

How to make it work

To get to good capabilities for exploring, brainstorm what a feature allows users to do, or what it prevents them from doing. When exploring user stories, try to focus on the user value part ('In order to...') rather than the feature description ('I want ...').

If you use impact maps for planning work, the third level of the map (actor impacts) are a good starting point for discussing capabilities. Impacts are typically changes to capabilities. If you use user story maps, the top-level item in the user story map spine related to the current user story is a nice starting point for discussion.

Tap into your emotions



As testers are usually very quick to point out, the happy path is just the tip of the iceberg when it comes to the types of tests needed for adequately covering the many risks of any new software feature.

Starting with the happy path scenario certainly makes sense, as it provides us with a strong initial key example and a basis from which to think about other possibilities, but we don't want to get stuck there.

It is not always easy to see what other paths to take, what other permutations to try and techniques to use. Commonly taught techniques like boundary value analysis and equivalence partitioning are good ways of flushing out specific tests and focusing coverage, but they are not enough in themselves.

Whether in a specification workshop, designing test ideas afterwards or in an exploratory test session, having a heuristic for test design can stimulate some very useful discussion and upturn some stones that otherwise might have been left untouched.

The heuristic we propose is based on nine emotions or types of behaviour: scary, happy, angry, delinquent, embarrassing, desolate, forgetful, indecisive, greedy, stressful. As a mnemonic, 'shaded figs' is the best we can come up with, but even if it is too long to remember what each one stands for, hopefully it will trigger the thought to look it up.

Key benefits

The ‘shaded figs’ heuristic helps teams design more complete tests, whether up-front, say in a specification workshop, or during an exploratory session. It stimulates new ideas for tests and exposes other areas of risk for consideration.

Using this spectrum of test type ideas can deliver good broad coverage pretty quickly when designing or executing tests. It can also be a nice reminder in a specification workshop if you are looking for alternatives to the initial key example and for negative cases from a variety of perspectives.

How to make it work

One way to make this work is to start with the happy path and look along it for alternatives. As we step through the happy path, start thinking of other paths that could be taken using on this checklist.

Have the heuristic by your side and refer to it or work through it as a team as you explore a story or feature.

Here is our set of emotional heuristics to stimulate test design, taking an emotional roller coaster of a ride along the way:

- *The scary path* – if this path was followed it would really tear the house down, and everything else with it. Flush out those areas of the highest risk to the stakeholders. Think what would scare each stakeholder the most about this piece of functionality or change.

- *The happy path* – the key example, positive test, that describes the straightforward case. This is the simplest path through the particular area of behaviour and functionality that we can think of, it is the simplest user interaction and we expect it to pass every time (other than its first ever run maybe).
- *The angry path* – with the angry path we are looking for tests which we think will make the application react badly, throw errors and get cross with us for not playing nicely. These might be validation errors, bad inputs, logic errors.
- *The delinquent path* – consider the security risks that need testing, like authentication, authorisation, permissions, data confidentiality and so on.
- *The embarrassing path* – think of the things that, should they break, would cause huge embarrassment all round. Even if they wouldn't be an immediate catastrophe in loss of business they might have a significant impact on credibility, internally or externally. This could be as simple as something like spelling quality as 'Qality', as we once saw on a testing journal (just think of the glee on all those testers' faces).
- *The desolate path* – provide the application or component with bleakness. Try zeros, nulls, blanks or missing data, truncated data and any other kind of incomplete input, file or event that might cause some sort of equally desolate reaction.
- *The forgetful path* – fill up all the memory and CPU capacity so that the application has nowhere left to store anything. See how forgetful it becomes and whether it starts losing data, either something that had just been stored, or something it was already holding.
- *The indecisive path* – simulate being an indecisive user, unable to quite settle on one course of action. Turn things on and off, clicking back buttons on the browser, move between breadcrumb trails with half-entered data. These kinds of actions can cause errors in what the system remembers as

the last known state.

- *The greedy path* – select everything, tick every box, opt into every option, order lots of everything, just generally load up the functionality with as much of everything as it allows to see how it behaves.
- *The stressful path* – find the breaking point of the functions and components so you can see what scale of solution you currently have and give you projections for future changes in business volumes.

This technique works really well in specification workshops when multiple people are present, because the non-happy-path ideas are likely to generate interesting conversations, asking questions that have not been thought of yet and that are hard to answer. Some questions may need to be taken away and investigated further (non-functional characteristics repeatedly have this tendency).

Snoop on the competition

As a general rule, teams focus the majority of testing activities on their zone of control, on the modules they develop, or the software that they are directly delivering. But it's just as irresponsible not to consider competition when planning testing as it is in the management of product development in general, whether the field is software or consumer electronics.

