



12 Secrets of Effective Test Management

By Paul Gerrard



Every software development project includes testing activities. These activities may be managed formally or informally, by a designated QA manager or by various individuals on a development team. Managing them properly is critical if you want to have the best chance of verifying that your application works as intended.

If you're involved in software development in any way (but especially if you're a tester) this ebook is for you. Inside, you'll find a range of essential test management topics including process, modeling, risk, delivery, reporting, and tools. Each section includes relevant stories from the author's experience, and concludes with some ideas to help you apply the concepts presented.

Contents

1. Understand Testing Goals and Stakeholders	4
2. Define Your Test Strategy	12
3. Use Models	20
4. Focus On Risk	34
5. Decide How Much Testing is Enough	45
6. Create Useful Documentation	54
7. Plan as a Journey, Not a Task	66
8. Execute Your Plan	79
9. Test as a Team	91
10. Test Performance	103
11. Have the Right Tools and Infrastructure	118
12. Develop Yourself	138

Introduction

It is helpful to think of testing as less of a role and more of an activity that people do. Everyone tests, but some people specialize and make a career of it. In the same way, test management is an activity associated with testing. Whether you are the sole tester on a team or oversee the testing for a massive project, you have test management activities.

For better or worse, many companies have decided that the role of a test manager is no longer essential. Responsibility for testing in a larger project is distributed to smaller, Agile teams. There might be only one tester on the team, while the developers on the team take more responsibility for testing and run their own unit tests. Perhaps the members of the team jointly fulfill the role of a virtual test manager.

Even if there is not a dedicated test manager, some critical test management activities still need to be done. This includes more mundane activities such as keeping good records of tests planned and/or executed. But it also includes more strategic activities such as developing a strategic plan for test coverage.

This ebook describes the secrets of essential test management activities and provides insights into how you can apply them in your own situation, or “context”. Your context might be as the sole tester on a team or as the leader of an army of testers. Your development team might be following an agile approach or a waterfall approach. Your testing might be entirely manual, or might be assisted by automation.

For that reason, the information in this ebook is presented in a context-neutral manner. But I will tell you stories from my (and others’) experience and identify thinking processes that will help you adapt these secrets for your own context.



12 Secrets of Effective Test Management

1. Understand Your Goals and Stakeholders

If you are the test manager on a project, it is very likely that people will assume you are the expert on all things testing-related. Other team members will have their own views on testing. Some may have more experience than you. Expectations of testing are often unrealistic. There may be people who doubt your competence, value to the team, or even your motivations. It can be tough.

In your career, you will have to adapt to new and changing circumstances. You will encounter business and senior project stakeholders. You will be joining teams of various sizes with different people in them who have widely varying backgrounds. Your role might be as the sole tester. Or you may be running a large team, providing oversight to the testing in a project, advising an Agile team, or figuring out how testing fits into a continuous delivery process.

Before we start using terms like **testing**, **stakeholder** and **stakeholder goals**, let's be clear what we're talking about.

Definition of "Test"

The word test is used as a noun and as a verb. It's about testing as an activity and the outcome of that activity. It's about the people or organizations who commission that activity and those who use the results. Very much it's about the people who call themselves testers and the complex systems on which we work.

To talk about context-neutral testing we'll need a definition of the word "test" that is context-neutral. I find the definition from the [American Heritage Dictionary](#) to be the most appropriate:



Test: (noun) a procedure for critical evaluation; a means of determining the presence, quality, or truth of something; a trial

There is not just one definition; rather, there are three variations. Well, this isn't so bad I think, as all three taken together give us the foundations we need. Let's take a closer look at each one.



A procedure for critical evaluation

Critical evaluation involves a skillful judgment as to the truth or merit of something. A test is a procedure, usually with a series of steps that include some form of preparation, execution, results gathering, analysis, and interpretation. This isn't a definitive description of a test procedure. There could be more steps and one could break these main steps down further.

The procedure doesn't necessarily require prepared documentation, but some tests are documented. Automated tests are always scripted in some way. The important thing is that there is a clear thought process at the heart of a test. This thought process is driven by the need to evaluate the system under test with respect to its adequacy, consistency, behavior, accuracy, reliability or any other significant aspect or property.



A means of determining the presence, quality, or truth of something

A test could determine the presence (or absence) of something easily enough, but quality is a different matter: the term is loaded with emotional connotations, but we are rescued by the same dictionary. Quality can be, "an essential or distinctive characteristic, property, or attribute". Now we can see that a test can reveal these properties.

Note that I'm not using the term quality to reflect the relationship between a user or stakeholder and a product. Quality is like beauty - in the eye of the user. Don't be drawn into discussions of how one measures quality (with testing).

Can a test determine the truth of something? Well, this makes good sense too. Typically, we need to test an assertion such as, "this system meets some requirement" or "this system behaves in such a way" or "this system is acceptable" and so on. There's a certain amount of subjective judgment involved but we can see that a test or tests could provide evidence for someone to exercise that judgment and make a decision.



A trial

The notion of a trial implies that the process of testing a system will help us to evaluate that system with respect to its qualities. The purpose of such an evaluation is normally to make a decision.

The decision might be to accept or reject the system, but it might also be to expose its shortcomings so they can be remedied in some way. A test might also influence an individual or organization to change direction – to rethink a design; to relax or change a requirement; to scrap a component and start again; to buy rather than build or build rather than buy.

A natural way of looking at a system under test is that it is on trial, and will be judged in some way.

Definition of Testing

From our definition of the noun test, we can derive a verb easily enough.



Test: (verb) to critically evaluate; to determine the presence, quality, or truth of something; to conduct a trial.

So far, so good. But not that good, really. Unfortunately, the testing profession is dogged with terminological problems. There's no way we can present a definite glossary here. In your projects to avoid misunderstandings, I suggest you ask what the goal of any defined test type is rather than rely on an assumed definition to tell you.

Testing Needs Stakeholders



What is a stakeholder? A tester's stakeholders are those people who have an interest in the outcome of tests, and the information that testers provide.

They could be the individuals or organizations who:

- Are funding the project.
- Are affected (in a beneficial or detrimental way) by the new or changed system.
- Contribute to the project in some way, such as designers or developers.
- Need to make key decisions to keep the project moving forward, such as project managers or product owners.
- Are custodians of the new system, such as asset managers or service management staff.
- Are or represent the user community, such as business users, customers, partners or society as a whole.
- If you are a designer or developer, testing your own work, you are your own stakeholder. You need to think about what you need from your own tests.

Know Your Stakeholders



If we are responsible for defining a test strategy, we need to identify and engage the people or organizations that will use and benefit from the evidence testing provides. If we don't do this, how can we answer the following questions?

- What should and should not be tested?
- What are the most important and least important things to test?
- Who will use the evidence we generate from testing?
- What information do they need?
- How will we squeeze all this into the time available?

If there is no stakeholder, no one will take any notice of the outcome of the testing, so there is little point in doing any testing. We won't have a mandate or any authority for testing. We won't get the resources or time we need. We can't report test execution passes or failures or ask questions with any expectation of anyone answering them.

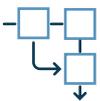
Who are the stakeholders and what do they want? There are four types of stakeholders based on their needs:



Sponsors: these stakeholders need evidence that the system can support their business goals and that the risk of failure is acceptable.



Users: these stakeholders need evidence that the system will 'work' for them.



Project Management: these stakeholders need to know the status of deliverables (i.e. application features): are they available, acceptable, and reasonably free of defects? If there is a problem, the manager may need to re-plan or at least manage expectations.



Designers and developers: these stakeholders need to know where their products fail – so they can fix defects or adjust their designs.

Stakeholders must articulate what evidence they need the testers to produce. This will start with the business goals and the product risks of most concern. The testers need to know what information is required from the test process. Sometimes, this will take the form of documentation, but in smaller projects, this information is often managed through shared wikis, Kanban or Scrum boards or ChatOps facilities.

Team communications may focus on these shared resources, visible to all. Exactly how the evidence is provided needs to be agreed with stakeholders in your project.

Testers are accountable to stakeholders: you need to engage, consult, refer to and defer to stakeholders. Our stakeholders are our customers, directors, and supporters. Our test approach should focus our energy and the testing effort towards meeting their needs at all times. Test managers and testers should be *stakeholder-obsessed*.

What If My Stakeholders Aren't Interested?

Here's a story from a project I worked on a few years ago:



I was hired to help a start-up business define and implement a test strategy. The start-up was set up by a parent company in the finance industry to operate a small, but lucrative financial service. The start-up had a CEO, a project manager and a handful of staff tasked with building and operating a new system. Everything was outsourced. One supplier would build the software system, another would host the system, and we would test the system.

I set up a risk workshop to explore the concerns of the various suppliers and stakeholders. The CEO was not invited, but he was the first to arrive at the meeting. Surprised, I asked him why he came – he was very welcome – but not on the invite list. He said, "I saw the agenda and thought it would be negligent of me not to attend as the topics are so important."

Senior stakeholders may not be interested in the technical detail of testing, but their personal performance is usually measured by the level of achievement of their goals in the context of systematic risk management. If business goals and risk are at stake, senior stakeholders will be interested – their job may be on the line.

Putting It Into Practice

Answer the following questions for your current project or a past project. It might help if you organize the answers into a table.



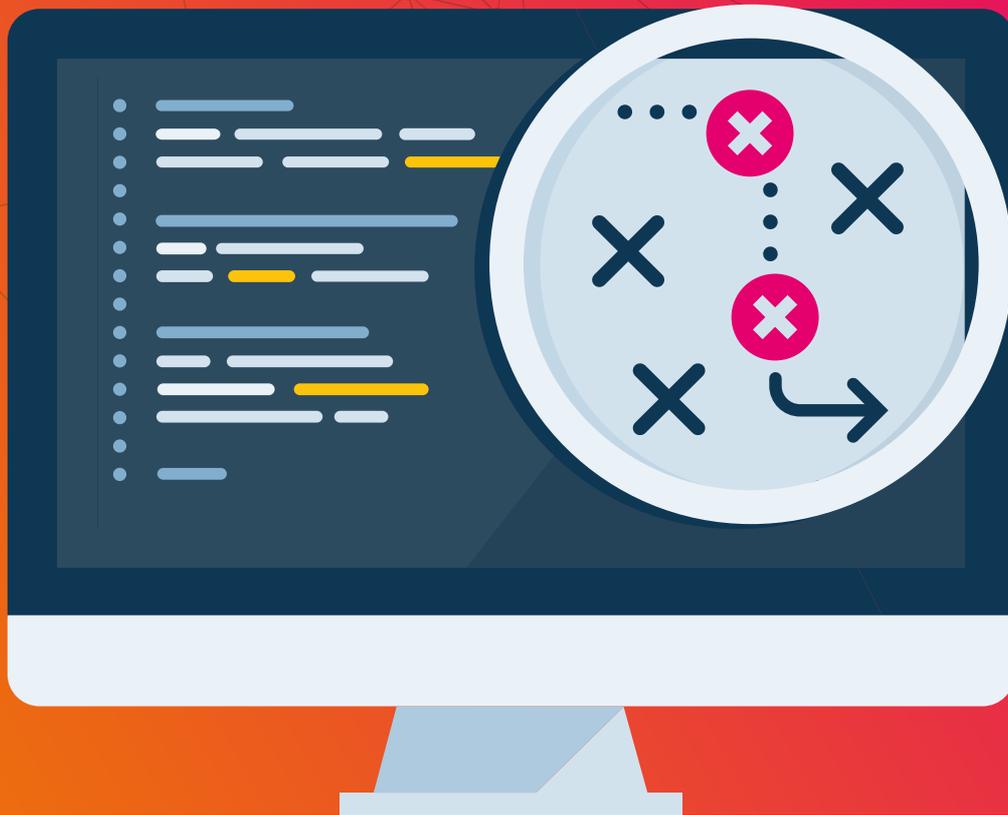
- Who are your testing stakeholders?
- What are their goals, with respect to delivering the system and its benefits?
- What are their concerns (risks of failure, delay, performance, insecurity, etc.)?
- What decisions do they need to make?
- What information, from testing, do they need to make those decisions?

The answers to these questions provide the top-level framework for your test strategy. If stakeholders see how the strategy relates to their goals and concerns you have a far better chance of them understanding and supporting it.

If you experience problems with stakeholders or managers not supporting your test approach, consider conducting a stakeholder analysis. Problems that you might encounter include:



- Stakeholders never read your test strategy, and so don't support it at a critical moment.
- Your time or resource estimates are ignored.
- People question why so much testing is required,
- Testing is squeezed because development slips.
- Decisions to go-live are made with insufficient testing.



12 Secrets of Effective Test Management

2. Define Your Test Strategy

Process, not Ceremony

A process is “a series of actions or steps taken in order to achieve a particular end.” If you ask an individual or organization how they achieved something, their story of what they did is their process, or at least, an example of their process in use. Process is really only a description of how things get done.

In the last thirty years or so, process has acquired a tarnished reputation. When structured methods in software came into being in the late 1970s, process people took what were intended to be a set of flexible, hand-written modeling tools and formalized them into large-scale software development methodologies and CASE products. These more or less supported large-scale system development, but were taken to be universal by their advocates and the market.

Trusted to excess, high process approaches got bogged down in formality, documentation and meaningless ceremony. These approaches simply did not work in smaller, more dynamic environments. The Agile movement was a welcome reaction to wasteful bureaucracy. The [Agile manifesto](#) set out a series of principles which put emphasis on working software over documentation, collaboration over contracts, responding to change over following a plan and so on.

With the emergence of continuous delivery and DevOps approaches, we now have a third way. Continuous delivery takes the cadence of software development to extremes. Development work is broken into extremely small packages which are developed, tested and deployed in days, hours, or possibly minutes. This depends on high levels of automation of environment provisioning, testing, deployment to production and post-deployment monitoring (among other things).

Although the continuous approach is gaining popularity, for most organizations some form of hybrid of the three approaches is the norm. Pure waterfall, Agile, or continuous is less common than their advocates would have you believe.

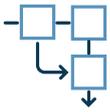
The challenge for testers is, given the possibly infinite range of development processes, how can a generic approach to defining a test process be formulated?

Separate Thinking from Logistics

As you encounter new project situations, what you do to achieve your testing goal will vary. The only way to offer generic advice in this respect is to separate out the things that are context-neutral from those that are likely to be context-specific. The test approach you use and almost every activity in that approach varies with the context. What does not change is the thought process you go through to define that approach.



Thinking: The definition of your test process may vary, but the questions you need to ask and the decisions you need to make don't. When you explore your situation, of course, the answers to your questions will vary. But the objective – meeting your testing goal – is generic. We'll explore the questions you should ask in order to formulate your test strategy.



Logistics: You might decide that testing will take place in stages, with each stage having different objectives, techniques, and responsibilities for test activities. You must also decide whether and how you automate some or all of the testing. Perhaps there are no stages, just a rapid series of activities (possibly automated) that must be completed within a project sprint. These are the practical, logistical choices you must make.

In the *Tester's Pocketbook*, I suggest there are a set of principles, or **axioms** that are universally applicable in all software projects (albeit to varying degrees). Each axiom has a series of questions, and the answers to these questions support the decisions that you make in testing. These fundamental questions are not definitive. Rather they are offered as a starting set for you to expand on as you explore your project situation in increasing detail. The answers to these initial questions lead to secondary and tertiary questions and their answers lead you to your context-specific conclusions.

Test Strategy

If you look up strategy in the dictionary, you tend to see definitions that relate to military battles – which isn't terribly useful. But there are some statements that can be made that set out a framework for how you define a test strategy.

First, your strategy is not simply a document. Your strategy is a result of exploration, thinking, and collaboration. A strategy seeks to define the process you will use to achieve your testing goals. It could be a brief set of guidelines that your team follow. It could be a document of 20 to 2,000 pages. The goal isn't a document, it's the thinking.

“Planning is everything. The plan is nothing.”
– Dwight D. Eisenhower, of the D-Day preparations.

Second, before you can plan a test, you usually need many questions answered and decisions made. Some can be answered now, others will have to wait. A strategy, therefore:

- Presents some decisions that can be made ahead of time.
- Defines the process or method or information that will allow decisions to be made during the project.
- Sets out the principles or process to follow for uncertain situations or unplanned events.

So, a strategy attempts to answer as many questions as possible ahead of time. Why bother doing this? Surely, we can address problems in testing as we encounter them?

Well, by raising these questions early, and getting people to think about the consequences, huge difficulties might be avoided, or at least mitigated, before they threaten the success of your project.



I worked for a UK regional electricity company (REC) on a project to digitize the electricity network so it could be displayed on graphical workstations, rather than wall-mounted boards. There were several of these 20 feet wide, 8 feet high boards in control rooms in different buildings. Each had plugs, patches,

switches and lights to simulate/show the live (and safe) routes through the 144,000-volt regional electricity network. The boards represented the state of the switches and network helping controllers to plan and manage maintenance activity. The safety of engineers on site depended on the accuracy of these boards.

I asked in a project meeting when/where the test environment would be ready. The plan implied we were to get access to a control room on a certain date. The control board was to be ripped out with a local network, a set of servers and workstations installed. Someone, somewhere, had assumed the live control board could be scrapped before the system that would replace it had been tested. Awkward.

Test Strategy Framework

Below are the most important starting points for your information gathering. The questions are divided into three topic areas, but you could ask more questions and/or structure the questions differently.



Stakeholder Objectives

- **Stakeholders:** See section 1 for questions regarding stakeholders.
- **Goal and risk management:** How will risks be identified? Who assesses them? Who approves the test approach?
- **Decisions:** What decisions must stakeholders make, and how will they be made?
- **Confidence:** How will the test results give stakeholders confidence in the testing?
- **Assessment:** How will the quality/thoroughness of the testing be assessed?
- **Scope:** How will scope be defined?



Design approach

- **Sources of knowledge:** What/where/who are the knowledge sources to scope and specify tests?
- **Sources of uncertainty:** What causes uncertainty in our sources of knowledge?
- **Models:** How will test models be derived? How will they relate to stakeholders?
- **Prioritization approach:** Under time pressure, how will priorities be assigned to tests?



Delivery approach

- **Test sequencing:** How will the sequence of tests be determined?
- **Retesting:** What is the policy for retesting? For regression testing?
- **Environment requirements:** Who provides test environments? How will they be delivered, controlled, and managed?
- **Information delivery approach:** How will the test execution team deliver results to the stakeholders?
- **Incident management approach:** How will incidents be managed?
- **End-game approach:** How will the test process end? (How) will outstanding bugs be fixed and retested?

In the table above, there is no mention of the planning process. This could be defined in the strategy or not. At any rate, we'll look at test planning later in this ebook.

Shift-Left

Shift-Left can mean developers take more ownership and responsibility for their own testing; it can also mean testers get involved earlier, challenge requirements, and feed examples through a Behavior-Driven Development (BDD) process to developers. It can mean users and BAs together with developers take full responsibility for testing, and it can mean no test team and no testers. We have seen all configurations and there is no “one true way.”



Sources of knowledge that provide direction for the design and development of software should be challenged and/or tested.

In a staged project, this might involve formal reviews. In an Agile project, the tester (or developer or BA or user) can suggest scenarios or examples that challenge the author of a requirement or story to think through concrete examples and discussion before any code is written.



Shift-Left is mostly about bringing the thinking about testing earlier in the process.

Shift-Left implies that, whenever possible, you should provide feedback that will help the team to understand, challenge, or improve goals, requirements, design or implementation. This behavior comes as second nature to many, but not all, testers. Users, BAs, developers, and the entire team should be ready to both provide and accept feedback in this way. There might be resistance, but the overall aim is to run a better, more informed project.

As a tester, you need to get involved as early as possible by engaging in the discussion. Be ready to collaborate on ideas, requirements, and every stage where the outcome of that stage has a bearing on the value of the final deliverable of the project. Put simply, a tester challenges sources of knowledge, whether these sources are stakeholders, users, developers, business stories, documents, or “received wisdom.”

Whether you have a Shift-Left approach or not, your test strategy should encourage and align with these principles.

Putting It Into Practice

In the previous section, you thought about the stakeholders for a recent or current project. For the same project, think about these questions:



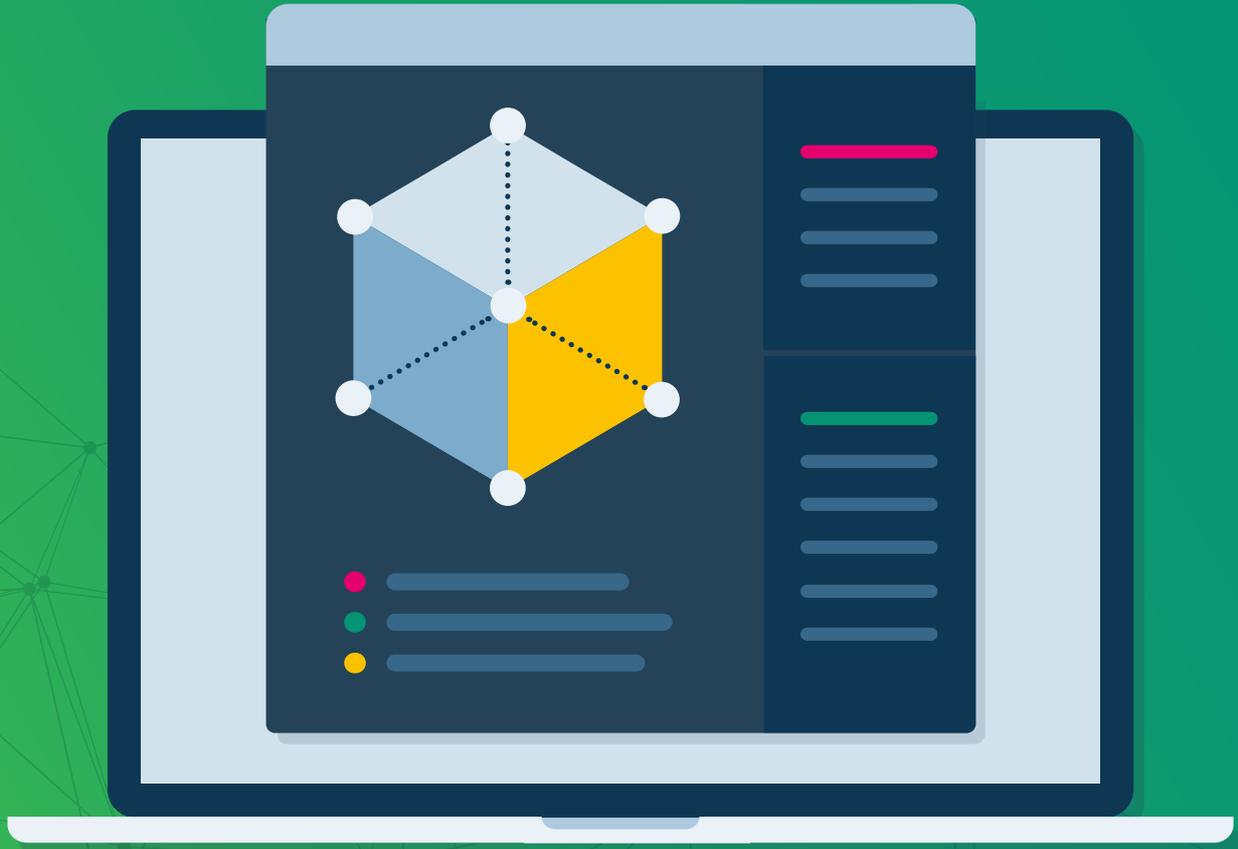
- What information did stakeholders need to make their project transition/release/acceptance decisions?
- How did that affect the way you reported test outcomes?
- How did you define/agree on the scope of testing?
- How did you decide how much testing was enough?
- How did you prioritize tests?

Whatever development approach you follow (waterfall, Agile, continuous, etc.), what criteria would you use to decide:



- When and where required tests should be run?
- Whether they are executed manually or through automation?

Do you think these criteria change depending on the development approach?



12 Secrets of Effective Test Management

3. Use Models

Models Are at the Heart of Testing

Test design is the process by which we select – from the infinite number possible – the tests that we believe will be most valuable to us and our stakeholders. Our test model helps us to select tests in a systematic way. Test models are fundamental to testing. This section discusses their importance and use to testers and test managers.

“Testing is a process in which we create mental models of the environment, the program, human nature, and the tests themselves. Each model is used until we accept the behaviour is correct or until the model is no longer sufficient ...”

– Boris Beizer, Software Test Techniques, 1990

The way testers use models is this:

- 1 We identify and explore sources of knowledge to build test models.
- 2 We use these models to challenge and validate our sources, improving our sources and our models.
- 3 We use these models to inform development and testing.

I use the term **exploration** to mean the uncovering of knowledge about the system to be tested from **sources of knowledge**. But what are the sources of knowledge? Given a mission for testing, our first task is to identify these sources. These sources of knowledge might be:

- **Documentation:** specifications, designs, requirements, standards, guidelines, etc.
- **People:** Stakeholders, users, analysts, designers, developers, and others.
- **Experience:** your own knowledge and experience of similar (or dissimilar systems), your preferences, prejudices, guesses, hunches, beliefs, and biases.
- **New System:** the system under test, if it exists, is available, and is accessible.
- **Old System:** the system to be replaced is obviously a source – it may, in some areas of functionality provide an oracle for expected behavior.

All our sources of knowledge are fallible and incomplete, and so are our models. Testers use experience, skill, and judgment to sift through these sources, to compare and contrast them, to challenge them, and to arrive at a consensus.

"All models are wrong, some are useful." – George Box, Economist.

What is a Test Model?

A test model might be a checklist or set of criteria. It could be a diagram derived from a design document or an analysis of narrative text. Many test models are never committed to paper. They can be mental models constructed specifically to guide testers while they explore the system under test.

We use models all the time. For example, a Gantt chart is a model of the dependencies and timings of activities in a project. Weather forecasts are based on mathematical models. A London underground map is a model. Use cases, user stories, decision tables, state-collaboration-and sequence-diagrams, and many others are all examples of models used to describe the functionality of programs or systems.

Models simplify complex situations by omitting detail that is not relevant at the moment. We use models to simplify a problem such as selecting things to test, for example. The model informs our thinking and we select things to test by identifying occurrences of some key aspect of the model.

We might select branches in a flowchart or control-flow diagram; state-transitions in a state model; boundaries in a model of an input (or output) domain; or scenarios derived from user stories written in the Gherkin domain-specific language.



Sometimes models make omissions that are not safe. Perhaps the model over-simplifies a situation. We need to pay attention to this.

