POCKET-SIZED INSIGHTS FOR SOFTWARE TEAMS



TEAM GUIDE TO SOFTWARE TESTABILITY

Ash Winter and Rob Meaney

3

conflux

Team Guide to Software Testability

Better software through greater testability

Ash Winter and Rob Meaney

This book is for sale at http://leanpub.com/softwaretestability

This version was published on 2021-08-19

ISBN 978-1-912058-69-3



This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2016-2021 Ash Winter and Rob Meaney

Contents

1. *Team Guides for Software* **1**

2. Praise for *Software Testability* **2**

3. Introduction **3**
- 3.1 Why is testability important? 3
- 3.2 What does hard-to-test feel like? 5
- 3.3 What does testable feel like? 6
- 3.4 What leads to testability being neglected? 6
- 3.5 What is covered in this book? 7
- 3.6 How to use this book 8
- 3.7 Why we wrote this book 8
- 3.8 Feedback and suggestions 9

4. Set a pragmatic direction for improving testability using trade off sliders **10**

5. Create testability targets to improve interactions with dependencies **11**

6. Adopt testability mapping to expose hard-to-test architectures **12**
- 6.1 Poor architectural testability causes slow feedback and flawed decisions 13
- 6.2 Identify the symptoms of poor architectural testability 16

6.3 Exercise: Use testing smells to diagnose poor architectural testability 17
6.4 Exercise: adopt testability mapping to measure testing feedback and waste 25
6.5 Summary 34

7. Use historical data to fix design problems that inhibit feedback 36

8. Adopt ephemeral development environments for fast feedback 37

9. Use production data to enhance your testing strategy 38

10. Use team testing reviews to enable sustainable delivery 39

11. Terminology 40

12. About the authors 41

13. Conflux Books 43

1. *Team Guides for Software*

Pocket-sized insights for software teams

The *Team Guides for Software* series takes a *team-first approach* to software systems with the aim of **empowering whole teams** to build and operate software systems more effectively. The books are written and curated by experienced software practitioners and **emphasize the need for collaboration and learning, with the team at the centre**.



Titles in the *Team Guides for Software* series include:

1. *Software Operability* by Matthew Skelton, Alex Moore, and Rob Thatcher
2. *Metrics for Business Decisions* by Mattia Battiston and Chris Young
3. *Software Testability* by Ash Winter and Rob Meaney
4. *Software Releasability* by Manuel Pais and Chris O'Dell



Find out more about the *Team Guides for Software* series by visiting: http://teamguidesforsoftware.com/

2. Praise for *Software Testability*

(This chapter is not available in this edition of the book.)

3. Introduction

3.1 Why is testability important?

How can software teams adapt and change the software systems they build and run? How can we get 'early warning' of performance, compatibility and integration problems? How can we design software architectures that enable ongoing evolution? The answer is to have a sustainable focus on testability.

Testability is a vital property of modern software. It enables software teams to make changes rapidly and safely. With clear feedback loops to understand the impact of changes. The testability of your software product links to its operability and maintainability. There is a close relationship between the extent to which the software meets customer needs and how testable it is. If you want to drive improvements in both speed and agility, testability is the fuel for organizations delivering modern software.

Team dynamics

Imagine if there was a constant focus on how to observe, control and understand the product within your team. Teams need all kinds of skills, and everyone brings different skills to the table. Developers need it to implement change confidently, testers need it to assess risk and provide actionable information, customers need it to achieve their goals with the product. Creating a focus on testability can help bring teams and their customers closer together.

Predictability

Have you ever been asked if your team could go a little bit faster? Or worse, as an individual on behalf of the team? We know we have. We believe that what your customers are *really* asking for is a little more predictability, delivering software consistently to a known level of quality. Being able to observe, control and understand a system describes predictability pretty well. If your system struggles with those three characteristics, how can you imagine delivering in a predictable fashion? Testability isn't the only piece of the predictability puzzle, but it's a big part.

Supporting your business and customers

A focus on testability helps to support many aspects of your business. Operations can control the system, product management can experiment with features for feedback, marketing and design can get feedback on who is using the product and if customer's needs are being met. Most products have peak consumption periods when many stakeholders use the system simultaneously. This could be due to a major music or sporting event, for example.



A testable system provides information about its own limits, giving your business and technical stakeholders the ability to make crucial decisions at peak times, when reputations can be won or lost.

Fast feedback

For us, one of the real questions in software development is: how can we get meaningful feedback on what we have created? Be it quantitative feedback through usage statistics and financial

measures, or qualitative interpretation of customer experience, enhancing testability can enrich this feedback.

This is also true for feedback on system failure. Meaningful information on failure modes enhances your ability to test a system and its time to recovery after an outage or error, which is crucial for those who support the system. When things go wrong, being able to observe and control the system will be invaluable to both business and technical operations.