Software products that are unique are very rare, and it's likely that someone else is working on something similar to the product or project that you are involved with at the moment. Although the products might be built using different technical platforms and address different segments, key usage scenarios probably translate well across teams and products, as do the key risks and major things that can go wrong.

When planning your testing activities, look at the competition for inspiration – the cheapest mistakes to fix are the ones already made by other people. Although it might seem logical that people won't openly disclose information about their mistakes, it's actually quite easy to get this data if you know where to look.

Teams working in regulated industries typically have to submit detailed reports on problems caught by users in the field. Such reports are kept by the regulators and can typically be accessed in their archives. Past regulatory reports are a priceless treasure trove of information on what typically goes wrong, especially because of the huge financial and reputation impact of incidents that are escalated to such a level.

For teams that do not work in regulated environments, similar sources of data could be news websites or even social media networks. Users today are quite vocal when they encounter problems, and a quick search for competing products on Facebook or Twitter might uncover quite a few interesting testing ideas.

Lastly, most companies today operate free online support forums for their customers. If your competitors have a publicly available bug tracking system or a discussion forum for customers, sign up and monitor it. Look for categories of problems that people typically inquire about and try to translate them to your product, to get more testing ideas.

For high-profile incidents that have happened to your competitors, especially ones in regulated industries, it's often useful to conduct a fake post-mortem. Imagine that a similar problem was caught by users of your product in the field and reported to the news. Try to come up with a plausible excuse for how it might have happened, and hold a fake retrospective about what went wrong and why such a problem would be allowed to escape undetected. This can help to significantly tighten up testing activities.

Key benefits

Investigating competing products and their problems is a cheap way of getting additional testing ideas, not about theoretical risks that might happen, but about things that actually happened to someone else in the same market segment. This is incredibly useful for teams working on a new piece of software or an unfamiliar part of the business domain, when they can't rely on their own historical data for inspiration.

Running a fake post-mortem can help to discover blind spots and potential process improvements, both in software testing and in support activities. High-profile problems often surface because information falls through the cracks in an organisation, or people do not have sufficiently powerful tools to inspect and observe the software in use. Thinking about a problem that happened to someone else and translating it to your situation can help establish checks and make the system more supportable, so that problems do not escalate to that level. Such activities also communicate potential risks to a larger group of people, so developers can be more aware of similar risks when they design the system, and testers can get additional testing ideas to check.

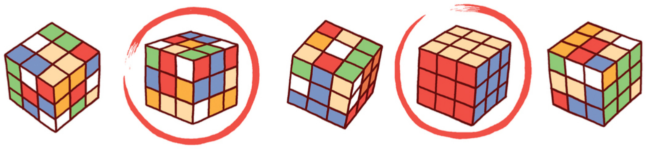
The post-mortem suggestions, especially around improving the support procedures or observability, help the organisation to handle 'black swans' – unexpected and unknown incidents that won't be prevented by any kind of regression testing. We can't know upfront what those risks are (otherwise they wouldn't be unexpected), but we can train the organisation to react faster and better to such incidents. This is akin to government disaster relief organisations holding simulations of floods and earthquakes to discover facilitation and coordination problems. It's much cheaper and less risky to discover things like this in a safe simulated environment than learn about organisational cracks when the disaster actually happens.

How to make it work

When investigating support forums, look for patterns and categories rather than individual problems. Due to different implementations and technology choices, it's unlikely that third-party product issues will directly translate to your situation, but problem trends or areas of influence will probably be similar.

One particularly useful trick is to look at the root cause analyses in the reports, and try to identify similar categories of problems in your software that could be caused by the same root causes.

Focus on key examples



User stories need clear, precise and testable acceptance criteria so that they can be objectively measured. At the same time, regardless of how many scenarios teams use for testing, there are always more things that can be tested. It can be tempting to describe acceptance criteria with loads of scenarios, and look at all possible variations for the sake of completeness. Although trying to identify all possible variations might seem to lead to more complete testing and better stories, this is a sure way to destroy a good user story.

Because fast iterative work does not allow time for unnecessary documentation, acceptance criteria often doubles as a specification. If this specification is complex and difficult to understand, it is unlikely to lead to good results. Complex specifications don't invite discussion. People tend to read such documents alone and selectively ignore parts which they feel are less important. This does not really create shared understanding, but instead just provides an illusion of precision and completeness.