If we don't have models directly from our sources, we must invent them. Where requirements are presented as narrative text, for example, we need to use the language of requirements to derive features and the logic of their behavior. This can be difficult to do for developers and testers, and often it's a collaborative effort, but we must persist.

Using Models to Test

We use test models to:



Simplify the context of the test. Irrelevant or minor details are ignored in the model.



Focus attention on one perspective of the system's behavior. These might be critical or risky features, technical aspects or user operations of interest, or aspects of the construction or architecture of the system.



Generate a set of tests that are unique and diverse within the context of the model.



Enable the testing to be estimated, planned, monitored, and evaluated for its completeness, or "**coverage.**"

From the tester's point of view, a model focuses our attention on areas of interest and helps us to recognize aspects of the system that could be the subject of a test.

Coverage and Models

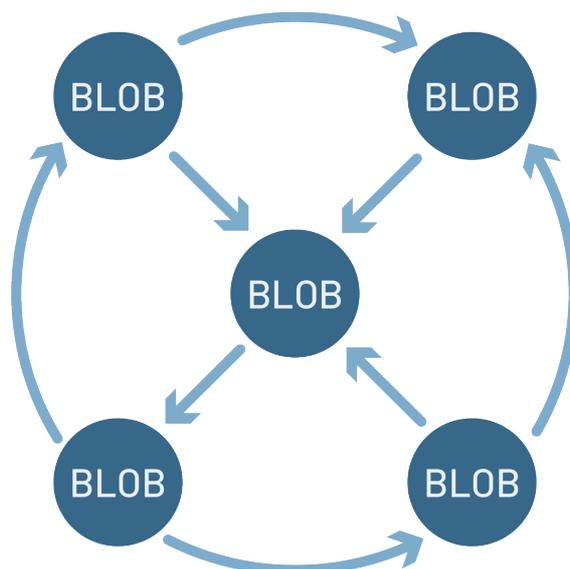
Coverage is the term we use to describe the thoroughness or completeness of our testing with respect to our test model. A coverage item is something we want to exercise in our tests. Ideally, our test model should identify coverage items in an objective way.

When we have planned or executed tests that cover items identified by our model, we can quantify the coverage achieved. We can also express that coverage as a percentage of all items on the model.

We can use any model that allows coverage items to be identified. Models are often graphical with examples such as flowcharts, use cases, sequence diagrams, and so on. These and many other models have elements (or blobs) connected by lines or arrows. These are usually called directed graphs.

Imagine a graphical model that comprises blobs and arrows, similar to the example below. At least two different coverage targets could be defined:

- All blobs coverage
- All arrows coverage



Any model that is a directed graph can be treated in the same way.

A **formal model** allows us to reliably identify coverage items. This allows us to define a quantitative coverage measure and use it as a measurable target.

Informal models tend to be checklists or criteria that we use to

- Brainstorm a list of coverage items
- Trigger ideas for testing
- Guide testing in an exploratory testing session.

These lists or criteria might be prepared as part of a test plan, or adopted in an exploratory test session.

Informal models differ from formal models in that the definition of coverage items depends on our experience, intuition, and imagination. As a result, we can't assign a number to the coverage. We can never know what complete coverage means when using an informal model. That said, tests derived from an informal model are just as valid as tests derived from a formal model if they increase our knowledge of the behavior or capability of our system.

Models Simplify, So Use More Than One

A good model provides a means of understanding complexity. It does this partly by excluding details that aren't relevant. Your model might use the concept of state or failure modes, flows, input combinations, domain values, etc. Obviously, one model is never enough to fully test a system. All models compromise, so we need multiple models. This concept is normally referred to as "**diverse half-measures.**"

Diverse half-measures in practice means that we need multiple, diverse, partial models to test a system.

Although not usually described in these terms, the test stages in a waterfall project use models from different perspectives. Unit, subsystem integration, system-level, and user testing all have differing goals; each uses a different model and perspective to create diversity.

Using Models to Manage

As stated earlier, models are at the heart of testing, and also of test management. There are four key aspects to this:

1. **Stakeholder engagement**
2. **Scope**
3. **Coverage**
4. **Estimation and progress monitoring**

Each of these is discussed below.

1 Stakeholder Engagement



When we plan and scope tests, and when we explain progress and the meaning of coverage to stakeholders, we must use models that they understand and are meaningful to their goals.

If we plan a user test, we'll probably adopt the business process flow as our model and as a template for tracing paths to exercise system features that matter to the user. If we are testing the integration of service components on behalf of a technical architect, we'll use the architectural model, collaboration diagrams, interface specifications and so on as the basis of our testing. If we are testing features defined by users, we'll use the user stories that were the outcome of earlier collaborative requirements work.

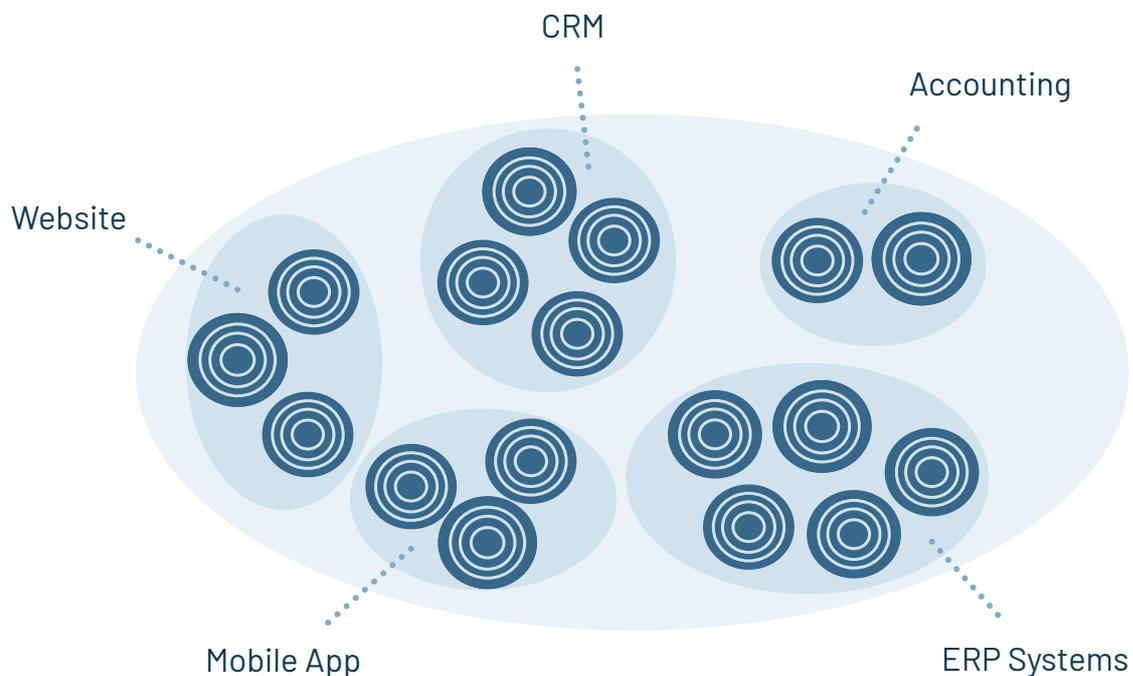
If stakeholders don't understand your models, they will not invest in testing, not understand or trust your testing. They may not even trust you.

2 Scope

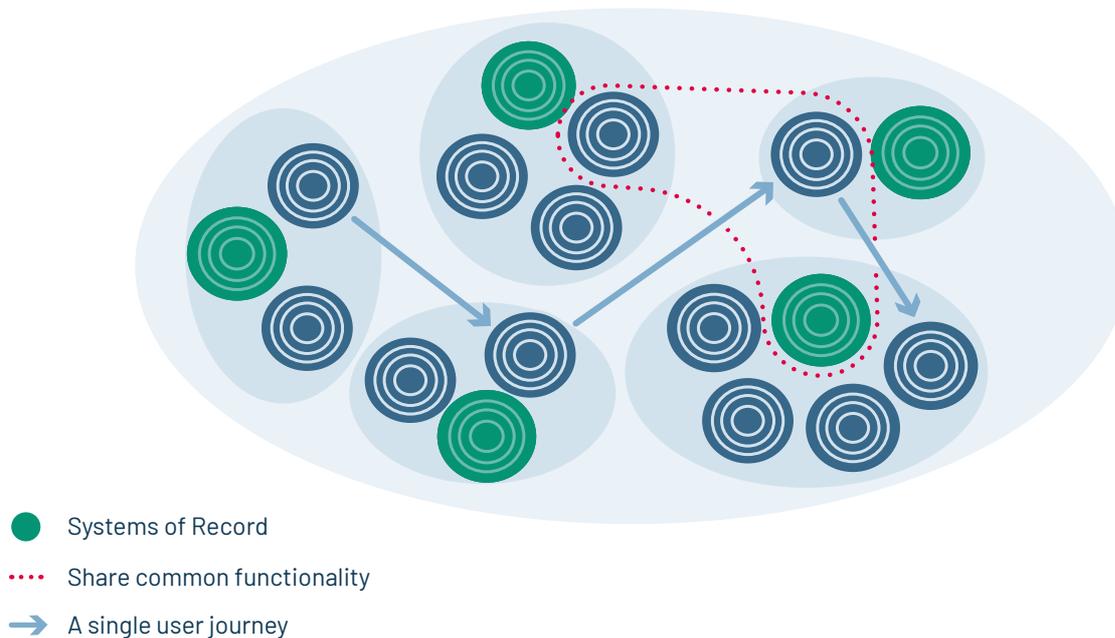


The first activity in a systems thinking discipline is to define a system boundary. In testing, the first model you define will help you to scope the testing. The diagram below is a schematic of the system architecture – a “system of systems” – in an organization. Each system (the blue concentric circles) sits within an application area such as CRM, Accounting, or the website. All of the systems and application areas sit inside the “system of systems.” There is no detail on the model of course, but it’s plain to see how each system fits into the overall architecture.

For example, we could easily define the scope of our testing to be just the ERP systems:



In the second diagram below, we have added some more detail to the system architecture and suggested three ways we could define scope more specifically.



- The systems shaded in green are the **systems of record**. These systems might share a database, for example, and changes to the database schema could affect any of these systems adversely – and be in scope for testing.
- The systems enclosed by the dotted red line might share some common functionality or infrastructure – perhaps they all use a common set of web services or a messaging system or run on the same server.
- The dashed blue line denotes a user journey which utilizes the systems connected by the line. Perhaps the user journey has changed and our focus is on the consistency and accuracy of the flow of data between these systems.

A model can show what is in scope for testing, but just as importantly, what is out of scope too. A model also explains scope to stakeholders in terms they understand.

When we use a model to define scope:

- The model defines the overall territory.
- The items in scope identify the areas within the overall territory which we intend to explore and test.

3 Coverage



Coverage measurement can help to make testing more manageable. If we don't have a notion of coverage, we may not be able to answer questions like:

- What has been tested?
- What has not been tested?
- Have we finished yet?
- How many tests remain?

This is awkward for a test manager. So, after the scope is defined, the natural next step is to identify the target coverage. The scoping model defines the places we will test. The coverage model tells stakeholders how thoroughly we plan to test in those places.

Test models and coverage measures can be used to define quantitative or qualitative targets for test design and execution. To varying degrees, we can use such targets to plan and estimate. We can also measure progress and infer the thoroughness or completeness of the testing we have planned or executed. But we need to be very careful with any quantitative coverage measures or percentages we use.

While you may calculate an objective coverage measure based on a formal test model, there is no formula that says X coverage means Y quality or Z confidence. All coverage measures give only indirect, qualitative, subjective insights into the thoroughness or completeness of our testing. There is no meaningful relationship between coverage and the quality or acceptability of systems.

Quantitative coverage targets are often used to define exit criteria for the completion of testing, but these criteria are arbitrary. A more stringent coverage target might generate twice as many items to cover. But while twice as many tests may cost twice as much, they do not make a system twice as tested or twice as reliable. Such an interpretation is meaningless and foolish.

Sometimes, the formal models used for the definition and build of a system may be imposed on the testers for them to use to define coverage targets. At other times, the testers may have little documentation to work with and have to invent models of their own. *The selection of any test model and coverage target is somewhat arbitrary and subjective.* Consequently, informal test models and coverage measures can be just as useful as established, formal ones.

4 Estimation and Progress Monitoring



“How long will these tests take to run?” You may dread this question. But managers need to plan, so you’ll probably be forced to give an answer.

The best that you can do is to set out, in the time allocated with the resources available, the tests that could be prepared and the tests that could be run. Since test failures require diagnosis and possibly rework and re-testing by developers, any estimate for testing needs to include an allowance for diagnosis and rework (and re-testing) by developers and re-testing by the testers.

Of course, it may be that the resources and time allowed for testing have already been defined for you. Perhaps the project has a fixed deadline, or the development and test activities are timeboxed. In this case, the estimation task is to determine what can be achieved in the time available.

If you have knowledge and experience of similar projects, perhaps you can make an allowance for the difficulty in getting environments set up, running tests, the number and nature of failures, time taken to fix, and the time and success rate of re-tests.

These estimates can never be reliable before testing begins. But, if you monitor progress closely when the test activities start, there’s a good chance that you will learn enough to refine your estimates and realign your plan as time passes. Even so, estimation is always an uncertain process.

Sources of knowledge are fallible. A tester is a human being and prone to error. The system is being tested because we are uncertain of its behavior or

reliability. As a consequence, any plan for any test worth running cannot be relied upon to be accurate before we follow it.

Predictions of test status (e.g. coverage achieved or test pass- rate) at any future date or time are notional. The planning quandary is conveniently expressed in the **testing uncertainty principle**:

You can predict test status, but not when it will be achieved.

You can predict when a test will end, but not its status.

When you are pressed for estimates, you can offer the uncertainty principle as a reason not to trust them. But it's inevitable you will be pressed. All you can do is make the best estimate you can, and offer some caveats to your estimate to set reasonable expectations.

Most of the information you need to estimate is either not in your control, or is available in detail only after you test. In most organizations, the phase after development is some form of testing. But it's easier to think of rework as the phase after development. The bugs that cause failures and the time taken to perform rework are not known before you test and it's the developers who put the bugs in (so to speak).

*If it is reasonable to ask a tester,
"How long will it take to test the system?"*

*then it is reasonable to ask developers,
"How many bugs will you put in the system?"*

We'll use the concept of models later in this ebook.

Putting It into Practice

Consider a current or recent project. Identify the sources of knowledge that are available to the project team. Sources are usually documents, people, team experience, the new or old system. Identify other sources if you have them.

For each source, consider whether the it is (or was) accurate, whether it is updated over time, what it is useful for, and what the risk of using it might be. Use a table like the one below:

Source	Accurate (at some time)	Kept up to date?	Useful for	Risks of using

In your list above, do any sources supersede or overrule other sources?

Do developers use different sources of knowledge than the test team?

Model Making

In the following examples, sketch out what you think the model might look like – its shape only – either as a picture/diagram, a table, or a list:

- Users are concerned about the end-to-end journeys in the system.
- A messaging service can be in four states, shut down, running, starting up and shutting down.
- An insurance premium calculator has 40 input values which, in combination, influence the calculation; there are dependencies between some inputs.
- An extract/transform/load process has seven stages. After extraction, each stage either rejects records, sends them to a suspense file, or transforms and passes records to the next stage. The last stage is a load process which handles rejections. Test that all extracted records are accounted for.

Final Thoughts



Consider your primary sources of knowledge for deriving tests. Are there any models built into these sources? For example are there any decision tables, or flowcharts, use cases, or BDD (Gherkin) stories that you can use directly?



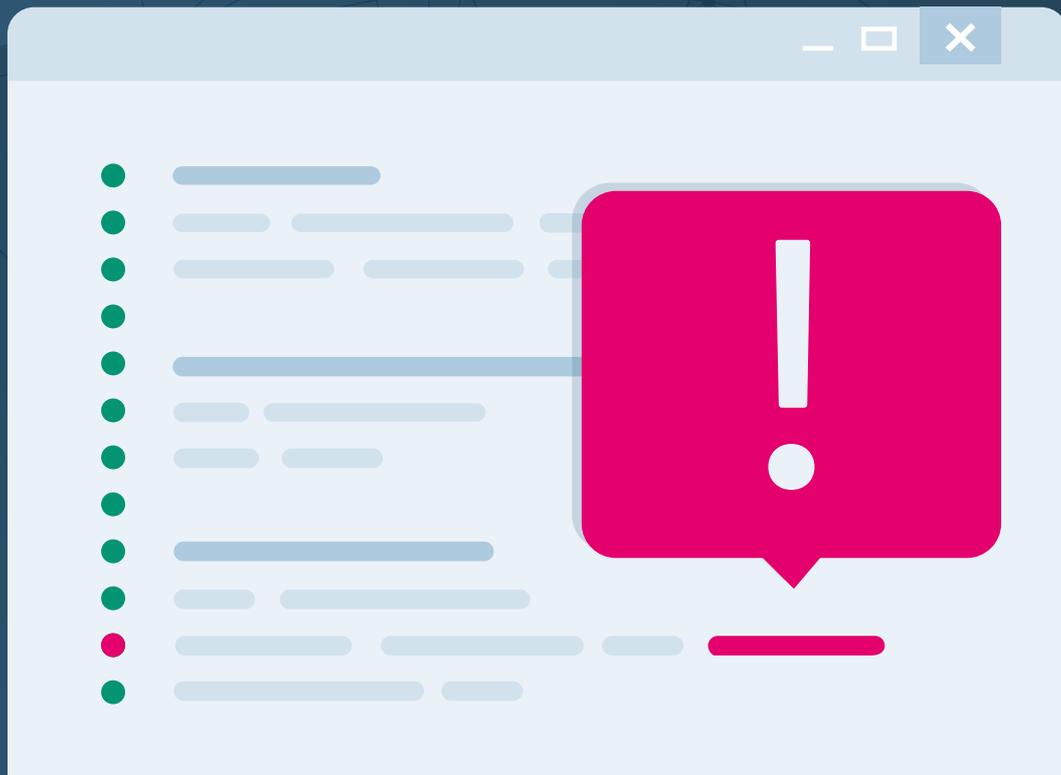
If you have user stories without scenarios or examples, could you create your own stories to enhance these stories?



Where requirements are in plain text, could you think up models for individual features?



If you must perform integration or end-to-end tests, what models are available? Would you need to create your own?



12 Secrets of Effective Test Management

4. Focus on Risk

The concept of risk was first studied and described in the late 18th century by the French mathematician and philosopher, Blaise Pascal. Pascal was asked by a gambler how to win games of chance. The mathematics of probability and the balance of risk and reward derives from this work. Risk and probability are not new concepts. Risk management as a discipline emerged in the years after the Second World War.

In some businesses, risk management is practiced with engineering and mathematical precision. In software, it's not possible to be very precise. Most organizations still don't practice risk-based testing systematically. But there is a good reason for this: software product risks are often terribly difficult to assess.

From the perspective of testing and assurance, a risk is a "mode of failure to be concerned about."

Risk-based testing is the practice of modeling potential system failure modes as product risks to support test scoping, scaling, and prioritization. In this section, we'll look at classic risk management and see how it can be adapted for testers. We'll then look at some practical ideas to apply your own risk-based test method.

We'll use this definition of risk:



A risk is a threat to one or more of the stakeholders' goals for a project and has an uncertain probability.

Risks, if they materialize, have an adverse effect on our projects. Risk management is thinking about what risks exist, and then taking action to achieve one of the following:

- Reduce their likelihood
- Eliminate them
- Reduce their impact on our stakeholders' goals.

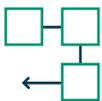
Risk-Based testing in a Nutshell

There are hundreds of books on risk and risk management and these set out approaches to managing risk at a project level. Typically these approaches focus on what we'll call project and process risks with perhaps a single risk titled something like 'failure to meet requirements'. Having a single requirements risk isn't very helpful. It's not as if you can prioritize one risk for testing.

There are three types of software risk:



Project risk: These risks relate to the project in its own context. Projects usually have external dependencies such as the availability of skills, dependency on suppliers, constraints such as a fixed-price contract, or fixed deadlines. External dependencies are project management responsibilities.



Process risk: These risks relate primarily to the internals of the project and the project's planning, monitoring, and control. Typical risks here are under-estimation of project complexity, effort, or the required skills. The internal management of a project such as good planning, progress monitoring, and control are all project management responsibilities.



Product risk: These risks relate to the definition of the product, the stability (or lack) of requirements, the complexity of the product, and the fault-proneness of the technology which could lead to a failure to meet requirements. Product risk is the main risk area of concern to testers.

We're going to concentrate on product risks. We'll use the term "product risk" as follows:

A product risk represents a mode or pattern of failure that would be unacceptable in a production environment.

In projects where risks will be taken (which is all of them, we think), testers need to make risks visible to management and propose reliable test methods to address these risks. In turn, management must oversee the following risk-based testing tasks:

1. **Assess software product risks.** The tester needs to identify and assess, through consultation, the major product risks of concern and where appropriate, propose tests to address those risks.
2. **Gain consensus on the amount of testing.** To achieve, through consensus, the right coverage, balance, and emphasis of testing.
3. **Plan, implement and execute tests.** To specify and run tests that address the risks of most concern.
4. **Provide information for a risk-based decision on release.** This is perhaps the most important task of all, as information is the major deliverable of all testing.

The Classic Approach to Risk

There are many variations, but the classic approach to risk attempts to be quantitative and is well-established.

Probability, Consequence, and Exposure

To assess a product risk we need to understand what the consequence of that mode of failure is. If a failure happens, we say the risk materializes. How serious is the risk? We need to understand how exposed we are to that risk.

Assess each risk in two dimensions:

- The **probability** of a risk materializing. This value is typically expressed as a percentage between (but not including) zero and 100 percent.
- The **consequence** of a risk materializing. This is the potential cost of the damage if this mode of failure happens.

The exposure of a risk – how serious a risk it is – is calculated as the product of the probability and the consequence.

$$\text{Exposure} = \text{Risk Probability} \times \text{Risk consequence.}$$

Quantitative or Qualitative Scales?

Now, it is often difficult to predict the potential cost of a failure. It is impossible to predict the probability of a failure with much certainty, without further information at least. So most practitioners adopt semi-quantitative or qualitative scales for probability and consequence.

It is common to use numeric ranges from one to five for both scales giving us possible values for exposure of one to twenty-five. But few testers, developers or stakeholders assign these numbers with any confidence, so it is common to simply grade risk exposure directly using a scale of 1 to five, or even simpler – a red, amber, green assessment. Some go as far as simply saying risks are in-scope or out of scope – but we would suggest this is a simplification too far.

Whether you quantify or color-code the risks on the table, the critical aspect of risk assessments is not the scoring, but the conversations and debate you have when assigning and discussing the scores. To misquote Eisenhower, the risk plan is nothing, risk planning is everything.

If you are asked to score risks using any more precision than scales of 1-3 or 1-5, you should seriously consider whether the numbers you assign have any real value. Numeric scoring might give an impression of scientific rigor, but if the numbers are guesswork you are fooling yourself. If numbers expose differences of opinion (e.g. a user says “5” and the developer says “1”), there’s a conversation to be had as expectations or perceptions are clearly different – and need resolution.

Beware of using a simple in scope/out of scope scheme for risks. It might be clear which risks of concern are to be addressed by testing. However, projects learn over time and often slip, so compromises have to be made. If risks are not prioritized in some way, you’ll have to review all risks of concern to decide which might need removing from or adding to scope.

We’ll discuss scoping, prioritization and scaling in more detail later in this ebook.

Risk-Based testing in a Nutshell

A discussion of product risks will expose stakeholders' fears of failure in critical areas of the project or system. Given a list of product risks, how do we translate that into action and specific plans for testing? Before we can answer that question, we need to understand a little better how testing affects or influences our risk assessments.

Testing Doesn't Reduce Risk: It Informs Risks Assessments

You will often hear people suggest that "testing reduces risk." That's not true – testing can increase risk sometimes. The "testing reduces risk" mantra comes from a misunderstanding of both risks and testing.

Let's say we have identified a risk that some feature or functionality might fail. We formulate a test and run it. The test might pass or might fail. How would a test passing or failing affect our assessment of risk?

First, tests have no effect on our understanding of the consequence of a failure. The consequence of a failure is what it is and testing doesn't change that.

There are four relevant outcomes of a test and consequent impacts on risk probability. These are summarized in the following table:

	Risk Probability Increases	Risk Probability Decreases
 Test Passes	No. A passing test can only make us feel more confident a failure mode will not occur. This assumes our tests are meaningful.	Yes, eventually. As we run more and more tests that pass, and we explore diverse scenarios, the likelihood of failure recedes.
 Test Fails	Possibly, depending on the failure. But the likelihood decreases over time. We predicted this mode of failure would happen; our test choice is vindicated. Thankfully, we can fix this now, and focus our testing to reduce the risk of other failures of this type.	Not likely, unless we got our risk assessment wrong.

Running tests that we associate with a particular risk gives us more and more information to refine our risk assessment. If tests fail, our choice is vindicated. When we fix and re-test, as tests pass our assessment of risk should reduce. But if we keep finding new/more bugs, we might reassess the risk as higher.

If you run a test and the outcome doesn't affect your probability assessment either way – it's probably a worthless test.

Now that you understand the potential of testing to inform our risk assessment, we can look at specifying tests from risk descriptions.

Risk-Based Test Planning

Once you have identified a product risk of concern, formulate a set of tests to assess the likelihood of that risk materializing. The test approach will be to stimulate the failure mode in as many ways as possible. In so doing, you will see one of two outcomes:



Failures, which detect bugs to be fixed, thus reducing the risk from that mode of failure



Passes, which increases confidence that a mode of failure is unlikely to occur.

Our tests, collectively, focus on ways in which a selected mode of failure can occur. Suppose, for example, the risk was “failure to calculate a motor insurance premium correctly.” An appropriate test objective might be “demonstrate using the all-pairs technique for input variables that the system calculates the motor premium correctly.” This test objective can be given to a tester to derive a covering set of test cases using all-pairs, for example.

Overcoming Challenges



What if stakeholders are not interested in helping with the risk assessment?

You might find that it's hard to get stakeholder's time to help you identify and assess risks and formulate your test plan. This is best answered by the stakeholder story from section 1 of this ebook.



What if there are multiple test options for a product risk?

This is almost always the case. For example, to test a feature of concern you could use any one of a range of test design techniques, or model the feature in some unique way and derive tests from that model. There is a range of coverage measures you might use to set a target. You might test a feature manually or using a tool or other technology. We'll look at how you might make choices in the next section.