Data, not opinion

Everyone in the team has opinions. However, focusing on testability as a team will help you go beyond opinions and have data to support you. Being able to objectively test the assumptions and hypotheses behind the functional and operational features you build helps establish a dialogue with product stakeholders. Enhancing testability allows quantifying the business and operational impact, while also moving conversations away from those who shout loudest to those with the right data.

3.2 What does hard-to-test feel like?

Take your mind back to 'that' product. The one which caused you quite a lot of pain, phone buzzing in the middle of the night, and yet another failed `cron` job. The product that caused a lot of distress to the poor operations people when you handed them a complete mess.

Do you recognize any of these traits?

- Interacting with a product gives you no feedback. No logs, no audit trail, only mysterious unmonitored dependencies. You don't know if anything went well. Or badly.

- Interacting with a product gives you vast amounts of feedback. Log files spam uncontrollably, lights twinkle on dashboards, integrated dependencies give strange, opaque answers. You don't know if anything went well. Or badly.
- You release your product. Scores of utterly baffling issues pop up. Seemingly unrelated but somehow intertwined. Next release makes you twitchy.

3.3 What does testable feel like?

We would like to evoke a different feeling when you talk about your next project or product. Imagine a product where:

- You are in control of the amount of feedback your product gives you, allowing either a deep, detailed view or a wide, shallow view. Rather than trying to parse what your product is doing, it will tell its story.
- The product can be set into a state of your choosing, whether that be data or configuration, allowing you to develop your product with much more certainty.
- After release, you are not dreading the call from support or account management that your customers are unhappy. Any problems are flagged early and can be proactively managed.

3.4 What leads to testability being neglected?

During our careers, we have seen various maturity models of the "'ilities" of software deployed within organizations. Testability might make an appearance every now and then but it rarely seems to rank as high as scalability or reliability. We hope to give strategies for overcoming apathy and some common constraints to testability:

- No paradigm - if you don't understand what it is, how can you ask for it? Never mind describing its benefits to those who are paying for the product development.
- Nobody knows who should be responsible for it - product people think it has nothing to do with them, developers think it's testers' responsibility, testers don't communicate the pain of a hard-to-test system to developers.
- There's no time - "we need to build the thing now", when the pressure is on, with deadlines looming, there is no time for testability.
- It's not a functional requirement - testability and other operational features never make it into the backlog. But these features are what turns functionality into a product.

3.5 What is covered in this book?

Testability is a vast topic, which permeates much of software development. Our focus will be on how the team can enhance their systems and interactions with stakeholders through a focus on testability. We will share our experiences, those of others we have encountered along the way and many practical resources and references we can use on our testability journey.

What we *will* cover includes:

- Improving interactions within and between teams to enhance testability both through conversation and documentation
- Practical approaches for creating testable architectures, addressing common risks and useful solutions
- Taming your environments from local, disposable environments all the way to live ones
- Keeping the focus on testability through the concept to customer and beyond the lifecycle of a product

Although this book touches on some technical aspects of software testing, it doesn't focus on how to make specific artifacts like legacy code or user interfaces more testable. There is a lot of great guidance out there for those, for example "Working Effectively with Legacy Code" by Michael Feathers (2004). Testability is not only about testers and, by extension, not solely about testing!

3.6 How to use this book

Ideally, each chapter is meaningful independently. You should be able to identify your situation and pick the thinking, tools and techniques that help you most in your current situation. However, like most books, this also represents our story, what we have learned over the years. Hopefully those lessons can help you with your lessons, as a holistic experience.

Concepts and theory are important, but we value practical examples highly in this book and have sourced them from various friends within the world of software development, as well as ourselves. These examples are critical to the book and will hopefully provide value to our readers.

Most of all, we believe the process of developing systems and products is a joyful, collaborative endeavor. We hope that shines through the book!

3.7 Why we wrote this book

As experienced testers, we have seen teams struggling with hard-to-test systems for a long time. This has usually manifested itself in pain for testers of all kinds so we felt it. In our experience, there has been little will do anything about it. Through lack of knowledge, advocacy and an acceptance that hard to test is normal. Team testing cultures are hard to create. Enhancing testability makes this journey much easier.

Looking around at the wider body of knowledge about testability, we wanted to fill some gaps. In particular, creating architectures designed for testability has light coverage. Once locked into a hard to test architecture, retrofitting testability is a challenging journey. We hope the book helps those working on systems of all ages to improve their architectures.

We are both testers who have looked beyond our role into other disciplines, engaged in the whole system, including people, processes, products and much more. Improving testability involves that wider system over optimizing for local gains. It's time for a new practical focus on testability and this book contains the tools for teams to do that.