Here is a typical example (this one was followed by ten more pages of similar stuff):

```
Feature: Payment routing

In order to execute payments efficiently
As a shop owner
I want the payments to be routed
using the best gateway

Scenario: Visa Electron, Austria
  Given the card is 4568 7197 3938 2020
  When the payment is made
  The selected gateway is Enterpayments-V2

Scenario: Visa Electron, Germany
  Given the card is 4468 7197 3939 2928
  When the payment is made
  The selected gateway is Enterpayments-V1

Scenario: Visa Electron, UK
  Given the card is 4218 9303 0309 3990
  When the payment is made
  The selected gateway is Enterpayments-V1

Scenario: Visa Electron, UK, over 50
  Given the card is 4218 9303 0309 3990
  And the amount is 100
  When the payment is made
  The selected gateway is RBS

Scenario: Visa, Austria
  Given the card is 4991 7197 3938 2020
  When the payment is made
  The selected gateway is Enterpayments-V1
...
```

The team that implemented the related story suffered from a ton of bugs and difficult maintenance, largely caused by the way they captured examples. A huge list such as this one is not easy to break into separate tasks. This means that only one pair of developers could work on it instead of sharing the load with others. Because of this, the initial implementation of underlying features took a few weeks. There was so much complexity in the scenarios, but nobody could say if they painted the complete picture. Because the list of scenarios was difficult to understand, automated tests did not give business users any confidence, and they had to spend time manually testing the story as well. The long list of scenarios gave the delivery team a false sense of completeness, so it prevented them from discussing important boundary conditions with business stakeholders. Several important cases were interpreted by different people in different ways. This surfaced only after a few weeks of running in production, when someone spotted increased transaction costs.

Although each individual scenario might seem understandable, pages and pages of this sort of stuff make it hard to see the big picture. These examples try to show how to select a payment processor, but the rules aren't really clear from the examples. The objective was to send low-risk transactions to a cheaper processor, and high-risk transactions to a more expensive processor with better fraud controls.

An overly complex specification is often a sign that the technical model is misaligned with the business model, or that the specification is described at the wrong level of abstraction. Even when correctly understood, such specifications lead to software that is hard to maintain, because small changes in the business environment can lead to disproportionately huge changes in the software.

For example, important business concepts such as transaction risk score, processor cost or fraud capabilities were not captured in the examples for payment routing. Because of this, small changes to the business rules required huge changes to a complex network of special cases in the software. Minor adjustments to risk thresholds led to a ton of unexpected consequences. When one of the processors with good fraud-control capabilities reduced prices, most of the examples had to change and the underlying functions were difficult to adjust. That means that the organisation couldn't take advantage of the new business opportunity quickly.

Instead of capturing complex scenarios, it is far better to focus on illustrating user stories with key examples. Key examples are a small number of relatively simple scenarios that are easy to understand, evaluate for completeness and critique. This doesn't mean throwing away precision – quite the opposite – it means finding the right level of abstraction and the right mental model that can describe a complex situation better.

The payment routing case could be broken down into several groups of smaller examples. One group would show transaction risk based on the country of residence and country of purchase. Another group of examples would describe how to score transactions based on payment amount and currency. Several more groups of examples would describe other transaction scoring rules, focused only on the relevant characteristics. One overall set of examples would describe how to combine different scores, regardless of how they were calculated. A final group of examples would describe how to match the risk score with compatible payment processors, based on processing cost and fraud capabilities. Each of these groups might have five to ten important examples. Individual groups would be much easier to understand. Taken together, these key examples would allow the team to describe the same set of rules much more precisely but with far fewer examples than before.

Key benefits

Several simple groups of key examples are much easier to understand and implement than a huge list of complex scenarios. Smaller groups make it easier to evaluate completeness and argue about boundary conditions, so they allow teams to discover and resolve inconsistencies and differences in understanding.

Breaking down complex examples into several smaller and focused groups leads to more modular software, which reduces future maintenance costs. If the transaction risk was modelled with examples of individual scoring rules, that would give a strong hint to the delivery team to capture those rules as separate functions. Changes to an individual scoring threshold would not impact all the other rules. This would avoid unexpected consequences when rules change. Changing the preferred processor when they reduce prices would require small localised changes instead of causing weeks of confusion.

Describing different aspects of a story with smaller and focused groups of key examples allows teams to divide work better. Two people can take the country-based scoring rules, two other people could implement routing based on final score. Smaller groups of examples also become a natural way of slicing the story – some more complex rules could be postponed for a future iteration, but a basic set of rules could be deployed in a week and provide some useful business value.