What if testing to address a risk costs too much?

Since there are almost always options, and there is no limit to how much testing one could apply to explore a risk and inform a risk assessment, a choice has to be made. Some options may be deemed too expensive. So, when you present options to stakeholders, you'll need to have at least one option that is affordable and one that might be unaffordable.



If we test high risk features more, we test other features less, don't we?

If the budget for testing is pre-set for a team, or the team size and number of testers is fixed, then the amount of testing that people can perform is limited (in any fixed time scale or timeboxed period). So, if we place more emphasis on some features, then the testing on others must reduce. The coverage or at least effort across features varies with risk.

The way to look at this is that when a test fails and a bug is found in the system, then the severity assigned to the bug is likely to be closely related to the consequence of that failure in production. More severe bugs get fixed because they are simply more important to stakeholders. A product risk

analysis highlights the areas where important bugs are more likely to be fixed. So, using a risk analysis to focus on areas for testing means that we are more likely to find important bugs and less likely to find bugs in features that stakeholders might care less about.

It would be nice sometimes to think that we can test everywhere in a more even fashion. But where resources and time are limited (and they always are, aren't they?), a risk-based approach helps you to utilize tester time more effectively and efficiently.



What if testing isn't the right response?

Before you commit time to figuring out a testing approach to a product risk in any detail, it's important that non-testing options are also considered. For example, suppose there's concern that a system component might not perform well enough in production and response times or reliability are poor. Of course, you could perform load tests and try and debug your way out of problems. But other options might be:

- Buy the component from a 3rd party: don't build and take the risk.
- Re-architect the solution to distribute loads across multiple component instances.
- Offload the processing to a batch system on a copy of the data.
- Rather than implement a big-bang implementation of all users, take a phased approach to implementing customers region by region and monitor performance closely.

Often, a different course of action to prevent problems happening is more effective and economic than testing to find those problems later.

Understanding the Role of Testing in Risk Management

Testing can reduce the risk of release. If we use a risk assessment to steer our test activity, the testers' aim becomes explicitly to design tests to detect faults so they can be fixed, and in so doing, reduce the risk of a faulty product.

Fault detection reduces the residual risk of failures in production, where costs increase very steeply. When a test finds a fault and it is corrected, the number of faults is reduced and consequently, the overall likelihood of failure is reduced. You could also look at it this way: the “micro-risk” due to that fault is eliminated. But it must be remembered that testing cannot find all faults, so there will always be latent faults which are untouched and on which testing has provided no direct information.

However, if we focus on critical features and find faults in these, undetected faults in these critical features are less likely. The faults left in the non-critical features of the system are of lower consequence.

One final note: the process of risk analysis is itself risky. Risk analysis can both overestimate and underestimate the risks leading to less-than-perfect decision making. Risk analysis can also contribute to a culture of blame if taken too far. Among other potential snags with risk analysis, you must understand that there is no such thing as absolute risk that could be calculated after the project is over. The nature of risks is that they are uncertain; just like the effectiveness of testing, the value of a risk analysis can only be determined with hindsight after the project is over.

Putting It into Practice

Think of your system and how it might fail in production in some way.

- Spend ten minutes making a list of failures that your system might suffer.
- Is your list getting longer? How many failure types have you identified so far?
- If you kept going would you have more than a hundred modes of failure?

Your list probably looks like a list of bugs. If you look at these bugs from the perspective of your stakeholders, you could assign a severity to each bug and create a prioritized list of bugs to fix.

As an exercise, look at a few of these bugs in more detail. For each one, use the language of the bug to create a test objective you could give to a tester. For example:

- Bug: order entry detail miscalculates the cost of an order item (quantity x price)
- Test objective: demonstrate order entry item cost is calculated correctly for a range of order quantities and prices including extremely large and small values as well as typical values.

You don't have to reuse the language of the risk to create an objective, but it can help. Knowing how you will test for a problem (a risk) will help you to write better risk descriptions. (Doesn't this sound just like testers seeing problems with requirements?)

One way of looking at risk-based testing is you are speculating what high severity bugs will find their way into your bug logging system. Just imagine, if you knew ahead of time what bugs would occur, how that might alter your test planning choices at the beginning? This is what risk-based testing does – it helps you to make better choices in test planning at the level of detail of your risk assessment.

Risk-based testing is a universal approach – you can use it to scope huge project test plans, or if you are exploring, to define the very next test you try on a system.

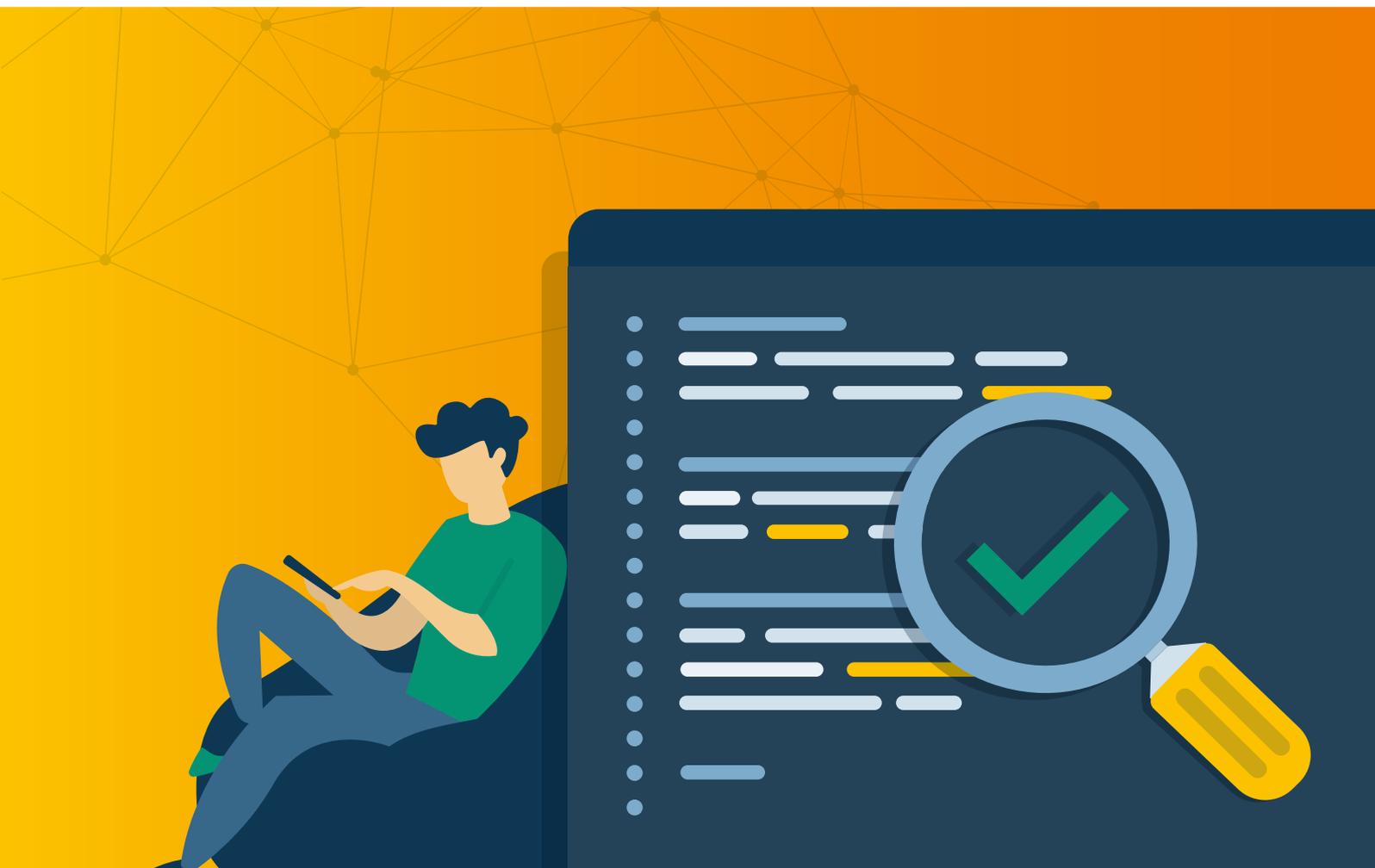
Final Thoughts

If you think about system failures at a low level – that is, the level of bugs or real failures – you might end up with a list of hundreds or thousands of bug types. Yes, you could assign a different risk to each and every bug type, but that list would be enormous and unusable. Rather, you need to look for patterns or modes of failure instead. There is no golden rule for this, but if you are working on even a large system, you really ought not to have much more than one hundred failure modes.

Some failure modes will be critical and relate to a single feature. They might be system-wide like performance or they might be systemic across all features. Usability shortcomings fall into this category. At any rate, a rule of thumb in your risk identification activity is to look for a number between say forty and one hundred product risks across the system.

Now, create a few risk descriptions based on your knowledge of the system.

Finally, create test objectives to address each risk in turn.



12 Secrets of Effective Test Management

5. Decide How Much Testing is Enough

How much testing is enough? This is the classic, unanswerable, philosophical question that all testers ask – because stakeholders want the answer! Stakeholders want confidence that systems have been adequately tested and will work. But they also want to know the potential cost of testing and how long it will take – after all, directly or indirectly, they are paying for the testing, and they have deadlines to meet.

Ultimately, it is for stakeholders to judge how much testing is enough, but before we can discuss that, we need to understand the value of testing to stakeholders.

The Value of Testing

Every test should have value to stakeholders. The tests that we run have value because they provide stakeholders with four types of evidence to support decision making:



Evidence that the system will meet the business goals of the project



Evidence that the system will not fail or if it does, that the impact of failures is bearable



Evidence to reproduce and diagnose failures and repair and re-test the failed system



Evidence to support decision-making in the context of a project (to accept, to release, to reject, etc.)

Our goal is to create tests that incrementally increase coverage of the system with respect to a recognizable test model. Our tests should demonstrate that the system will meet the business goals or requirements of the system and that the risk of failure is known, and hopefully, acceptable.

Of the four types of evidence above, the first three underpin the fourth. Ultimately, the stakeholders need to make an evidence-based decision. It is their decision to make, not the testers, so it is for them to judge whether they have enough to be confident. At any rate, the value of testing is in the eye of the stakeholder.

*Beauty is in the eye of the beholder;
but the value of testing is in the eye of the stakeholder.*

Now, every tester makes choices on what to test. Whether they use a formal model or even gut feel, these choices are made on the basis of some perceived value. Unless the tester is also the stakeholder, the tester usually judges value on the basis of some measure of completeness or coverage or, if they know the minds of stakeholders, on whether the test will support some acceptance decision. But ultimately, it is the stakeholders who are accountable for their decisions and can judge value.

*The value of testing is measured by the confidence
stakeholders have in their decision-making.*

These principles have some important consequences.

Firstly, if you do not know the mind of the stakeholder, then your perception of test value is unlikely to be the same as theirs. If you design tests without reference to their values, when the time comes to present your results, the stakeholders may find they have insufficient data in some areas and an excess in others. They certainly won't feel as confident as they should. Do not presume you know the mind and values of stakeholders; testers must engage with stakeholders.

Secondly, when you design or run a test, what is its contribution to the stakeholder decision-making? If a test doesn't provide some incremental information supporting a decision or if stakeholders don't care whether your test passes or fails, then it has no place in a test plan. In section 3, we said that the test models you use must be relevant to stakeholders. If your test results map to models that the stakeholders understand and value, then they will value your contribution.

The Significance of Tests

There are two further principles relating to the value and significance of tests. I call one the Quantum Theory and the other the Theory of Relativity. These labels sound pretentious (they are tongue in cheek for sure) but they describe two phenomena that underpin all discussions of test value, prioritization and scoping.

When we run a test, we usually interpret the outcome as a pass or a fail. The pass/fail judgment is a binary outcome – a true or false, yes or no, a one or a zero. That test outcome generates a discrete quantum of evidence. Evidence builds up as we execute more and more tests. Whatever the outcome, that test incrementally increases the coverage of our test model. It incrementally increases the knowledge we have of the behavior of the system.

If a test does not increase knowledge in some way, it has little value.

The second aspect is the value of a test. What is the value of a test? Could you really put a Dollar, Pound or Euro value on a single test? Probably not. But what we can do – and often rather easily – is say, ‘this test is more valuable than that one’.

Suppose we use a code coverage model like **statement coverage**. We could run a test that exercises say, five lines of code or perhaps five thousand lines of code. What is the value of each? It’s hard to say. But if our goal is statement coverage, surely the second test has more value.

We cannot put an absolute value on any test. But, we can usually compare tests and determine their relative value. That is, if we are under time pressure, we can usually say one test has less value than another and therefore de-scope the first test if we have to.

We can compare the value of tests but only if they derive from the same model.

We need to stress however, that these comparisons are really only meaningful if they share the same model. A test that covers a large range of extreme conditions in a process probably has more value than a test of the straight-through path. End-to-end tests of a complex process can’t be compared directly with the unit tests of critical components, for example.

Note that our treatment of the value of tests is very similar to our assessment of risks. It is very hard to put numeric values on the scale or exposure to risk. However, it is usually possible to compare one risk with another and rank them to make choices on the risks in scope for testing.

There is one more important consideration. When we run a test once, it has some value. When we run the same test on the same system version under the same conditions, and it produces the same results, does it have the same value? No, of course not. The first test told us that the system works (or does not work) under some set of conditions. The second test tells us only that nothing has changed (under those same conditions). This might not be so interesting. But where the software or environment does change, then a repeat test like this has more value. This is the essence of testing to detect regressive behavior. Understanding this also helps us to formulate a test automation strategy. Regression tests don't aim to cause failure or find defects; they aim to demonstrate what is called **functional equivalence** – a different objective.

Using the Right Language

Using the language of risk, testers will be heard by management. Managers listen to development team leaders, even when they talk in technical terms. The difference is that developers present the technology as exciting and beneficial. When testers talk in their own technical terms – of the clerical details of tests, such as incident statistics – the message tends to be negative, and managers can become bored or irritated. Management may already think testers are slightly dull as a species, but this is arguably because many managers don't really understand what testers do and what value they add. So testers should raise their language to management level.

Risk-based testing talks to user management and project management in their language. These are people who think very much in terms of risks and benefits. If testers address them in similar terms, management is more likely to listen to testers and make better decisions.

By relating tests to the goals of the project, we can focus attention on the deliverables of most value to stakeholders. This enables us to spend our time testing the most important things. Further, as progress is made through testing, we can demonstrate that the most valuable benefits are now available. Because the release decision is ultimately a judgment of whether the benefits to be gained outweigh the risks, tests will be providing better data to management for them to base decisions on.

*Use the language of risk (and benefits) to scope,
plan and to report progress on testing.*

Testers obviously need to talk technically with developers and other technical staff. For example, the quality of incident reports is a key factor in getting faults corrected quickly and reliably. We are simply suggesting that, when talking to management, the tester talks in terms of risks addressed and outstanding, rather than tests, incidents, and faults.

Estimates and Negotiation

Your project manager asks you, early in a new project, "I need to schedule and resource the testing in good time, please can you give me an estimate for how many people and how long you need to do the system testing?"

You do some thinking and go to talk to the boss. "I need six testers for eight weeks."

The project manager thinks for a moment, and consults his draft schedule and resource plan. "You can have four testers for six weeks and that's all."

You object, and say, "But it'll take longer than six weeks! It'll cost more than you've allocated to test this system. The system is bigger than last time. It's more complicated. It's certainly too risky for us to skimp on the testing this time. It's just not enough!"

But the manager is adamant, muttering something about other dependencies, higher authorities and so on...

What, you think, was the point of doing an estimate, if the manager knew what the budget must be all along? What relevance does an arbitrary budget have to the job in hand? Why don't they ever take testing seriously? The developers always get the time they ask for, don't they? It doesn't seem fair.

You might feel aggrieved and that your professional judgment is undermined. But the problem is, and always was, that the test budget in this situation was fixed. All you can do is figure out what is the best or most valuable testing you can squeeze into your budget.

But often, the Project Manager really does want to know how long things will take so the plan can be adjusted. If you think you are in a negotiation, you need something to bargain with. You also need to discuss the outcomes of the plan, not the inputs. Scope is an outcome of the planning, the effort you apply is an input. You need to negotiate scope.

Negotiation of test budgets should always be about scope, not effort.

Scope might be in one or several formats and the way you present scope and discuss it will vary, but here are some common patterns. Whatever scope you have, you will use that as the basis of your estimate and your defense of it:

- **Scope as an inventory of requirements or features:** When the estimate is cut by 30% ask, 'which 30% of the system should I not test?'
- **Scope as an inventory of risks:** When the estimate is cut by 30% ask, 'which stakeholder risks should be cut from the plan?'
- **Scope as a tabulated or graphical model:** When the estimate is cut by 30% ask, 'which paths/journeys/items won't be tested?'

I hope you see what's going on here.

The scope of testing is based on some model that has been discussed and agreed with stakeholders first. This scope is provisional, and subject to resources and time being available, and you should make stakeholders aware of that.

You estimate on the basis of that provisional scope. Use the model (risks, requirements, business process or any other model) to set a coverage target and count coverage items and estimate on that basis.

Have the discussion with the Project Manager. If the estimate is too high, then use the responses above to trigger the discussions with stakeholders.

As the tester or test manager, you are not in a good position to negotiate testing with the project manager. Stakeholders are the priority. If you share the models that define testing scope they will have opinions, but will buy into the scope and defend it. Stakeholders are also ultimately responsible for justifying the budget for their system, so they are best placed to balance the cost of testing with the need to address their concerns.

Negotiations of scope take place between project managers and stakeholders; your role is to facilitate those conversations.

The tester's role is to facilitate that discussion, not to pre-empt decisions that are ultimately the stakeholders' responsibility.

Putting It into Practice

In your projects, who takes responsibility for the amount of testing to be performed?



Do you, as a tester, define the scope and the amount of testing?



Does the project manager set a budget and you do the best you can with the time you are allocated?



Is the scope negotiated with the project manager and stakeholders to agree on a balance between effort and the scope of testing to be performed?

One of the key responsibilities of a tester is to ensure the team is aware of the product risks being taken and to document them. Only if these risks are visible can management recognize the risks they take by squeezing testing.

There's no formula for the right amount of testing. In most environments, the level of testing can only be determined by consensus between project management, customer sponsors, developers, technical specialists, and the testers. Tests are deemed in scope if they address the risks of concern.

Tests can be de-scoped if they address risks that are deemed too low to worry about, or the cost of the tests is too high, or the test might not be the only way to address a particular risk. If no one can see the value of doing a test, it's probably not worth doing.

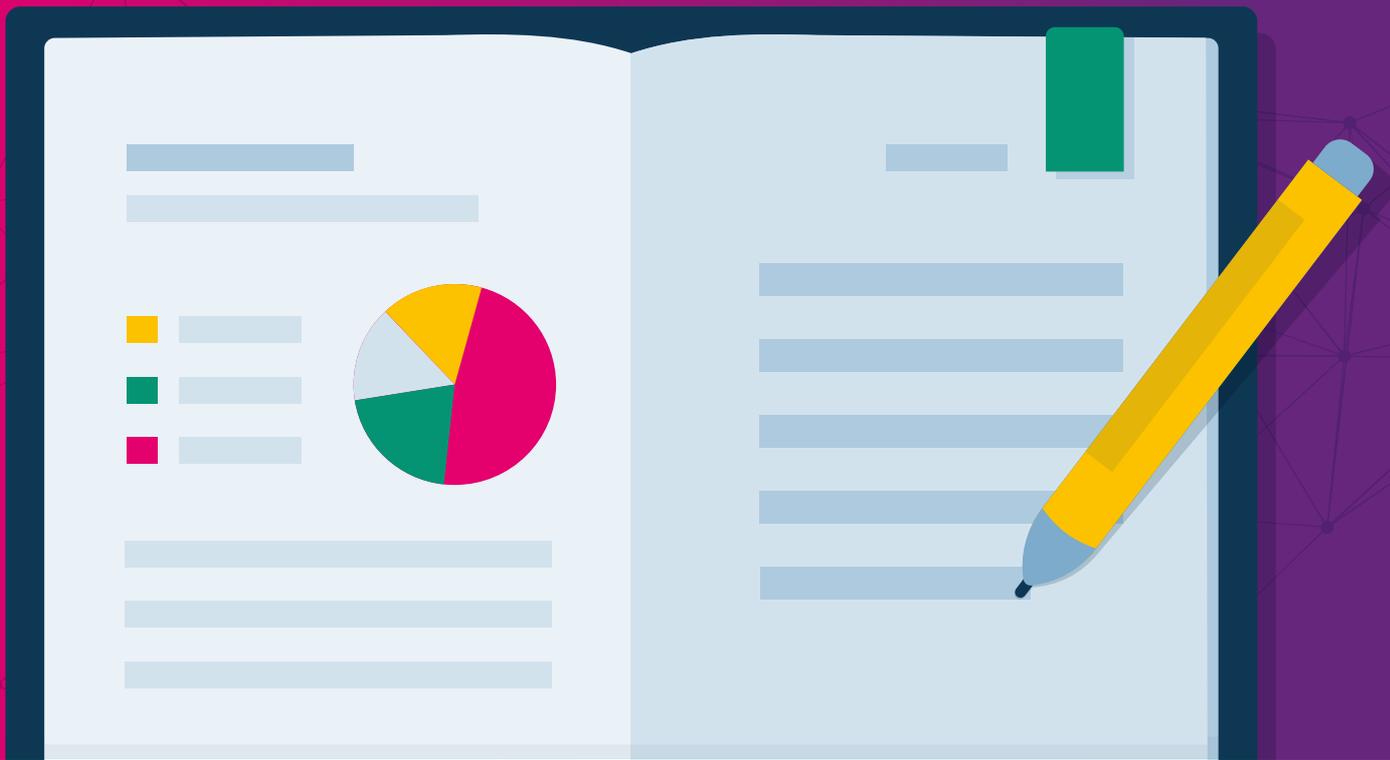
Enough testing should be determined by consensus, with the tester facilitating and contributing information to the consensus process.

Final Thoughts

You already know that we use models to simplify the problem of scoping the testing. When we use graphical models in particular, there are things we can identify as coverage items to set a coverage target. Most models allow several different targets to be set and each target has a different cost. By offering projects varying targets, you offer different levels of thoroughness and cost.

You can use a table similar to identify the things you can count and use as a coverage target. Then, you can rank these targets in order of completeness and potential cost.

Model / Knowledge Source	Coverage Item	Coverage Target
Source code		
Tabulated requirements		
Requirements as narrative text		
User stories and scenarios in Cucumber syntax		
Risk register		
State transition diagram / state model		
Business process / flowchart		
UML Use Case		



12 Secrets of Effective Test Management

6. Create Useful Documentation

In any project, in any organization and using any approach, there is a place for documentation. Documentation provides a useful record of approach, scope, plans, and designs, as well as the outcomes of analysis, development and test activities. In structured projects, documents are usually deemed to be deliverables in their own right. In agile or continuous regimes, documentation might be produced as by-products of more or less value.

Documentation can be a valuable repository of knowledge.

Document writing may be the primary activity of professional technical writers, but for most practitioners, documentation is a chore, no matter how useful it may be. Document writing may be boring for some people, but the real problem with documentation is that in many contexts, most documentation is simply a waste of time. It is of little value or out of date or inaccurate – or all three.

Every test manager has written test strategies that people did not read or buy into. Testers write reams of test plans, scripts and reports and the only content of value to stakeholders are the one-page summaries at the start or end.

*We have all written documents that we know have little value
and no one will read.*

There are problems with documentation that we encounter in projects large and small. For every document we write, there are several questions we need to address:

- What type of document? A policy or strategy, an approach or plan, a design or implementation, or an outcome and interpretation?
- What is the aim of the document?
- What content is required to meet that aim?
- What sources of knowledge are required to create the content?
- If the document must change over time, how will it be maintained?
- What level of detail is required?

As a test manager or as a team, you'll need to figure out what types and formats of documentation are appropriate and, if they are to be accurate records or so-called living documents, how they are maintained.

The Perils of Templates and Cut/Paste

If your project decides that a certain document is required, say a system test plan or a risk-register, it is tempting and easy to find an off the shelf template for these on the internet. Some templates may claim to adhere to some standard or convention or to have been downloaded and used thousands of times. Sometimes a template may appear to suit your purpose exactly, but even if the table of contents looks comprehensive, it can get you into trouble.

It may be that you or other people in your company have prepared similar documents for previous projects. Why don't we just copy and rename the document, change the references to the old project and edit the content to suit? In my experience, as an independent reviewer, this is very common and often a real problem. Firstly, it's usually blatantly obvious that a copy/edit has been done. The language used in the text often seems disconnected from the project and there are gaps and superfluous text everywhere. Why is this?

Using a pre-formed template or existing document as a source carries several risks:



It looks comprehensive, but it might include topics that are inappropriate and exclude others that are essential.



It provides headings for a document, but absolutely no guidance as to what content is appropriate for each heading.



It might contain text that looks reusable and is copied unchanged from a previous unrelated project, but that text might give a wrong impression, be inaccurate or incomplete.

Using templates to get headings and basic formatting layout might be helpful, but the main problem with templates is this:

*Using a template might save some time;
the risk is you won't put enough thought into the writing.*

The temptation with templates to trust the template and write boilerplate for the various sections. After all, you might think the document completes a “tick box” and no one will read it anyway. The risk of templates is that you stop thinking and write a document that has little value.

Types of Test Documentation

Let's look at the various forms of test documentation and discuss some considerations in structured or agile/continuous projects. We won't attempt to align with any certification scheme here. These schemes tend to offer one-size-fits-all templates, which are unlikely to suit any particular context.

The core set of test documents tend to fall into the following categories:

- Policy and strategy (sometimes known as Master Test Plan)
- Test definition (also called specification or test plans)
- Test design
- Test cases
- Test procedures or scripts
- Test execution
- Schedule
- Log
- Test report

This range of document types cover the definition of the test process, key activities of definition, and execution and reporting. There are several other test-related documents that, in more bureaucratic environments, would include test environment definitions and management processes, acceptance procedures, incident management processes and so on. Incident management will be covered in section 8.

An obvious omission from the above would be an overall plan or schedule for test activities. A schedule isn't really a test document, it's really a subset of an overall project plan for a structured project at least. We'll cover schedule planning in section 7.

A Documentation Framework

Below are profiles of typical document categories for a structured/waterfall project. The aim is to provide an example for what you might regard as a documentation-heavy context, and then suggest where compromises might be made for agile or continuous contexts. Since agile or continuous projects almost invariably require less content and rely more on constant communication and collaboration, we present for each document category a summary of the kinds of compromises that would be made in a typical agile or continuous project.