3.8 Feedback and suggestions

We'd welcome feedback and suggestions for changes: please contact us at info@confluxbooks.com, via @TestabilityBook on Twitter, or on the Leanpub discussion:
https://leanpub.com/SoftwareTestability/feedback

Ash Winter & Rob Meaney - August 2021

4. Set a pragmatic direction for improving testability using trade off sliders

(This chapter is not available in this edition of the book.)

5. Create testability targets to improve interactions with dependencies

(This chapter is not available in this edition of the book.)

6. Adopt testability mapping to expose hard-to-test architectures

Key points

- Low-testability architectures contribute to the conditions for shallow testing, or a neglect of testing entirely.
- Symptoms of low testability architectures are present in a team's everyday work. Problems need to be surfaced before they can be addressed.
- High-testability architectures enable different types of testing and a comprehensive view of quality.

This chapter covers techniques for detecting low testability in software architectures.

6.1 Poor architectural testability causes slow feedback and flawed decisions

In order to make changes to your architecture which will have the biggest positive impact on your testability, we need to understand what symptoms are present, their roots and how to judge progress on an ongoing basis. Without gathering data first and regularly, we are at risk of:

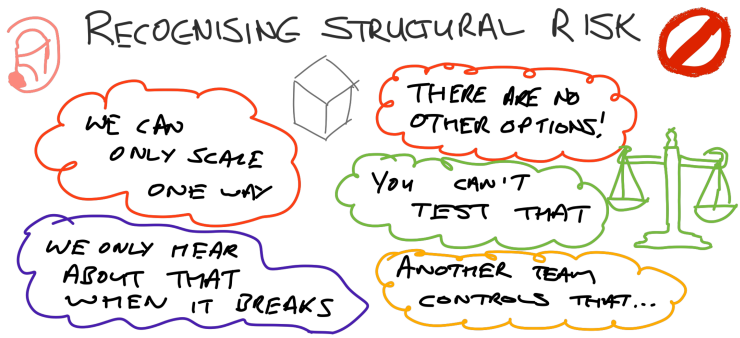
- Prematurely optimizing our architecture based on a single point in time rather than allowing the picture to evolve as we discover more.
- Changing our architecture with a bias to local optimization, rather than removing the overall constraints of the system.
- Focusing on our own architecture when some of our biggest testability challenges might lie within our dependencies.

Over the course of a system's life, many decisions are made along the way based on needs of stakeholders, available technologies, contributors' knowledge, and even the problems that occur during development. Often, these decisions can fundamentally influence the testability of an architecture.

> When Ash was working at a large credit broker he experienced this:
>
> *I was working on a team performing a major replatforming exercise. They were essentially creating a replica of an existing system built on LAMP (Linux, Apache, MySQL and PHP) during which the database was being ported to Microsoft SQL Server. This was mainly because the rest of the organization was Microsoft-based and had little MySQL knowledge.*

> *Doing so put the system architecture into a "bespoke" state, which no-one had seen within the organization before [which no-one within the organization had foreseen?]. Plus, support for database drivers between PHP and Microsoft SQL Server was sparse, as was knowledge of successful implementations in the wider development world.*
>
> *In addition, all the monitoring tools were subsequently split between two technology stacks, which made it hard to observe the whole system easily, as it removed the previously complete monitoring which existed for the LAMP application.*
>
> *The team dynamic changed too. The PHP development team used to support MySQL to a significant degree (as they were the only ones with sufficient knowledge to do so), but after moving to the new technology, enhancements, updates and support were assigned to a large team of Database Administrators, which limited our control of a layer of our own architecture.*

We knew we had a fundamental, structural risk within our architecture. How does a team protect the system from future impacts of such a change?



Structural Risk

You will likely have examples like this within your own history. We will go into recognizing the symptoms of poor testability in more detail later in the chapter. When faced with decisions about the architectural testability, consider the following risks:

Risk	Impact
If it's hard to test, it won't get tested	If your architecture is hard to configure, operate and analyze, testing will be too. Moreover, tests likely won't be run, or at least they will uncover less useful information
Important problems are hard to find	We want to find the problems that have the greatest impact on our stakeholders. High complexity and low observability can hide problems during testing and in operation. Production problems that are hard to reproduce in earlier environments occur often.
If it requires I/O it's going to be slow	If your tests rely on network or disk I/O then they are going to be slow and unreliable without test doubles.
Feedback slows when complexity grows	A tightly coupled architecture pushes testing into the higher layers of the system, which can lead to slow feedback and many combinations of tests to run without knowing which components are covered.

Risk	Impact
Test as close to the change as possible	When designing your system architecture provide mechanisms that allow the team to test the logic as close to source as possible. This allows for faster, more accurate defect isolation.