Finally, focusing on key examples significantly reduces the sheer volume of scenarios that need to be checked. Assuming that there are six or seven different scoring rules and that each has five key examples, the entire process can be described with roughly eighty thousand examples (five to the power of seven). Breaking it down into groups would allow us to describe the same concepts with forty or so examples (five times seven, plus a few overall examples to show that the rules are connected correctly). This significantly reduces the time required to describe and discuss the examples. It also makes the testing much faster, whether it was automated or manual. Clearer coverage of examples and models also provide a much better starting point for any further exploratory testing.

How to make it work

The most important thing to remember is that if the examples are too complex, your work on refining a story isn't complete. There are many good strategies for dealing with complexity. Here are four that we often use:

- Look for missing concepts
- Group by commonality and focus only on variations
- Split validation and processing
- Summarise and explore important boundaries

Overly complex examples, or too many examples, are often a sign that some important business concepts are not explicitly described. In the payment routing examples, transaction risk is implied but not explicitly described. Discovering these concepts allows teams to offer alternative models and break down both the specification and the overall story into more manageable chunks. We can use one set of examples to describe how to calculate the risk score, and another for how to use a score once it is calculated.

Avoid mixing validation and usage – this is a common way of hiding business concepts. For example, teams often use the same set of examples to describe how to process a transaction and all the ways to reject a transaction without processing (card number in incorrect format, invalid card type based on first set of digits, incomplete user information etc). The hidden business concept in that case is 'valid transaction'. Making this explicit would allow splitting a single large set of complex examples into two groups – determining whether a transaction is valid, and working with a valid transaction. These groups can then be broken down further based on structure.

Long lists of examples often contain groups that are similar in structure or have similar values. In the payment routing story, there were several pages of scenarios with card numbers and country of purchase, a cluster of examples involving two countries (residence and delivery, and some scenarios where the value of a transaction was important. Identifying commonalities in structure is often a valuable first step for discovering meaningful groups. Each group can then be restructured to show only the important differences between examples, reducing the cognitive load.

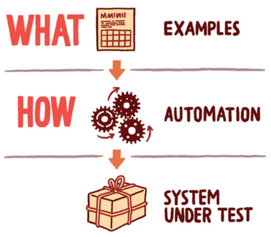
The fourth good strategy is to identify important boundary conditions and focus on them, ignoring examples that do not increase our understanding. For example, if 50 USD is the risk threshold for low-risk countries, and 25 USD for high-risk countries, then the important boundaries are:

- 24.99 USD from a high-risk country
- 25 USD from a high-risk country
- 25 USD from a low-risk country
- 49.99 USD from a low-risk country
- 50 USD from a low-risk country

A major problem causing overly complex examples is the misunderstanding that testing can somehow be completely replaced by a set of carefully chosen examples. For most situations we've seen, this is a false premise. Checking examples can be a good start, but there are still plenty of other types of tests that are useful to do.

Don't aim to fully replace testing with examples in user stories – aim to create a good shared understanding, and give people the context to do a good job. Five examples that are easy to understand and at the right level of abstraction are much more effective for this than hundreds of very complex test cases.

Describe what, not how



By far the most common mistake inexperienced teams make when describing acceptance criteria for a story is to mix the mechanics of test execution with the purpose of the test. They try to describe *what* they want to test and *how* something will be tested all at once, and get lost very quickly.

Here is a typical example of a description of how something is to be tested:

```
Scenario: basic scenario
Given the user Mike logs on
And the user clicks on "Deposit"
And the page reloads
Then the page is "Deposit"
And the user clicks on "10 USD"
And the page reloads
Then the page is "Card Payment"
When the user enters a valid card number
And the user clicks on "Submit"
And the payment is approved
And the page reloads
```

```
Then the page is "Account"
And the account field shows 10 USD
And the user clicks on "Find tickets"
And the user clicks on "Quick trip"
And the page reloads
Then the page is "Tickets"
And the price is 7 USD
And the user clicks on "Buy tickets"
Then the purchase is approved
And the page reloads
And a ticket confirmation number is displayed
And the account field shows 3 USD
```

This is a good test only in the sense that someone with half a brain can follow the steps mechanically and check whether the end result is 3 USD. It is not a particularly useful test, because it hides the purpose in all the clicks and reloads, and leaves the team with only one choice for validating the story. Even if only a tiny fraction of the code contains most of the risk for this scenario, it's impossible to narrow down the execution. Every time we need to run the test, it will have to involve the entire end-to-end application stack. Such tests unnecessarily slow down validation, make automation more expensive, make tests more difficult to maintain in the future, and generally just create a headache for everyone involved.

An even worse problem is that specifying acceptance criteria like this pretty much defeats the point of user stories – to have a useful conversation. This level of detail is too low to keep people interested in discussing the underlying assumptions.