Compared to a structured project, the goals of an agile or continuous test strategy will look similar but the approach and documentation are likely to be much less formal.

The profiles of structured documentation are not comprehensive but are intended to give you an impression of the nature and content of documents in these projects.

Each document type is described in terms of its purpose, content, sources and maintenance considerations in a structured context. These document profiles provide insight into the type of information required to meet the document aims. Then follows a discussion of how the document might be amended, merged into other documents or avoided entirely in an agile or continuous delivery context.

Structured projects might have very detailed test strategies of tens or hundreds of pages or even have strategies for distinct test stages. An agile project might have many of the same aims as a large project but the approach used might never be documented. It's just, 'the way we work around here'. Every organization, every project, and every team is different.

There is no one formula for documentation, but it has to be fit for purpose; you will need to talk to your stakeholders and team members to find that purpose.

Policy, Strategy, Master Test Plan

Purpose	<p>A policy usually covers an organization and includes a subset of topics covering all projects. A strategy usually covers a single project (or application)</p> <p>Overall, the strategy provides decisions or choices made on logistical questions of approach, handoffs, responsibilities, environments, etc. Some of these decisions can be made ahead of time and documented in the strategy</p> <p>Some decisions can't be made now, but the strategy can document the process or method or information that will allow decisions to be made (in project). For uncertain situations or unplanned events where decisions need to be made, the strategy will document the general principles (or process) to follow.</p>
Content	<ul style="list-style-type: none">• Stakeholders, goals, key risks of concern• Test principles/approach to be adopted, e.g. risk-based test approach• Test process (stages of testing):• Goals and scope• Acceptance criteria• Methods, techniques• Deliverables (documents)
Sources	Stakeholders, users, BAs, developers, operations
Maintenance	Usually a one-off document defined for a project or program
Agile/Continuous Considerations	<p>The test strategy for agile projects using Scrum, for example, are likely to be rather brief and comprising just a few pages, if they are documented at all. The test process won't have stages perhaps, but there is likely to be definitions of testing at different levels, for example:</p> <ul style="list-style-type: none">• Testing in a sprint or iteration• Testing for a release• System Integration testing (with other systems or interfaces)• User testing (in sprint and/or release-level acceptance) <p>How tools are used in developer testing (and using BDD or TDD for example) is likely to go undocumented, but development teams would be expected to evolve an approach and liaise with other team members as they commit new code. The role of the tester might be to test features interactively as they are released by developers or to act as a testing coach to the rest of the team. Like the use of tools and or TDD, the way of working would evolve over time and might never be documented formally.</p>

Test Definition (Design, Cases, Procedures)

Purpose	<ul style="list-style-type: none">• Demonstrate the connection between sources of knowledge and the tests to be performed• Document the coverage of the requirements, system features, or user behavior, against multiple models• Enable stakeholders to review the scope, approach, coverage, and choices made in creating tests.• Provide instructions for the performance of tests at a pre-determined level of detail.
Content	<ul style="list-style-type: none">• Scope of testing – at both a high level, e.g. features, and a lower level, e.g. models of behaviour• Coverage of tests versus items in scope (e.g. a requirements coverage matrix or other test model)• Test cases identifying features, pre-conditions, inputs, and post-conditions (including expected results)• Test procedures, reusable to execute selected test cases.
Sources	Stakeholders, users, requirements, designs, and specifications
Maintenance	In principle, with fixed requirements there should be one agreed version of these documents. Where requirements or scope changes occur, testers will need to adjust the documents, maintain the traceable aspects of the documentation, and provide a configuration management or change log.
Agile/Continuous Considerations	<p>The area of test definition is where the approach to agile is most markedly different from structured projects. Potentially, testers who are focusing on features as they are delivered may not create any documentation at all. This is appropriate if there is a system-wide policy or charter for feature testing. More likely, there would be a brief charter for testing each features in an exploratory test session. A charter is a plan for a short period of exploration. The charter would typically identify:</p> <ul style="list-style-type: none">• The scope of the test session – the feature(s) to cover and/or some specified functionality or behaviour of the system• The aim of the session – to explore certain aspects of behaviors, to focus on some risk or mode of failure, to apply some selected scenarios• The duration of the session is typically 45-120 minutes. The session is necessarily limited in scope, but testers are free to explore outside the scope if they see it as valuable.• A charter may focus on exploration: to learn what a feature does and identify specific behaviors worth testing. A charter may focus specifically on testing a feature, but may highlight some areas that need more attention than others.

BDD tools and stories in a format such as Cucumber or Gherkin might provide the traceability and content that test designs and procedures do. Scenarios with 'given/when/then' clauses identify pre-conditions, inputs, and post-conditions. They are referenced to a single feature, so they provide a minimal test case/procedure and are traceable – to features at least.

Tests that have been scripted to be run by tools might have intermediate documentation. Teams rely more on observation of automated tests and tables of test data used by automated scripts than documented test designs.

Test Execution (Schedule, Log)

Purpose	<ul style="list-style-type: none"> • To specify the running order of tests • To record the status of tests – run/not run and status • To provide the test execution results for reporting
Content	<p>Typically includes a test identifier, tester, date/time run, status. For tests that appear to show anomalous behaviour (optionally):</p> <ul style="list-style-type: none"> • Details of the test as run where details differ from the script • Actual vs. expected outcomes • Other observations, interpretation • Test status (defect, setup or environment anomaly, etc.) • Observation or defect report ID (where appropriate)
Sources	Test case/procedure inventory, testers
Maintenance	The schedule would change in line with the scope of testing and procedures amended, removed or added to the plan. Tests are likely to be run several times as either re-tests or regression tests. The Log should hold a complete history for all tests in scope.
Agile/Continuous Considerations	<p>If agile/continuous projects do not commit to test definition documents, they compensate somewhat by encouraging testers to keep better logs of test execution. Where testing is performed in sessions against charters, the tester is expected to keep good notes of the tests run. There are few dedicated test-logging tools that are more than notebooks, so many testers use simple text editors, note-takers, or hardcopy notebooks.</p> <p>Use logs to record all the significant activity and observations in sessions while tests are performed. A typical exploratory testing log would contain aspects such as:</p> <ul style="list-style-type: none"> • Structure of the features explored (a map of the territory) • Observations, questions relating to features explored in the session

- Models, lists, tables of test items and test ideas
- Tests run, captured in enough detail to replay them
- Anomalies found – failures, questionable behaviour, slow responses, poor user experience, etc.
- Time spent on exploration, test setup, testing, investigation, bug logging, retesting, regression testing, non-productive time
- Date/time of entry

Where testers log their session activity in note-taking or other tools, some use markup or other domain-specific language to structure their notes. These can be parsed by simple in-house tools to provide summaries of activity for use in the test report.

Tests executed by tools (whether proprietary or open source) keep logs automatically. Typically these logs can be queried by the tool or by user-written procedures.

Test Report

Purpose

- Communicate the outcome of a test stage, selected tests or a test session. This can also apply to technical requirements or non-functional test activities, in which case the content would be differ to match the test objective
- Partially inform stakeholders enabling them to make a decision on acceptance or release of a system or sub-system.

Content

- Test start/end times and duration
- Test environment, including software and system version(s) under test
- Goals and scope of testing (from test strategy, test definition)
- Narrative summary of results
- Features deemed to be working as required
- Risks deemed to have been addressed
- Outstanding tests of significance (failed or blocked)
- Features partially and not tested
- Risks partially or not addressed
- Test results detail with output from test logs etc.
- Workaround status for outstanding anomalies
- Test analyses
- Test progress and status over time
- Incident statistics

Sources	Sources include test strategy, test definitions, test log, incident reports. Much of the content of a test report will be the tools used to record test definition(s), the test log and the incident or defect log.
Maintenance	This is a snapshot in time for a test execution phase and is not maintained.
Agile/Continuous Considerations	<p>The purpose of a test report in an agile project could cover a single iteration or sprint, testing for a release or a higher-level test phase such as integration, or overall system acceptance. Even so, the purpose is unchanged.</p> <p>Much of the content of a test report will be from tools or the notes from testers. The narrative summary of results is written by a test lead or the tester for a smaller scale test phase.</p> <p>As usual in Agile, the report is likely to be less formal, and there would probably be less raw data that could form the basis of sophisticated analyses. Certainly, during the iteration(s), progress through the iteration in terms of features or stories delivered, tested and approved by the users might be recorded in a tool or a public KanBan board for example. In this way, the stakeholders are kept informed of progress throughout the iteration and there is less need for a formal report at the end of a period of testing.</p> <p>Visibility of progress is a key concern for agile teams. With regular, perhaps daily stand-ups around a Scrum board, for example, team members share their understanding of progress, question each other and agree on a position (and next steps) constantly. In this way a formal test report might never be required the team are always informed and up to date.</p> <p>If the testers are keeping written notes of their session activities, then there is no analyzable data available to present automated reports, so session and progress reporting might be presented in a public, visual way. This requires a high degree of discipline and good communication skills on the part of the testers. The test lead or manager will have to provide a correspondingly informative report to stakeholders based on verbal reports.</p>

Documentation Doesn't Have to Be a Chore

The subject of documentation in projects is a sensitive one for testers as well as other project team members. Most people regard writing documentation as a chore.

Here are some things to bear in mind in designing your documentation.



1. Documentation must have a well-defined purpose and audience. If your audience doesn't need the documentation, they won't read it. If it doesn't resonate with their own goals, they won't buy into it.



2. It is generally better to capture the record of an activity before or as you perform it. For example, TDD captures tests before code is written. Test session logs should be captured during the session and not written afterward. And so on.



3. A test logged in a notebook might be sufficient for the tester, but this format isn't easy to analyze. Perhaps a simple text log with some markup would be as fast to capture, and then you'd have something that could be analyzed by a custom tool.



4. Test procedures might not be necessary at all if testers know the system under tests well. Perhaps only a test objective or charter is necessary. Prepared test cases can be minimally documented in a spreadsheet.

Necessity is the mother of documentation. If you conscientiously prepare comprehensive documentation for your stakeholders and they don't read it, it's because they don't see value in it, or they can't find the value in it. It's better to present a stakeholder with a blank piece of paper and jointly add the topics they need to see in a document and work from there. It may be they ask for reams of content, but what they actually need is rather simple. Keep asking, 'why do they want this?'

Design your documentation with purpose; don't just provide content you think might be useful to others because you might have the data.

Putting It into Practice

Think about some of the major issues you experienced in a recent project. Perhaps some could have been avoided, or resolved more easily? Perhaps these issues were a matter of knowledge management? At any rate, for each issue you can think of, write down the issue and whether:



1. If it had been captured in a plan or process document, this would have helped.



2. If we had a checklist of strategy we could have raised the issue sooner, discussed, and agreed on a way forward.



3. The issue wasn't on the list, but could have been (and #1 or #2 above applied).



4. We did document the issue, but the management/project team did not comply with the recommendation (even if they agreed to).

If you choose #4 above, what could you have done differently?



12 Secrets of Effective Test Management

7. Plan as a Journey, Not a Task

To quote Eisenhower yet again, "Planning is everything. The plan is nothing." This sentiment is so important it bears repetition in almost every context of work in systems projects. But what does it mean to say the plan is nothing? What is the point of planning if the outcome is a worthless plan? The plan you end up with is never worthless, but Eisenhower's sentiment relates to the act of planning and its value compared with the plan. Let's look more closely at how planning works in longer structured and agile/continuous projects separately.

In a longer-term project, your business, suppliers, and internal IT staff need to know what commitment is required of them so they can schedule the availability of people and physical resources. It takes time to gather the required information to create a plan. Of all the dependencies on resources and people and the commitment and performance of suppliers as well as internal people, many things can go wrong. Some things will go wrong. Because of this, the plan, as a prediction of the future, is fraught with difficulty - things rarely go to plan in reality.

The day after a plan is published, and every day subsequently, new information comes to light and the plan needs some adjustment. As the project team follows the plan, some things go well and others don't and challenges present themselves. Perhaps this requirement isn't so good after all? Suppliers deliver late, environments or test data or tools are not ready in time. Software components are held up for various reasons and don't get delivered. Things that are assumed to work, don't. So, planning is never a one-off task - it is a continuous activity because unanticipated events cause adjustments to the plan almost every day.

Often unplanned events are treated as noise and don't get much attention. But later, some of these minor glitches become major problems. One of the challenges of waterfall projects is that this continuous adjustment is burdensome because of the change process or sometimes seen as tinkering. Projects often carry on regardless, hoping that things will be 'all right on the night'. But this is rarely the case and it has been said many times:

How does a project get to be a year late? One day at a time.

The agile approach is partly a reaction to the frustration of fixed or inflexible plans. Agility is the proven alternative to the inertia of cumbersome, staged approaches. But does this mean that agile projects don't do planning? No, what it means is that some up-front planning is necessary to structure the work, to marshal resources and to schedule at least at a high level, the release process over the coming months. But iteration by iteration, and often day by day, the overall plan is continuously adjusted to take account of events and new information that comes to light.

Planning is a continuous learning journey, not a task with a deliverable.

The Purpose of Planning

If you take a project's tasks, estimated durations, dependencies, and deliverables together, the plan that combines them into a schedule is a model of reality. If you regard the plan as a prediction of the future, you would be very wary of relying on it. But, that's what we usually do. We hope that it represents reality with a degree of certainty. You might have encountered project managers who treat their plan as their personal, and possibly delusional, reality. But we should not be so harsh with project managers. They are often motivated to create a plan and deliver against it, no matter what.

A plan is not reality; it is a model of reality requiring constant change.

The purpose of planning is to create agreement on a set of commitments, dependencies, costs, and schedule to deliver a project. A plan is an understanding between project stakeholders, suppliers, and participants. It typically includes elements like:

- What resources are required and when
- When tasks need to start and end and who will perform them
- The skills required to complete tasks
- The tools and technologies that support the plan
- The deliverables and their due dates

- The costs of effort and resources required
- The process for advancing the project/process through its stages
- The risks that threaten delivery.

Some of these aspects might be specified in a strategy, or may already be in place. For example, we may already know which suppliers are involved and what they will do for the project. The tools and technology to be used may already be in place. The people, test environments and data may be ready. In this case, the strategy defines the process, approach or techniques to be used. A **strategy** sets out how a project will be executed in principle. A **plan** identifies how the project will be executed in practice.

A viable plan depends on all the project participants knowing what they are doing and how. To achieve this, the plan and knowledge of how things will be done need to be communicated and commitment given by all involved.

So far, we've focused on project planning as a continuous activity. Now we'll focus specifically on **test planning** in projects. Test planning is very similar to project planning – it's just on a smaller scale. Naturally, the test plan must integrate with the larger project plan, because it is dependent on other project activities and deliverables, and other tasks depend on testing. Frequently, test plans are disrupted because these dependencies are not met.

Testing activities are distributed through a team, with perhaps a large inventory of test items to plan and execute tests for. This level of detail is not so relevant to the project manager's schedule. As a result, testing activity is usually managed at a local level within the test team.

The Deliverables of Test Planning

What are the deliverables of testing? Isn't it the test specifications, the tests, the results and reports? The deliverables are essentially contained in the documentation, so all we need worry about is getting the paperwork out and ticking some boxes, right? But this approach, although common, is partly to blame for testing (and testers) gaining a reputation for being expensive, flat-footed bureaucrats who have little value. After all, who reads these voluminous documents?

Suppose there is no intention of producing test documentation in an agile project. This is more than likely, so what does testing actually deliver in those situations? What value does testing have at all?

When a component, sub-system or the entire system is tested, the output is evidence of how the system behaves in some situation or context. Evidence of system behavior is collected and collated to be presented to stakeholders for them to make a decision: to fix bugs, to integrate a component, to accept or reject functionality, to release or deploy a sub-system or the system as a whole.

The key deliverable of testing is evidence of system behavior that is useful to stakeholders for decision-making.

Now, in some projects, it is essential to provide test documentation to provide a record of how a test was scoped, designed, implemented and executed. But it is the evidence of system behavior that is the ultimate deliverable of value to stakeholders.

The value of testing is the level of confidence stakeholders have in making decisions based on test evidence.

This evidence might be systematically collected, tabulated, analyzed in sophisticated test management tools; then presented to stakeholders in elegant graphical formats. Or it might be handwritten notes that a tester uses to present the testing story verbally to a product owner in a stand-up meeting. However collected and presented, the final task of testing is the handover of evidence.

The Essentials of Test Planning

When planning, it's important to consider the broad range of activities required to deliver. If you fail to include essential activities in your plan, your estimates of time and effort will be too low. After all, you'll have to perform these activities anyway, but you won't have allowed for it. As a result, the schedule might slip. In addition, if you don't consider dependencies on other people or suppliers in the plan, they might go unnoticed until they become critical. Not a good situation.

The good news is that there is a typical set of activities that apply to most testing projects, regardless of scale or methodology. These are presented below in two tables, one for **test activities**, and another for test support or **logistical activities**.

You can, of course, adapt the schedule for these activities as necessary. For example, you might find it useful to conduct exploration and modeling concurrently, or perhaps execution and reporting. The key is that your plan allocates the necessary time and resources for these activities to be completed.

Test Activities

Exploration, reading, inquiry	Reading documentation, interviewing users, exploring the existing or new system.
Modeling, exempling	Understanding the system through documented or mental test models.
Feedback, review, challenge	Using the test model, derive examples of the system in use (aka test ideas). This can be used to highlight contradictions, omissions, ambiguities in requirements to be resolved.
Test design	Development of test cases from a trusted test model.
Test execution, interpretation	Execution of tests and interpretation of test outcomes.
Reporting	Reporting on the progress through testing, and telling the testing story – what is known and not yet known – about the system’s behavior.
Logging	Reporting anomalies requiring investigation. Often these are defects requiring repair.

Test Logistics

Create test environment	Creation, configuration, and availability of test environments including necessary user accounts and permissions.
Prepare test data	Generation, selection, scrambling, anonymizing, importing, presenting, securing test data.
Prepare test documentation	Specifications, scripts, test cases as required
Create test automation framework	Create software to manage and control automated test scripts, environment builds, preparation and clean-up/tear down.
Create test automation scripts	Create scripts to execute automated tests using prepared test data, generate automated execution reports.

Resource Allocation

Having identified the activities required to deliver testing, you need people to perform them. Those people require material or physical resources to do their work.

We often use the word **resources** to refer to people on our projects. For some people, that's not comfortable. People are not things, they are human beings. In a small project it might be possible to simply use names to identify who does what. However, in larger organizations and projects, teams are involved. These might not yet be staffed, so a number of resources is simply shorthand for a number of people.

More importantly, it's not necessarily the number of people that really matters – it's their skills and capabilities. In addition, different people work in different ways, and usually at different speeds. You will need to take this into account when estimating.

Beyond people, you'll need, at different stages, various physical resources to complete your testing. These range from the mundane to the highly specific. Lacking any of these may threaten the testing mission. The table below lists some typical resources required for testing. Your list will differ considerably, no doubt. You might already be lucky enough to have access to a fully equipped, managed, dedicated test lab. If you don't, you may need to specify everything – from physical space and furniture down to the whiteboards and erasers – to define your working environment.

Example Resources

Working environment	Office space, desks, chairs, phones, paper, power, network access point
Communications	Local, corporate wires and wireless network and internet access.
Technology	Workstations, tablets, mobile phones, printers, plotters, scanners, etc.
Test environment	Probably in a data center or in the cloud somewhere.
System Under Test	The application or system under test, correctly versioned, configured and accessible.
SUT Access	System roles, access rights, accounts, passwords.

Support Network

If the people and skills you need to do the testing were under your control, projects could be much simpler! But few teams have all the skills they need, or authorization and access to the necessary physical resources.

Many projects are staffed and run in the style of matrix management. Key members of the team actually report to managers of other specialist departments and you'll get a limited commitment from them. You might need to specify a number of hours per day or scheduled times to access specialist skills. Sometimes, you'll be asked what level of service is acceptable e.g. 'high-priority requests will be responded to in thirty minutes' and so on.

Every project differs, but these are typical requirements:

Test Support Personnel

Senior stakeholders	To provide input to scoping decisions, to advise on appropriate levels of coverage and suitable test models, to sign-off documentation, approve releases, deployments, sign-off exit-reports.
Subject matter experts (business managers, users)	To provide background information on the business, the system, and business work practices; also to advise on defects.
Developers	To advise on technical matters; to be advised on developer testing as appropriate; to create utilities, tools, harnesses, drivers for testers; to diagnose and fix defects.
Operations, sysadmins	Provide and support environments, user accounts, system access, security certificates etc; to monitor, tune/optimize system resources; to perform deployments, upgrades, patches.
Network admins	Provide and support connectivity; to monitor, reconfigure, and optimize networks.
Help desk/user support	You may need to work through a help desk to reach technical resources
External suppliers	Some or all of the above, as appropriate in your circumstance.

Estimates

Estimation in software projects is tricky. Much has been written about how estimation, as an exercise in predicting the future, is difficult or even impossible. But estimates are required in all projects; and the bigger the project, the more we depend on them.

We need accurate estimates to create a schedule and allocate resources. But estimation does not give you accuracy. We can only calculate an effort or elapsed time with some level of confidence or probability. There is much dispute over whether estimation can ever be accurate enough or is even a good practice in software projects, as shown by the existence of the #NoEstimates movement.

What makes estimation so difficult?



Requirements are fallible and imprecise.



We can't control whether people are competent, conscientious and hard-working.



As requirements and software become more complex, it takes longer to develop and test. There is a non-linear relationship between complexity and effort, but no one knows what it is.



Software development is a learning process. The more you learn, the more accurate your estimate to completion. The only perfect estimate is when you're done.



You can anticipate some bad things happening, but you can't predict their likelihood or impact on your estimates with confidence.



You can't anticipate all the bad things that could have an impact.

Estimation is imprecise; foreseeable events that affect your estimate aren't measurable; some impactful events are not foreseeable.

Having said all that, there are some rules of thumb for all estimation.



The smaller the item of work, the easier it is to estimate. Break large tasks into smaller units wherever possible and roll-up the estimates to the larger task.



Estimation is based on experience. If you don't have any, find where the experience of others is relevant and can be adapted.



Your work item is unique. Look for patterns of work in other known situations where you have experience.



Ask others to estimate and compare. Discussing discrepancies exposes differences in expectations, confidence, and thoroughness.



Estimate the best-case situation. Estimate the worst-case situation. A good estimate is somewhere in-between.

Estimate today and start work; tomorrow and every subsequent day, with the knowledge gained, your estimate to completion will improve.

Dependencies, Risks, and Assumptions

Earlier, we discussed **product risk**, which can impact the ability of our product to meet user expectations. Now, we're going to focus on **project risk**, which threatens our ability to deliver the product on time and within budget.

Many things can go wrong in projects. In your planning activity, regardless of industry, technology, approach or organization, you should clarify which risks of failure in the project itself you have considered and allowed for. There are three aspects to this: **dependencies**, **risks**, and **assumptions**.

To be viable, your plan requires certain things to be in place before and during the activities that you are planning. Dependencies represent a list of human and physical resources required, and predecessor activities that must complete for your planned

activities to succeed and deliver. There are always dependencies for a test activity, whether it is a large-scale system-level test or an exploratory test session of a feature.

Dependencies cover three main aspects:

- Predecessor activities that must complete, or deliver something to allow your planned activity to start without delay or to succeed at all.
- Human resources with specified skills or capabilities that must be available to perform the work or to support the work of your team.
- Inputs to your planned activity e.g. requirements and subject-matter expertise to create a test; the system under test, environment and data to support a system test and so on.

What could go wrong with your plan? The risks of your plan failing during execution relate to things like:



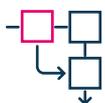
Your estimates: what could cause your estimates to be wrong? The system being more complex, or defective or incomplete or constantly changing could all affect your estimates.



People not being available or lacking experience or skills or not being fully committed to your phase of the project. These might be team members or people in your support network.



Infrastructure such as test environments, tools (or training in tools use) or office space not being available, being improperly prepared or defective.



Predecessor activities not completing on time (or being abandoned, delivering partial or defective products or without delivering at all).

For each of these risks, you need to assess in some way, the likelihood and impact, and where significant, an appropriate response or expected consequence. These responses tend to take one of these forms:

- **Assumption:** the risk is deemed low enough to ignore or regard as inconsequential. But it is on the radar, and recorded as an assumption (of availability, completeness, etc.)
- **Response:** the risk is significant, but there is some action that can reduce its impact on the plan. For example, if the system under test is partially delivered, with features missing, then the plan can be adjusted to test only those features that are available.
- **Consequence:** some risks cannot be assimilated. If the system is delivered late, then the testing cannot start until it is delivered.

Communication, Commitment, and Progress

You might produce a project plan with all tasks, participants, resources, responsibilities, dependencies, timings, effort, and costs included. Or it might be a verbal understanding between the members of an agile team. Either way, the plan needs to be communicated to everyone so all know what is required and when. More importantly, the plan is a contract between participants. If a team manager signs a plan off, it is implicitly or explicitly a commitment to fulfill their side of the contract.

The plan is a contract between participants.

Now, in less formal projects there may be no written plan or commitments at all. In these circumstances, the plan is an ongoing conversation between the members of the team. Where the team meets daily, the active tasks in the plan are discussed in real time. What are people working on? Is progress going according to expectations? What problems are people experiencing? What problems are blocking progress?

In all projects, the purpose of progress reporting is twofold. Obviously, there is a need to communicate where everyone is in terms of their current project activity. But the progress report is also a continuing periodic check that progress can be maintained and to verify that the commitment of participants can be relied on.