6.2 Identify the symptoms of poor architectural testability

The testability of your architecture is not always obvious. In our experience, you are more likely to notice the symptoms in your daily work. This can lead to a number of frustrations for the team and the stakeholders interested in the delivery of product. An architecture with poor testability limits the team in many ways.

> Ash experienced this while working at a large gaming company:
>
> *"We were testing a hybrid iOS application, a website with limited native components. Just to begin testing on a mobile device, multiple configurations had to be navigated. Pull a feature branch, build a development environment, configure relevant external feeds, join a specific WiFi network, change DNS, configure a local proxy to intercept and change certain headers and URI's. It would take a couple of hours on average to get going with testing, and that's when it all went well."*

The symptom here was that it took a long time to begin testing, which had a direct impact on the team:

- Poor quality of exploratory testing - everyone was exhausted

by the time testing began! This led to a lot of limited, shallow testing, more akin to demonstrations.
- Long queues of ready to test work - team boards had long queues of tickets, waiting to be tested, in a mobile environment where time to device is critical.
- Limited test automation - writing automated tests for customer journeys that didn't yield false positives or false negatives was difficult. They fell behind the cadence of delivery, became regarded as unreliable and were removed from the build pipeline to stop them blocking releases.

It took a while to shrink the time taken to get from "ready to test" to "testing". However, once we did, the team and product stakeholders were much happier.

- Whole-team testing: since it now took less time to get our development environment onto a real device, the whole team started doing it more often, developers got feedback on real devices quickly, and key user journeys were protected with automation.
- Less context switching: shorter queues in testing columns meant developers and testers were synchronized, and were delivering as a team again.

We want to help you to recognize these symptoms so you can begin to act on them to discover your own root causes.

6.3 Exercise: Use testing smells to diagnose poor architectural testability

A testing "smell" is a symptom of a testability problem. We believe each of the smells listed below can be tamed by establishing a

more testable architecture. This exercise aims to determine which testability smells impact your team and the extent of the impact on both the team as a whole and each of the individuals. The findings of this exercise will allow your team to make informed decisions about which testability smells to tackle first.

What does the exercise involve?

In this exercise we will first ask each individual in the team to rate 15 common testing smells in terms of the impact the smell has on their ability to do their job effectively. The 15 smells are listed below.

#	Testing Smell	Example Impacts
1	**Too many production issues**	Does your team feel that too many issues are escaping into production? Is your team's planned work frequently interrupted and delayed as a result of dealing with production issues?
2	**Pre-release regression cycles**	Does your team have to execute a lengthy regression test cycle before releasing? Does your team often find important issues during this regression cycle?
3	**Lack of automation & exploratory testing**	Does your team frequently check and confirm things that should be done using automation? Does your team overlook exploratory testing?

#	Testing Smell	Example Impacts
4	**Hesitance to change code**	Is your team hesitant to make small, regular code improvements for fear it will introduce undetected issues? Does your team feel uncomfortable refactoring the code even when they believe it's necessary?
5	**Testing not considered during architectural design**	Does your team neglect to involve testers in the architectural design discussions? Does your team neglect the impact on testing when making design decisions?
6	**Team constantly seeking more testers**	Does your team feel like they need to add more testers as a result of mounting workload and complexity?
7	**Too many slow user interface tests**	Does your team waste a lot of time preparing, executing and waiting for feedback from slow GUI tests, either manual or automated?
8	**Important scenarios not tested**	Does your team release the system without testing important scenarios because they are either impossible or impractical to test? Are there areas of significant risk that are not being tested?

#	Testing Smell	Example Impacts
9	**Ineffective unit and integration tests**	Does your team write unit tests and integration tests that often miss important problems? Does your team endeavor to continuously improve your unit and integration tests?
10	**Cluttered, ineffective logging**	Do your logs contain lots of errors and warnings even when the system is considered to be behaving as normal? Can team members quickly and easily isolate and debug issues using the logs?
11	**Flaky nondeterministic automation**	Does your team spend a large proportion of their time investigating failures, debugging and maintaining automation? Does your team re-run automation when it fails expecting it to pass the second time?
12	**Tests that contain duplication & irrelevant detail**	Does your team have tests that contain a lot of duplicate steps (usually setup) in order to get it in a state to perform the essential part of the test? Does your team have tests that contain lots of details that have nothing specifically to do with what you're actually trying to test?

#	Testing Smell	Example Impacts
13	**Issues are difficult to reproduce**	Does your team often encounter issues that are difficult, time consuming or impractical to reproduce either in your test environments or production?
14	**Issues are difficult to isolate & debug**	Does your team struggle to isolate and debug issues when they occur either in your test environments or production? Does it take days of investigation to find the root cause of a problem?
15	**Too much effort spent writing, maintaining and debugging automation**	Does your team rely too heavily on automation written at the UI level? Does your team test business logic through the UI?