Avoid describing the mechanics of test execution or implementation details with user stories. Don't describe *how* you will be testing something, keep the discussion focused on *what* you want to test instead. For example:

```
Scenario: pre-paid account purchases
Given a user with 10 USD in a pre-paid account
When the user attempts to buy a 7 USD ticket
Then the purchase is approved
And the user is left with 3 USD in the account
```

When most of the clutter is gone, it's easier to discuss more examples. For example, what if there is not enough money in the account?

Pre-paid balance	Ticket cost	Purchase status	Resulting balance
10 USD	7 USD	approved	3 USD
5 USD	7 USD	rejected	5 USD

This is where the really interesting part comes in. Once we remove the noise, it's easy to spot interesting boundaries and discuss them. For example, what if the pre-paid balance is 6.99 and someone wants to buy a 7 USD ticket?

As an experiment, go and talk to someone in sales about that case – most likely they'll tell you that you should take the customer's money. Talk to a developer, and most likely they'll tell you that the purchase should be rejected. Such discussions are impossible to have when the difficult decisions are hidden behind clicks and page loads.

Key benefits

It's much faster to discuss what needs to be done instead of how it will be tested in detail, so keeping the discussion on a higher level allows the team to go through more stories faster, or in more depth. This is particularly important for teams that have limited access to business sponsors, and need to use their time effectively.

Separately describing the purpose and the mechanics of a test makes it easier to use tests for communication and documentation. The next time a team needs to discuss purchase approval rules with business stakeholders, such tests will be a great help. Although the mechanics of testing will probably be irrelevant, a clear description of what the current system does will be an excellent start for the discussion. In particular it will help to remind the team of all the difficult business decisions that were made months ago while working on previous stories. An acceptance criterion that mixes clicks and page loads with business decisions is useless for this.

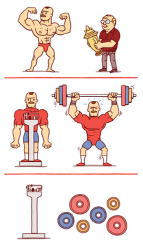
Decoupling *how* something will be tested from *what* is being tested significantly reduces future test maintenance costs. When a link on a web page becomes a button, or users are required to log in before selecting products, we only have to update the mechanics of testing. If the purpose and the mechanics are mixed together, it is impossible to identify what needs to change. That's the reason why so many teams suffer from record-and-replay test maintenance.

How to make it work

A good rule of thumb is to split the discussions on *how* and *what* into two separate meetings. Business sponsors are most likely not interested in the mechanics of testing, but they need to make decisions such as the $6.99 purchase. Engage decision-makers in whiteboard discussions on what needs to be tested, and postpone the discussion on how to test it for the delivery team later.

If you use a tool to capture specifications with examples, such as Cucumber, FitNesse or Concordion, keep the human-readable level focused on what needs to be tested, and keep the automation level focused on how you're checking the examples. If you use a different tool, then clearly divide the purpose of the test and the mechanics of execution into different layers.

Separate decisions, workflows and technical interactions



Any good test automation book will suggest that user interface interactions need to be minimised or completely avoided. However, there are legitimate cases where the user interface is the only thing that can actually execute a relevant test. A common example is where the architecture dictates that most of the business logic sits in the user interface layer (such applications are often called 'legacy' even by people who write them, but they are still being written). Another common situation is when an opaque, third-party component drives an important business process, but has no sensible automation hooks built into it. In such cases, teams often resort to record-and-replay tools with horrible unmaintainable scripts. They create tests that are so difficult to control and so expensive to maintain that it's only possible to afford to check a very small subset of interesting scenarios. Teams in such situations often completely give up on any kind of automation after a while.

There are two key problems with such tests. One is that they are slow, as they often require a full application stack to execute. The other is that they are extremely brittle. Small user interface changes, such as moving a button on the screen somewhere else, or changing it to a hyperlink, break all the tests that use that element. Changes in the application workflow, such as requiring people to be logged in to see some previously public information, or introducing a back-end authorisation requirement for an action, pretty much break all the tests instantly.

There might not be anything we can do to make such tests run as fast as the ones below the user interface, but there are definitely some nice tricks that can significantly reduce the cost of maintenance of such tests, enough to make large test suites manageable. One of the most important ideas is to apply a three-layer approach to automation: divide business-oriented decisions, workflows and technical interactions into separate layers. Then ensure that all business decision tests reuse the same workflow components, and ensure that workflow components share technical interactions related to common user interface elements.

We've used this approach with many clients, from financial trading companies working with thick-client administrative applications, to companies developing consumer-facing websites. It might not be a silver bullet for all possible UI automation situations, but it comes pretty close to that, and deserves at least to be the starting point for discussions.