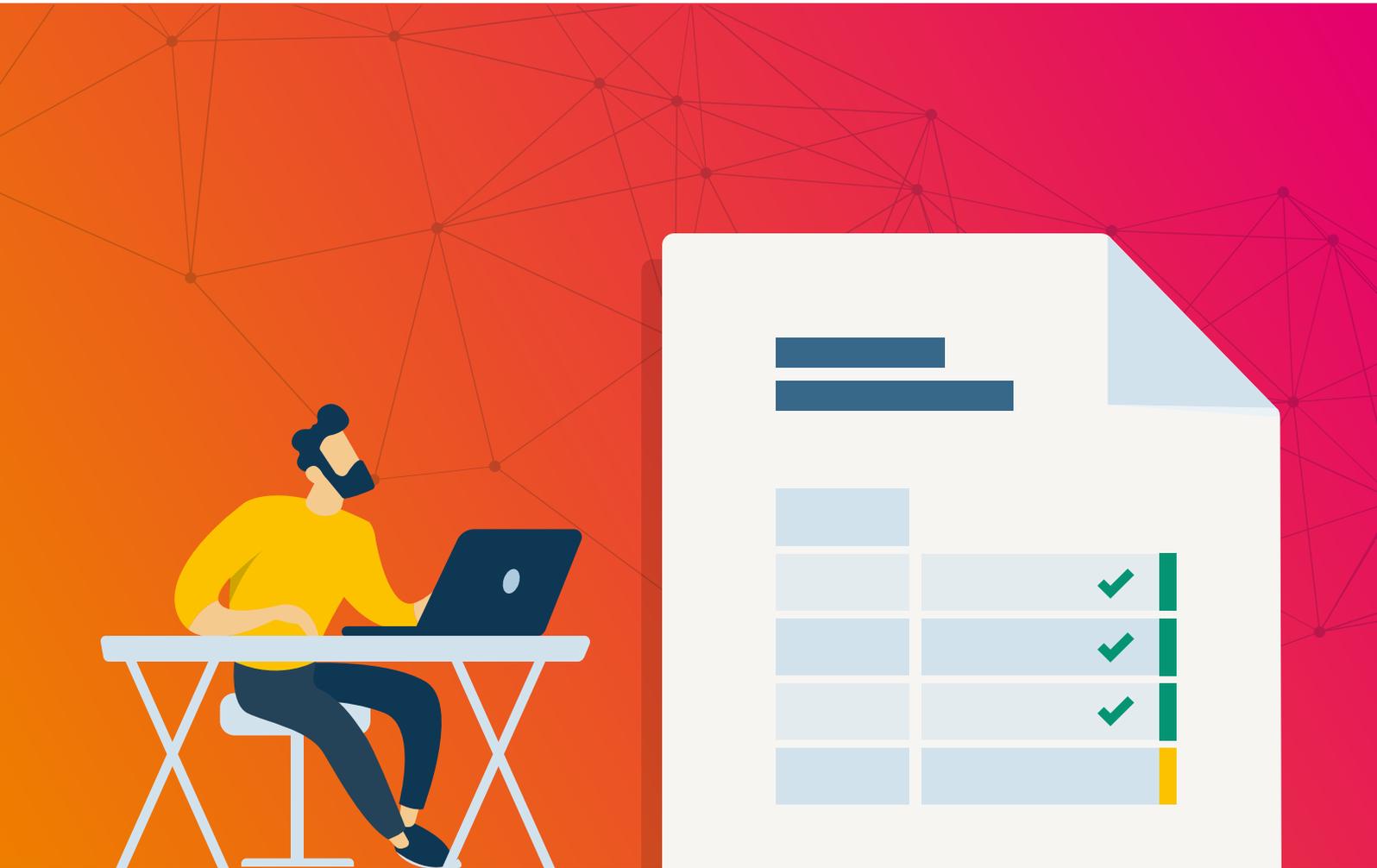
Putting It into Practice

Assume that you are responsible for planning the test design, preparation, and execution of the system test. For each of these activities make a list of inputs and dependencies, the activity(ies) required and the outputs.

Now, assume that you are the tester on an agile team following the scrum approach. Each sprint is intended to deliver features described by user stories in the Gherkin format. Make a list of inputs and dependencies, the activity(ies) required and the outputs required to test a feature sufficiently to say it is 'done'.

Consider the functionality of your current project, waterfall or agile or continuous. For a piece of functionality, whether it is a feature or subsystem or other subset of a system, suggest three ways of estimating the effort to plan, prepare and execute tests of that functionality. Where your estimation makes assumptions, list these too.

For each estimation type, list the advantages and disadvantages.



12 Secrets of Effective Test Management

8. Execute Your Plan

When we discussed planning in the previous section, it became clear that the basis of a workable plan (for any size of project or activity) is that the resources – people and physical – are available, with the required capability and capacity, in a timely manner. The time has come to do the job. Now, are you ready? The larger the test, the earlier you may need to prepare, and only you can figure out when you need to start preparing. But you are dependent on multiple factors. Some are out of your control.

There are four keys to the successful execution of your plan:

- 1 People – is your team ready?
- 2 Environments – do you have the technologies, data, devices, interfaces to implement meaningful tests?
- 3 Knowledge – have you prepared your tests at an appropriate level of detail, or are your team ready and able to explore and test the system in a dynamic way?
- 4 System Under Test – is the software or system you are to test available?

The first three aspects are either under your control, or you have the means to monitor, manage and coordinate action to provide people, environments and knowledge. At least you should be able to assess where you are with respect to being ready to start and set expectations; or adjust the schedule to account for the lack of people, environments or knowledge. The system under test is another matter.

If the system under test is delivered late, then you cannot make any meaningful start to the testing. Anyone who has tested systems has experienced late delivery of the system to be tested. At almost every level from components to entire systems, developers encounter problems and delivery is either delayed or incomplete. In most circumstances, where a time period has been assigned to conduct the testing, the deadline does not move, and the testing is squeezed.

Partial and Incremental Delivery

Although the complete system cannot be delivered for testing, some functionality – a partial system – can be delivered on the promise that later releases will contain the remaining functionality. At any time, the status of the features in a release will be one of the following:

-  Completed as required: These features are testable – at least in isolation.
-  Incomplete: Features omit functionality and/or are known to be faulty.
-  Missing: Postponed to be delivered in a later release.

With regards to the features that are available, it may be possible to test them in isolation, but they may depend on data created by other features not yet available, which makes testing more difficult perhaps. Features may be available, but the output of those features cannot be verified by other features not yet available so examination of the before- and after-test database is required. It is almost certain that your end-to-end tests requiring testable strings of features will be mostly blocked.

In almost all respects, system level testing of partial systems is severely hampered.

Defending Testing

Your team must make progress. If the system is not available or is only partially available to the team, you'll have to manage expectations and defend your plan. Your plan for testing depends on the system being available, so your plan must change if it isn't. You may not document formal entry criteria, but the message is the same:

Entry Criteria are planning assumptions – if these criteria are not met, your planning assumptions are wrong, and the plan needs to be adjusted.

Whether you are waiting for the system to be delivered or you have access to a partial system, you will run out of useful things to do quite quickly. Then you will have a difficult conversation with your product owner or project manager. If the deadline for completion

doesn't move, then you will be forced to do less testing. Some features may be tested less, or de-scoped from testing altogether.

The manager may believe you can make up time, later, but this is hardly ever achievable in practice.

The testing time lost because of late or partial deliveries cannot be recovered through 'working harder'.

Why is testing delayed? There are many possible causes, but they tend to follow one of these patterns:

- Environments cannot be configured in time. People started late, were too busy or not qualified to create a meaningful test environment. What is available is partial, mis-configured or incomplete.
- Delivery is delayed because the scale of work was underestimated.
- Delivery is delayed because the software is buggy, or difficult to test and fix.
- Delivery is delayed because the development team lacks the skills, experience or competence in the business domain or the technologies they are using.
- Delivery is delayed because development started late.
- Delivery is delayed because requirements keep changing.

Excluded from the list above are acts of God and other external factors beyond the control of the project team. If the project manager insists that the deadline for testing does not change, and the scope of testing is also fixed, you have a significant challenge. In every case above, the causes of late delivery argue for doing *more* testing, not less.

If the development work is underestimated, then the testing probably is too. If the software is buggy, testing will take longer. If developers lack skills, then the software will likely be poor and take longer to test and fix. If developers started late (why?) and the scope is unchanged, why should testing be reduced? If requirements change, it's likely your plans are wrong anyway – working to a bad plan inevitably makes life harder.

*How many of the common causes of delayed delivery suggest that less testing is required?
None of them. Defend your plan.*

Reporting Success and Failure

Effective testing requires curiosity, persistence, and a nose for the problem. Your goal is to stimulate failures along with the evidence required to trace those failures to defects so that they can be fixed. Although finding (and fixing) defects is good for the quality of the product, communicating defects often feels like you are giving bad news to someone. This could be a developer who has made a mistake somewhere and now has a defect to fix. But you could also be reporting to stakeholders that some critical functionality does not work correctly, the system is not ready and/or delivery will be delayed.

No one likes to give bad news. It's a natural feeling to be reluctant to upset other people, especially close colleagues. But the sense that your news is good or bad is not a feeling the messenger should be concerned with. Obviously, defects are always bad news for someone, but the role of testing is not to be judgemental in this way.

In some respects, the tester is like a journalist, seeking out the truth. In the same way a journalist tells a news story, you are telling the testing story that reveals what you discovered as you test a system. The truth – the news – may be good or bad, but your responsibility is simply to seek out both the problems and successes as best you can. You are attempting to discover what a system does and how it does it. Whether this is good or bad news is for others to judge.

At the very end of a project, large or small, the goal is to deliver to production with few problems outstanding. Ultimately, you want all your tests to pass, but the journey to success is hampered by test failures which need to be investigated and resolved. Your tactical goal is to find problems fast, but your ultimate goal is to have no problems to report.

As a tester, you need to behave much like an investigative journalist – looking for the story with a critical and independent mind. Then you will be doing a good job for your project and stakeholders.

Coverage Erosion

Despite the coverage target(s) you have at the start of testing, several factors conspire to reduce the coverage actually achieved. **Erosion** is an appropriate term for this because it truly reflects the little by little reduction of the scope of planned tests, followed by the inevitable realization that not all the planned tests can be executed in the time available.

Coverage erosion can occur prior to testing for several reasons:



Test plans identify the risks to be addressed and the approach to be used to address them. Plans usually assume a budget for testing – which is always a compromise and may not be sufficient.



Poor, unstable or unfinished system requirements, designs, and specifications make test specification harder. Coverage of the system is compromised by a lack of specification detail.



The late availability or inadequacy of test environments make certain planned tests impractical or not meaningful. Larger scale integration testing may be impossible to execute as planned because not all interfaces or interfacing systems can be made available.



Performance testing might be compromised because environments lack scale or don't reflect production.



Late delivery of the software under test means that, when deadlines remain fixed, the amount of testing in scope must reduce.

Likewise, coverage can erode during testing for several reasons:



The software to be tested is of poor quality. The most basic tests might fail, and the faults found could be so fundamental the developers need more time to fix than anyone anticipated. If testing is suspended because the software quality is too poor, you'll be running late. Some tests will be de-scoped if the deadline doesn't move.



Where more faults occur than anticipated, the fix and re-test cycle itself will take more time and you'll run out of time to complete all your tests.



When time does run out, and the decision to release is made, not all of your testing will complete. Either some tests were never reached in the plan or some faults that remain outstanding block the completion of failed tests. Where the go-live date does not move, this is the classic "squeeze" of testing.

Dealing with test coverage erosion is one of the challenges the testers face in all projects. Things never go smoothly, and where the testing is under pressure, reducing the time for testing (and coverage) is usually the only option to keep the project on track. It is not wrong to reduce the amount of testing; it is only wrong to reduce the testing arbitrarily. Consequently, when making choices on which tests to cut, the impact on your test objectives and the risks to address need to be reviewed. You might have some awkward conversations with stakeholders to get through.

Where the impact is significant, you may need to facilitate a meeting of those who are asking for the cuts (typically project management) and those whose interests might be affected by the cuts (the stakeholders). Your role is to set out the situation with regards to tests completed, the current known state of the tested system, the tests that fail and/or block progress, and the amount of testing remaining to be done.

Your plans and models articulate the original scope of testing and are critical to helping stakeholders and management understand the gaps and outstanding risks and to make the decision to continue testing or to stop.

Incident Management

Once the project moves into System Testing and Acceptance Testing stages, it is largely driven by the incidents occurring during test execution. Incidents trigger activities in the remainder of the project; and incident statistics can sometimes provide a good insight as to the status of the project. When categorizing incidents, we need to think ahead to how that information will later be used.

An incident is an unplanned event that occurs during testing that may have some bearing on the successful completion of testing, the acceptance decision or the need to take some other action.

We have used a neutral term for these unplanned events – incident. But these events are often referred to using other terms; some are more neutral than others. Tests that fail might be referred to as observations, anomalies or issues – neutral terms that don't presume a cause. But sometimes, problems, bugs, defects or faults are used – which presumes the system is faulty. This might be a premature conclusion and these labels can mislead.

We suggest you reserve the terms bug, defect, or fault, for the outcomes of the diagnosis of failures in the construction of the system under test which usually generate rework for the development team.

There are two types of incidents:

- 1 Failure of the system:** the system does not behave as expected in a test i.e. it fails in some way or appears not to meet some requirement.
- 2 Interruption to or undermining of testing or tests:** some event that affects the ability of testers to complete their task, such as loss or failure of the test environment, or data or interfaces or supporting, integrated systems or services, or some other external influence.

Handling Type 2 Incidents

Type 1 incidents are often of the most direct concern because they undermine confidence in the system's quality. Some organizations do not treat type 2 incidents

as incidents at all. Interruptions are part of the rough-and-tumble of projects reaching a rather chaotic conclusion. Undermined tests – those where the environment or test set up is wrong – might be blamed on the test team. Perhaps they did the setup wrong or at least failed to check it before testing.

In both cases, progress through the testing is affected, and if you are managing the process, you are accountable for explaining delays. For this reason, you should either capture these events as incidents, or ask the team to keep a testing log and record environment outages, configuration problems or lack of appropriate software versions to test. If you don't, you'll have difficulty justifying delays in progress and it could reflect badly on you and the team.

To Log Incidents or Not?

With the advent of agile and continuous delivery approaches, the traditional view of incident management has been challenged. In staged projects, incidents are treated as potential work packages for developers which are approved according to severity and/or urgency. There is a formal, often bureaucratic process to follow whereby incidents are reviewed, prioritized and actioned by the development (or other) team. Sophisticated incident management tools may be involved.

In smaller, agile teams, the relationship between tester and developer is close. The whole team might meet daily to discuss larger incidents. Often bugs are detected, diagnosed, fixed and retested without any need to log an incident or involve others in the team or external business or IT personnel. Most bugs are fixed in an informal way. More serious bugs might be discussed and absorbed into the work for a user story or bundled into dedicated bug fix iterations or sprints.

We discussed the purpose and need for documentation in an earlier article. That discussion is appropriate for incidents too. The team needs to consider whether an incident tool and process is required, either for within the team or for communicating with outside groups. Larger teams tend to rely on process and tools to manage incidents, for several reasons:

- To ensure that incidents aren't forgotten
- To ensure that serious problems are reviewed by stakeholders and project management
- To capture metrics that might be valuable during and after the project.

Separating Urgency from Severity

Whatever incident management process you adopt, we advocate assigning both a priority and a severity code to all of your incidents.

Priority is assigned from a testing viewpoint. It influences when an incident will be resolved. The tester should decide whether an incident is of high or low priority, or some degree in between. The priority indicates the urgency of this fault to the testers themselves and is based on the impact that the test failure has on the rest of testing.

Severity is assigned from a user's viewpoint. It indicates the acceptability of a defect. The severity reflects the impact of that defect on the business if it wasn't fixed before delivery. Typically, a severe defect makes the system unacceptable. If a defect is of low severity, it might be deemed too trivial to need fixing before go-live, but could be fixed in a later release.

A high priority incident stops all testing, and usually the project.

The important thing to bear in mind with incident classification schemes is that not every urgent incident is severe and that not every severe incident is urgent.

Managing the End Game

We call the final stages in our test process the **end game**. This is because the management of the test activities during these final, possibly frantic and stressful days requires a different discipline from the earlier, seemingly much more relaxed period of test planning.

The purpose of testing is to deliver information to stakeholders to enable them to decide – to accept, to fix, to reject, to extend the project or abandon it entirely. If you have a shared understanding of the models to be used for testing, it becomes much easier to explain what “works” (with respect to the model), where things fail to work, and the risks of these failures. It is for the stakeholders to use this information to make their decisions – guided by you.

One of the values of the risk-based test approach is that where testing is squeezed, late in a project, we use the residual risk to make the argument for continuing to test, or even adding more testing. Where management insists on squeezing the testing, testers should simply present the risks that are being traded. This is much easier when an early risk assessment has been performed, used to steer the test activities, and monitored throughout the project. When management is continually aware of the residual risks, it is less likely that they will squeeze testing in the first place.

Putting It into Practice

On the following pages, you'll find one list of incident priorities and another for incident severities. Notice that the priorities map closely to the impact on testing progress. The severities map to the effect on end users, using the terms of a risk assessment.

Compare these to your current priority/severity scheme.

- Do you separate priority from severity in your incident management?
- What differences do you see between the example tables and your own?

Incident Priority	Description
0	Critical: all streams of testing for this stage (e.g. System Testing) are halted
1	Urgent: one or more streams of testing are halted for this stage, but testing can continue in other streams
2	Non-urgent: no streams of testing for this stage are halted, but modifications or compromises to other tests may be necessary
3	Minimal: no effect on test execution other than incorrect results
4	Improvement: testware to be corrected / extended for future runs

Incident Severity	Description
0	Not negotiable (or worth evaluating further): problem absolutely must be fixed
1	Major: affects a large and/or particularly important part of the functionality or non-functional characteristics. The release overall could not be allowed to go live with this incident outstanding
2	Significant: affects enough of the functionality or non-functional characteristics to have an adverse effect on the business, but they could be worked around. The release overall could go live if workarounds are adequate
3	Minimal: no effect on test execution other than incorrect results. Minor: the application does not conform to its specification, but the release overall could go live without unacceptable adverse effect
4	Enhancement: the application does conform to its specification, but still may not meet business needs



12 Secrets of Effective Test Management

9. Test as a Team

In recent years, responsibility for testing has been distributed. Rather than dedicated teams owning testing, both small and large teams are encouraging users, analysts, developers, and testers to redistribute responsibility for testing to support better collaboration. Some test activities and responsibilities move to the left, so this change of approach is commonly called **Shift-Left**.

Shift-Left can mean developers take more ownership and responsibility for their own testing. It can also mean testers get involved earlier, challenge requirements, and feed examples through a Behavior-Driven Development (BDD) process to developers. It can mean users and BAs together with developers take full responsibility for testing. It can mean no test team and no testers. All these configurations are possible – there is no one true way of course.

Shift-Left isn't new. For many years testing advocates have preached the mantra "test early, test often". As long ago as 1993 we suggested that all artifacts in a staged process – both documentary and software – can (and should often) be tested.

Shift-Left is mostly about bringing the thinking about testing earlier in the process.

Although Waterfall was the dominant life-cycle approach at the time, the number or duration of stages are not what is important. The underlying principle was that *the Sources of Knowledge that provide direction for the design and development of software should be challenged or tested*. In a staged project, this might involve formal reviews. In an Agile project, the tester (or developer or BA or user) can suggest scenarios (examples) that challenge the author of a requirement or story to think through concrete examples and discuss them before any code is written.

These are the main changes involved in the shift-left phenomena:

1. The Behavior Driven Development (BDD) approach has allowed developers, users/BAs, and testers to engage around what might be called business stories. Test-Driven Development has been used by many, but not all, developers for 15 or more years. BDD is being adopted more widely as it encourages better collaboration in agile teams as well as introducing tools that can be used by developers. It is a less onerous Test-First approach.

2. Continuous Delivery (CD) has been around for 5-10 years and its roots are in the highly automated build and release automation approaches pioneered by large online businesses. It is now adopted by most organizations with an online presence.
3. CD systematized and accelerated the release process through automation. But it then highlighted the delays in production deployment and infrastructure change that had previously been masked by slow build, test and release processes. **DevOps** is a cultural and mindset change whereby developers collaborate much more closely with operations staff. Right now, new tools appear almost daily and vendors promote DevOps as the 'next big thing'. It is a very hyped and dynamic situation.
4. SMAC, or Social, Mobile, Analytics and Cloud, represents a shift in the way organizations are managing business and systems change in the mobile space. Experimentation in business, implemented as production systems' changes, are monitored at a detailed level. The "Big" data captured is processed and business decisions are made based on the analytics obtained.

Frequent experimentation with production systems enables business innovation 'at the speed of marketing'. Experimentation is at the heart of what seems to be the most important current bandwagon: **Digital Transformation**.

Testing is an Activity, Not a Role

Shift-Left changes the role of testers. The redistribution of testing triggered by the shift-left approach makes it clear that testers are not solely responsible for testing; testers don't own testing anymore. If you think about it, they never really did own testing, but there was a tacit understanding in projects that whatever the rest of the team, in particular developers, did in testing, the testers provided a safety-net. If developers were squeezed on time and delivered but reduced their testing, the safety-net was provided by the testers.

Developers are adopting better test practices and visibility into their work. Good component-level or unit testing has specific goals that are distinct from system testing, so the scope (or amount) of system-level testing could be reduced. The correct distribution of testing goals and testing to meet those goals is the primary purpose of the test strategy. Unfortunately, until recently, developers were not held accountable very effectively, so they relied on late, system testing to compensate.

Shift-left makes developers more accountable

Overall, the responsibility for testing is redistributed, so the tester's role changes. Testers might do less tactical testing, but their strategic contribution increases. Testers might own the test strategy: they challenge requirements, consult with stakeholders, and forge a closer relationship with developers to support better developer testing and test automation.

The Tester's New Role

Shift-Left implies that whenever it is possible to provide feedback that will help the team to understand, challenge and improve goals, requirements, design or implementation – that feedback should be provided. This behavior comes as second nature to some, but not all, testers. Users, BAs, developers, and the entire team should be ready to both provide and accept feedback in this way. Sometimes there is resistance, but the overall aim is to run a better, more informed project – that's all.

The easiest way to summarize this behavior is, "get involved as early as possible." Engage in the discussion and collaborate on ideas, requirements and every stage where the outcome of that stage has a bearing on the value of the final deliverable of the project. Put simply, the tester challenges sources of knowledge, whether these sources are stakeholders, users, developers, business stories, documents or received wisdom.

The most common approach is to challenge through example. At all stages, these examples can be regarded as tests. They might be discarded quickly after use, or be codified into test automation or manual checks. These examples could just be used tactically to point out flaws in peoples' thinking, or be provided to developers as ideas or seeds for developer tests. They might also be used as a coaching aid to help users or developers to see how better tests can be created.

Think of your software project as a knowledge acquisition process. This knowledge is gathered throughout the project and often evolves over time. The goal of shift-left is to assure this knowledge through challenge and testing close to its source and to ensure where possible that it is trusted before it is frozen in code.

Shift-left takes the test-first philosophy further. Agile has always promoted collaboration and rapid feedback. Shift-left could be viewed simply as a concerted rapid-feedback approach.

Agile Test Interventions

The shift-left approach is fundamental to a test strategy for agile projects. In an agile context, test strategy can be viewed as a series of test interventions. There are critical moments in all projects where opportunities to gather and give feedback present themselves. The tester focuses on these critical moments and is ready to contribute at those times.

In your own projects, identify the critical moments where intervention is possible, and the choices that you and your team can make. For example, should the tester write unit tests for developers? Should you provide examples to get them started or coach them to improve their testing ability? Only you and your team can decide this.

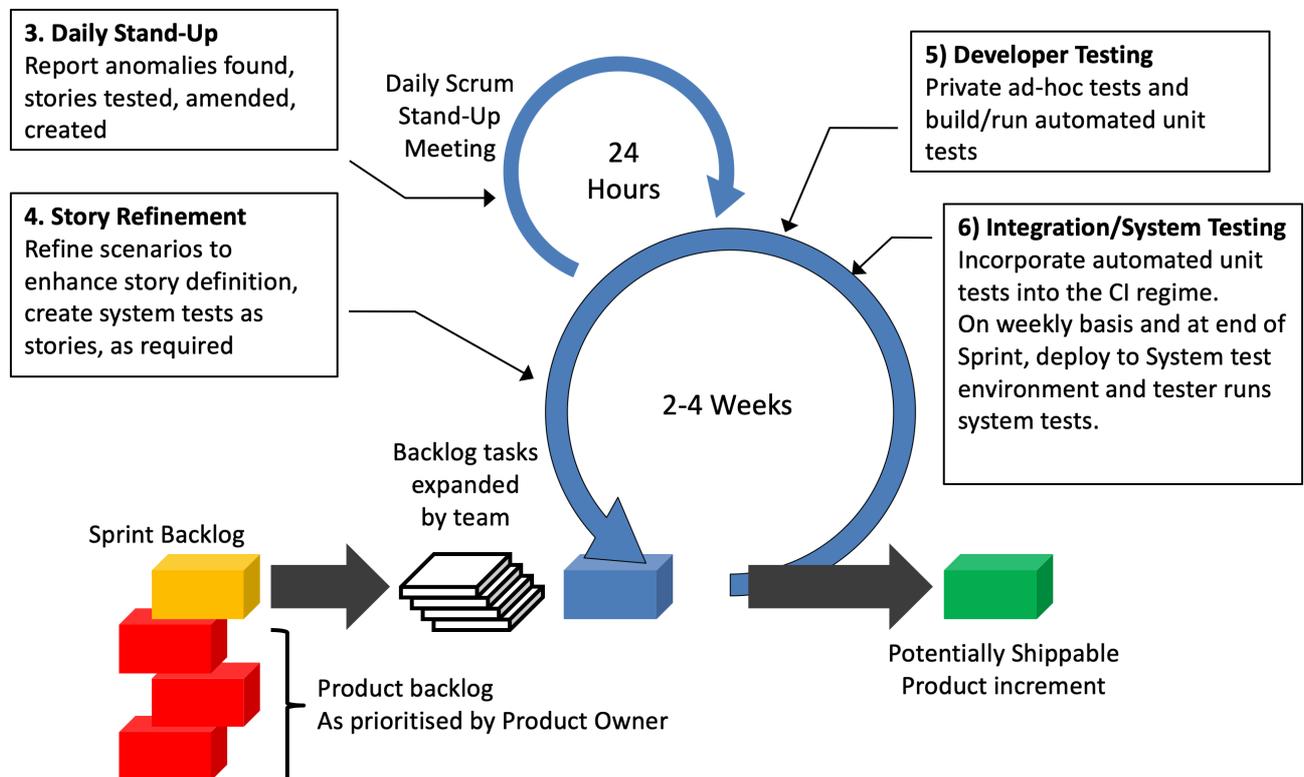
We'll use a typical Scrum process to demonstrate how test interventions can be positioned. In the table below, you can see the typical intervention types. You might have more or less interventions active at different points in your own unique process.