When we talk about the impact of each of the testing smells consider the effects in terms of time, effort, your ability to deliver value and the satisfaction of the team in their work. The ratings are simple:

1. **No impact on team effectiveness**
2. **Small impact on team effectiveness**
3. **Moderate impact but rarely impacts team effectiveness**
4. **Moderate impact but often impacts team effectiveness**
5. **Large impact, almost always impacts team effectiveness**

Follow this facilitator's checklist to help you run the exercise:

Who do we need?

We suggest involving architects, developers, testers, ops and analysts.



For full details of the roles, see Team Test for Testability exercise in Chapter 1.

Preparation

- Time - One hour
- Type of Space - An open space with a whiteboard
- Attendee Preparation
  - Distribute testing smells sheet & explanations before the exercise.
- Physical Materials
  - Hard copies of the testing smells sheet for every member of the team
  - A whiteboard or flipchart
  - Sticky notes and pens (one color)
- Digital Materials - None

Sample agenda for pre-communication

- Set the Stage (5 mins)
  - Explain goals and FAQ
  - Introduction to the testing smells.
  - Introduction to scale of impact on team effectiveness.
- Gather Data (10 mins)
  - Create a matrix of the 15 testing smells and allow team members to vote using post its or some other indicator.
- Generate Insights (40 mins)

  - Explore the obvious differences in rating, particularly any outliers from the general consensus by encouraging individual contributions.
  - Capture insights from the discussion and a team consensus on the rating for each smell.
- Agree actions (5 mins)
  - The team discusses and agrees the top 3 smells they would like to address first.

Facilitation

- Change the wording of the questions for common terminology at your organization if you need to.
- The gather data step is better done anonymously, we want to learn about individual perceptions and the differences/similarities between them.
- Actively seek to drive discussion to attain shared understanding.
- If discussions about outliers result in deeper understanding of a smell, allow team members to adjust their rating.

Frequently Asked Questions

- What if someone wants to add a smell?
  - Check if it's covered already, but mostly say yes!
- What if the team can't find consensus on a rating?
  - If the team can't agree then go with the lower proposed rating. Most likely if there's a lack of consensus it's not the most important issue.

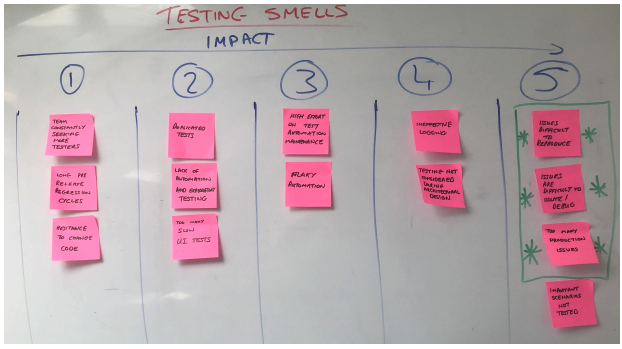
Goals

- Cultivate a shared understanding of the impact of not having a testable architecture, on individuals and also on the team as a whole.
- Create an appetite within the team to establish a testable architecture.
- Capture insights that other people within the team may not have previously considered.
- Reach consensus within the team about which testing smells need to be acted upon.

Outputs

- A matrix of smells, team consensus and interesting points.
- A list of the top 3 testing smells that the team is committed to trying to address.

After the exercise you should have something like this:



Testing Smells

What do the results mean?

As you can see each of the smells has been allocated to an impact level based on the team consensus. One could infer from the results that this team believes the following:

- They have a reasonably balanced test strategy without any significant problems in the effectiveness of their test automation, aside from a high maintenance cost of their automation due to flaky tests.
- By far their biggest concern is when issues during development are hard to reproduce, isolate and debug, which may contribute to the incidence of issues leaking into production.

The next step is to determine what this might mean for architectural testability. The team agreed to tackle three areas highlighted with asterisks in the photo above (*issues difficult to reproduce*, *issues are difficult to isolate/debug*, and *too many production issues*). We hope this inspires a few ideas for tackling poor architectural testability that can be refined in the subsequent exercises of the chapter, such as:

- Difficulty in reproducing issues might be mitigated by implementing controllable log levels by request or a unique ID which is attributed to a transaction throughout its journey through the system.
- Issues that are difficult to debug and isolate might be assisted by "stubbing" dependent services in order to fix the response type or response time to limit variance and narrow down possible causes.