Key benefits

A major benefit of the three-layer approach, compared to record-and-replay tests, is much easier maintenance. Changes are localised. If a button suddenly becomes a hyperlink, all that needs to change is one technical activity. Workflows depending on that button continue to work. If a workflow gets a new step, or loses one, the only thing that needs to change is the workflow component. All technical activities stay untouched, as do any business rule specifications that use the workflow. Finally, because workflows are reused to check business decisions, it's easy to add more business checks.

The three-layer design pattern is inspired by similar ideas from the popular page object pattern, but instead of tying business tests too tightly to current web page structures, it decouples all common types of change. Tests automated using page objects are easily broken by workflow changes that require modifications to transitions between pages or affect the order of interactions. Because of this, the three-layer approach is better for applications with non-trivial workflows.

Applications with a lot of messy user interface logic often need a good set of integration tests as well as business checks. Another big benefit of the three-layer approach is that the bottom layer, technical interactions, can be easily reused for technical integration tests. This reduces the overall cost of test maintenance even further, and allows the delivery team to automate new tests more easily.

How to make it work

Most test automation tools work with one or two layers of information. Tools such as FitNesse, Concordion or Cucumber provide two layers: the business specification and the automation code. Developer-oriented tools such as Selenium RC and unit-testing tools tend to offer only one layer, the automation code. So do tester-oriented tools. This misleads many teams into flattening their layer hierarchy too soon. Automation layers for most of these tools are written using standard programming languages, which allow for abstractions and layering. For example, using Concordion, the top-level (human readable specification) can be reserved for the business-decision layer, and the automation code below can be structured to utilise workflow components, which in turn utilise technical activity components.

Some tools, such as Cucumber, allow some basic reuse and abstraction in the test specification (top level) as well. This theoretically makes it possible to use the bottom automation layer only for technical interactions, and push the top two layers into the business-readable part of the stack. Unless your team has a great many more testers than developers, it's best to avoid doing this. In effect, people will end up programming in plain text, without any support from modern development tool capabilities such as automated refactoring, contextual search and compilation checks.

Decouple coverage from purpose

Because people mix up terminology from several currently popular processes and trends in the industry, many teams confuse the purpose of a test with its area of coverage. As a result, people often write tests that are slower than they need to be, more difficult to maintain, and often report failures at a much broader level than they need to.

For example, integration tests are often equated with end-to-end testing. In order to check if a service component is talking to the database layer correctly, teams often write monstrous end-to-end tests requiring a dedicated environment, executing workflows that involve many other components. But because such tests are very broad and slow, in order to keep execution time relatively short, teams can afford to exercise only a subset of various communication scenarios between the two components they are really interested in. Instead, it would be much more effective to check the integration of the two components by writing more focused tests. Such tests would directly exercise only the communication scenarios between the two interesting areas of the system, without the rest.

Another classic example of this confusion is equating unit tests with technical checks. This leads to business-oriented checks being executed at a much broader level than they need to be. For example, a team we worked with insisted on running transaction tax calculation tests through their user interface, although the entire tax calculation functionality was localised to a single unit of code. They were misled by thinking about unit tests as developer-oriented technical tests, and tax calculation clearly fell outside of that. Given that most of the risk for wrong tax calculations was in a single Java function, decoupling coverage (unit) from purpose (business test) enabled them to realise that a business-oriented unit test would do the job much better.

A third common way of confusing coverage and purpose is thinking that acceptance tests need to be executed at a service or API layer. This is mostly driven by a misunderstanding of Mike Cohn's test automation pyramid. In 2009, Cohn wrote an article titled *The Forgotten Layer of the Test Automation Pyramid*, pointing out the distinction between user interface tests, service-level and unit tests. Search for 'test automation pyramid' on Google Images, and you'll find plenty of examples where the middle tier is no longer about API-level tests, but about acceptance tests (the top and bottom are still GUI and unit). Some variants introduce additional levels, such as workflow tests, further confusing the whole picture.

To add insult to injury, many teams try to clearly separate unit tests from what they call 'functional tests' that need different tools. This makes teams avoid unit-testing tools for functional testing, instead introducing horrible monstrosities that run slowly, require record-and-replay test design and are generally automated with bespoke scripting languages that are quite primitive compared to any modern programming tool.

To avoid this pitfall, make the effort to consider an area of coverage separately from the purpose of a test. Then you're free to combine them. For example, you can have business-oriented unit tests, or technical end-to-end checks.

Key benefits

Thinking about coverage and purpose as two separate dimensions helps teams reduce duplication between different groups of tests, and leads to more focused, faster automation. In addition to speeding up feedback, such focused tests are less brittle, so they will cause fewer false alarms. By speeding up individual test execution, teams can then afford to execute more tests and run them more frequently.