Step	Level	Activity	When?
1	Project	Story Challenge	As stories are added to the Product Backlog
2	Project	Story Definition	As stories are added to a Sprint Backlog
3	Sprint	Daily Stand-Up	Once per day during the Sprint
4	Sprint	Story Refinement	Occurs throughout the Sprint as new information emerges
5	Sprint	Developer Testing	Occurs throughout the Sprint as the developer codes the stories
6	Sprint	Integration (and incremental System) Testing	During and at the end of each sprint, including the final sprint
7	Project	System Testing	At the end of each sprint, including the final sprint
8	Project	User Acceptance Testing	At the end of each sprint, including the final sprint
9	Project	Non-functional Testing and Pre-Production Testing	As needed.

Sprint-Level Interventions

A project usually has multiple sprints. The diagram below shows the four sprint interventions that are repeated for each sprint:

- The **daily stand-up (3)** is an opportunity to report progress, raise concerns, identify risks, or discuss questions raised and answers received during the sprint.
- **Story refinement (4)** and contributions to **developer testing (5)** are day-to-day activities that occur as part of discussions with users, analysts and developers.
- The tester incorporates developer and new system tests into a growing **collection of tests to be automated (6)**.



Identify the critical moments, propose your contribution, and negotiate with your team. You offer more test leadership and guidance rather than volunteering simply to take on responsibility for the testing work. It will be much easier to demonstrate your value to the team if you take this approach

Relationships with Developers

In some organizations, the relationship between developers can be mistrustful, blame-worthy and antagonistic. At worst, the relationship is toxic: developers don't test much at all and testers are regarded as the servants of developers. Testers adopt what might be called co-dependent behaviors and act like victims. This situation is exactly what the shift-left approach aims to avoid.



There are many metaphors that have been used to describe a good developer-tester relationship. We'll use a pilot-navigator style of working to illustrate how the comparable developer-tester relationship can work. But first, let's look at a dysfunctional situation.

The navigator doesn't get on the plane, but waves to the pilot as they take off. The navigator travels by bus, separately but slowly. Eventually, the navigator arrives at the destination, but, after some time, the plane is found to have flown in the wrong direction, crashed into a mountain and everyone is dead.

No one would fly a plane without the navigator on board. Surely the navigator should be on the plane? Imagine how pilots and navigators work together in reality.

- The pilot cannot fly the plane without a navigator.
The navigator cannot fly the plane.
- The pilot and navigator agree to the flight plan before the journey begins.
- The pilot takes off and flies the plane.
- The navigator tracks the course and compares it to the flight plan and/or final destination taking into account adverse events, in particular the weather.
- The navigator looks for variances, plots a new course if necessary, and notifies the pilot who makes adjustments to the flight path.

The pilot/navigator relationship is comparable to the programmer/tester relationship. Separating the developers and testers into separate teams working sequentially makes no sense either – and yet, that is what we have conventionally done for the past thirty years or so – particularly in larger, longer-term projects.



Shift-left redistributes the thinking about testing and effectively brings it forward. Testers act as full partners to the developers, just like navigators do with pilots:

- The tester and developer jointly collate information on requirements.
- The tester challenges requirements through example (potential or actual test ideas).
- The developer thinks about implementation, using examples to provide direction for their design.
- The tester, developer, stakeholders, users, and analysts agree on a shared understanding of a trusted requirement.
- The tester is in constant communication with the developer, discussing requirements change, risks of failure, and how testing can show features working or expose failures.
- The tester looks at how the feature being worked on will integrate at both a technical and user-journey level, and how integration can be tested.
- The tester looks outside the immediate feature the developer is working on to identify future challenges, risks, changes, and uncertainties.

The ideal developer-tester relationship just described doesn't happen automatically. It has to be worked out by the team. You can think of each of the activities above as interventions but interventions are not always comfortable for all parties. Give this approach the best chance of succeeding by discussing each intervention type with your partners at the very beginning – when you first know that you'll be working closely together.

Interventions, like good user stories, are triggers for conversations.

Each intervention type requires assent from both sides that it is a valid thing to do. To be comfortable, each has to trust the other that there is a good reason for asking questions, raising issues, challenging requirements or understanding in stand-up meetings, planning sessions or retrospectives. For software teams to be productive, they must communicate and collaborate.

Challenges of Distributed and Outsourced Teams

In the discussions above, it's assumed everyone works on the same team and are co-located. When software development or testing is outsourced and/or offshored several negative factors come into play. The table on the following page shows three typical factors to consider.

Physical (and temporal) Separation	Teams can be distributed in an office, different buildings, towns or countries and time zones. Communication is disrupted, the channels are constrained and less information flows.
Different Motivations	The supplier team works for an organization that is being paid to do the testing work. The team's motivation is ultimately to make a profit. On fixed-price contracts, the pressure is to work quickly. On time and materials contracts the temptation is to spin out the work.
Cultural Differences	National and cultural differences can be significant. These can sometimes take time to acknowledge and make an allowance for.
Company/Corporate Culture	Company cultures differ too – companies tend to partner best with companies of comparable size, flexibility, and formality. Companies are more or less cautious when it comes to privacy, security, confidentiality, and so on. Companies have different leadership styles and the difference between government and commercial organizations also takes some getting used to.
Suppliers work to Contracts	Your supplier has no loyalty to your project stakeholders or their business goals. They work to the rules specified in their contract. Make sure that your contracts identify all responsibilities on both sides, reward good behavior, and penalize bad behavior.

Some companies augment existing teams to build up capabilities for larger projects. Or they may hire specialist testing companies rather than rely on contractors or internal user staff to do the testing. In other situations, all development and/or testing can be handed off to suppliers. In all cases, the client company needs to manage their suppliers. This does not mean that writing a restrictive contract with severe penalty clauses will suffice.

When things go wrong, you want rapid responses and collaboration from your supplier; you don't want them hiding behind legal clauses in contracts. The keys to success are:



If you don't manage your supplier, they will manage you. (If you are not full partners, and leave the supplier to define terms, the supplier will hold all the cards).



The goal is to define a good working relationship at all levels – stakeholder and manager as well as practitioner.



Contracts should be worded to identify all responsibilities on both sides, with appropriate measures, thresholds, stage payments and acceptance criteria defined to encourage good behavior (on both sides of the agreement).



Agreements should encourage openness and buy-in to the shared goal of project success.

Putting It into Practice

Good developers are usually fond of rules of thumb that guide them when they are writing code. Here are three examples:

- Don't Repeat Yourself (DRY) – rather than write the same functionality in several different places in the system, modularize the code into dedicated functions which you can reuse and share.
- Single Source of Truth (SSOT) – for every piece of information collected by a system, store it once and once only.
- Keep it Simple, Stupid (KISS) – break complex tasks into simpler tasks; each function in your code should do just one, simple, self-contained task so it is understandable.

For each of these three rules, make a list of benefits of adopting the rule. If you can think of adverse side effects, then add these to the right-hand column. You might find it easier to pair with a programmer and to do some research on these rules to complete the table. The adverse consequences of adoptions of each rule should give you some ideas for what the risks could be but also what tests you might apply in each case.

Final Thoughts



What relationship do you have as a tester with your developers?
Or, if you are a developer, what is your relationship with testers?



Ask your partners in development or testing how they would describe your relationship. Compare notes!



Do you have the same relationship with each person or is each relationship unique? Why do you think they differ (or are the same)?



Think of metaphors or analogies for the developer-tester relationship (for good and bad relationships). How many can you suggest?



What is your favorite collaboration metaphor?
Does it match your working relationship(s)?



12 Secrets of Effective Test Management

10. Test Performance

The quality of a web site or mobile application includes attributes such as functionality, performance, reliability, usability, security and so on. In this section, we'll focus on three particular objectives that we'll call Performance Testing. These objectives are:



Responsiveness: the service must be responsive to users while supporting the loads imposed upon it.



Stability: the service must be reliable and/or continue to provide a service even when a failure occurs if designed to be resilient to failure.



Manageability: the service must be capable of being managed, configured or changed without a degradation of service being noticeable to end users.

In all three cases, we need a simulation of a user load to conduct the tests effectively.

The responsiveness of a site is directly related to the resources available within the technical architecture. As more customers use a service, fewer technical resources are available to service each user's requests and response times will degrade. Obviously, a service that is lightly loaded is less likely to fail. Much of the complexity of software and hardware exists to accommodate the demands for resources within the technical architecture when a site is heavily loaded. When a site is loaded (or overloaded) the conflicting requests for resources must be managed by various infrastructure components such as server and network operating systems, database management systems, web server products, object request brokers, middleware and so on. These infrastructure components are usually more reliable than the custom-built application code that demands the resource but failures can occur in either.

Infrastructure components fail because application code (through poor design or implementation) imposes excessive demands on resources. The application components may fail because the resources they require may not always be available (in time). By simulating typical and unusual production loads over an extended period, testers can expose flaws in the design or implementation of the system. When these flaws are resolved, the same tests will demonstrate the system to be resilient.

On all services, there are usually a number of critical management processes that keep the service running smoothly. It might be possible to shut down a service to do routine maintenance, backups or upgrades for example outside normal working hours. But most online services operate 24 hours a day. These procedures need to be tested while there is a load on the system to ensure they do not adversely impact the live service.

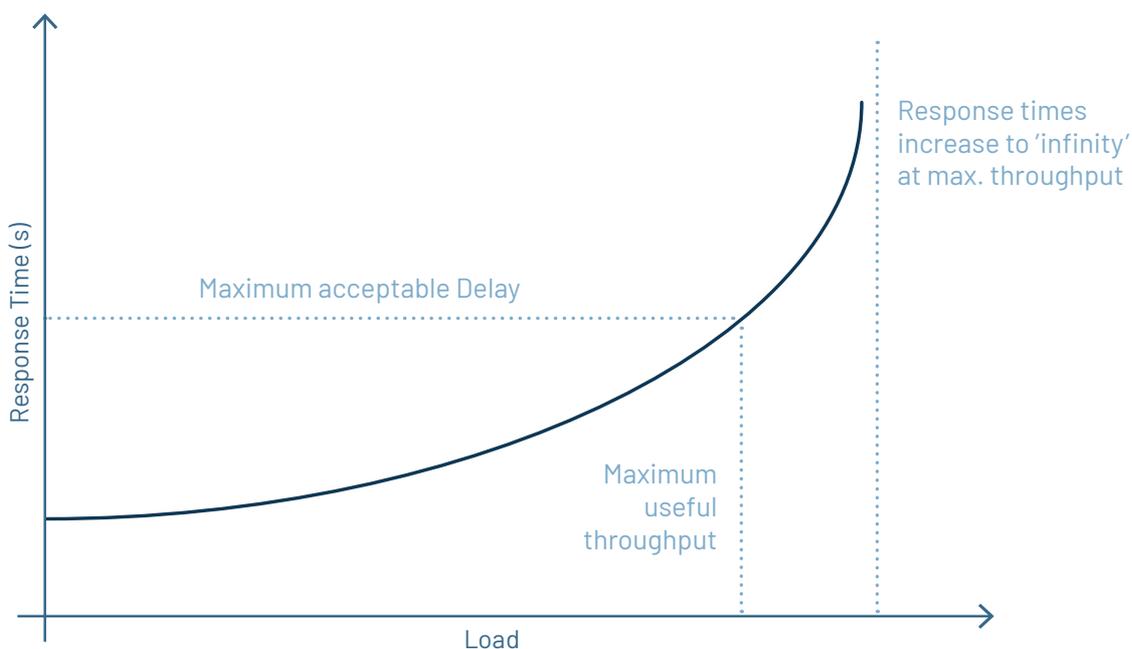
What is Performance Testing?

Here is performance testing in a nutshell:

- Performance testing consists of a range of tests at varying load where the system reaches a steady state (loads and response times at constant levels).
- We measure load and response times for each load simulated for a 15-30 minute period to get a statistically significant number of measurements.
- We monitor and record the vital signs for each load simulated. These are the various resources such as CPU and memory usage, network bandwidth, I/O rates. etc.

We plot a graph of these varying loads against the response times experienced by our “virtual” users. When plotted, our graph looks something like the figure below.

Graph: Response time/load

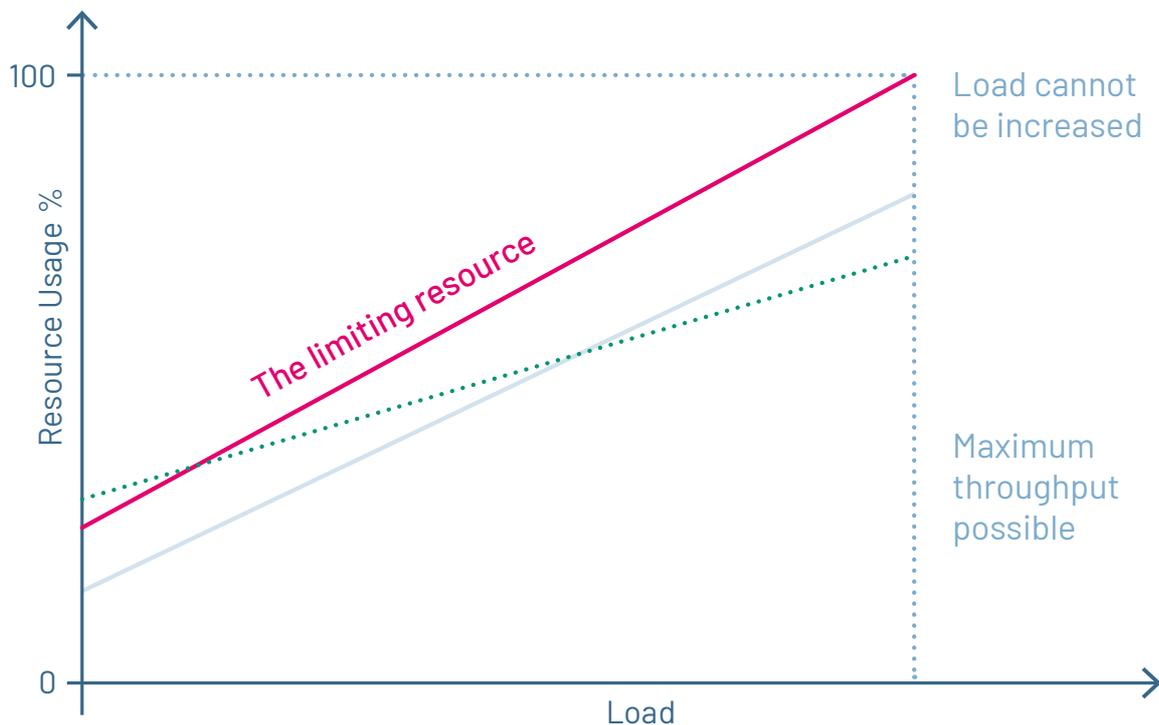


At zero load, where there is only a single user on the system, they have the entire resource to themselves and the response times are fast. As we introduce increased loads and measure response times, they get progressively worse until we reach a point where the system is running at maximum capacity. It cannot process any more transactions beyond this maximum level. At this point, the response time for our test transactions is theoretically infinite because one of the key resources of the system is completely used up and no more transactions can be processed.

As we increase the loads from zero up to the maximum, we also monitor the usage of various resource types. These resources include server processor usage, memory usage, network bandwidth, database locks and so on. At the maximum load, one of these resources is used up 100%. This resource is the limiting resource, because it runs out first. Of course, at this point response times have degraded to the point that they are probably much slower than would be acceptable.

The graph below shows the usage/availability of several resources plotted against load.

Graph: Response usage/load



To increase the throughput capacity and/or reduce response times for a system we must do one of the following:



Reduce the demand for the resource, typically by making the software that uses the resource more efficient. This is usually a development responsibility.



Optimize the use of the hardware resource within the technical architecture. By configuring the DBMS to cache more data in memory or by prioritizing some processes above others on the application server, for example.



Make more resource available. Normally by adding processors, memory or network bandwidth and so on.

Performance testing needs a team of people to help the testers. These are the technical architects, server administrators, network administrators, developers, and database designers/administrators. These technical experts are qualified to analyze the statistics generated by the resource monitoring tools and judge how best to adjust the application, to tune or upgrade the system.

If you are the tester, unless you are a particular expert in these fields yourself, don't be tempted to pretend that you can interpret these statistics and make tuning and optimization decisions. You'll need to involve these experts early in the project to get their advice and buy-in and later, during testing, to ensure bottlenecks are identified and resolved.

Performance Testing Prerequisites

We can define the primary objective of performance testing as:

"To demonstrate that the system functions to specification with acceptable response times while processing the required transaction volumes on a production sized database."

Your performance test environment is a test bed that can be used for other tests with broader objectives that we can summarize as:

- Assessing the system's capacity for growth
- Identifying weak points in the architecture
- Tuning the system
- Detect obscure bugs in software
- Verifying resilience and reliability.

Your test strategy should define the requirements for a test infrastructure to enable all these objectives to be met.

There are **five prerequisites** for a performance test, as described below. If any of these prerequisites are missing, be very careful before you proceed to execute tests and publish results. The tests might be difficult or impossible to perform, or the credibility of any results that you publish may be seriously flawed and easy to undermine.

1 Quantitative, Relevant, Measurable, Realistic, Achievable Requirements



As a foundation to all tests, performance requirements (objectives) should be agreed prior to the test so that a determination of whether the system meets those requirements can be made. Requirements for system throughput or response times, in order to be useful as a baseline to compare performance results, should have the following attributes. They must be:

- Expressed in quantifiable terms.
- Relevant to the task a user wants to perform.
- Measurable using a tool (or stopwatch) and at a reasonable cost.
- Realistic when compared with the duration of the user task.
- Achievable at a reasonable cost.

Often, performance requirements are vague or non-existent. Seek out any documented requirements if you can, and if there are gaps, you may have to document them retrospectively.

Before a performance test can be specified and designed, requirements need to be agreed for:

- Transaction response times.
- Load profiles (the number of users and transaction volumes to be simulated).
- Database volumes (the numbers of records in database tables expected in production).

It is common for performance requirements to be defined in vague terms. You may have to do some requirements analysis yourself and document these requirements as the target performance objectives. These requirements are often based on guesstimated forecasted business volumes. You may have to get business users to think about performance requirements realistically.

2 Stable System



If the system is buggy and unreliable, you won't get far with a performance test. Performance tests stress all architectural components to some degree, but for performance testing to produce useful results, the system and the technical infrastructure has to be reasonably reliable and resilient to start with.

3 Realistic Test Environment



The test environment should to be configured so the test is meaningful. You probably can't replicate the target or production system. But for the results of the test to be meaningful, the test environment should be comparable in whole or part to the final production environment. You'll need to agree with the architect of the system what compromises are acceptable and which are not, or at least what useful interpretation could be made of test results.

4 Controlled Test Environment



Performance testers require stability in not only the hardware and software in terms of its reliability and resilience, but also need changes in the environment or software under test to be minimized. Test scripts designed to drive user interfaces are prone to fail immediately, if the interface is changed even slightly, for example.

Changes in the environment should be strictly controlled. If the change fixes bugs that are unlikely to affect performance, you might consider not accepting the release. Only changes intended to improve performance or reliability could be accepted.

5 Performance Testing Toolkit

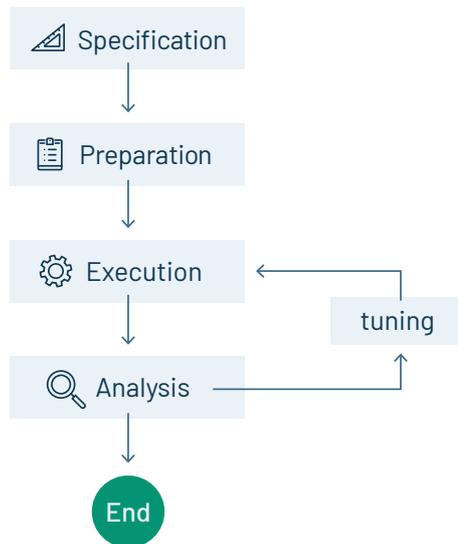


The key tool requirements for our “Performance Testing Toolkit” are summarized here:

- **Test Data Creation/Maintenance** – to create the large volumes of data on the database that will be required for the test. We’d expect this to be an SQL-based utility or perhaps a PC based product like Microsoft Access™, connected to your test database.
- **Load generation** – the common tools used to test drivers that simulate virtual clients by sending HTTP messages to web servers.
- **Application Running Tool** – this drives one or more instances of the application using the browser interface and records response time measurements. (This is usually the same tool used for load generation, but doesn’t have to be).
- **Resource Monitoring** – utilities that monitor and log client and server system resources, network traffic, database activity, etc.
- **Results Analysis and Reporting** – test running and resource monitoring tools generate large volumes of results data for analysis.

The Performance Testing Process

The diagram below shows a generic process for performance testing and tuning. Tuning is not really a part of the test process, but it's an inseparable part of the task of improving performance and reliability. Tuning may involve changes to the architectural infrastructure but should not affect the functionality of the system under test.



Incremental Test Development

Test development is usually performed incrementally in four stages:

- Each test script is prepared and tested in isolation to debug it.
- Scripts are integrated into the development version of the workload and the workload is executed to test that the new script is compatible.
- As the workload grows, the developing test framework is continually refined, debugged and made more reliable. Experience and familiarity with the tools also grow.
- When the last script is integrated into the workload, the test is executed as a “dry run” to ensure it is completely repeatable and reliable, and ready for the formal tests.

Interim tests can provide useful results. Runs of the partial workload and test transactions may expose performance problems. Tests of low volume loads can also provide an early indication of network traffic and potential bottlenecks when the test is scaled up. Poor response times can be caused by poor application design. This can be investigated and cleared up by the developers earlier. Early tests can also be run for extended periods as soak tests.

Test Execution

Test execution requires some stage management or coordination. You should collaborate with the supporting participants who will monitor the system as you run tests. The “test monitoring” team might be distributed so you need to keep them in the loop if the test is to run smoothly and results captured correctly.

Beyond the coordination of the various team members, performance test execution typically follows a standard routine.

1. Prepare database/data files.
2. Prepare test environment as required and verify its state.
3. Start monitoring processes (network, clients and servers, database).
4. Start the load simulation and observe system monitor(s).
5. If a separate tool is used, when the load is stable, start the application test running tool and response time measurement.
6. Monitor the test closely for the duration of the test.
7. If the test running tools do not stop automatically, terminate the test when the test period ends.
8. Stop monitoring tools and save results.
9. Archive all captured results, and ensure all results data is backed up securely
10. Produce interim reports; confer with other team members concerning any anomalies.
11. Prepare final reports with an analysis of the results.

Tuning usually follows testing when there are problems or where there are known optimizations possible. For repeated tests, it is essential that any changes in environment are recorded, so that any differences in test results can be matched with the changes in configuration.

As a rule, it is wise to change only one thing at a time so that when differences in behavior are detected, they can be traced back to the changes made.

Results Analysis and Reporting

The most typical report for a test run will summarize these measurements. For each measurement taken, the following will be reported:

- The count of measurements.
- Minimum response time.
- Maximum response time.
- Mean response time.
- Nth (typically 95th) percentile response time.

The load generation tool should record the count of each transaction type for the period of the test. Dividing these counts by the duration of the test gives the transaction rate or throughput actually achieved. The count of transactions is the load applied to the system. This assumes the proportions of transactions executed match the load profile you are trying to apply.

The load applied should match the load profile simulated - but might not if the system responds slowly and transactions run at varying speeds.

Usually, you will execute a series of test runs at varying load. Using the results of a series of tests, a graph of response time for a transaction plotted against the load applied can be prepared.

Resource monitoring tools usually have statistical or graphical reporting facilities that plot resource usage over time. Enhanced reports of resource usage versus load applied are very useful and can assist identification of bottlenecks in a system's architecture.

Reliability/Failover Testing

Assuring the continuous availability of a service may be a key objective of your project. Reliability testing helps to flush out obscure faults that cause unexpected failures so they can be fixed. Failover testing helps to ensure that the designed-in failover measures for anticipated failures actually work.

Failover Testing

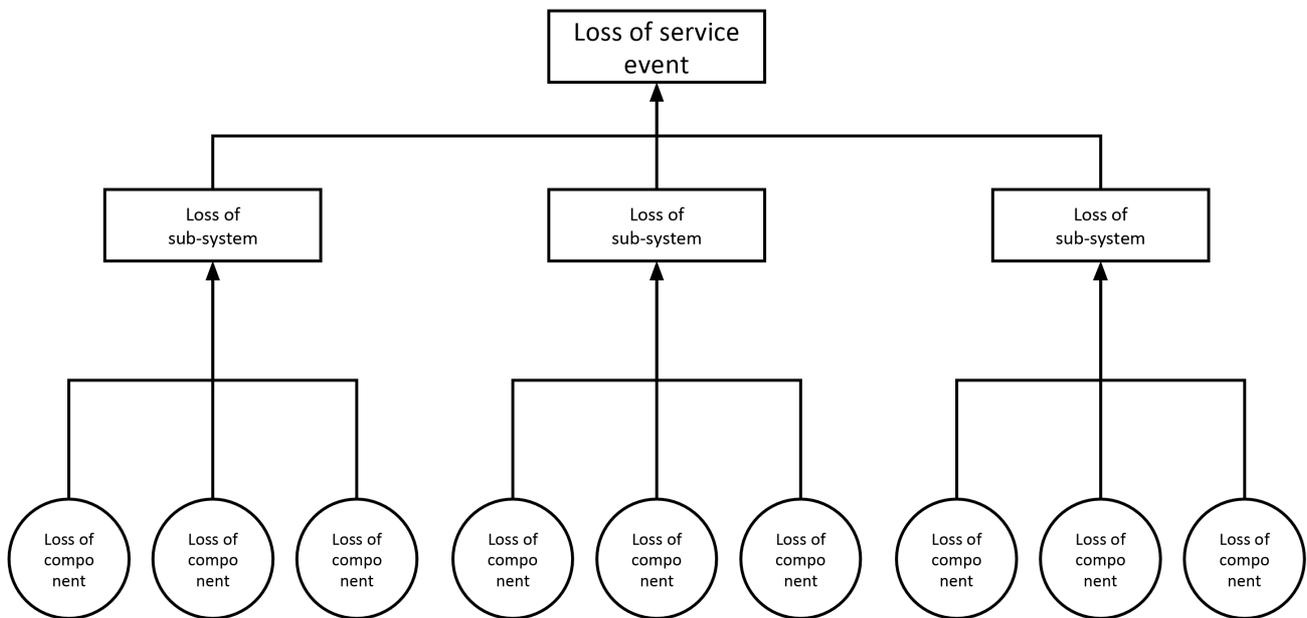
Where sites are required to be resilient and/or reliable, they tend to be designed with reliable systems components with built-in redundancy and failover features that come into play when failures occur. These features may include diverse network routing, multiple servers configured as clusters, middleware and distributed service technology that handles load balancing and rerouting of traffic in failure scenarios.