6.4 Exercise: adopt testability mapping to measure testing feedback and waste

Typically, when we encounter hard-to-test systems there is a distinct separation between the testing performed by the developers and testers in a team. Often, the developers invest their efforts in writing unit and integration tests while the testers write UI tests

and perform the vast majority of exploratory testing. Typically, the tester has little visibility or involvement in the testing performed by the developer and vice versa. As a result, the unit and integration tests (performed by the developers) rarely find important problems, while the UI automation requires constant attention (from the dedicated testers), which reduces the amount of exploratory testing done.

A testable architecture helps break this "silo culture" and promotes whole-team involvement in design discussions. By involving testers in design you bring a wealth of testing knowledge and context to the discussion. Team members work together to understand the important quality attributes, critical paths, core components and associated risks and agree on a design that allows the team to mitigate those risks in the most effective manner.

A testable architecture allows the team to build effective automation that's easy to write, fast to run and whose results can be trusted. From a developer's perspective, it provides the confidence to make small changes continuously, knowing that regressions will be detected almost immediately.

There's also the added benefit of making defect isolation easier by encouraging the creation of isolatable components which allow for pinpointing defects; this defect isolation is further helped by greater observability which allows us to determine the defective components internal state from metrics, logs and alerting (Waterhouse).



From a tester's perspective, a testable architecture allows them to increase the depth of their testing, to perform tests that would otherwise be impossible or impractical, and to provide more detailed information to developers when issues are found.

What does this exercise involve?

As a team you will identify the different types of testing needed to deliver a product increment. This could be within whatever planning sessions you use as a team. This might be release or sprint planning, for example, or a story planning session (often known as a "Three Amigos" session (Dinwiddie)).

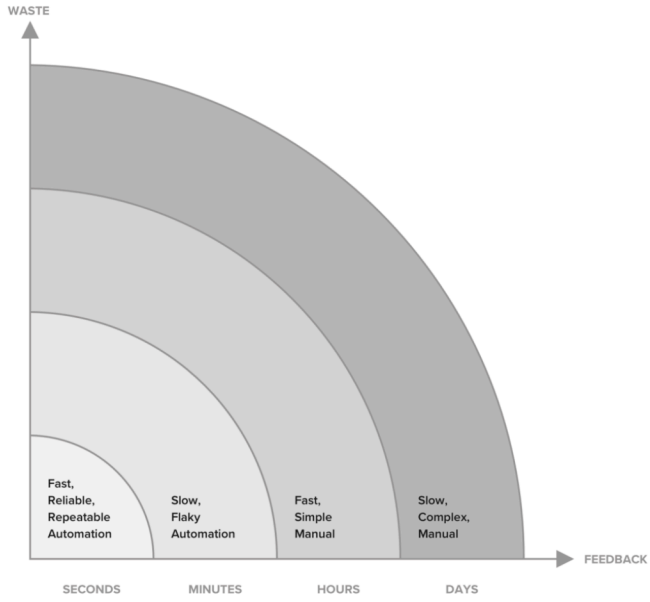
The key to this exercise is collaboration between product, development and test stakeholders. For each type of testing that is meaningful to you, create what we have called a 'testability map.'

You might run this exercise when you have a major decision to make regarding your system architecture such as:

- Larger new features that represent a departure from current architectural patterns, such as adding a new service as a microservice.
- The introduction of a new technology, such as a container orchestration engine or resource-intensive operations into cloud services.
- A new project or product would benefit from this thinking too. Adding functionality which uses File Input and Output for example will present testing challenges which are best anticipated and mitigated earlier.

What is a testability map?

A testability map looks like this:



Example Testability Map

- The Y axis of the maps concern the amount of waste that occurs when attempting to test. If you have dependencies which are unreliable and suffer from significant downtime then testing may take longer or you might do more testing to gain confidence.
- The X axis concerns feedback over time, the longer it takes to receive feedback on a deployment the more risk is accumulated. For example, if you have a system with limited test automation and data dependent manual tests which require a lot of setup, then it may take days to get meaningful feedback.
- Each segment (for example - Slow, Complex, Manual) is a representation of the amount of waste you will incur in order to get the feedback you need.

You need to run the exercise for these 4 types at a minimum. if you

don't perform one or more of them, then your testing lacks balance. For example, too much focus on automated acceptance testing at the expense of exploratory testing could increase your exposure to unexpected problems later in a feature's development cycle, when you could have applied variance earlier on to avoid them. You can of course add more testing types that are important/challenging for you:

Testing Type	Description	Examples
Acceptance Testing	Demonstrates that the system can successfully provide the capability that it was designed to provide. These tests are often derived from user story acceptance criteria or BDD scenarios and aim to confirm the feature works as designed	Basic flows, sunny day scenarios, business logic, validation, functional tests, contract tests.
Exploratory Testing	Exploring, experimenting and interacting with the system in varied ways in order to unearth the unknown and unexpected. This type of testing is focused on finding problems and developing a deeper understanding of the system.	User journeys driven by personas, resilience, error handling, failure scenarios, usability, rainy-day scenarios, risk-based tests, stress tests.