By thinking about technical tests separately from whether they are unit-level, component level or end-to-end tests, teams can also make better decisions on how and where to automate such tests. This often leads to technical tests being written with tools that developers are already familiar with, and helps teams maintain automated tests more easily.

How to make it work

Decide on purpose first, and let the purpose drive the choice of the format in which you capture the test. Business-oriented tests should be written in a language and format that allows teams to discuss potential problems with business domain experts. Technical checks can be written with a technical tool.

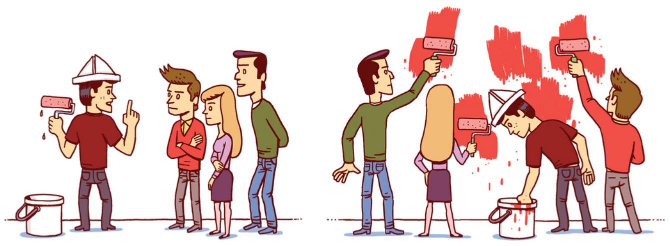
Once you've decided on the format and the purpose, think about the minimal area of coverage that would serve the purpose for that particular test. This will mostly be driven by the design of the underlying system. Don't force a test to execute through the user interface just because it's business oriented. If the entire risk for tax calculation is in a single unit of code, by all means write a unit test for it. If the risk is mostly in communication between two components, write a small, focused integration test involving those two areas only.

It’s perfectly fine to use tools commonly known as acceptance testing frameworks for writing business-oriented unit tests. They will run faster and be more focused.

Likewise, it’s perfectly fine to use tools commonly known as unit testing frameworks for more than just unit tests, as long as such groups of tests are clearly separated so they can be managed and executed individually. If the programmers on the team already know how to use JUnit, for example, it’s best to write technical integration tests with this tool, and just execute them with a separate task. In this case, the team can leverage their existing knowledge of a tool for a slightly different purpose.

Beware though of mixing up tests with different areas of coverage, because it becomes impossible to run individual groups in isolation. For example, split out tests into separate libraries so you can run true unit tests in isolation.

Make developers responsible for checking



Many organisations start with test automation as an auxiliary activity – it needs to be done, but without interrupting the development schedule. This often leads to test automation specialists working after development, or even entire teams of people charged with making testing faster and cheaper. This is a false economic premise, and can lead to a lot of trouble later on.

By decoupling development and test automation, teams either introduce a lot of duplicated work or unnecessarily delay feedback. Manually running all the tests during development is rarely sustainable, so development can officially finish without any serious testing. If a developer receives feedback about potential problems only after a different team automates tests, the code that needs to be fixed might have been modified by some other people meanwhile. This introduces a further delay, because developers need to coordinate more and research other potential changes to fix problems.

In addition, when specialists are hired to automate tests, they are often overwhelmed by work. Ten developers can produce a lot more code than a single person can test, so specialist automation often introduces a bottleneck. The delivery pipeline slows down to the speed of test automation, or software gets shipped without completed testing. The first scenario is horrible because the organisation loses the ability to ship software quickly. The second scenario is horrible because developers stop caring about testing, and automated tests then just come to seem like a waste of time and money. Developers do not design the system to be testable, and it becomes even more difficult to automate tests, causing more delay between development and testing. It's a lose-lose situation.

Separate automation specialists rarely have the insight into system internals, so the only option for them is to automate tests end-to-end. Such tests will be unnecessarily slow and brittle, and take a lot of time to maintain. Slow, difficult tests bolster the argument for not disrupting the critical delivery path with tests.

Test automation specialists often use tools that developers are not familiar with, so it is not easy for them to ask for help from the rest of the team. Any potential test automation problems have to be investigated by test automation experts, which creates a further bottleneck. It's a vicious circle where testing only gets further separated from delivery, creating more problems.

The only economically sustainable way of writing and automating hundreds of tests is to make developers responsible for doing it. Avoid using specialist groups and test automation experts. Give people who implement functionality the responsibility to execute tests, and ensure they have the necessary information to do it properly.

Key benefits

When the same people are responsible for implementing and changing code and automating the related tests, tests are generally automated a lot more reliably and execute much faster. Programmers have insight into system internals, they can use lots of different automation hooks, and they can design and automate tests depending on the real area of risk, not just on an end-to-end basis. This also means that developers can use the tools they like and are familiar with, so any potential problem investigations can be delegated to a larger group of people.