Failover testing aims to explore the behavior of the system under selected failure scenarios before deployment and normally involves the following:

- Identification of the components that could fail and cause a loss of service. (Looking at failures from the inside out).
- Identification of the hazards that could cause a failure and cause a loss of service. (Looking at threats from the outside in).
- An analysis of the failure modes or scenarios that could occur where you need confidence that the recovery measure will work.
- An automated test that can be used to load the system and explore the behavior of the system over an extended period.
- The same automated test that can be used to load the system under test and monitor the behavior of the system under failure conditions.

A technique called Fault Tree Analysis (FTA) can help to understand the dependencies of a service on its underlying components. Fault tree analysis and fault tree diagrams are a logical representation of a system or service and the ways in which it can fail.

The simple schematic on the following page shows the relationship between basic component failure events, intermediate sub-system failure events, and the topmost service failure event. Of course, it might be possible to identify more than three levels of failure event.



These tests need to be run with an automated load running to explore the system's behavior in production situations and to gain confidence in the designed-in recovery measures. In particular, you want to know:

- How does the architecture behave in failure situations?
- Do load-balancing facilities work correctly?
- Do failover capabilities absorb the load when a component fails?
- Does automatic recovery operate? Do restarted systems "catch up"?

Ultimately, the tests focus on determining whether the service to end-users is maintained and whether the users actually notice the failure occurring.

Reliability (or soak) testing

Reliability testing aims to check that failures don't occur under load. Most hardware components are reliable to the degree that their mean time between failures may be measured in years. Reliability tests require the use (or reuse) of automated tests in two ways to simulate:

- Extreme loads on specific components or resources in the technical architecture.
- Extended periods of normal (or extreme) loads on the complete system.

When focusing on specific components, we are looking to stress the component by subjecting it to an unreasonably large number of requests to perform its designed function. It is often simpler to stress test critical components in isolation with large numbers of simple requests first before applying a much more complex test to the whole infrastructure.

Soak tests are tests that subject a system to a load over an extended period of perhaps 24, 48 hours or longer to find (what are usually) obscure problems. Obscure faults often only manifest themselves after an extended period of use. The automated test does not necessarily have to be ramped up to extreme loads. (Stress testing covers that). But we are particularly interested in the system's ability to withstand continuous running of as wide a variety of test transactions to find if there are any obscure memory leaks, locking or race conditions.

Service Management Testing

When the service is deployed in production, it has to be managed. Keeping a service up and running requires it to be monitored, upgraded, backed up and fixed quickly when things go wrong. The procedures that service managers use to perform upgrades, backups, releases, and restorations from failures are critical to providing a reliable service so they need testing, particularly if the service is will undergo rapid change after deployment.

The particular problems to be addressed are:

- Procedures fail to achieve the desired effect
- Procedures are unworkable or unusable.
- Procedures disrupt the live service.

The tests should be run in as realistic a way as possible.

Putting It Into Practice

Consider the system you are working on now, or perhaps another system you worked on in the past that you know well. Suppose you had to design a performance test for that system. You have access to system logs that tell you the number and type of transactions used in a 'peak hour'. This peak is your design load which you will need to simulate.

You can't automate all transactions so you'll have to be selective of course.

What criteria might you apply to select which transactions will be used (or ignored) when you define the simulated load? (There are several).

Final Thoughts

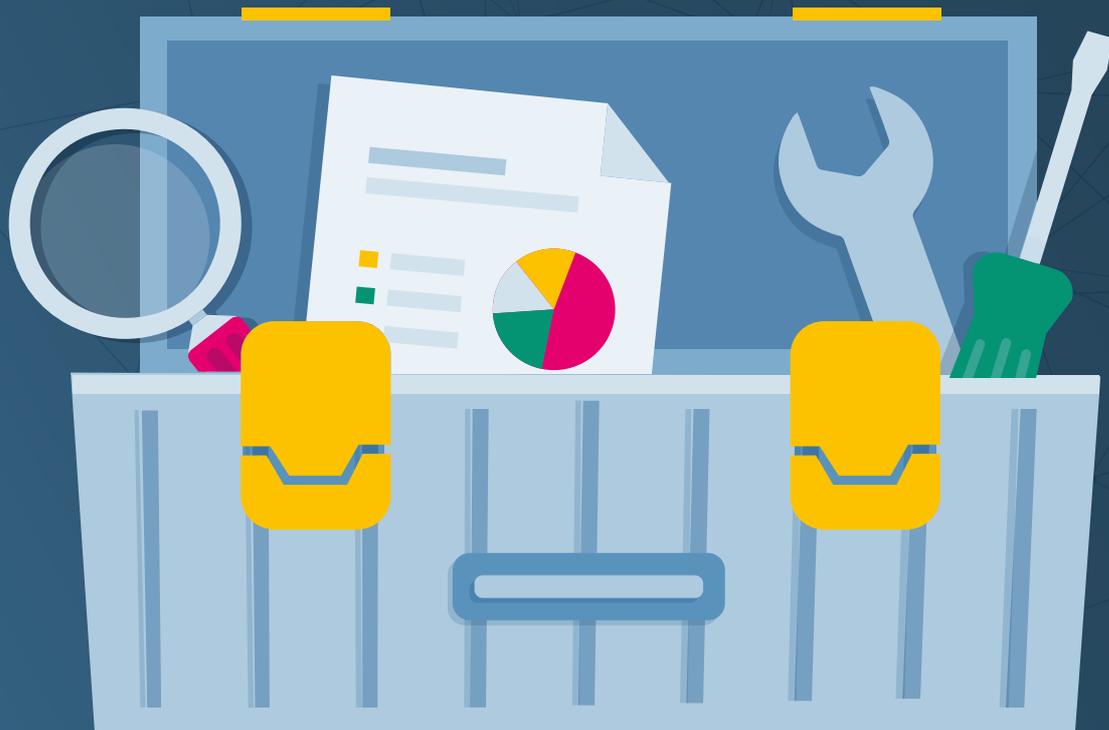
Some systems are prone to extreme loads when certain events occur. For example, a CRM system might be overloaded by salespeople updating their content in the system in the last few days of a sales quarter. An online business would expect peak loads just after they advertise offers on TV. A national news website might be overloaded when a tragic accident or other incident occurs.

Consider a system you know well that was affected by unplanned incidents in your business or national news.

What incidents or events could trigger excess loads in your system?

Can (or could) you capture data from system logs that give you the number of transactions executed? Can you scale this event to predict a 1 in 100 years or 1 in 1000 years critical event?

What measures could you apply (or have applied) to either reduce the likelihood of peaks, the scale of peaks or eliminate them altogether?



12 Secrets of Effective Test Management

11. Have the Right Tools and Infrastructure

Infrastructure is the term we use for all the hardware, cloud services, networking, and supporting software required to develop, test, deploy, and operate our application under test. It makes sense not to limit our definition to technology though. The data centers, office space, desks, desktops, laptops, tablet computers and mobile phones with their own apps are all part of the ecosystem.

You must also include the developer tools, DevOps tools and procedures, test tools and the business processes and subject-matter expertise required. The most mundane things – down to the access codes or smart cards used to gain access to buildings – can turn critical if they aren't in place.

Infrastructure, in all its variety, exists to support development, testing, deployment, and operations of your systems. It is either critical to testing, or it needs testing – by someone. In this section, we'll consider what most people would regard as a **test environment**, and also look briefly at what is often termed infrastructure testing. We'll also discuss tools for development, testing, and collaboration.

Test Environments

All testing makes an implicit, critical, simplifying assumption: that our tests will be run in a known, realistic environment. What this means is that for a test to be meaningful, the system to be tested must be installed, set up, deployed or built in an environment that simulates the real world in which the system will be used. We might use test scenarios that push the systems' capability in terms of functionality, performance or security, for example, but these are properties of the tests, not the environment.

Make It As Realistic As Possible

A realistic environment would replicate all of the business, technical and organizational environment too. Much of this environment comprises data that is used to drive business processes, provide reference data and the configuration of the system itself.

But perfectly realistic environments may be impractical or far too expensive. Even testers of very high criticality systems such as airplanes, nuclear reactors or brain scanners have to compromise at some point. Almost all testing takes place in environments that simulate, with some acceptable level of compromise, the real world.

Cars are tested on rolling-roads, in wind-tunnels, on vibration-beds and private test-tracks before they are tested on the open road. Computer systems are tested in software labs by programmers and software testers before end-users are engaged to try them out in a production-like environment.

Simulated environments are fallible just like our requirements and test models, but we just have to live with that.

We must stage tests that are meaningful in the environments we have available so that test outcomes really do mean what we interpret them to mean.

The reliability of test outcomes is dependent on the environment in which tests are run. If a test is run in an environment that is incorrectly set up:



A test that fails may imply the system is defective when in fact it is correct.



A test that passes may imply the system is correct when in fact it is defective.

Both situations are highly undesirable, of course.

Have It Ready on Time

Even with the emergence of cloud infrastructure, test environments can be difficult and expensive to set up and maintain. Often, just when support teams are working on the new production environment, the testers demand test environments. Late in projects, there always seem to be competing demands on support teams.

Environments for developers or any later test activity may be delivered late, or not at all, or they are not configured or controlled as required. Inevitably, this will delay testing and/or undermine the confidence in any outcome from testing.

A critical task is to establish the need and requirements for an environment to be used for testing, including a mechanism for managing changes to that environment – as soon as possible.

Infrastructure as code is a recent evolution in how environments can be constructed with tools following procedures and using declarative code to define the environment set up. Although base operating systems platforms – servers – can be created very easily in the cloud or as virtual machines in your own environment, fully specified special-purpose servers with all required software, data, configurations, and interfaces take more effort. Once set up, they provide a highly efficient means of environment creation. Infrastructure code can be source-controlled just like any application code and managed through change of course.

A major tenet of Continuous Delivery is that as soon as possible, some software, not even anything useful, should be pushed through the delivery pipeline to prove the processes work. Of course, this needs viable environments for builds, continuous integration, system-level testing, and deployment. The aim is to deploy test and production environments without constraint. Once the environment definitions and deployment processes are in place to do this, generating environments becomes an automated and routine task.

Under any circumstance, the definitions of these environments are an early deliverable from the project.

Development Environments

Developer testing focuses on the construction of software components that deliver features internally to the application or at the user or presentation layer. Tests tend to be driven by a knowledge of the internal structure of the code and tests may not use or require ‘realistic’ data to run. Tests of low-level components or services are usually run through an API using a custom built or proprietary drivers or tools.

The range of development tools, platforms and what are usually called Integrated Development Environments (IDEs) is huge. In this article, we can only touch upon some of the principal test-related requirements and features of environments.

To support development and the testing in scope for developers, environments need to support the following activities. This is just a selection – there may be additional or variations of these activities in your situation:



A **sandbox** environment to experiment with new software. Sandboxes are often used to test new libraries, to develop throwaway prototype code or to practice programming technique. All common programming languages have hundreds or thousands of (possibly free) software libraries. Sandboxes are used to install and test software that is not yet part of the main thread of development to evaluate it and to practice using it. These environments may be treated as disposable environments.



A **local development environment**. This is where developers maintain a local copy of a subset or all of the source code for their application from a shared code repository and can create builds of the system for local testing. This environment allows developers to make changes to code in their local copy and to test their changes. Some tests are ad-hoc and perhaps never repeated. Other tests are automated. Automated tests are usually retained for all time particularly if they follow a Test-Driven approach.



A **shared (continuous) integration environment**. When developers trust that their code is ready, they push their changes to the shared, controlled code repository. Using the repository, the CI environment performs automated builds, and executes automated tests. At this point, the new or changed code is integrated and tested. The CI system runs automated tests on demand, hourly or daily and the whole team gets notifications and sees the status of tests of the latest integrated build. Failures are exposed promptly and dealt with as a matter of urgency.

A development or CI environment supports developer testing, but other application servers, web services, messaging or database servers that complete the system might not be available. If these interfacing systems don't exist because they haven't been built, or because they belong to a partnering company and there is only a live system and no test version, then developers have to stub or mock these interfaces so at least they can test their own code. Mocking tools can be sophisticated, but mocked interfaces cannot usually support tests that require integrated data across multiple systems.

If an interface to a test database server is available to developers, the test data they use might be minimal, not integrated or consistent, and not representative of

production data. Development databases that are shared across a team are usually unsatisfactory – developers might reuse and corrupt or delete each other’s data if there isn’t a good regime for managing this shared resource.

System-Level Test Environments

System-level testing focuses on the integration of components and sub-systems in collaboration. The purpose of these environments is to provide a platform to support the objectives of larger-scale integration, validation of functionality, and the operations of systems in the context of user or business processes. Environments might also be dedicated to the non-functional aspects of the system, such as performance, security or service management.

One of the most common testing pitfalls is where a system tester in their environment experiences some kind of failure but no matter how hard they try, the developer or tester cannot reproduce the failure in the development environment.

“It works on my machine!” – yes, of course it does.

This is almost certainly caused by some lack of consistency in the two environments. The difference in behavior might be caused by the configuration, or some software version difference or the difference in data in the database.

The first thing to check for is differences in data. This is usually easy to identify and often can be resolved quickly.

When there is a software version or configuration discrepancy, testers and developers can waste a lot of time tracing the cause of the difference in behavior. When these problems occur, it often means there is either a failure in communication between dev and test, or that there is a loss of configuration control in the developer or test environment setup or the deployment process.

Infrastructure as code and automated environment provisioning may make environment consistency problems a thing of the past.

Types of Dedicated Test Environments

To support system, acceptance and non-functional testing, environments need to support the following activities (there may be more in your organization):



(Functional) system testing environment. A system is validated against the requirements documented for the system as a whole. Requirements may be large text documents, with tabulated sets of test cases defined for a system test. In agile projects, this environment may be necessary to allow testers to explore the integrated system without limiting themselves to specific features.



End-to-end test environment. Where a CI environment allows components to be integrated with subsystems, the business processes may require other interfacing systems to be available. Full-scope environments are required to conduct large-scale integration or business process or overall acceptance testing. Usually, data is a copy of production, or at least of appropriate scale. Where large-scale integration needs to be proven, the flows of data and control are exercised using longer user-journeys and independent reconciliations of the data across integrated systems.



Performance environment. These environments must provide a meaningful platform for evaluating the performance of a system (or selected sub-system(s)). Compromises in the architecture may be possible where there is redundant or cloning of servers. But the data volumes need to be of production scale even if the data is synthetic. Certainly, the environment needs to be of a scale to support production transaction volumes to enable useful prediction of the performance of systems in production to be made.



Availability, Resiliency, Manageability (ARM) environments. These environments are similar to the performance environments in some respects but may vary depending on the test objective. The goal of **availability testing** is to verify that the system can operate for extended periods without failing. **Resilience testing** (often called failover testing) checks that system components, when they do fail, do not cause an unacceptable disruption to the delivered service. **Manageability** or operations testing aims to demonstrate that system administrative, management and backup and recovery procedures operate effectively.

Tools and Test Management

Software teams that self-manage use a wider range of tools than ever before. In a typical software team, there might be twenty or even thirty tools in use across the team. Of course, the testers and other team members use different tool subsets, and you or someone in your team need to be the owner or be the expert in some of these tools. Some tools you need to be aware of because the output from them or increasingly, the decisions made by them affect you.

It is convenient to split the tools that are relevant to testing into three types:



Collaboration tools: these support the capture of ideas and requirements, communication across the team with some integration with automated processes, sometimes bots.



Testing tools: a large spectrum of tools that support test data management, test design, unit test frameworks, functional test execution, performance and load testing, static testing, test design, management of the test process, test cases, test logging, and reporting.



DevOps or infrastructure management tools: these tools support environment and platform management, deployment using infrastructure as code and container technologies and production logging, monitoring, and analytics.

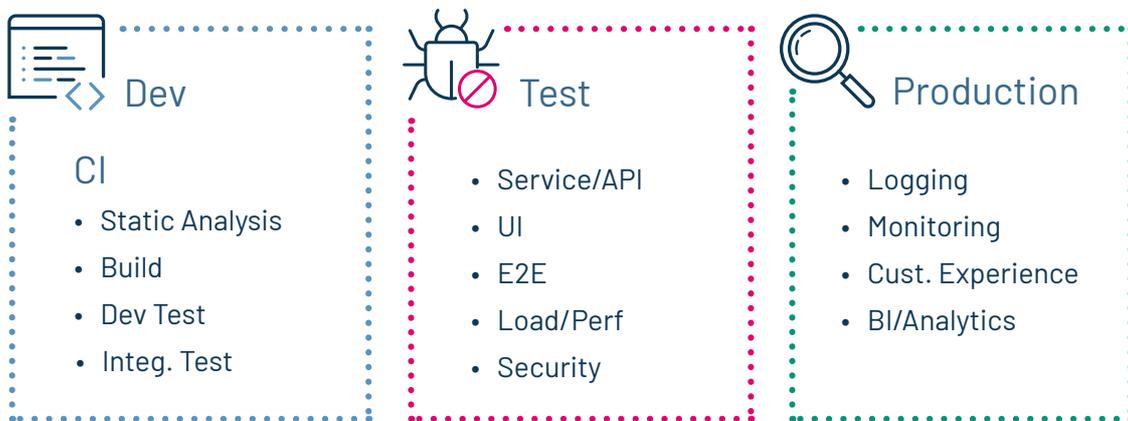
The Tools Knowledge Base (<https://tkbase.com>) is an online register for tools that differs from most online registers in that the scope of the register spans collaboration, testing, and DevOps. There are over 1700 tools across these three areas. Tools web pages are indexed and are searchable. The website also aggregates and indexes over 300 bloggers and over 52,000 blogs are also indexed and searchable. There are URLs to the major tools categories and shortcuts for searches of these categories.

Tool Architecture

The graphic below shows the range of tool types that most modern software teams use. These tools are supported by infrastructure tools that provide platforms, virtual machines, and containers to host environments and tools that perform automated deployments. The tools used to manage deployments and releases are called release and pipeline orchestration tools. Communications within the team and also with many of the automated processes are managed by collaboration or ChatOps tools.

Although the move towards DevOps is driving the development and adoption of tools to support continuous development, almost all of these tools are useful to any software development or operations team.

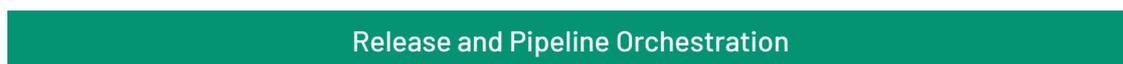
ENVIRONMENTS



PLATFORM



DEPLOYMENT



Test Management Tools

Test management tools are a must-have in all projects of scale. Agile projects usually adopt a tool for incident management and as for tests, place some reliance on the use of business stories and scenarios to track at least key examples, if not all tests. Test management tools vary in scope from very simplistic e.g. Microsoft Excel to comprehensive Application Lifecycle Management (ALM) products.

There are several types of test management tools:

- **Test Coverage Model:** Most test management tools allow you to define a set of requirements against which to map test cases and/or checks in tests. These requirements can sometimes be hierarchical to reflect a document table of contents. Increasingly, other models, e.g. use cases, business process flows or other models can be captured. Test plan and execution coverage reports are usually available.
- **Test Case Management:** Test cases and their content can be managed to provide a documented record of tests. The content of test cases might be prepared ahead of testing or as a record of tests that were executed. Test cases might be free text format or structured into steps with expected results. Importing documents and images to store against tests or steps is common.
- **Test Execution Planning:** Tests can be structured into a hierarchy or tagged to provide a more dynamic structure. Test team members might have tests assigned to them. Planned durations of tests might be used to publish a synchronized schedule of tests to be run across the team. Subsets of tests can be selected to achieve requirements coverage, to exercise selected features, to re-run regression test sets. Tests logged as not yet run, blocked, failed, or another status can be selected for execution, too.
- **Test Execution and Logging:** As tests are run by the team, the status of tests is recorded. All tests run would have the tester identified, the date/time and duration noted. Passed tests might be assigned a simple pass status. Failed, blocked or anomalous test results might have screenshots, test results assigned, and an incident report assigned. Many tools provide hooks to test execution tools that manage and run tests, log results and can even create draft incident reports.
- **Incident Management:** Test failures would be logged in the execution log. These usually require further investigation, with debugging and fixing where a failure was caused by a bug. Failures requiring investigation are usually recorded using incident

or observation or bug reports. Incident reports may have a large amount of supporting information captured. Usually, incidents are assigned a type, an object under test, a priority and severity. Some companies log a huge amount of information and match it with a sophisticated incident management process.

- **Reporting:** Reports and analyses of data from all of the above features as appropriate. The range of reports varies from planned v actual test coverage, incident report status to track outstanding investigation, fix and re-test work, analyses of time to fix for various failure types, by feature, severity and urgency and so on.

The most popular test management tool on the planet is still Microsoft Excel.

Test Design Tools

Test design is based on models. In the case of system and acceptance testers, typical models are requirements documents, use cases, flowcharts or swim-lane diagrams. More technical models such as state models, collaboration diagrams, sequence charts and so on also provide a sound basis for test design.

In many projects, models are used to capture requirements or high-level designs and when made available to testers, they can be used to trace paths to pick up coverage items off the model directly. If models like this are not available then it's often useful for the test team to capture for example process flowcharts or swim-lane diagrams. These help testers to have more meaningful discussions with stakeholders, particularly when it comes to the coverage approach.

In the proprietary space, tools are emerging which allow models such as flowcharts to be captured and used to generate test cases by tracing paths according to some coverage target e.g. all links, all processes, all decision outcomes, all-pairs and all paths. These tools can be linked with test data management and generation tools to generate test data combinations for use with manual or automated tests.

There are also tools which allow modeling to be done in test execution tools. These tools allow the test developer to capture all the fields on a web page for example, and create links to 'join up' the fields and create a navigation model for the page – all in a graphical format. The model is then used to create navigation paths to create a suite of tests that meet some coverage criteria – just like the modeling tools above. These

execution tools can create automated test paths using selected criteria or randomly generate them and also report coverage against these models.

Test Automation Tools

The topic of test automation (usually through a graphical user interface) is very high on most testers (and manager's) agendas. Superficially, tools appear to hold much promise, but many organizations wanting to automate some or all of their functional testing hit problems. We won't go into very much technical detail. But we'll touch on some of the issues relevant to test and project managers who need to create a business case for automation, but need to set realistic expectations for what automation can and cannot do. The next few sections might appear somewhat pessimistic.

What You Get and Don't Get from Test Execution Tools

Whether you are testing Windows desktop applications, web sites or mobile devices, the principles of test automation, the benefits and pitfalls are similar.

What you get from tools are:

- A tireless, supposedly error-free robot that will run whatever scripted tests you like, on demand, as frequently as you wish.
- A precise comparison between the output and/or outcome of tests with prepared expected results at whatever level of granularity you are willing to program into the tool.

What you don't get is:

- Flexibility and smooth responses to anomalies, failures or questionable behavior.
- The eyes and thinking of a human tester, capable of making decisions to explore, question, experiment and challenge the behavior of the system under test.
- Tests for free. You still have to design tests, prepare test data and expected results for example.

Let's explore the significance of these benefits and problems.

If you are a reasonably competent programmer, it is straightforward to implement procedural tests run by humans as automated procedures in your tool's scripting language for the tool to execute accurately. As long as the environment and data used by your test application are consistent, you can expect your tests to run reliably again and again. This is the obvious promise of automated test execution. But automated tests do not accurately simulate what (good) testers do when testing.

A test run by a tool is NOT equivalent to the same test run by a human.

A test, if scripted, might tell the tester what to do. But testers, by and large, look beyond the simple comparison between an expected result and the displayed result on a screen. Testers must navigate to the location of the test, and throughout the procedure, look out for anomalous behaviour – the response of commands; the speed of response; the appearance of the screen; the behaviour of objects affected by the changing state of the application under test as you navigate, enter data or make choices.

The human tester, when observing an anomaly might pause, re-trace their steps, look deeper into the application or data before concluding the application works correctly or not and whether to continue with the scripted test, to adjust the data and script of the test or to carry on as intended.

Tools do exactly what you program them to do. In a screen-based transaction, the tool enters data, clicks a button and checks for a message or displayed result – and that is all. With human testers, you get so much more that we take for granted. In principle, it is possible to program a tool to perform all the checks that a human performs instinctively – but you will have to write a lot of code and spend time debugging your tests. Even then, you don't get the human's ability to decide what to do next – to pause, to carry on, to change the test midstream or to explore an anomaly more deeply.

Nowhere is the inflexibility of tools more evident than when reacting to a loss of synchronization of the script with the system under test. Perhaps an expected result does not match, or the system crashes, or the behavior of the UI (a changed order of fields or a new field, for example) differs from what the script expects. What does the tool do? It attempts to carry on and a torrent of script failures or crashes ensues. Yes, you can and should have unplanned event handlers in your script – to trap test failures, for sure.

Loss of synchronization, mismatches between system state and test data, changes in the order of fields, new fields or removed fields are things the tester can recognize and decide on the next action without their testing to grind to a halt. Tools can't do this without programming effort on your part.

There are disadvantages in using humans to follow scripted tests. Testers can become obsessed with obeying the script and forget to use their observational skills. They might overlook blatant anomalies because they concentrate on following the script. This less-than-optimal approach is exactly what we get from test execution automation.

*Strict adherence to test scripts is a bad idea for human testers,
but it's the best we can do with test automation.*

In this comparison of people following scripts to tools running tests, we haven't considered what exploratory testers do. The exploratory approach gives testers the freedom to investigate and test wherever and however they want. Clearly, tools cannot simulate this activity. More importantly, tools cannot scope, prioritize and model a system's functionality and risks, and design tests to be run.

- Test analysis and design activities are required regardless of whether a test is run by a human or tool.
- There are further restrictions on what test execution tools can do:
- Test execution tools do exactly what you program them to do – no more, no less
- Tools don't usually perform environment and application builds and configuration, test data loading
- Tools don't do test case or script design, or prepare test data and expected results.
- Tools can't make thoughtful decisions when unexpected events occur.

GUI Test Automation

It has been twenty-five or so years since GUI test automation tools came onto the market, but the number of failed GUI test tool implementations is still high. This is mostly because expectations for automation are set too high – and tools used without discipline will never meet them.

GUI test automation tools have a reputation for being easy to use as products, but hard to manage when the applications under test (and therefore test scripts) need to change. These tools are very sensitive to change of the user interface. Changes of location, order, size or addition or removal of screen objects can all affect scripts. More technical changes such as the renaming of screen objects or the amendment of ‘invisible’ embedded GUI libraries cause problems too.