Testing Type	Description	Examples
Regression Testing	Detects unintended behavioral changes in the existing functionality that have been introduced as a result of adding new functionality. Typically, acceptance tests become a part of your regression test suite once a feature has been released.	Basic flows, sunny day scenarios, business logic, validation, functional tests, contract tests.
Operability Testing	Detects and isolates issues when the code is deployed in a fully functional pre-production or production environment. These typically focus on ensuring that business critical functionality is working when deployed in a given environment.	End to End Synthetics, User journey tests, monitoring, alerting, logging, analytics, instrumentation, performance testing.

How does the exercise work?

Follow this facilitator's checklist below to help you run the exercise.

Who do we need?

We suggest involving architects, developers, testers, ops and analysts.



For full details of the roles, see Team Test for Testability exercise in Chapter 1.

Preparation

- Time
  - 1 hour - as part of planning a product increment or architectural change.
- Type of Space
  - An open space with a large whiteboard
- Attendee Preparation
  - Familiarity with the current system architecture and the proposed changes.
  - Familiarity with the four testing types:, acceptance, regression, exploratory and operability.
- Physical Materials
  - A whiteboard or flipchart
  - Sticky notes and pens
- Digital Materials - None

Sample agenda for pre-communication

- Set the Stage (5 mins)
  - Goals, Outputs, FAQ
- Draw the architecture (20 mins)
  - A member of the team draws the system architecture with assistance and feedback from the team.
  - The facilitator needs to timebox this exercise and ensure that the discussion stays focused on the topic.
- Associate Test Ideas with the System Architecture (20 mins)
  - Get the team to brainstorm test ideas, adding a sticky note for each test idea they believe is necessary in order to release the product increment with confidence.
- Position stickies on the testability maps (20 mins)
  - The team works through each of the segments placing each of the stickies in the appropriate area.

Facilitation

- Ask for a volunteer from the team to draw the system architecture with assistance & feedback from all members.
- Run through each question & seek clarity on each of the answers until you feel the team is happy to move on.
- Take a picture of the system architecture (in case you have to remove it).
- Draw four testability maps on the whiteboard labelled acceptance, regression, exploratory and operability.
- Explain which test ideas belong in each of the segments.
- Prompt the whole team to add stickies to the appropriate segment for each testing idea or task they believe is necessary to deliver the feature with confidence.
- In order to help generate test ideas, consider using a testing mnemonic as a framework (McKee). This will help the test idea generation to flow, or at least get started.

Frequently Asked Questions

- What if the team can't agree on the architecture?
  - Allow each member of the team express their views; if the team can't agree, the team lead decides.

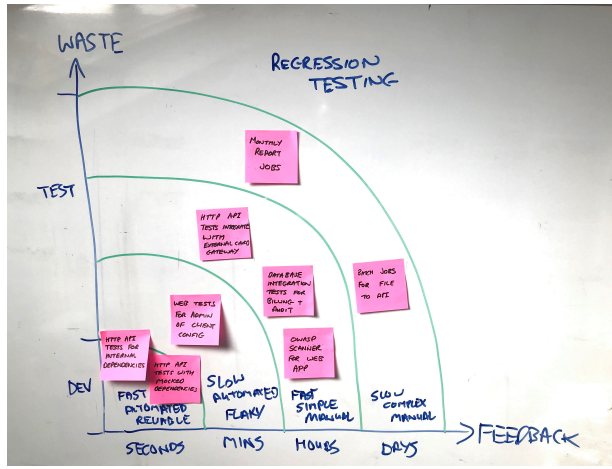
Goals

- Create a shared understanding in the team of how the current design may inhibit efficient testing.

Outputs

- A set of testability maps which express the types of testing most meaningful to your team for that feature, product increment or architectural change.

After the exercise you should have something like this:



Example Testability Map - with details

What do the results mean?

This team were focused on regression testing testability map, as they were looking to upgrade a number of their dependencies, specifically their versions of PHP using to build their HTTP API, Website and Batch File Processing system, plus MySQL for data storage.