In addition, when developers are responsible for automation, they will design the system to make it easy to control and observe functionality in the first place. They will build modules that can be tested in isolation, and decouple them so tests can run quickly. This brings the benefit of faster testing, but a modular design also makes it easier to evolve the system and implement future requests for change.

When developers are responsible for test automation, the tests will deliver fast feedback. The time between introducing a problem and spotting it is significantly shorter, and the risk of someone else modifying the underlying software meanwhile is pretty much eliminated.

These three factors significantly change the economics of test automation. Tests run faster, cheaper, they are more reliable, and the system is more modular so it's easier to write tests. There is no artificial bottleneck later in testing, and no need to choose between higher quality and faster deployment.

How to make it work

A common argument against letting developers automate tests is to ensure independent feedback and avoid tunnel vision. The right way to counter this is to ensure that the right people are involved in designing the tests. Developers should be responsible for automating the tests, but the entire team (including business stakeholders and testers) should be involved in deciding what needs to be tested.

Teams without test automation experience should not hire automation experts to take on the work. External experts should only be hired to teach developers how to use a particular tool for automation, and provide advice on how best to design tests.

Teams with a high risk of automation being done wrongly can further reduce the risk by pairing up testers and developers during automation work, and by running some quick exploratory tests to investigate the automation code.

The End

This book is part of a series of books on improving various aspects of iterative delivery. If you like it, check out the other books from the series at 50quickideas.com.

Authors

Gojko Adzic is a strategic software delivery consultant who works with ambitious teams to improve the quality of their software products and processes. Gojko won the 2012 Jolt Award for the best book, was voted by peers as the most influential agile testing professional in 2011, and his blog won the UK Agile Award for the best online publication in 2010. To get in touch, write to gojko@neuri.com or visit gojko.net

David Evans is a consultant, coach and trainer specialising in the field of Agile Quality. David helps organisations with strategic process improvement and coaches teams on effective agile practice. He is regularly in demand as a conference speaker and has had several articles published in international journals. Contact David at david.evans@neuri.com or follow him on Twitter @DavidEvans66

Tom Roden is a delivery coach, consultant and quality enthusiast, helping teams and people make the improvements needed to thrive and adapt to the ever changing demands of their environment. Tom specialises in agile coaching, testing and transformation. Contact Tom at tom.roden@neuri.com or follow him on Twitter @tommroden.

Bibliography and resources

- Fifty Quick Ideas To Improve Your User Stories by Gojko Adzic and David Evans, ISBN 978-0993088100, Neuri Consulting 2014
- Domain-Driven Design: Tackling Complexity in the Heart of Software by Eric Evans, ISBN 978-0321125217, Addison-Wesley Professional 2003
- How Google Tests Software by James A. Whittaker, Jason Arbon and Jeff Carollo, ISBN 978-0321803023, Addison-Wesley Professional 2012
- More Agile Testing: Learning Journeys for the Whole Team by Lisa Crispin and Janet Gregory, ISBN 978-0321967053, 978-0321967053
- Lessons Learned in Software Testing: A Context-Driven Approach by by Cem Kaner, James Bach and Bret Pettichord, ISBN 978-0471081128, Wiley 2001
- Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing by Elisabeth Hendrickson, ISBN 978-1937785024, Pragmatic Bookshelf 2013
- A Practitioner's Guide to Software Test Design by Lee Copeland, ISBN 978-1580537919, Artech House 2004
- The Checklist Manifesto: How to Get Things Right by Atul Gawande, ISBN 978-0312430009, Picador 2011
- Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems by Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm, University of Toronto, from 11th USENIX Symposium on Operating Systems Design and Implementation, ISBN 978-1-931971-16-4, USENIX Association 2014

- User Story Mapping: Discover the Whole Story, Build the Right Product by Jeff Patton, ISBN 978-1491904909, O'Reilly Media 2014

Useful web sites

Access these links quickly at http://www.50quickideas.com

- Fifty Quick Ideas discussion group
- QUPER web site
- Chaos Monkey Released Into The Wild, by Ariel Tseitlin 2012
- Improving Testing Practices at Google, a conference report on Mark Striebeck's presentation at XPDay 2009 by Gojko Adzic
- The Forgotten Layer of the Test Automation Pyramid by Mike Cohn, 2009

Legal Stuff

Title: Fifty Quick Ideas to Improve Your Tests
Print ISBN: 978-0-9930881-1-7
Published On: 15. May 2015.
Copyright (c) Neuri Consulting LLP
Authors: Gojko Adzic, David Evans and Tom Roden
Copy-editor: Marjory Bisset
Design and layout: Nikola Korac

Published by:

Neuri Consulting LLP
25 Southampton Buildings
London WC2A1AL
United Kingdom