GUI test automation works best when there is discipline in:

- The development process and changes and releases are carefully managed. For example, changes are analyzed for impact and testers are kept informed.
- The test script development approach. Experienced test automation engineers adopt systematic naming conventions, directory structures, modular, reusable components, defensive programming techniques and so on.

Test automation scripting is a task that requires more than basic programming skills and above all, experience with automation and the tool used. Where tests are automated at scale, there is also a need for design skills. No matter how tools are described by vendors – “scriptless”, “usable by non-programmers” or “users can automate too” – you still need a programmer’s mentality, design skills and a systematic approach.

API and Service Test Automation

Where components or functionality deployed as services called by thin or mobile clients, testing will be performed using an API or via service calls as appropriate. This mode of testing is either done using custom code written by developers or using dedicated tools to do API or service testing. At any rate, the usual approach is to automate testing one way or another.

Because the API is 'closer' to the code to be tested, tests can be much more focused on the functionality to be tested. Certainly, the complexities of navigation through the UI and testing through the UI itself can be bypassed. For this reason, the general rule is:

If the functionality under test can be tested by a function call, an API or service call, then automation will be easier – and is recommended.

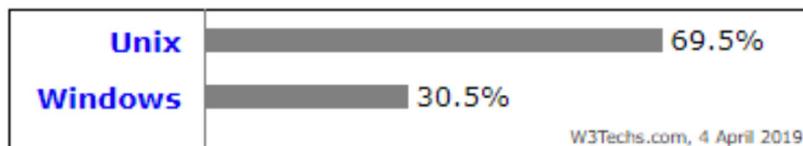
Testing through the API has definite advantages.

- Once an API call is scripted, it's just a case of tabulating the range of test cases you want to apply. There is no limit to the number of tests you can apply.
- API-based tests usually run much faster, making this style of testing well-suited to continuous delivery approaches.

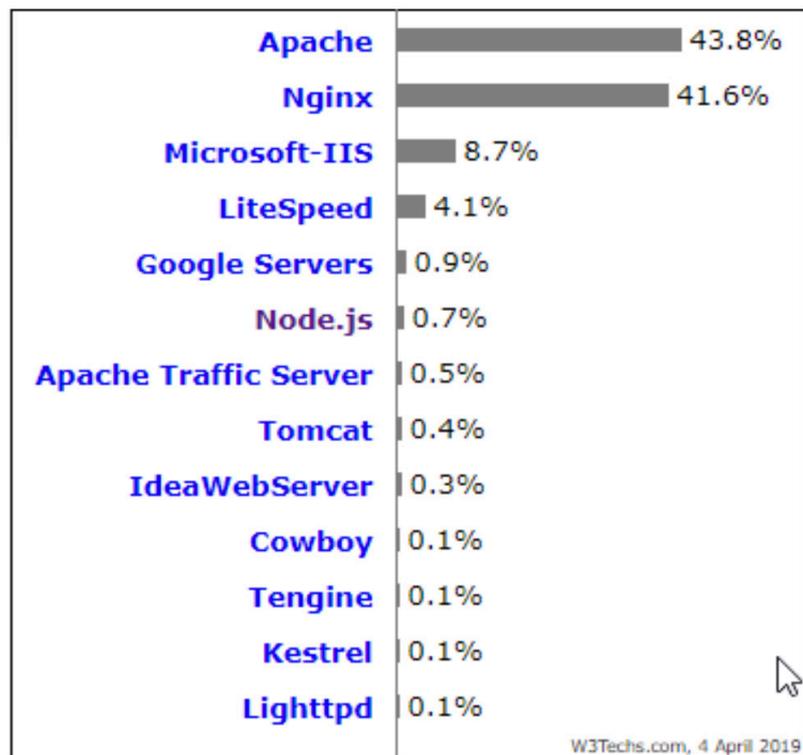
Use UI tests where the user workflow must be exercised, but in low volume because tests are expensive and can be slow to run. Use API or service calls where the functionality under test can be isolated and where speed is essential.

Proprietary or Open Source?

Over the past twenty years the use of free and open source software (FOSS) products, especially to run infrastructure, has been widespread. The cost of operating system licenses and associated webserver software from Microsoft, plus the general view that Linux/Unix is more reliable and secure than Windows, means that for many environments, Linux/Unix is the operating system of choice for servers. The two tables below (updated daily on w3techs.com) show the relative popularity of operating systems and web server products. Around 85% of sites run the best known open source Web server products, Apache and Nginx. Less than 0.1% are using macOS.



Percentage of operating systems used to host web sites. A website may use more than one O/S.
Source: https://w3techs.com/technologies/overview/operating_system/all



Web server software usage. A website may use more than web server.
Source: https://w3techs.com/technologies/overview/web_server/all

The popularity of these FOSS infrastructure products proves that open source can be as reliable as, if not more reliable than, proprietary products. For a software team needing the twenty or thirty software tools to support their activities, there are reliable and functional FOSS and proprietary tools to do every job. How do you choose between a proprietary and FOSS product?

The table below summarizes key considerations in selecting a tool type.

	Proprietary	FOSS
Availability	Tools available for every area.	Some areas, particularly development and infrastructure tools, are better supported than others.
Purchase cost	Often expensive, particularly 'enterprise' products.	Free or community-use license at zero cost. Commercial licenses may exist for enterprise or hosted versions.
Documentation	Usually very good.	Varies. Sometimes excellent, sometimes non-existent. Often written by programmers for programmers, so less usable than commercial documentation.
Technical Support	Very good, at a cost.	Varies. Some tool authors provide excellent support and will even add features on request. Many tools have online forums – but can be very technical. Other tools are poorly supported.
Reliability/quality	Usually very good.	Variable. Products with many users, locales, large support teams tend to be excellent. Some tools written by individuals with few contributors and few users can be flaky.
Richness of features	Feature sets tend to follow published product roadmaps and usually comprehensive.	Products tend to evolve based on user demand and the size of the team of contributors. Contributors tend to add features they need rather than based on customer surveys.
Release/patch frequency	Major releases tend to be months, sometimes years apart. Regular patch releases. Warnings and release notes tend to be very good.	Varies. Large infrastructure products may have major releases – like proprietary tools. Smaller, less popular products tend to release more frequently with little or no warning, poor release notes, and loss of backward compatibility on occasion.

FOSS products may be cheaper to obtain, but other costs and responsibilities can be significant. The deciding factor between the two is usually a mix of your culture, appetite for risk, and technical capability. When you buy proprietary products and support contracts, the risks associated with incompatibility (with other products), reliability, ease of use and attentive technical support are generally low, if sometimes expensive.

With FOSS products you usually have to do much more extensive research before you commit to using one. After all, there isn't a salesperson to talk to and documentation can be functional, rather than informative. Of course, a trial period is easy to set up and you can take on as many tools as you like, but you'll have to do a fuller investigation as to the tool's capability. Poorer usability and incompatibility with your existing tools might present problems, so you might have to write interfacing software or plug-ins, reporting or data import/export utilities. Also, you'll have to train yourself and team to bring them up to speed and usually do your own software support. But, your team will have a more intimate knowledge of the workings of the tool and be largely self-supporting.

A FOSS tool could help you to gain experience of a new tool type for little expense. With that experience, you'll be better placed to choose a proprietary tool for the long term.

Putting It into Practice

If you're in the market for a tool, make a list of 15-20 features that are either mandatory or desirable. This can include functional capabilities, integrations, a focus on ease of use, support, a large user base, or online forums/FAQs, for example.

Using the text of your requirements, search the Tools Knowledge Base to find three tools (including a proprietary and a FOSS product) that seem to match your requirements. Using the feature descriptions of the tools, create a feature comparison table of the three products. Add a fourth column for the tool you are actually using for comparison.

- How do the tools stack up in terms of features?
- Which features do the FOSS tool(s) lack, compared to the proprietary tools?
- How many tools exist that broadly meet your requirements?

For your current project or a recent one, make a list of all of the tools that your team uses to cover requirements, development, project management, testing, infrastructure, and release management. Map them to the Tools Architecture diagram at the start of the article.

Find some candidate tools that fill the gaps in your tools architecture. Try to find tools that integrate with your existing toolset.



12 Secrets of Effective Test Management

12. Develop Yourself

In the very first section of this ebook, this was the introduction:



If you are the test manager on a project, it is very likely that people will assume you are the expert on all things testing-related. Other team members will have their own views on testing. Some may have more experience than you. Expectations of testing are often unrealistic. There may be people who doubt your competence, value to the team, or even your motivations. It can be tough.

In your career, you will have to adapt to new and changing circumstances. You will encounter business and senior project stakeholders. You will be joining teams of various sizes with different people in them who have widely varying backgrounds. Your role might vary from being the tester to running a large team or to provide oversight to the testing in a project, being tasked with advising an Agile team or figuring out how testing fits into a continuous delivery regime.

This summary of what it is to be a test manager bears repeating. It sets out the real and significant challenges in the role:

- You work in a discipline that your management may not understand or appreciate
- Expectations placed on you and your team will probably be too high
- Your peers in development and DevOps may believe that 'anyone can test'
- Next year, and the year after, you'll be using different technologies from today
- The core skills you need to advance your career could include customer experience optimization (CXO), DevOps, assurance, automation, real-user monitoring, analytics, machine learning, artificial intelligence
- Testing is changing dramatically, and you need to adapt to survive.

In this final section, we'll explore these challenges and the future of test management and what steps you might take to survive and thrive in a new world.

Predictions of the future have a shelf life and in many respects, so does career advice. The technology industry is evolving all the time, but at the moment, the Digital wave is

dominating everything else. The effect on test managers, where companies are undertaking Agile or Digital Transformation can be serious. Many test manager roles are being lost – there is no need for test managers when testers are embedded with developers, or there is no test team, or testers at all. We'll also look at opportunities for test managers to reskill and take on alternative roles.

Evolving Test Management Skills

Whether you are a test manager running a team of testers or you are part of a self-managed team – the concepts, activities, and value don't really change.

What does change is the environment you work in. Certainly, the technologies change in minor ways every year, and every few years, there is a spike. Since mainframes ruled the world, we've had minicomputers, PCs, client/server, the internet, mobile, the internet of things and so on. Each technological advance has demanded we re-think what we do as technologists, architects, programmers, and testers. Our development approach is transforming from structured methods to agile and most recently continuous delivery and DevOps. Some are transformed, some are still on the journey and others work in environments where the old ways still work well enough.

The motivation for these articles was that we would focus not on the technology or the ways of working. Rather we'd focus on the principles of testing and test management, free from what we call logistics – process, technology, business environment and so on. By understanding the underlying principles, we prepare ourselves to deal with anything that is thrown at us in projects. Of course, you need to stay up to date with technologies and new thinking on approaches. It's impossible to stay on top of everything, but you know what? Technologies, methodologies and tools come and go, but the principles stay the same.

Learn to Adapt

The first advice we have is that you must learn to adapt. Remember the subtitle of Kent Beck's book, "Extreme Programming Explained" was, "Embrace Change". The Agile Manifesto builds on that simple message and encourages you to adopt a set of values that help you to be responsive, not resistant to change.

*Some people think that agile is an approach – it is not.
It is an attitude to change, or perhaps even a philosophy.*

So, as your technology landscape, processes and business environment change, you must trust that your principles are constant. Identify your stakeholders and ask them what they need from you. Use models to simplify complex systems and requirements. Use those models to communicate the testing challenge, to identify coverage measures and to explain how much testing will be done and has been done. And so on.

To be confident of the universality of test principles takes experience, but you must be open to that suggestion in the first place.

As you progress in your career and experience different working and technical environments, identify and separate the logistics and working practices and the thinking of project participants. Ask “based on the fixed inputs and context to this process how did this team decide to work this way?” Understand the difference between the two and how logistics are transient, but principles are universal.

Keep Volunteering, Keep Learning

As you and your team encounter new situations and problems, always be open to volunteering to take ownership of them or take the leadership role in solving them. In almost every respect, software projects are actually challenges in group learning.

- Stakeholders and users learn how to describe their problems and how systems and software can help to resolve them.
- Developers learn how to map software design to business outcomes and solve technical problems.
- Testers learn how to demonstrate how a system meets the needs of stakeholders and users and use the risk of failure to path-find mistakes and hazards and advise their teams on how to avoid them or provide evidence that they are mitigated.

In all respects, these learning challenges depend on information gathering, rational thinking, evidence-based decision-making, and persuasion skills. Volunteering to address or manage these challenges will teach you valuable lessons. Your reputation in the wider software team will be enhanced, and when projects reach critical stages,

senior managers will see you as a safe pair of hands, able to handle difficult tasks.

A willingness to volunteer to take on new, unique challenges gives you valuable experience and sets you apart from other members of your team.

Business, Leadership, Communication Skills

There are three non-technical skills areas that you need to excel. Having these skills will mark you out as an exceptional tester or manager. The three areas are:

- **Business Skills:** Consider learning more about your project stakeholders' business. If you are building systems for marketing, manufacturing, healthcare or banking for example, put effort into learning more about marketing, manufacturing, healthcare or banking. This will help you to have more intelligent conversations with stakeholders, to understand their motivations and concerns, and to be taken more seriously by senior decision makers. If you work for a software product company, learn how the company operates from the perspective of product sales and marketing, licensing, the product development lifecycle, maintenance and billing. In this way, you will be seen as a more knowledgeable and rounded professional and again, will understand the motivations and concerns of key stakeholders.
- **Leadership Skills:** Leadership is probably the most important management skill there is. Leadership is required to build strong, effective and stable teams. The logistics of recruitment, delegation, supervision and direction are important, but the effective leader needs to set a vision for the team, devise an approach or process to achieve the mission and the ability to build personal relationships with team members to gain their respect, loyalty and commitment. Leadership courses might help, but most people learn through experience and with the support of a leadership coach or mentor (see later). Don't rely on just your technical and organisational skills. Great leadership skills will set you apart from your peers.
- **Communication Skills:** Perhaps the biggest difference between being a developer or tester is the need to interact and communicate with other team members, management, stakeholders and suppliers. A developer's main interaction may be with a small number of peers, a business analyst and a tester. A tester or test manager on the other hand, at one time or another, needs to deal with almost every role within a software team. Beyond that, they might be liaising with senior stakeholders, and external suppliers of services, software and hardware.

Communication is a two way street. Your ability to assimilate large amounts of possibly conflicting information from diverse sources is a constant challenge. Of course, you also need to communicate outwards – to your team, to stakeholders, to developers and suppliers and it's not just handing off data. You need to inform, to persuade and influence in various ways and situations.

Find a Coach or Mentor

It goes without saying that to enhance your skills and progress in your career, you'll need some help. Help in this case comes in the guise of coaches and mentors and although the two terms are often confused, we think of them as distinct roles. There are many variations of the definitions, styles, purpose and value of mentoring and coaching. We can only skim over the subject here, so do some research to see what would be most useful to you. This is our summary of what coaches and mentors can do for you.

Coaching: the goal of coaching is usually to help a coachee to achieve a specific goal such as learning how to better manage a team and thus improve performance. Coaches are not necessarily experts in the field you need coaching in. Coaches tend to ask endless questions and challenge you all the time. The goal of coaching is to facilitate thinking and action on the part of the coachee. Typical questions would be:



- What is your goal?
- What do you need to achieve it? What support, resources, time, money... and so on.
- Is your goal meaningful, attainable, worthwhile?
- How will you measure progress?
- As you progress, is your goal still a good motivator? Is the goal changing?
- Are you making progress? What are the barriers to make progress?
- How will you overcome barriers? Can you finesse or avoid them?

The coachee has to think their way through the problem and do all the work. By thinking clearly, your goals, motivation and understanding of how to make progress all come from within.

Mentoring: The goal of mentoring is usually to develop some skill or capability and make an incremental change for example to transform to agile or to implement test automation. A mentor is typically an expert in the field to be worked in. (This does not preclude them from being a coach too). Typically, the dialogue between mentor and mentee is less one-sided, with the mentee asking as many questions and the mentor offering suggestions. In this way, the mentor helps the mentee to think more clearly, but also may give advice or at least set out options for the mentee to consider.

A testing mentor may suggest an approach or offer a framework to putting together a test strategy, for example. The mentee collects the data, and writes up the various sections of the strategy and the mentor might give feedback and advice on how to improve the content, writing style, level of detail, filling in gaps and so on.

These definitions of coaching and mentoring are not standardised. In your own organisations there may be accepted definitions and even recommended coaching and mentoring protocols – so pay attention to those.

Be a Coach or Mentor

A large part of management and leadership in particular, is helping your team to develop and perform better. In some ways, as you first meet and get to know your team members, you will develop a lasting personal relationship with your role emerging as a coach or mentor to each member of your team. The relationships you have as coach and mentor are obviously different and cover different career aspects than the relationship you have with your own coaches and mentors. But by playing both roles you will gain insights on how best to develop people and develop yourself.

A lot of people who enjoy coaching or mentoring volunteer for roles outside their working environment. If you have a background in a sport such as soccer or athletics you might coach children in your favourite sport. Other avenues include adult education, business coaching, teaching something you do as a hobby such as gardening or home maintenance.

At any rate, being a coach and mentor will expand your outlook, improve your interpersonal and communication skills and enhance your confidence with people.

Presentation

Everyone from time to time has to do a presentation at work. Presentations are used to communicate and share knowledge in a large range of situations and contexts such as:

- Explaining how an organisational change will affect your team
- Communicating the outcome of a phase of testing to stakeholders for them to make a decision
- Sharing an experience or case study with your team to facilitate a conversation about new ways of working
- Introducing the background, goals, risks and plan for a new application to be built and tested by your team
- Showcasing the work of your team to peers and managers in an internal conference

Presentation goals vary from the sharing factual information to giving persuasive pitches of something you want to sell – a new process or your own skills to a prospective employer. Formal presentations might not come your way very often, or you might present a progress report daily in a stand-up or a management meeting. At any rate, the skill of presentation is extremely valuable and whether you are talking to one person or one hundred, it's a matter of practice and knowing your subject that will make you confident and assured.

A common route to better presentation is to submit proposals to technical conferences. These range from the international shows at conference centres or four star hotels with a thousand delegates, a large exhibition, pre-conference tutorials and keynote speakers to local meetups with ten or fifteen enthusiastic peers in a pub.

Needless to say, if you are new to presenting, it's easier to start with a local meetup than to pitch for a keynote at an international show. Meetups are always looking for speakers, especially experience talks from practitioners. Find a meetup in your area, attend one or two, ask the organiser what they are looking for in talks – and make your offer. All internationally renowned keynote speakers started somewhere and a meetup is as good a place as any.

Moving Forward

If your organization embarks on an agile transformation, it is possible your test management role will be phased out (or quickly eliminated). Whether you think it is a good decision to drop the test management role or not, you have to deal with it. In these circumstances, you usually have a choice. To revert back to being a tester in an agile team, or a specialist in technical or performance testing, to evolve your role to be assurance-related or to leave the company. If you leave the company, you'll either find another internal test management role or join a testing service company.

We'll look at two of these options: assurance and testing services.

Moving to Assurance

Assurance (not quality assurance) is an emerging discipline which is testing related, rather than being a testing discipline. Ex-test managers are well qualified because they know how to run an internal testing service, have experience of dealing with suppliers (of development or test services) and of dealing with senior and stakeholder management.

Assurance roles embrace a range of disciplines:

- Test Assurance roles involve overseeing the testing of external or internal organizations. Typically, you might be involved in the definition or review of test strategy. Once the project kicks off you provide consulting support to test teams in the early phases, but later, review and assess the performance of testing and give an independent perspective of the reporting done by suppliers to project boards.
- Assurance leadership roles include the test assurance activities above, but the scope of responsibility might extend in several other directions. You might have an advisory and independent review responsibility for all deliverables of the project. So you might review requirements, designs, test plans, risk assessments, phase entry and exit meetings, phase-end reporting and attend regular project status reporting session to give your independent view of progress. Some roles extend into data migration, talent acquisition, cutover and roll-out phases of a project. In principle, where there is a project activity with dependencies and deliverables, there is a potential role of assurance.

- Assurance Management might require you to run a small team of people who specialize in, for example, test assurance, performance testing, information assurance, user experience design and assessment and so on. In larger projects and the largest projects for central government, for example, assurance teams are common.
- Agile testers perform “assurance in the small”. An agile tester performs all of the activities mentioned above but in the context of a smaller team. They liaise and challenge stakeholders; they support and coach developers in testing; they advise and steer the work of users who tests; sometimes they do some testing themselves.

Typically, assurance roles require that you have no day-to-day involvement or responsibility for delivery. It is important that you are able to take an independent view of the performance of teams throughout a project – both internal and external. It can be a rather isolated or lonely job – you are not part of the project team except by secondment. But you are usually accountable to the project sponsor and business stakeholders. It can be a very senior and influential role on occasion.

Assurance requires a broad range of technical and non-technical skills and strong management and communication skills.

Becoming a Consultant for a Service Company

A common pathway for skilled test managers is to join a testing service company. Testing services span all roles from junior tester to large program test management and assurance positions. Providing professional services requires a slightly different perspective than you might be used to, having run an internal test team in the past and it is not always a comfortable transition.

You might be called on to get involved in a project very early and be responsible for drafting test strategies for projects in business domains you are not familiar with. The client’s and their suppliers’ working practices might be very different from what you are used to. This can be daunting in critical business programmes, especially where timescales and projects are under pressure to deliver. Sometimes you might find yourself in a role where you feel underqualified; sometimes the reverse is true and you have to work with or for less-experienced client personnel.

At any rate, becoming a consultant and working with a range of clients over time, is a great way to accumulate a lot of experience in what might be a short period of time. As a consultant, your range of capabilities will increase over time, and if you choose to leave testing services to return to a permanent role in an IT department, you should be much better qualified to join at a more senior level.

Testing services requires good technical and interpersonal skills, but more than anything, you will need to have a flexible and pragmatic attitude. Working with clients can be alternately exhilarating and boring, exciting and frustrating. Again, it's not for everyone and sometimes, you have to travel more than you would like, but it could give you a lot of experience in just a few years.

Whatever your career path, we wish you the best of luck.

Putting It Into Practice

The first step in making a career plan is to assess and understand your own competencies. There are many assessment questionnaires on the internet for example, but you might find the table below helpful. It allows you to assess yourself in nine specific dimensions. The table is a transcription of the text of page 89 in "Development and Assessment Centres" by Charles Woodruffe, ISBN 9-780852-928523.

Rate yourself on a scale of 1 to 10 and make a note of where you need to improve.

Competency	Score (1-10)	Areas for improvement
<p>Breadth of awareness to be well-informed: Develops and maintains networks and formal channels of communication, within the organization and in the outside world; keeps abreast of relevant local, national and international political and economic developments; monitors competitor activity; has a general awareness of what should be happening and what progress is being made.</p>		
<p>Incisiveness to have a clear understanding: Gets a clear overview of an issue; grasps information accurately; relates pieces of information; identified causal relationships; gets to the heart of the problem; identifies the most productive lines of inquiry; appreciates all the variables affecting an issue; identifies limitations to information; adapts thinking in the light of new information; tolerates and handles conflicting/ambiguous information and ideas.</p>		
<p>Imagination to find ways forward: Generates options; evaluates options by examining positive and negative results if they were put into effect; anticipates effects of options on others; foresees others' reactions; demonstrates initiative and common sense.</p>		
<p>Organization to work productively: Identifies priorities; thinks back from deadline; identifies elements of tasks; sets objectives for staff; manages own and others' time.</p>		

Competency	Score (1-10)	Areas for improvement
<p>Drive to achieve results: Is prepared to compromise to achieve a result; installs solution within timeframe; innovates or adapts existing procedures to ensure a result; takes on problems; suffers personal inconvenience to ensure problems are solved; comes forward with ideas; sets challenging targets; sets out to win new business; sets own objectives; recognises areas for self-development; acquires new skills and capabilities; accepts new challenges.</p>		
<p>Self-confidence to lead the way: Expresses and conveys a belief in own ability; is prepared to take and support decisions; stands up to seniors; is willing to take calculated risks; admits to areas of inexpertise.</p>		
<p>Sensitivity to identify others' viewpoints: Listens to others' viewpoints; adapts to other people; take account of others' needs; sees situation from others' viewpoint; empathizes; is aware of others' expectations.</p>		
<p>Co-operative to work with other people: Involves others in own area and ideas; keeps others informed; makes use of available support services; utilises skills of team members; is open to others' ideas and suggestions.</p>		
<p>Patience to win in the long term: Sticks to a strategic plan; does not get side-tracked; sacrifices the present for the future; bides time when conditions are not favourable.</p>		

About the Author

Paul Gerrard is a consultant, teacher, author, webmaster, developer, tester, conference speaker, rowing coach, and a publisher. He has conducted consulting assignments in all aspects of software testing and quality assurance, specializing in test assurance. He has presented keynote talks and tutorials at testing conferences across Europe, the USA, Australia, South Africa and occasionally won awards for them.

Educated at the universities of Oxford and Imperial College London, in 2010, Paul won the Eurostar European Testing Excellence Award and in 2013, won The European Software Testing Awards (TESTA) Lifetime Achievement Award.

In 2002, Paul wrote, with Neil Thompson, "Risk-Based E-Business Testing". In 2009, Paul wrote "The Tester's Pocketbook" and in 2012, with Susan Windsor, Paul co-authored "The Business Story Pocketbook".

He is Principal of Gerrard Consulting Limited and is the host of the UK Test Management Forum and the UK Business Analysis Forum.

About TestRail

We build popular software testing tools for QA and development teams. Many of the world's best teams and thousands of testers and developers use our products to build rock-solid software every day. We are proud that [TestRail](#) – our web-based test management tool – has become one of the leading tools to help software teams improve their testing efforts.

Gurock Software was founded in 2004 and we now have offices in Frankfurt (our HQ), Dublin, Austin & Houston. Our world-wide distributed team focuses on building and supporting powerful tools with beautiful interfaces to help software teams around the world ship reliable software.

Gurock is part of the [Idera, Inc.](#) family of testing tools, which includes [Ranorex](#), [Kiuwan](#), [Travis CI](#). Idera, Inc. is the parent company of global B2B software productivity brands whose solutions enable technical users to do more with less, faster. Idera, Inc. brands span three divisions – Database Tools, Developer Tools, and Test Management Tools – with products that are evangelized by millions of community members and more than 50,000 customers worldwide, including some of the world's largest healthcare, financial services, retail, and technology companies.