- **Seconds** - The team recognized that they had a number of tests for their HTTP API, which used real internal dependencies and mocked external dependencies. These would be very useful for fast feedback on PHP version changes in their API, but they had no fast mechanism for getting feedback on their website.
- **Minutes** - The team's automated user interface tests sat in this segment, and the website received less attention as API traffic increased so the risk of upgrading versions felt greater. As the PHP version changes and the code changes

for deprecated methods were made, these would be better (at least partially) be run in their development environment.
- **Hours** - the team felt that database integration tests for crucial information for billing customers and audit information for compliance purposes would be needed. In addition, the team wanted to do some testing of their API with the real version of the payment gateway, rather than the mocked version in their development environment. They decided to bring these external card gateway tests into the 'Minutes' segment, but run them on a more infrequent basis in their pipeline.
- **Days** - The pattern you can see in the testability map for regression testing is fairly common, and the team that completed the exercise knew their major problems lay in the less frequently run monthly reporting and batch jobs. Although they run less frequently, they are still critical functionality but have very slow feedback; plus, they are only set up in test environments closer to Production. The team agreed that testing these in earlier test environments would be prudent.

6.5 Summary

This chapter covered:

- The risks of low architectural testability, which we believe not only compromises your ability to test effectively but also your overall ability to deliver.
- Listing the common smells of low architectural testability and which ones matter the most to the team currently. This shows where your pain lies in the team's perception. Problems oftem get underrated as time goes by and familiarity increases.
- Diagnosing how your current architecture and future changes affect the effectiveness of your acceptance, exploratory, regression and operability testing.

Low architectural testability is a real source of pain for teams. We often inherit systems we don't understand from previous teams, then we do our best to deliver with a hard-to-test system. This chapter has expressed that the key is to detect and diagnose architectural challenges to effective testing.



7. Use historical data to fix design problems that inhibit feedback

(This chapter is not available in this edition of the book.)

8. Adopt ephemeral development environments for fast feedback

(This chapter is not available in this edition of the book.)

9. Use production data to enhance your testing strategy

(This chapter is not available in this edition of the book.)

10. Use team testing reviews to enable sustainable delivery

(This chapter is not available in this edition of the book.)

11. Terminology

(To follow in a future version)

12. About the authors



Ash Winter

I'm Ash Winter, a consulting tester and conference speaker, working as an independent consultant providing testing, performance engineering, and automation of both build and test. I have been a team member delivering mobile apps and web services for start-ups and a leader of teams and change for testing consultancies and their clients. I spend most of my time helping teams think about testing problems, asking questions and coaching when invited. I am most proud of being a co-organizer for the Leeds Testing Atelier, a free full-day community testing workshop. Its mission is to give those involved in testing a platform to share their stories, particularly those who haven't been heard before.



Rob Meaney

I'm Rob Meaney. I came to work in the software industry soon after finishing college (where I gained a degree in electrical & electronic engineering) and began working as a tester without even knowing

what testing was. I learned my trade testing desktop applications for the manufacturing safety automation industry. Soon I got bored of manually checking the same thing over and over to I decided to try automating some of my tests. Since then, I have worked in start-ups, gaming, data storage, medical, fraud detection and communication companies building test and automation frameworks. I've worked as a manual tester, automation architect and test manager. I love testing and read and learn about testing and development continuously.



13. Conflux Books

Books for technologists by technologists

Our books help to accelerate and deepen your learning in the field of software systems. We focus on subjects that don't go out of date: fundamental software principles & practices, team interactions, and technology-independent skills.

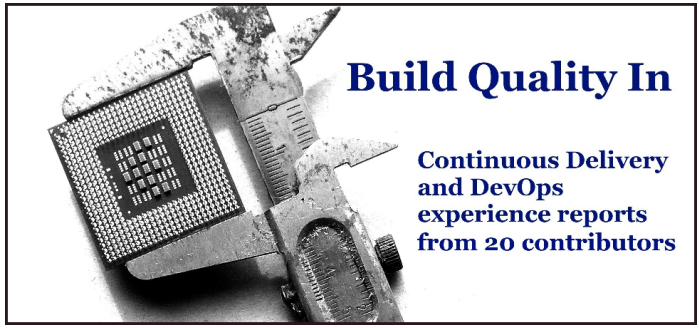
Current and planned titles in the *Conflux Books* series include:

1. *Build Quality In* edited by Steve Smith and Matthew Skelton (B01)
2. *Better Whiteboard Sketches* by Matthew Skelton (B02)
3. *Internal Tech Conferences* by Victoria Morgan-Smith and Matthew Skelton (B03)
4. *Technical Writing for Blogs and Articles* by Matthew Skelton (B04)



Find out more about the *Conflux Books* series by visiting: confluxbooks.com

conflux



Build Quality In - a book of Continuous Delivery and DevOps experience reports. Edited by Steve Smith and Matthew Skelton. Conflux Books, April 2015



Internal Tech Conferences by Victoria Morgan-Smith and Matthew Skelton. Conflux Books, April 2019



Better Whiteboard Sketches by Matthew Skelton. Conflux Books, August 2019



Team Guides for Software: *Software Operability, Metrics for Business Decisions, Software Testability, Software Releasability.* Skelton Thatcher Publications, 2016-2019